Tell us about your PDF experience.

ASP.NET documentation

Learn to use ASP.NET Core to create web apps and services that are fast, secure, cross-platform, and cloud-based. Browse tutorials, sample code, fundamentals, API reference and more.



GET STARTED
Create an ASP.NET Core app on any...



GET STARTED
Create your first web UI



OVERVIEW
ASP.NET Core overview



DOWNLOAD
Download .NET



GET STARTED
Create your first web API



GET STARTED
Create your first real-time web app

Develop ASP.NET Core apps

Choose interactive web apps, web API, MVC-patterned apps, real-time apps, and more

HTTP API apps

Develop HTTP services with ASP.NET Core

- Overview
- Create a minimal web API with ASP.NET Core
- Create a web API with ASP.NET Core Controllers
- Generate web API help pages with Swagger / OpenAPI

Interactive client-side Blazor apps

Develop with reusable UI components that can take advantage of WebAssembly for near-native performance

- Overview
- Blazor hosting models
- Build your first Blazor app
- Build your first Blazor app with reusable components

Page-focused web UI with Razor Pages

Develop page-focused web apps with a clean separation of concerns

- Create your first Razor Pages web app
- Create a page-focused web UI that consumes a web API
- Razor syntax
- See more

Data-driven web apps

Page-focused web UI with MVC

Develop web apps using the Model-View-Controller design pattern

- Overview
- Create your first ASP.NET Core MVC app
- Views
- Controllers
- Routing to controller actions

Create data-driven web apps in ASP.NET Core

- Data binding in ASP.NET Core Blazor
- SQL Server Express and Razor Pages
- Entity Framework Core with Razor Pages
- Entity Framework Core with ASP.NET Core MVC
- Get started with ASP.NET Core MVC (SQL Server Express and SQLite)

Real-time web apps with SignalR

Add real-time functionality to your web app, enable server-side code to push content instantly

- Overview
- Create your first SignalR app
- SignalR with Blazor WebAssembly
- SignalR with TypeScript
- Samples

Remote Procedure Call (RPC) apps - gRPC services

Develop contract-first, high-performance services with gRPC in ASP.NET Core

- Overview
- Create a gRPC client and server
- gRPC services concepts in C#
- Samples
- Compare gRPC services with HTTP APIs

Previous ASP.NET framework versions

Explore overviews, tutorials, fundamental concepts, architecture and API reference for previous ASP.NET...

- ASP.NET 4.x

ASP.NET Core video tutorials

- ASP.NET Core 101 video series
- Entity Framework Core 101 video series with .NET Core and ASP.NET Core
- Microservice architecture with ASP.NET Core
- Focus on Blazor video series
- .NET Channel

Concepts and features

API reference for ASP.NET Core

.NET API browser

Servers

Overview

Kestrel

IIS

HTTP.sys

Host and deploy

- Overview
- Deploy to Azure App Service
- DevOps for ASP.NET Core Developers
- Linux with Apache
- Linux with Nginx
- Kestrel
- IIS
- HTTP.sys
- Docker

Security and identity

- Overview
- Choose an identity solution
- Authentication
- Authorization
- Course: Secure an ASP.NET Core web app with the Identity framework
- Data protection
- Secrets management
- Enforce HTTPS
- Host Docker with HTTPS

Globalization and localization

- Overview
- Portable object localization
- Localization extensibility
- Troubleshoot

Test, debug and troubleshoot

- Razor Pages unit tests
- Remote debugging
- Snapshot debugging
- Integration tests
- Load and stress testing
- Troubleshoot and debug
- Logging
- Load test Azure web apps by using Azure DevOps

Azure and ASP.NET Core

- Deploy an ASP.NET Core web app
- ASP.NET Core and Docker
- Host a web application with Azure App Service
- App Service and Azure SQL Database
- Managed identity with ASP.NET Core and Azure SQL Database
- Web API with CORS in Azure App Service
- Capture Web Application Logs with App Service Diagnostics Logging

Performance

- Overview
- Memory and garbage collection
- Response caching
- Response compression
- Diagnostic tools
- Load and stress testing

Advanced features

- Model binding

Migration

- ASP.NET Core 5.0 to 6.0

- Model validation
- Write middleware
- Request and response operations
- URL rewriting

- ASP.NET Core 5.0 code samples to 6.0 minimal hosting model
- ASP.NET Core 3.1 to 5.0
- ASP.NET Core 3.0 to 3.1
- ASP.NET Core 2.2 to 3.0
- ASP.NET Core 2.1 to 2.2
- ASP.NET Core 2.0 to 2.1
- ASP.NET Core 1.x to 2.0
- ASP.NET to ASP.NET Core

Architecture

- Choose between traditional web apps and Single Page Apps (SPAs)
- Architectural principles
- Common web application architectures
- Common client-side web technologies
- Development process for Azure

Contribute to ASP.NET Core docs. Read our contributor guide.

Overview of ASP.NET Core

Article • 06/18/2024

By Daniel Roth, Rick Anderson, and Shaun Luttin

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps.

With ASP.NET Core, you can:

- Build web apps and services, Internet of Things (IoT) apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on .NET.

Why choose ASP.NET Core?

Millions of developers use or have used ASP.NET 4.x to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, including architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
  - Kestrel
  - IIS
  - HTTP.sys
  - Nginx
  - Docker

- Side-by-side versioning.
- Tooling that simplifies modern web development.

Build web APIs and web UI using ASP.NET Core MVC

ASP.NET Core MVC provides features to build web APIs and web apps:

- The Model-View-Controller (MVC) pattern helps make your web APIs and web apps testable.
- Razor Pages is a page-based programming model that makes building web UI easier and more productive.
- Razor markup provides a productive syntax for Razor Pages and MVC views.
- Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.
- Built-in support for multiple data formats and content negotiation lets your web APIs reach a broad range of clients, including browsers and mobile devices.
- Model binding automatically maps data from HTTP requests to action method parameters.
- Model validation automatically performs client-side and server-side validation.

Client-side development

ASP.NET Core includes Blazor for building richly interactive web UI, and also integrates with other popular frontend JavaScript frameworks like Angular, React, Vue, and Bootstrap. For more information, see ASP.NET Core Blazor and related topics under *Client-side development*.

ASP.NET Core target frameworks

ASP.NET Core 3.x or later can only target .NET.

There are several advantages to targeting .NET, and these advantages increase with each release. Some advantages of .NET over .NET Framework include:

- Cross-platform. Runs on Windows, macOS, and Linux.
- Improved performance
- Side-by-side versioning
- New APIs
- Open source

Recommended learning path

We recommend the following sequence of tutorials for an introduction to developing ASP.NET Core apps:

1. Follow a tutorial for the app type you want to develop or maintain.

Expand table

App type	Scenario	Tutorial
Web app	New server-side web UI development	Get started with Razor Pages
Web app	Maintaining an MVC app	Get started with MVC
Web app	Client-side web UI development	Get started with Blazor
Web API	RESTful HTTP services	Create a web API†
Remote Procedure Call app	Contract-first services using Protocol Buffers	Get started with a gRPC service
Real-time app	Bidirectional communication between servers and connected clients	Get started with SignalR

2. Follow a tutorial that shows how to do basic data access.

Expand table

Scenario	Tutorial
New development	Razor Pages with Entity Framework Core
Maintaining an MVC app	MVC with Entity Framework Core

3. Read an overview of ASP.NET Core fundamentals that apply to all app types.

4. Browse the table of contents for other topics of interest.

†There's also an interactive web API tutorial. No local installation of development tools is required. The code runs in an Azure Cloud Shell in your browser, and curl is used for testing.

Migrate from .NET Framework

For a reference guide to migrating ASP.NET 4.x apps to ASP.NET Core, see Update from ASP.NET to ASP.NET Core.

How to download a sample

Many of the articles and tutorials include links to sample code.

1. Download the ASP.NET repository zip file.
2. Unzip the `AspNetCore.Docs-main.zip` file.
3. To access an article's sample app in the unzipped repository, use the URL in the article's sample link to help you navigate to the sample's folder. Usually, an article's sample link appears at the top of the article with the link text *View or download sample code*.

Preprocessor directives in sample code

To demonstrate multiple scenarios, sample apps use the `#define` and `#if-#else/#elif-#endif` preprocessor directives to selectively compile and run different sections of sample code. For those samples that make use of this approach, set the `#define` directive at the top of the C# files to define the symbol associated with the scenario that you want to run. Some samples require defining the symbol at the top of multiple files in order to run a scenario.

For example, the following `#define` symbol list indicates that four scenarios are available (one scenario per symbol). The current sample configuration runs the `TemplateCode` scenario:

```
C#

#define TemplateCode // or LogFromMain or ExpandDefault or FilterInCode
```

To change the sample to run the `ExpandDefault` scenario, define the `ExpandDefault` symbol and leave the remaining symbols commented-out:

```
C#

#define ExpandDefault // TemplateCode or LogFromMain or FilterInCode
```

For more information on using C# preprocessor directives to selectively compile sections of code, see #define (C# Reference) and #if (C# Reference).

Breaking changes and security advisories

Breaking changes and security advisories are reported on the Announcements repo. Announcements can be limited to a specific version by selecting a **Label** filter.

Next steps

For more information, see the following resources:

- Get started with ASP.NET Core
- Publish an ASP.NET Core app to Azure with Visual Studio
- ASP.NET Core fundamentals
- The weekly ASP.NET community standup covers the team's progress and plans. It features new blogs and third-party software.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Choose between ASP.NET 4.x and ASP.NET Core

Article • 04/11/2023

ASP.NET Core is a redesign of ASP.NET 4.x. This article lists the differences between them.

ASP.NET Core

ASP.NET Core is an open-source, cross-platform framework for building modern, cloud-based web apps on Windows, macOS, or Linux.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.
- Open-source and community-focused.
- Integration of modern, client-side frameworks and development workflows.
- Support for hosting Remote Procedure Call (RPC) services using gRPC.
- A cloud-ready, environment-based configuration system.
- Built-in dependency injection.
- A lightweight, high-performance, and modular HTTP request pipeline.
- Ability to host on the following:
  - Kestrel
  - IIS
  - HTTP.sys
  - Nginx
  - Apache
  - Docker
- Side-by-side versioning.
- Tooling that simplifies modern web development.

ASP.NET 4.x

ASP.NET 4.x is a mature framework that provides the services needed to build enterprise-grade, server-based web apps on Windows.

Framework selection

The following table compares ASP.NET Core to ASP.NET 4.x.

Expand table

ASP.NET Core	ASP.NET 4.x
Build for Windows, macOS, or Linux	Build for Windows
Razor Pages is the recommended approach to create a Web UI as of ASP.NET Core 2.x. See also MVC, Web API, and SignalR.	Use Web Forms, SignalR, MVC, Web API, WebHooks, or Web Pages
Multiple versions per machine	One version per machine
Develop with Visual Studio, Visual Studio for Mac, or Visual Studio Code using C# or F#	Develop with Visual Studio using C#, VB, or F#
Higher performance than ASP.NET 4.x	Good performance
Use .NET Core runtime	Use .NET Framework runtime

See ASP.NET Core targeting .NET Framework for information on ASP.NET Core 2.x support on .NET Framework.

ASP.NET Core scenarios

- Websites
- APIs
- Real-time
- Deploy an ASP.NET Core app to Azure

ASP.NET 4.x scenarios

- Websites
- APIs
- Real-time
- Create an ASP.NET 4.x web app in Azure

Additional resources

- Introduction to ASP.NET
- Introduction to ASP.NET Core
- Deploy ASP.NET Core apps to Azure App Service

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

.NET vs. .NET Framework for server apps

Article • 11/22/2024

There are two supported .NET implementations for building server-side apps: .NET and .NET Framework. **The latest .NET version (currently .NET 9) is the preferred version of .NET to use for server development.** The reasons to continue using .NET Framework are specific and limited.

Expand table

Implementation	Included versions
.NET	.NET Core 1.0 - 3.1 .NET 5 and later versions
.NET Framework	.NET Framework 1.0 - 4.8

Choose .NET

.NET has the following advantages for server apps:

- **Works cross-platform.**

  .NET enables your web or service app to run on multiple platforms, for example, Windows, Linux, and macOS. You can also use any of these operating systems as your development workstation. Use the Visual Studio integrated development environment (IDE) on Windows, or use Visual Studio Code on macOS, Linux, or Windows. Visual Studio Code supports IntelliSense and debugging. Most third-party editors, such as Sublime, Emacs, and VI, work with .NET. These third-party editors get editor IntelliSense using Omnisharp. You can also skip the code editor and directly use the .NET CLI.

- **Lets you target microservices.**

  A microservices architecture allows a mix of technologies across a service boundary. This technology mix enables a gradual embrace of .NET for new microservices that work with other microservices or services. For example, you can mix microservices or services developed with .NET Framework, Java, Ruby, or other monolithic technologies.

  There are many infrastructure platforms available. Azure Service Fabric is designed for large and complex microservice systems. Azure App Service is a

good choice for stateless microservices. Microservices alternatives based on Docker fit any microservices approach, as explained in the next section (Supports Docker containers). All these platforms support .NET and make them ideal for hosting your microservices.

For more information about microservices architecture, see .NET Microservices: Architecture for containerized .NET apps.

- **Supports Docker containers.**

  Containers are commonly used with a microservices architecture. Containers can also be used to containerize web apps or services that follow any architectural pattern. While .NET Framework can be used on Windows containers, the modularity and lightweight nature of .NET make it a better choice for containers. When you're creating and deploying a container, the size of its image is much smaller with .NET than with .NET Framework. Because it's cross-platform, you can deploy server apps to Linux Docker containers.

  You can host Docker containers in your own Linux or Windows infrastructure or in a cloud service such as Azure Kubernetes Service. Azure Kubernetes Service can manage, orchestrate, and scale container-based applications in the cloud.

- **Is high-performance and scalable.**

  When your system needs the best possible performance and scalability, .NET and ASP.NET Core are your best options. The high-performance server runtime for Windows Server and Linux makes ASP.NET Core a top-performing web framework on TechEmpower benchmarks.

  Performance and scalability are especially relevant for microservices architectures, where hundreds of microservices might be running. With ASP.NET Core, systems run with a much lower number of servers or virtual machines (VMs), which saves costs on infrastructure and hosting.

- **Supports side-by-side .NET versions per application.**

  The .NET implementation supports side-by-side installations of different versions of the .NET runtime on the same machine. That capability allows multiple services on the same server, each on its own version of .NET. It also lowers risks and saves money in application upgrades and IT operations.

  Side-by-side installation isn't possible with .NET Framework. It's a Windows component, and only one version can exist on a machine at a time: each version of .NET Framework replaces the previous version. If you install a new app that targets

a later version of .NET Framework, you might break existing apps that run on the machine because the previous version was replaced.

- **Is more secure.**

When to choose .NET Framework

As previously mentioned, the *.NET* implementation offers significant benefits for new applications and application patterns. However, in some specific scenarios, you might need to use .NET Framework for your server apps, and .NET Framework will continue to be supported. Use .NET Framework for your server app when:

- Your app currently uses .NET Framework.

  In most cases, you don't need to migrate your existing applications to .NET. Instead, we recommend using .NET as you extend an existing app, such as writing a new web service in ASP.NET Core.

- Your app uses third-party libraries or NuGet packages that aren't available for .NET.

  .NET Standard enables sharing code across all .NET implementations, including .NET 6+. With .NET Standard 2.0, a compatibility mode allows .NET Standard and .NET projects to reference .NET Framework libraries. For more information, see Support for .NET Framework libraries.

  You should only use .NET Framework when the libraries or NuGet packages use technologies that aren't available in .NET Standard or .NET.

- Your app uses .NET Framework technologies that aren't available for .NET.

  Some .NET Framework technologies aren't available in .NET. The following list shows the most common technologies not found in .NET:
  - **ASP.NET Web Forms applications**: ASP.NET Web Forms are only available in .NET Framework. ASP.NET Core can't be used for ASP.NET Web Forms.
  - **ASP.NET Web Pages applications**: ASP.NET Web Pages aren't included in ASP.NET Core.
  - **Workflow-related services**: Windows Workflow Foundation (WF), Workflow Services (WCF + WF in a single service), and WCF Data Services (formerly known as "ADO.NET Data Services") are only available in .NET Framework.
  - **Language support**: Visual Basic and F# are supported in .NET but *not for all project types*. For a list of supported project templates, see Template options for dotnet new.

  For more information, see .NET Framework technologies unavailable in .NET.

- Your app uses a platform that doesn't support .NET.

  Some Microsoft or third-party platforms don't support .NET. Some Azure services provide an SDK not yet available for consumption on .NET. In such cases, you can use the equivalent REST API instead of the client SDK.

See also

- Choose between ASP.NET and ASP.NET Core
- ASP.NET Core targeting .NET Framework
- Target frameworks
- .NET introduction
- Porting from .NET Framework to .NET 5
- Introduction to .NET and Docker
- .NET implementations
- .NET Microservices. Architecture for Containerized .NET Applications

Tutorial: Get started with ASP.NET Core

Article • 09/18/2024

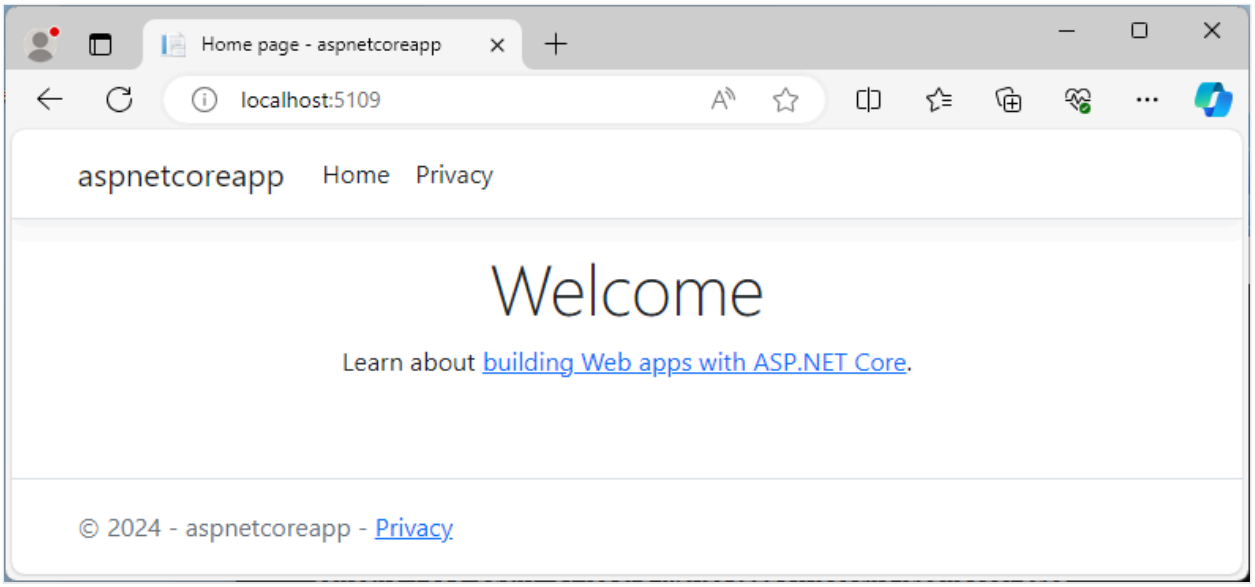
This tutorial shows how to create and run an ASP.NET Core web app using the .NET CLI.

For Blazor tutorials, see ASP.NET Core Blazor tutorials.

You'll learn how to:

- ✓ Create a web app project.
- ✓ Run the app.
- ✓ Edit a Razor page.

At the end, you'll have a working web app running on your local machine.



Prerequisites

.NET 8.0 SDK

Create a web app project

Open a command shell, and enter the following command:

.NET CLI

```
dotnet new webapp --output aspnetcoreapp --no-https
```

The preceding command creates a new web app project in a directory named `aspnetcoreapp`. The project doesn't use HTTPS.

Run the app

Run the following commands:

.NET CLI

```
cd aspnetcoreapp
dotnet run
```

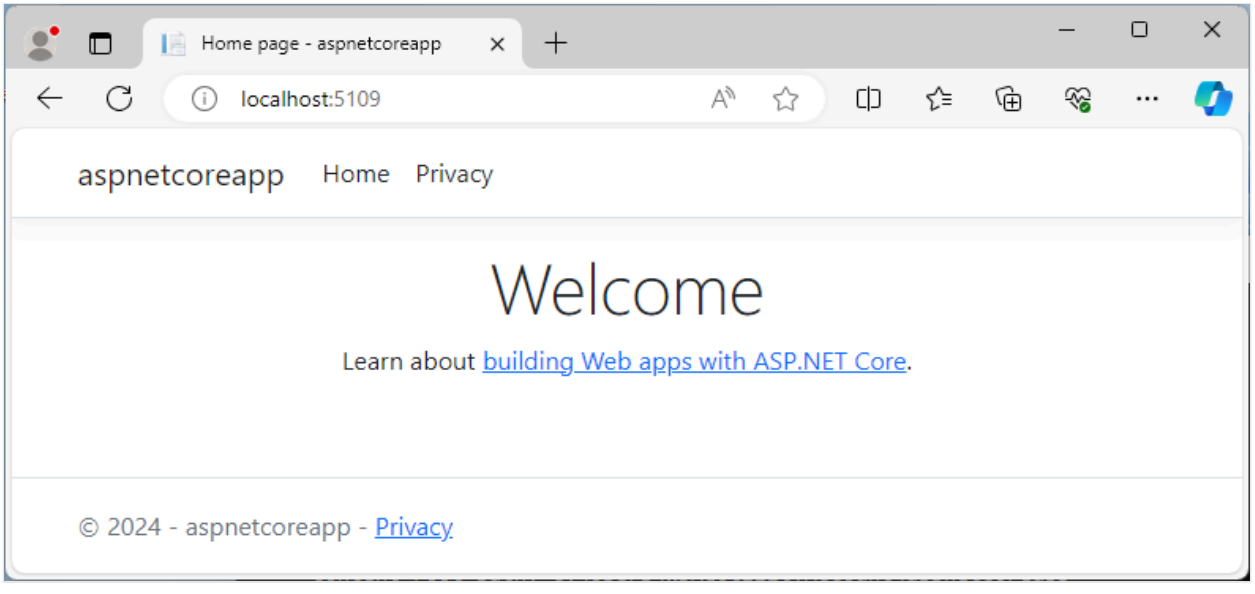
The `run` command produces output like the following example:

Output

```
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5109
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\aspnetcoreapp
```

Open a browser and go to the URL shown in the output. In this example, the URL is `http://localhost:5109`.

The browser shows the home page.



Edit a Razor page

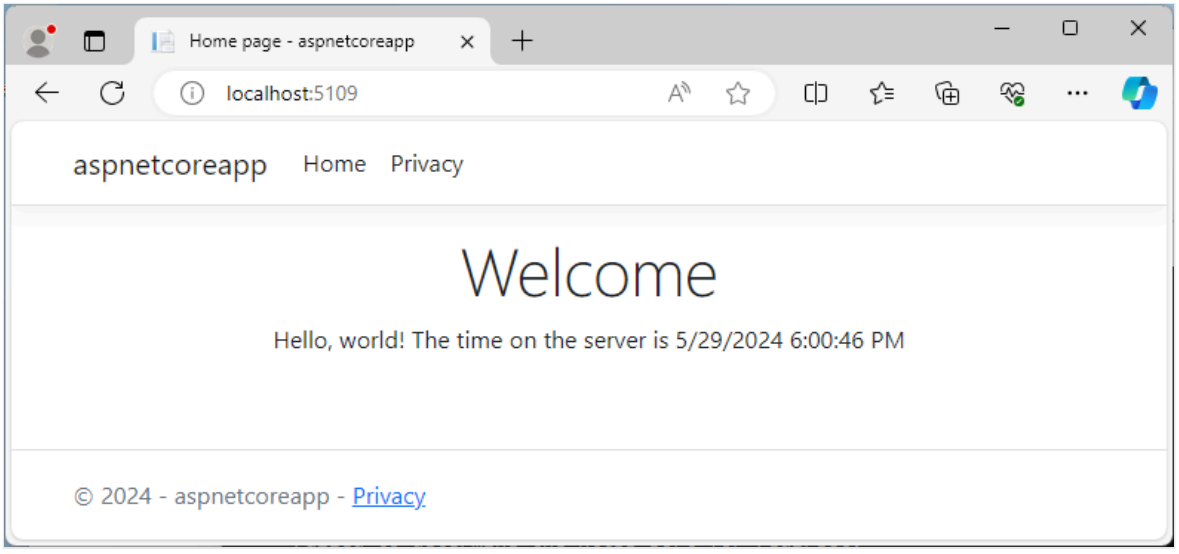
Change the home page:

- In the command shell, press Ctrl+C (Cmd+C in macOS) to exit the program.
- Open `Pages/Index.cshtml` in a text editor.
- Replace the line that begins with "Learn about" with the following highlighted markup and code:

CSHTML

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Hello, world! The time on the server is @DateTime.Now</p>
</div>
```

- Save your changes.
- In the command shell run the `dotnet run` command again.
- In the browser, refresh the page and verify the changes are displayed.



Next steps

In this tutorial, you learned how to:

- ✓ Create a web app project.
- ✓ Run the project.
- ✓ Make a change.

To learn more about ASP.NET Core, see the following:

Overview of ASP.NET Core

What's new in ASP.NET Core 9.0

Article • 11/19/2024

This article highlights the most significant changes in ASP.NET Core 9.0 with links to relevant documentation.

Static asset delivery optimization

MapStaticAssets routing endpoint conventions is a new feature that optimizes the delivery of static assets in ASP.NET Core apps.

For information on static asset delivery for Blazor apps, see ASP.NET Core Blazor static files.

Following production best practices for serving static assets requires a significant amount of work and technical expertise. Without optimizations like compression, caching, and fingerprints:

- The browser has to make additional requests on every page load.
- More bytes than necessary are transferred through the network.
- Sometimes stale versions of files are served to clients.

Creating performant web apps requires optimizing asset delivery to the browser. Possible optimizations include:

- Serve a given asset once until the file changes or the browser clears its cache. Set the ETag header.
- Prevent the browser from using old or stale assets after an app is updated. Set the Last-Modified header.
- Set up proper caching headers.
- Use caching middleware.
- Serve compressed versions of the assets when possible.
- Use a CDN to serve the assets closer to the user.
- Minimize the size of assets served to the browser. This optimization doesn't include minification.

MapStaticAssets is a new feature that optimizes the delivery of static assets in an app. It's designed to work with all UI frameworks, including Blazor, Razor Pages, and MVC. It's typically a drop-in replacement for UseStaticFiles:

```
diff
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

+app.MapStaticAssets();
-app.UseStaticFiles();
app.MapRazorPages();

app.Run();
```

`MapStaticAssets` operates by combining build and publish-time processes to collect information about all the static resources in an app. This information is then utilized by the runtime library to efficiently serve these files to the browser.

`MapStaticAssets` can replace UseStaticFiles in most situations, however, it's optimized for serving the assets that the app has knowledge of at build and publish time. If the app serves assets from other locations, such as disk or embedded resources, `UseStaticFiles` should be used.

`MapStaticAssets` provides the following benefits not found with `UseStaticFiles`:

- Build time compression for all the assets in the app:
  - `gzip` during development and `gzip + brotli` during publish.
  - All assets are compressed with the goal of reducing the size of the assets to the minimum.
- Content based `ETags`: The `Etags` for each resource are the Base64 encoded string of the SHA-256 hash of the content. This ensures that the browser only redownloads a file if its contents have changed.

The following table shows the original and compressed sizes of the CSS and JS files in the default Razor Pages template:

Expand table

File	Original	Compressed	% Reduction
bootstrap.min.css	163	17.5	89.26%
jquery.js	89.6	28	68.75%
bootstrap.min.js	78.5	20	74.52%
Total	331.1	65.5	80.20%

The following table shows the original and compressed sizes using the Fluent UI Blazor components library:

Expand table

File	Original	Compressed	% Reduction
fluent.js	384	73	80.99%
fluent.css	94	11	88.30%
Total	478	84	82.43%

For a total of 478 KB uncompressed to 84 KB compressed.

The following table shows the original and compressed sizes using the MudBlazor Blazor components library:

Expand table

File	Original	Compressed	Reduction
MudBlazor.min.css	541	37.5	93.07%
MudBlazor.min.js	47.4	9.2	80.59%
Total	588.4	46.7	92.07%

Optimization happens automatically when using `MapStaticAssets`. When a library is added or updated, for example with new JavaScript or CSS, the assets are optimized as part of the build. Optimization is especially beneficial to mobile environments that can have a lower bandwidth or an unreliable connections.

For more information on the new file delivery features, see the following resources:

- Static files in ASP.NET Core

- ASP.NET Core Blazor static files

Enabling dynamic compression on the server vs using `MapStaticAssets`

`MapStaticAssets` has the following advantages over dynamic compression on the server:

- Is simpler because there is no server specific configuration.
- Is more performant because the assets are compressed at build time.
- Allows the developer to spend extra time during the build process to ensure that the assets are the minimum size.

Consider the following table comparing MudBlazor compression with IIS dynamic compression and `MapStaticAssets`:

Expand table

IIS gzip	`MapStaticAssets`	`MapStaticAssets` reduction
≅ 90	37.5	59%

Blazor

This section describes new features for Blazor.

.NET MAUI Blazor Hybrid and Web App solution template

A new solution template makes it easier to create .NET MAUI native and Blazor web client apps that share the same UI. This template shows how to create client apps that maximize code reuse and target Android, iOS, Mac, Windows, and Web.

Key features of this template include:

- The ability to choose a Blazor interactive render mode for the web app.
- Automatic creation of the appropriate projects, including a Blazor Web App (global Interactive Auto rendering) and a .NET MAUI Blazor Hybrid app.
- The created projects use a shared Razor class library (RCL) to maintain the UI's Razor components.
- Sample code is included that demonstrates how to use dependency injection to provide different interface implementations for the Blazor Hybrid app and the Blazor Web App.

To get started, install the .NET 9 SDK and install the .NET MAUI workload, which contains the template:

.NET CLI

```
dotnet workload install maui
```

Create a solution from the project template in a command shell using the following command:

.NET CLI

```
dotnet new maui-blazor-web
```

The template is also available in Visual Studio.

> **Note**
>
> Currently, an exception occurs if Blazor rendering modes are defined at the per-page/component level. For more information, see BlazorWebView needs a way to enable overriding ResolveComponentForRenderMode (dotnet/aspnetcore #51235).

For more information, see Build a .NET MAUI Blazor Hybrid app with a Blazor Web App.

Detect rendering location, interactivity, and assigned render mode at runtime

We've introduced a new API designed to simplify the process of querying component states at runtime. This API provides the following capabilities:

- **Determine the current execution location of the component**: This can be useful for debugging and optimizing component performance.
- **Check if the component is running in an interactive environment**: This can be helpful for components that have different behaviors based on the interactivity of their environment.
- **Retrieve the assigned render mode for the component**: Understanding the render mode can help in optimizing the rendering process and improving the overall performance of a component.

For more information, see ASP.NET Core Blazor render modes.

Improved server-side reconnection experience:

The following enhancements have been made to the default server-side reconnection experience:

- When the user navigates back to an app with a disconnected circuit, reconnection is attempted immediately rather than waiting for the duration of the next reconnect interval. This improves the user experience when navigating to an app in a browser tab that has gone to sleep.
- When a reconnection attempt reaches the server but the server has already released the circuit, a page refresh occurs automatically. This prevents the user from having to manually refresh the page if it's likely going to result in a successful reconnection.
- Reconnect timing uses a computed backoff strategy. By default, the first several reconnection attempts occur in rapid succession without a retry interval before computed delays are introduced between attempts. You can customize the retry interval behavior by specifying a function to compute the retry interval, as the following exponential backoff example demonstrates:

  JavaScript

  ```
  Blazor.start({
    circuit: {
      reconnectionOptions: {
        retryIntervalMilliseconds: (previousAttempts, maxRetries) =>
          previousAttempts >= maxRetries ? null : previousAttempts * 1000
      },
    },
  });
  ```

- The styling of the default reconnect UI has been modernized.

For more information, see ASP.NET Core Blazor SignalR guidance.

Simplified authentication state serialization for Blazor Web Apps

New APIs make it easier to add authentication to an existing Blazor Web App. When you create a new Blazor Web App with authentication using **Individual Accounts** and you enable WebAssembly-based interactivity, the project includes a custom AuthenticationStateProvider in both the server and client projects.

These providers flow the user's authentication state to the browser. Authenticating on the server rather than the client allows the app to access authentication state during prerendering and before the .NET WebAssembly runtime is initialized.

The custom AuthenticationStateProvider implementations use the Persistent Component State service (PersistentComponentState) to serialize the authentication state into HTML comments and read it back from WebAssembly to create a new AuthenticationState instance.

This works well if you've started from the Blazor Web App project template and selected the **Individual Accounts** option, but it's a lot of code to implement yourself or copy if you're trying to add authentication to an existing project. There are now APIs, which are now part of the Blazor Web App project template, that can be called in the server and client projects to add this functionality:

- AddAuthenticationStateSerialization: Adds the necessary services to serialize the authentication state on the server.
- AddAuthenticationStateDeserialization: Adds the necessary services to deserialize the authentication state in the browser.

By default, the API only serializes the server-side name and role claims for access in the browser. An option can be passed to AddAuthenticationStateSerialization to include all claims.

For more information, see the following sections of ASP.NET Core Blazor authentication and authorization:

- Blazor Identity UI (Individual Accounts)
- Manage authentication state in Blazor Web Apps

Add static server-side rendering (SSR) pages to a globally-interactive Blazor Web App

With the release of .NET 9, it's now simpler to add static SSR pages to apps that adopt global interactivity.

This approach is only useful when the app has specific pages that can't work with interactive Server or WebAssembly rendering. For example, adopt this approach for pages that depend on reading/writing HTTP cookies and can only work in a request/response cycle instead of interactive rendering. For pages that work with interactive rendering, you shouldn't force them to use static SSR rendering, as it's less efficient and less responsive for the end user.

Mark any Razor component page with the new [ExcludeFromInteractiveRouting] attribute assigned with the `@attribute` Razor directive:

razor

```
@attribute [ExcludeFromInteractiveRouting]
```

Applying the attribute causes navigation to the page to exit from interactive routing. Inbound navigation is forced to perform a full-page reload instead resolving the page via interactive routing. The full-page reload forces the top-level root component, typically the `App` component (`App.razor`), to rerender from the server, allowing the app to switch to a different top-level render mode.

The RazorComponentsEndpointHttpContextExtensions.AcceptsInteractiveRouting extension method allows the component to detect whether the [ExcludeFromInteractiveRouting] attribute is applied to the current page.

In the `App` component, use the pattern in the following example:

- Pages that aren't annotated with the [ExcludeFromInteractiveRouting] attribute default to the `InteractiveServer` render mode with global interactivity. You can replace `InteractiveServer` with `InteractiveWebAssembly` or `InteractiveAuto` to specify a different default global render mode.
- Pages annotated with the [ExcludeFromInteractiveRouting] attribute adopt static SSR (`PageRenderMode` is assigned `null`).

razor

```
<!DOCTYPE html>
<html>
<head>
    ...
    <HeadOutlet @rendermode="@PageRenderMode" />
</head>
<body>
    <Routes @rendermode="@PageRenderMode" />
    ...
</body>
</html>

@code {
    [CascadingParameter]
    private HttpContext HttpContext { get; set; } = default!;

    private IComponentRenderMode? PageRenderMode
        => HttpContext.AcceptsInteractiveRouting() ? InteractiveServer :
```

```
null;
}
```

An alternative to using the RazorComponentsEndpointHttpContextExtensions.AcceptsInteractiveRouting extension method is to read endpoint metadata manually using `HttpContext.GetEndpoint()?.Metadata`.

This feature is covered by the reference documentation in ASP.NET Core Blazor render modes.

Constructor injection

Razor components support constructor injection.

In the following example, the partial (code-behind) class injects the `NavigationManager` service using a primary constructor:

C#

```
public partial class ConstructorInjection(NavigationManager navigation)
{
    private void HandleClick()
    {
        navigation.NavigateTo("/counter");
    }
}
```

For more information, see ASP.NET Core Blazor dependency injection.

Websocket compression for Interactive Server components

By default, Interactive Server components enable compression for WebSocket connections and set a `frame-ancestors` Content Security Policy (CSP) directive set to `'self'`, which only permits embedding the app in an `<iframe>` of the origin from which the app is served when compression is enabled or when a configuration for the WebSocket context is provided.

Compression can be disabled by setting `ConfigureWebSocketOptions` to `null`, which reduces the vulnerability of the app to attack but may result in reduced performance:

C#

```
.AddInteractiveServerRenderMode(o => o.ConfigureWebSocketOptions = null)
```

Configure a stricter `frame-ancestors` CSP with a value of `'none'` (single quotes required), which allows WebSocket compression but prevents browsers from embedding the app into any `<iframe>`:

C#

```
.AddInteractiveServerRenderMode(o => o.ContentSecurityFrameAncestorsPolicy =
"'none'")
```

For more information, see the following resources:

- ASP.NET Core Blazor SignalR guidance
- Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering

Handle keyboard composition events in Blazor

The new `KeyboardEventArgs.IsComposing` property indicates if the keyboard event is part of a composition session. Tracking the composition state of keyboard events is crucial for handling international character input methods.

Added `OverscanCount` parameter to `QuickGrid`

The QuickGrid component now exposes an `OverscanCount` property that specifies how many additional rows are rendered before and after the visible region when virtualization is enabled.

The default `OverscanCount` is 3. The following example increases the `OverscanCount` to 4:

razor

```
<QuickGrid ItemsProvider="itemsProvider" Virtualize="true"
OverscanCount="4">
    ...
</QuickGrid>
```

`InputNumber` component supports the `type="range"` attribute

The InputNumber<TValue> component now supports the type="range" attribute, which creates a range input that supports model binding and form validation, typically rendered as a slider or dial control rather than a text box:

razor

```razor
<EditForm Model="Model" OnSubmit="Submit" FormName="EngineForm">
    <div>
        <label>
            Nacelle Count (2-6):
            <InputNumber @bind-Value="Model!.NacelleCount" max="6" min="2"
                step="1" type="range" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private EngineSpecifications? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() {}

    public class EngineSpecifications
    {
        [Required, Range(minimum: 2, maximum: 6)]
        public int NacelleCount { get; set; }
    }
}
```

SignalR

This section describes new features for SignalR.

Polymorphic type support in SignalR Hubs

Hub methods can now accept a base class instead of the derived class to enable polymorphic scenarios. The base type needs to be annotated to allow polymorphism.

C#

```csharp
public class MyHub : Hub
{
    public void Method(JsonPerson person)
```

```
    {
        if (person is JsonPersonExtended)
        {
        }
        else if (person is JsonPersonExtended2)
        {
        }
        else
        {
        }
    }
}

[JsonPolymorphic]
[JsonDerivedType(typeof(JsonPersonExtended), nameof(JsonPersonExtended))]
[JsonDerivedType(typeof(JsonPersonExtended2), nameof(JsonPersonExtended2))]
private class JsonPerson
{
    public string Name { get; set; }
    public Person Child { get; set; }
    public Person Parent { get; set; }
}

private class JsonPersonExtended : JsonPerson
{
    public int Age { get; set; }
}

private class JsonPersonExtended2 : JsonPerson
{
    public string Location { get; set; }
}
```

Improved Activities for SignalR

SignalR now has an ActivitySource for both the hub server and client.

.NET SignalR server ActivitySource

The SignalR ActivitySource named `Microsoft.AspNetCore.SignalR.Server` emits events for hub method calls:

- Every method is its own activity, so anything that emits an activity during the hub method call is under the hub method activity.
- Hub method activities don't have a parent. This means they are not bundled under the long-running SignalR connection.

The following example uses the .NET Aspire dashboard and the OpenTelemetry packages:

XML

```xml
<PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
Version="1.9.0" />
<PackageReference Include="OpenTelemetry.Extensions.Hosting" Version="1.9.0"
/>
<PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
Version="1.9.0" />
```

Add the following startup code to the `Program.cs` file:

C#

```csharp
using OpenTelemetry.Trace;
using SignalRChat.Hubs;

// Set OTEL_EXPORTER_OTLP_ENDPOINT environment variable depending on where
your OTEL endpoint is.
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddSignalR();

builder.Services.AddOpenTelemetry()
    .WithTracing(tracing =>
    {
        if (builder.Environment.IsDevelopment())
        {
            // View all traces only in development environment.
            tracing.SetSampler(new AlwaysOnSampler());
        }

        tracing.AddAspNetCoreInstrumentation();
        tracing.AddSource("Microsoft.AspNetCore.SignalR.Server");
    });

builder.Services.ConfigureOpenTelemetryTracerProvider(tracing =>
tracing.AddOtlpExporter());

var app = builder.Build();
```

The following is example output from the Aspire Dashboard:

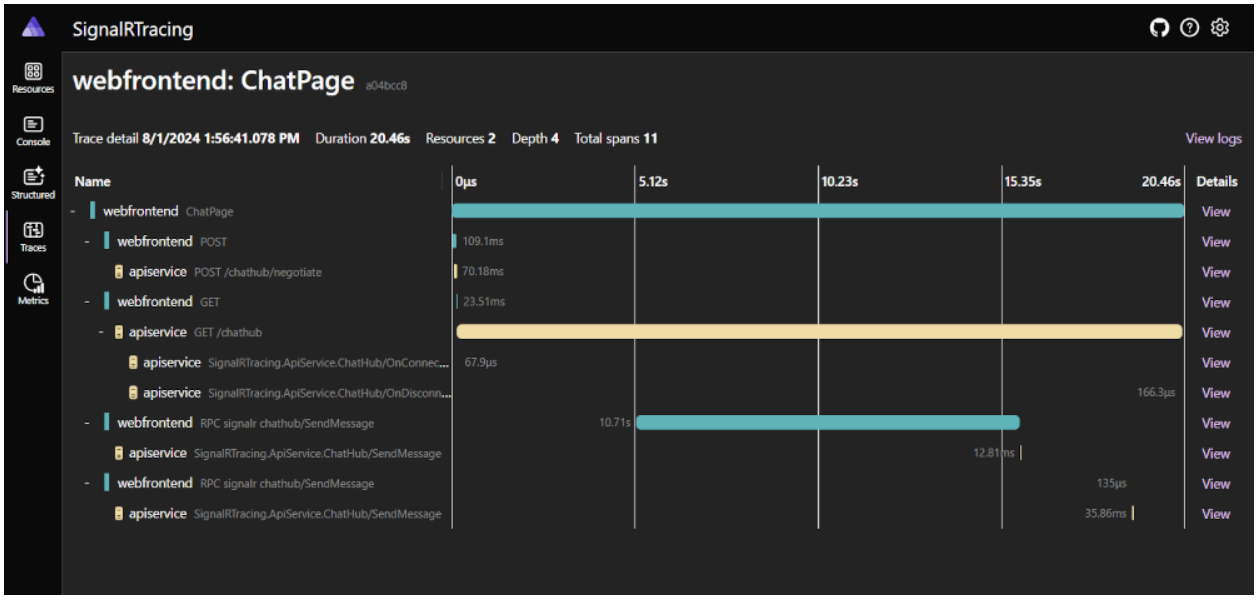
Timestamp	Name	Spans	Duration
7/5/2024 8:47:18.28...	unknown_service:WebApplication27: GET d3b7fd9	unknown_service:WebApplication27 (1)	80.64ms
7/5/2024 8:47:18.35...	unknown_service:WebApplication27: POST /default/negotiate 6d3ef74	unknown_service:WebApplication27 (1)	10.57ms
7/5/2024 8:47:18.39...	unknown_service:WebApplication27: GET /default 31dd5ac	unknown_service:WebApplication27 (1)	22.18s
7/5/2024 8:47:18.41...	unknown_service:WebApplication27: MyHub/OnConnectedAsync 5ec2ebf	unknown_service:WebApplication27 (1)	25.5µs
7/5/2024 8:47:22.47...	unknown_service:WebApplication27: MyHub/SendMessage e63a6ed	unknown_service:WebApplication27 (1)	7.83ms
7/5/2024 8:47:22.96...	unknown_service:WebApplication27: MyHub/SendMessage c80a8e6	unknown_service:WebApplication27 (1)	155.7µs
7/5/2024 8:47:23.13...	unknown_service:WebApplication27: MyHub/SendMessage 2ebf38b	unknown_service:WebApplication27 (1)	107.1µs
7/5/2024 8:47:40.55...	unknown_service:WebApplication27: GET 0b106a2	unknown_service:WebApplication27 (1)	4.82ms
7/5/2024 8:47:40.56...	unknown_service:WebApplication27: MyHub/OnDisconnectedAsync 7fa3a43	unknown_service:WebApplication27 (1)	16.5µs

.NET SignalR client ActivitySource

The SignalR ActivitySource named `Microsoft.AspNetCore.SignalR.Client` emits events for a SignalR client:

- The .NET SignalR client has an `ActivitySource` named `Microsoft.AspNetCore.SignalR.Client`. Hub invocations now create a client span. Note that other SignalR clients, such as the JavaScript client, don't support tracing. This feature will be added to more clients in future releases.
- Hub invocations on the client and server support context propagation. Propagating the trace context enables true distributed tracing. It's now possible to see invocations flow from the client to the server and back.

Here's how these new activities look in the .NET Aspire dashboard:



SignalR supports trimming and Native AOT

Continuing the Native AOT journey started in .NET 8, we have enabled trimming and native ahead-of-time (AOT) compilation support for both SignalR client and server scenarios. You can now take advantage of the performance benefits of using Native AOT in applications that use SignalR for real-time web communications.

Getting started

Install the latest .NET 9 SDK.

Create a solution from the `webapiaot` template in a command shell using the following command:

.NET CLI

```
dotnet new webapiaot -o SignalRChatAOTExample
```

Replace the contents of the `Program.cs` file with the following SignalR code:

C#

```
using Microsoft.AspNetCore.SignalR;
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.AddSignalR();
builder.Services.Configure<JsonHubProtocolOptions>(o =>
{
    o.PayloadSerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapHub<ChatHub>("/chatHub");
app.MapGet("/", () => Results.Content("""
<!DOCTYPE html>
<html>
<head>
    <title>SignalR Chat</title>
</head>
<body>
    <input id="userInput" placeholder="Enter your name" />
    <input id="messageInput" placeholder="Type a message" />
    <button onclick="sendMessage()">Send</button>
    <ul id="messages"></ul>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-
signalr/8.0.7/signalr.min.js"></script>
```

```
    <script>
        const connection = new signalR.HubConnectionBuilder()
            .withUrl("/chatHub")
            .build();

        connection.on("ReceiveMessage", (user, message) => {
            const li = document.createElement("li");
            li.textContent = `${user}: ${message}`;
            document.getElementById("messages").appendChild(li);
        });

        async function sendMessage() {
            const user = document.getElementById("userInput").value;
            const message = document.getElementById("messageInput").value;
            await connection.invoke("SendMessage", user, message);
        }

        connection.start().catch(err => console.error(err));
    </script>
</body>
</html>
""", "text/html"));

app.Run();

[JsonSerializable(typeof(string))]
internal partial class AppJsonSerializerContext : JsonSerializerContext { }

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

The preceding example produces a native Windows executable of 10 MB and a Linux executable of 10.9 MB.

Limitations

- Only the JSON protocol is currently supported:
  - As shown in the preceding code, apps that use JSON serialization and Native AOT must use the `System.Text.Json` Source Generator.
  - This follows the same approach as minimal APIs.
- On the SignalR server, Hub method parameters of type `IAsyncEnumerable<T>` and `ChannelReader<T>` where `T` is a ValueType (`struct`) aren't supported. Using these types results in a runtime exception at startup in development and in the published app. For more information, see SignalR: Using IAsyncEnumerable<T>

and ChannelReader<T> with ValueTypes in native AOT (dotnet/aspnetcore #56179).

- Strongly typed hubs aren't supported with Native AOT (`PublishAot`). Using strongly typed hubs with Native AOT will result in warnings during build and publish, and a runtime exception. Using strongly typed hubs with trimming (`PublishedTrimmed`) is supported.
- Only `Task`, `Task<T>`, `ValueTask`, or `ValueTask<T>` are supported for async return types.

Minimal APIs

This section describes new features for minimal APIs.

Added `InternalServerError` and `InternalServerError<TValue>` to `TypedResults`

The TypedResults class is a helpful vehicle for returning strongly-typed HTTP status code-based responses from a minimal API. `TypedResults` now includes factory methods and types for returning "500 Internal Server Error" responses from endpoints. Here's an example that returns a 500 response:

C#

```
var app = WebApplication.Create();

app.MapGet("/", () => TypedResults.InternalServerError("Something went
wrong!"));

app.Run();
```

Call `ProducesProblem` and `ProducesValidationProblem` on route groups

The `ProducesProblem` and `ProducesValidationProblem` extension methods have been updated to support their use on route groups. These methods indicate that all endpoints in a route group can return `ProblemDetails` or `ValidationProblemDetails` responses for the purposes of OpenAPI metadata.

C#

```
var app = WebApplication.Create();

var todos = app.MapGroup("/todos")
    .ProducesProblem();

todos.MapGet("/", () => new Todo(1, "Create sample app", false));
todos.MapPost("/", (Todo todo) => Results.Ok(todo));

app.Run();

record Todo(int Id, string Title, boolean IsCompleted);
```

`Problem` and `ValidationProblem` result types support construction with `IEnumerable<KeyValuePair<string, object?>>` values

Prior to .NET 9, constructing Problem and ValidationProblem result types in minimal APIs required that the `errors` and `extensions` properties be initialized with an implementation of `IDictionary<string, object?>`. In this release, these construction APIs support overloads that consume `IEnumerable<KeyValuePair<string, object?>>`.

C#

```
var app = WebApplication.Create();

app.MapGet("/", () =>
{
    var extensions = new List<KeyValuePair<string, object?>> { new("test",
"value") };
    return TypedResults.Problem("This is an error with extensions",
                                                    extensions:
extensions);
});
```

Thanks to GitHub user joegoldman2 for this contribution!

OpenAPI

This section describes new features for OpenAPI

Built-in support for OpenAPI document generation

The OpenAPI specification is a standard for describing HTTP APIs. The standard allows developers to define the shape of APIs that can be plugged into client generators, server generators, testing tools, documentation, and more. In .NET 9, ASP.NET Core provides built-in support for generating OpenAPI documents representing controller-based or minimal APIs via the Microsoft.AspNetCore.OpenApi package.

The following highlighted code calls:

- `AddOpenApi` to register the required dependencies into the app's DI container.
- `MapOpenApi` to register the required OpenAPI endpoints in the app's routes.

C#

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddOpenApi();

var app = builder.Build();

app.MapOpenApi();

app.MapGet("/hello/{name}", (string name) => $"Hello {name}"!);

app.Run();
```

Install the Microsoft.AspNetCore.OpenApi package in the project using the following command:

.NET CLI

```
dotnet add package Microsoft.AspNetCore.OpenApi --prerelease
```

Run the app and navigate to `openapi/v1.json` to view the generated OpenAPI document:




```
{
  "openapi": "3.0.1",
  "info": {
    "title": "OpenApiExample | v1",
    "version": "1.0.0"
  },
  "paths": {
    "/hello/{name}": {
      "get": {
        "tags": [
          "OpenApiExample"
        ],
        "parameters": [
          {
            "name": "name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "text/plain": {
                "schema": {
                  "type": "string"
                }
              }
            }
          }
        },
        "x-aspnetcore-id": "9ae10d8b-2d81-4b2d-b9f6-2a8a614b05de"
      }
    }
  },
  "tags": [
    {
      "name": "OpenApiExample"
    }
  ]
}
```


OpenAPI documents can also be generated at build-time by adding the Microsoft.Extensions.ApiDescription.Server package:

.NET CLI

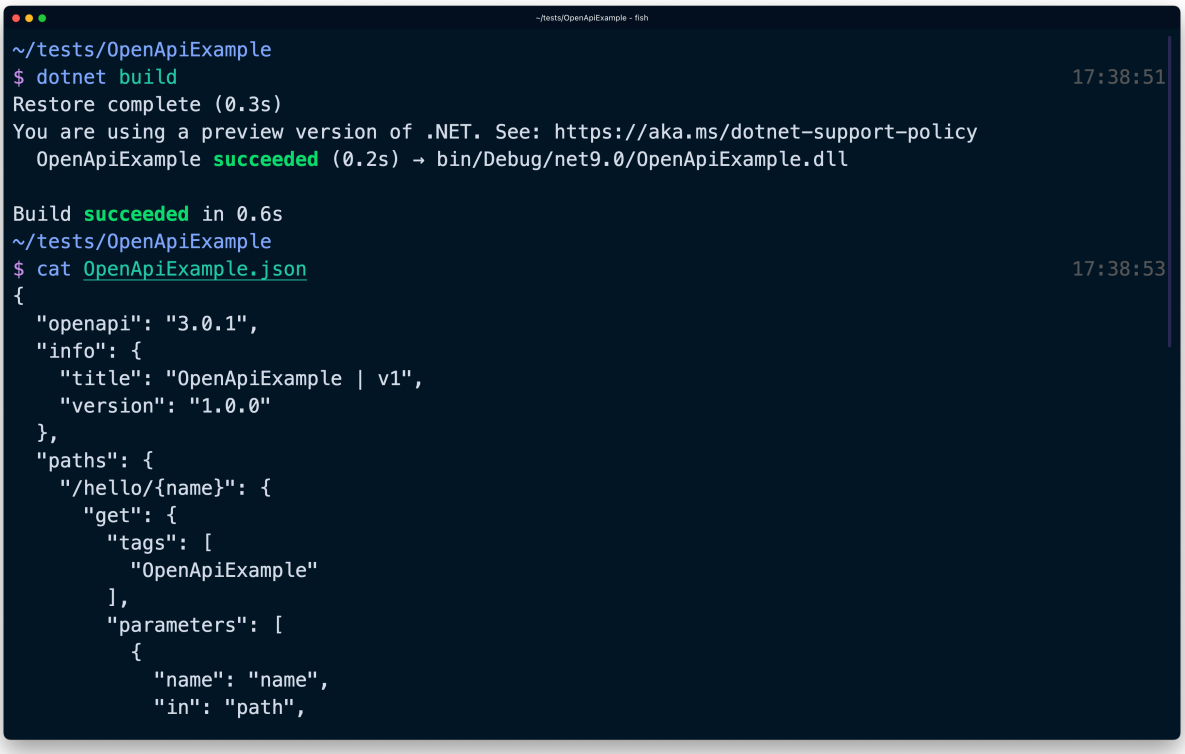
```
dotnet add package Microsoft.Extensions.ApiDescription.Server --prerelease
```

To modify the location of the emitted OpenAPI documents, set the target path in the OpenApiDocumentsDirectory property in the app's project file:

XML

```
<PropertyGroup>
  <OpenApiDocumentsDirectory>$(MSBuildProjectDirectory)
</OpenApiDocumentsDirectory>
</PropertyGroup>
```

Run `dotnet build` and inspect the generated JSON file in the project directory.



ASP.NET Core's built-in OpenAPI document generation provides support for various customizations and options. It provides document, operation, and schema transformers and has the ability to manage multiple OpenAPI documents for the same application.

To learn more about ASP.NET Core's new OpenAPI document capabilities, see the new Microsoft.AspNetCore.OpenApi docs.

Microsoft.AspNetCore.OpenApi supports trimming and Native AOT

The new built-in OpenAPI support in ASP.NET Core now also supports trimming and Native AOT.

Get started

Create a new ASP.NET Core Web API (Native AOT) project.

```
Console

dotnet new webapiaot
```

Add the Microsoft.AspNetCore.OpenAPI package.

```
Console

dotnet add package Microsoft.AspNetCore.OpenApi --prerelease
```

For this preview, you also need to add the latest Microsoft.OpenAPI package to avoid trimming warnings.

```
Console

dotnet add package Microsoft.OpenApi
```

Update *Program.cs* to enable generating OpenAPI documents.

```
diff

+ builder.Services.AddOpenApi();

var app = builder.Build();

+ app.MapOpenApi();
```

Publish the app.

```
Console

dotnet publish
```

The app publishes using Native AOT without warnings.

Authentication and authorization

This section describes new features for authentication and authorization.

OpenIdConnectHandler adds support for Pushed Authorization Requests (PAR)

We'd like to thank Joe DeCock from Duende Software for adding Pushed Authorization Requests (PAR) to ASP.NET Core's OpenIdConnectHandler. Joe described the background and motivation for enabling PAR in his API proposal as follows:

> Pushed Authorization Requests (PAR) is a relatively new OAuth standard that improves the security of OAuth and OIDC flows by moving authorization parameters from the front channel to the back channel. Thats is, moving authorization parameters from redirect URLs in the browser to direct machine to machine http calls on the back end.
>
> This prevents a cyberattacker in the browser from:
>
> - Seeing authorization parameters, which could leak PII.
> - Tampering with those parameters. For example, the cyberattacker could change the scope of access being requested.
>
> Pushing the authorization parameters also keeps request URLs short. Authorize parameters can get very long when using more complex OAuth and OIDC features such as Rich Authorization Requests. URLs that are long cause issues in many browsers and networking infrastructures.
>
> The use of PAR is encouraged by the FAPI working group within the OpenID Foundation. For example, the FAPI2.0 Security Profile requires the use of PAR. This security profile is used by many of the groups working on open banking (primarily in Europe), in health care, and in other industries with high security requirements.
>
> PAR is supported by a number of identity providers, including
>
> - Duende IdentityServer
> - Curity
> - Keycloak
> - Authlete

For .NET 9, we have decided to enable PAR by default if the identity provider's discovery document advertises support for PAR, since it should provide enhanced security for providers that support it. The identity provider's discovery document is usually found at `.well-known/openid-configuration`. If this causes problems, you can disable PAR via OpenIdConnectOptions.PushedAuthorizationBehavior as follows:

```
C#
```

```
builder.Services
    .AddAuthentication(options =>
    {
        options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddOpenIdConnect("oidc", oidcOptions =>
    {
        // Other provider-specific configuration goes here.

        // The default value is PushedAuthorizationBehavior.UseIfAvailable.

        // 'OpenIdConnectOptions' does not contain a definition for
'PushedAuthorizationBehavior'
        // and no accessible extension method 'PushedAuthorizationBehavior'
accepting a first argument
        // of type 'OpenIdConnectOptions' could be found
        oidcOptions.PushedAuthorizationBehavior =
PushedAuthorizationBehavior.Disable;
    });
```

To ensure that authentication only succeeds if PAR is used, use PushedAuthorizationBehavior.Require instead. This change also introduces a new OnPushAuthorization event to OpenIdConnectEvents which can be used customize the pushed authorization request or handle it manually. See the API proposal for more details.

OIDC and OAuth Parameter Customization

The OAuth and OIDC authentication handlers now have an `AdditionalAuthorizationParameters` option to make it easier to customize authorization message parameters that are usually included as part of the redirect query string. In .NET 8 and earlier, this requires a custom OnRedirectToIdentityProvider callback or overridden BuildChallengeUrl method in a custom handler. Here's an example of .NET 8 code:

C#

```
builder.Services.AddAuthentication().AddOpenIdConnect(options =>
{
    options.Events.OnRedirectToIdentityProvider = context =>
    {
        context.ProtocolMessage.SetParameter("prompt", "login");
        context.ProtocolMessage.SetParameter("audience",
"https://api.example.com");
```

```
        return Task.CompletedTask;
    };
});
```

The preceding example can now be simplified to the following code:

C#

```
builder.Services.AddAuthentication().AddOpenIdConnect(options =>
{
    options.AdditionalAuthorizationParameters.Add("prompt", "login");
    options.AdditionalAuthorizationParameters.Add("audience",
"https://api.example.com");
});
```

Configure HTTP.sys extended authentication flags

You can now configure the HTTP_AUTH_EX_FLAG_ENABLE_KERBEROS_CREDENTIAL_CACHING and HTTP_AUTH_EX_FLAG_CAPTURE_CREDENTIAL HTTP.sys flags by using the new `EnableKerberosCredentialCaching` and `CaptureCredentials` properties on the HTTP.sys AuthenticationManager to optimize how Windows authentication is handled. For example:

C#

```
webBuilder.UseHttpSys(options =>
{
    options.Authentication.Schemes = AuthenticationSchemes.Negotiate;
    options.Authentication.EnableKerberosCredentialCaching = true;
    options.Authentication.CaptureCredentials = true;
});
```

Miscellaneous

The following sections describe miscellaneous new features.

New `HybridCache` library

ⓘ Important

`HybridCache` is currently still in preview but will be fully released *after* .NET 9.0 in a future minor release of .NET Extensions.

The HybridCache API bridges some gaps in the existing IDistributedCache and IMemoryCache APIs. It also adds new capabilities, such as:

- **"Stampede" protection** to prevent parallel fetches of the same work.
- Configurable serialization.

`HybridCache` is designed to be a drop-in replacement for existing `IDistributedCache` and `IMemoryCache` usage, and it provides a simple API for adding new caching code. It provides a unified API for both in-process and out-of-process caching.

To see how the `HybridCache` API is simplified, compare it to code that uses `IDistributedCache`. Here's an example of what using `IDistributedCache` looks like:

C#

```
public class SomeService(IDistributedCache cache)
{
    public async Task<SomeInformation> GetSomeInformationAsync
        (string name, int id, CancellationToken token = default)
    {
        var key = $"someinfo:{name}:{id}"; // Unique key for this
combination.
        var bytes = await cache.GetAsync(key, token); // Try to get from
cache.
        SomeInformation info;
        if (bytes is null)
        {
            // Cache miss; get the data from the real source.
            info = await SomeExpensiveOperationAsync(name, id, token);

            // Serialize and cache it.
            bytes = SomeSerializer.Serialize(info);
            await cache.SetAsync(key, bytes, token);
        }
        else
        {
            // Cache hit; deserialize it.
            info = SomeSerializer.Deserialize<SomeInformation>(bytes);
        }
        return info;
    }

    // This is the work we're trying to cache.
    private async Task<SomeInformation> SomeExpensiveOperationAsync(string
name, int id,
        CancellationToken token = default)
```

```
    { /* ... */ }
}
```

That's a lot of work to get right each time, including things like serialization. And in the cache miss scenario, you could end up with multiple concurrent threads, all getting a cache miss, all fetching the underlying data, all serializing it, and all sending that data to the cache.

To simplify and improve this code with `HybridCache`, we first need to add the new library `Microsoft.Extensions.Caching.Hybrid`:

XML

```
<PackageReference Include="Microsoft.Extensions.Caching.Hybrid"
Version="9.0.0" />
```

Register the `HybridCache` service, like you would register an `IDistributedCache` implementation:

C#

```
builder.Services.AddHybridCache(); // Not shown: optional configuration API.
```

Now most caching concerns can be offloaded to `HybridCache`:

C#

```
public class SomeService(HybridCache cache)
{
    public async Task<SomeInformation> GetSomeInformationAsync
        (string name, int id, CancellationToken token = default)
    {
        return await cache.GetOrCreateAsync(
            $"someinfo:{name}:{id}", // Unique key for this combination.
            async cancel => await SomeExpensiveOperationAsync(name, id,
cancel),
            token: token
        );
    }
}
```

We provide a concrete implementation of the `HybridCache` abstract class via dependency injection, but it's intended that developers can provide custom implementations of the API. The `HybridCache` implementation deals with everything related to caching, including concurrent operation handling. The `cancel` token here

represents the combined cancellation of *all* concurrent callers—not just the cancellation of the caller we can see (that is, `token`).

High throughput scenarios can be further optimized by using the `TState` pattern, to avoid some overhead from captured variables and per-instance callbacks:

C#

```
public class SomeService(HybridCache cache)
{
    public async Task<SomeInformation> GetSomeInformationAsync(string name,
int id, CancellationToken token = default)
    {
        return await cache.GetOrCreateAsync(
            $"someinfo:{name}:{id}", // unique key for this combination
            (name, id), // all of the state we need for the final call, if
needed
            static async (state, token) =>
                await SomeExpensiveOperationAsync(state.name, state.id,
token),
            token: token
        );
    }
}
```

`HybridCache` uses the configured `IDistributedCache` implementation, if any, for secondary out-of-process caching, for example, using Redis. But even without an `IDistributedCache`, the `HybridCache` service will still provide in-process caching and "stampede" protection.

A note on object reuse

In typical existing code that uses `IDistributedCache`, every retrieval of an object from the cache results in deserialization. This behavior means that each concurrent caller gets a separate instance of the object, which cannot interact with other instances. The result is thread safety, as there's no risk of concurrent modifications to the same object instance.

Because a lot of `HybridCache` usage will be adapted from existing `IDistributedCache` code, `HybridCache` preserves this behavior by default to avoid introducing concurrency bugs. However, a given use case is inherently thread-safe:

- If the types being cached are immutable.
- If the code doesn't modify them.

In such cases, inform `HybridCache` that it's safe to reuse instances by:

- Marking the type as `sealed`. The `sealed` keyword in C# means that the class can't be inherited.
- Applying the `[ImmutableObject(true)]` attribute to it. The `[ImmutableObject(true)]` attribute indicates that the object's state can't be changed after it's created.

By reusing instances, `HybridCache` can reduce the overhead of CPU and object allocations associated with per-call deserialization. This can lead to performance improvements in scenarios where the cached objects are large or accessed frequently.

Other `HybridCache` features

Like `IDistributedCache`, `HybridCache` supports removal by key with a `RemoveKeyAsync` method.

`HybridCache` also provides optional APIs for `IDistributedCache` implementations, to avoid `byte[]` allocations. This feature is implemented by the preview versions of the `Microsoft.Extensions.Caching.StackExchangeRedis` and `Microsoft.Extensions.Caching.SqlServer` packages.

Serialization is configured as part of registering the service, with support for type-specific and generalized serializers via the `WithSerializer` and `.WithSerializerFactory` methods, chained from the `AddHybridCache` call. By default, the library handles `string` and `byte[]` internally, and uses `System.Text.Json` for everything else, but you can use protobuf, xml, or anything else.

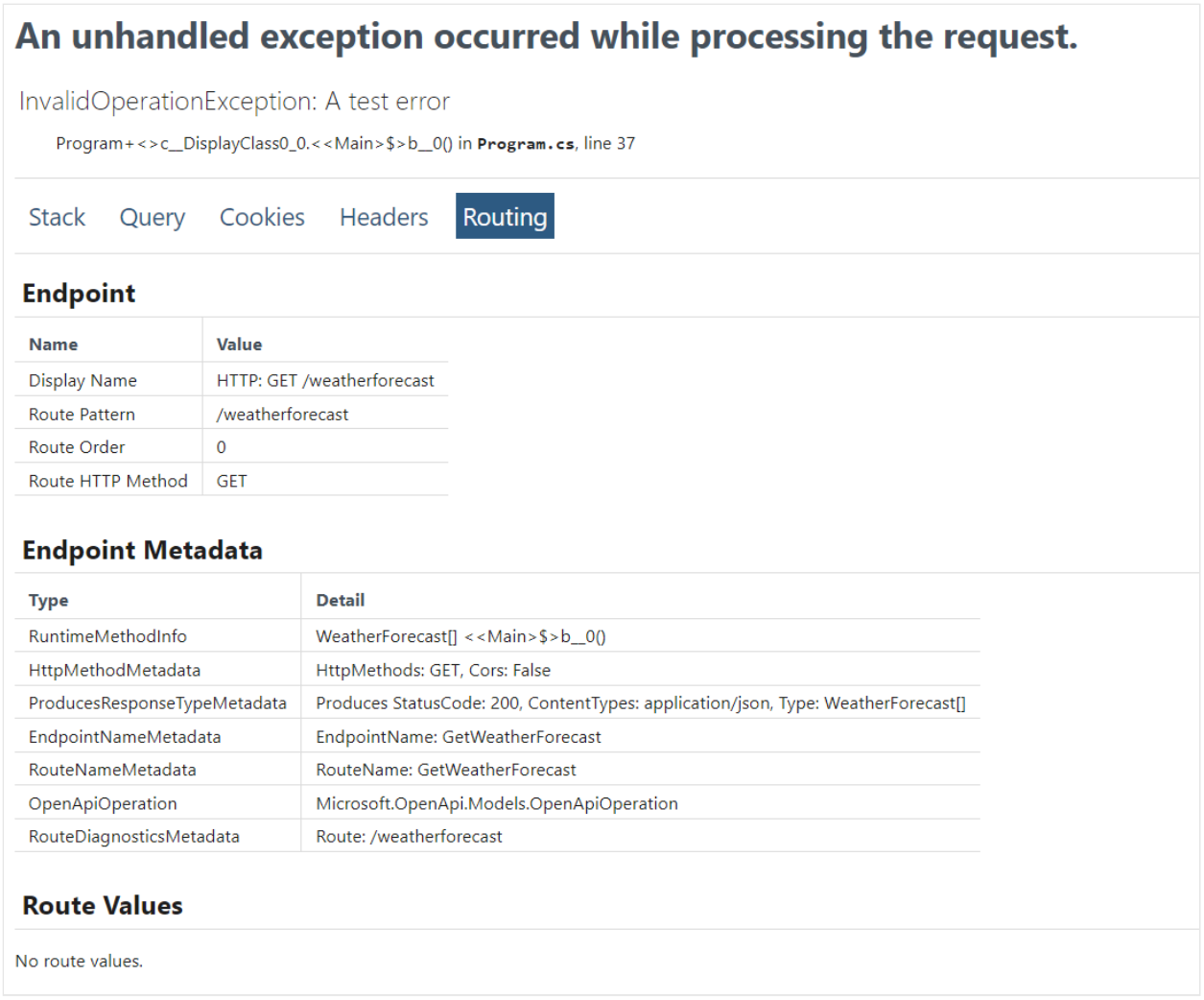
`HybridCache` supports older .NET runtimes, down to .NET Framework 4.7.2 and .NET Standard 2.0.

For more information about `HybridCache`, see HybridCache library in ASP.NET Core

Developer exception page improvements

The ASP.NET Core developer exception page is displayed when an app throws an unhandled exception during development. The developer exception page provides detailed information about the exception and request.

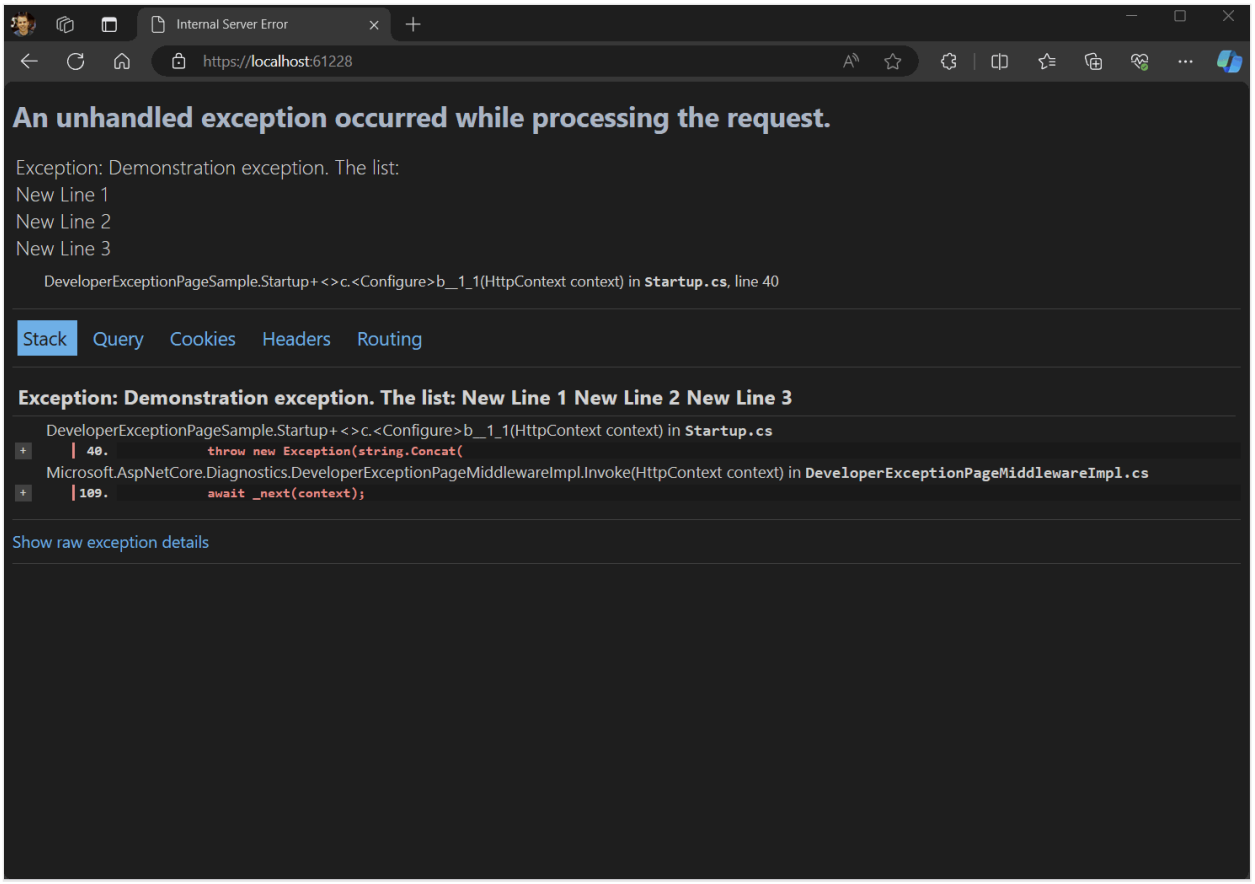
Preview 3 added endpoint metadata to the developer exception page. ASP.NET Core uses endpoint metadata to control endpoint behavior, such as routing, response caching, rate limiting, OpenAPI generation, and more. The following image shows the new metadata information in the `Routing` section of the developer exception page:



While testing the developer exception page, small quality of life improvements were identified. They shipped in Preview 4:

- Better text wrapping. Long cookies, query string values, and method names no longer add horizontal browser scroll bars.
- Bigger text which is found in modern designs.
- More consistent table sizes.

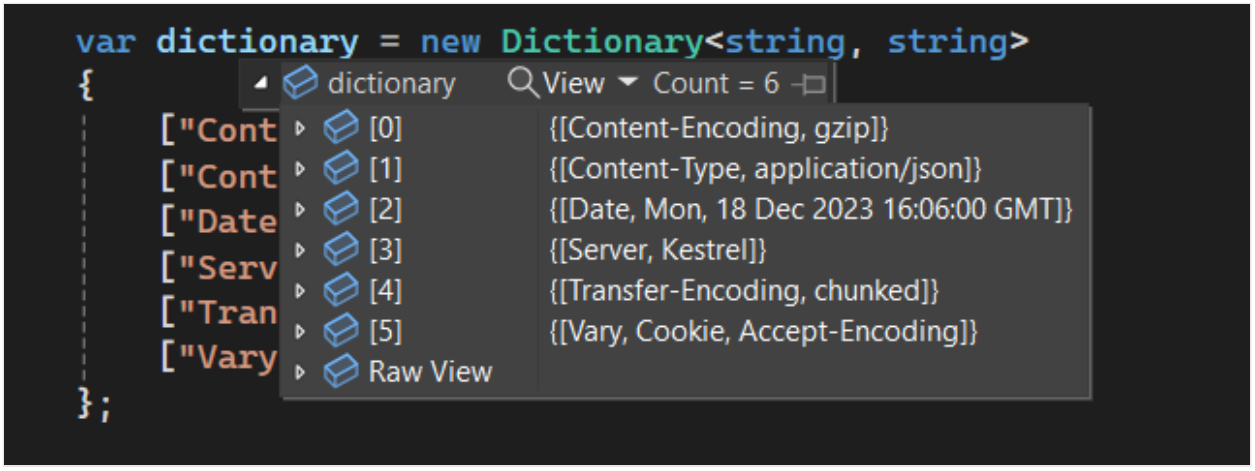
The following animated image shows the new developer exception page:



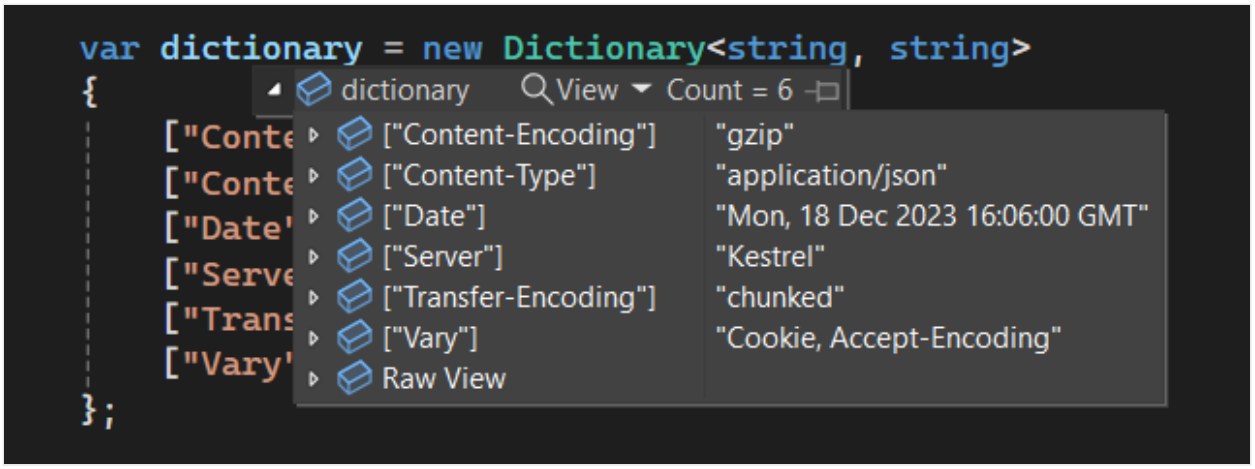
Dictionary debugging improvements

The debugging display of dictionaries and other key-value collections has an improved layout. The key is displayed in the debugger's key column instead of being concatenated with the value. The following images show the old and new display of a dictionary in the debugger.

Before:



After:



ASP.NET Core has many key-value collections. This improved debugging experience applies to:

- HTTP headers
- Query strings
- Forms
- Cookies
- View data
- Route data
- Features

Fix for 503's during app recycle in IIS

By default there is now a 1 second delay between when IIS is notified of a recycle or shutdown and when ANCM tells the managed server to start shutting down. The delay is configurable via the `ANCM_shutdownDelay` environment variable or by setting the `shutdownDelay` handler setting. Both values are in milliseconds. The delay is mainly to reduce the likelihood of a race where:

- IIS hasn't started queuing requests to go to the new app.
- ANCM starts rejecting new requests that come into the old app.

Slower machines or machines with heavier CPU usage may want to adjust this value to reduce 503 likelihood.

Example of setting `shutdownDelay`:

XML

```
<aspNetCore processPath="dotnet" arguments="myapp.dll"
stdoutLogEnabled="false" stdoutLogFile=".logsstdout">
  <handlerSettings>
    <!-- Milliseconds to delay shutdown by.
    this doesn't mean incoming requests will be delayed by this amount,
```

```
    but the old app instance will start shutting down after this timeout
occurs -->
    <handlerSetting name="shutdownDelay" value="5000" />
  </handlerSettings>
</aspNetCore>
```

The fix is in the globally installed ANCM module that comes from the hosting bundle.

ASP0026: Analyzer to warn when [Authorize] is overridden by [AllowAnonymous] from "farther away"

It seems intuitive that an `[Authorize]` attribute placed "closer" to an MVC action than an `[AllowAnonymous]` attribute would override the `[AllowAnonymous]` attribute and force authorization. However, this is not necessarily the case. What does matter is the relative order of the attributes.

The following code shows examples where a closer `[Authorize]` attribute gets overridden by an `[AllowAnonymous]` attribute that is farther away.

C#

```
[AllowAnonymous]
public class MyController
{
    [Authorize] // Overridden by the [AllowAnonymous] attribute on the class
    public IActionResult Private() => null;
}
```

C#

```
[AllowAnonymous]
public class MyControllerAnon : ControllerBase
{
}

[Authorize] // Overridden by the [AllowAnonymous] attribute on
MyControllerAnon
public class MyControllerInherited : MyControllerAnon
{
}

public class MyControllerInherited2 : MyControllerAnon
{
    [Authorize] // Overridden by the [AllowAnonymous] attribute on
MyControllerAnon
    public IActionResult Private() => null;
}
```

C#

```
[AllowAnonymous]
[Authorize] // Overridden by the preceding [AllowAnonymous]
public class MyControllerMultiple : ControllerBase
{
}
```

In .NET 9 Preview 6, we've introduced an analyzer that will highlight instances like these where a closer `[Authorize]` attribute gets overridden by an `[AllowAnonymous]` attribute that is farther away from an MVC action. The warning points to the overridden `[Authorize]` attribute with the following message:

```
ASP0026 [Authorize] overridden by [AllowAnonymous] from farther away
```

The correct action to take if you see this warning depends on the intention behind the attributes. The farther away `[AllowAnonymous]` attribute should be removed if it's unintentionally exposing the endpoint to anonymous users. If the `[AllowAnonymous]` attribute was intended to override a closer `[Authorize]` attribute, you can repeat the `[AllowAnonymous]` attribute after the `[Authorize]` attribute to clarify the intent.

C#

```
[AllowAnonymous]
public class MyController
{
    // This produces no warning because the second, "closer"
[AllowAnonymous]
    // clarifies that [Authorize] is intentionally overridden.
    // Specifying AuthenticationSchemes can still be useful
    // for endpoints that allow but don't require authenticated users.
    [Authorize(AuthenticationSchemes = "Cookies")]
    [AllowAnonymous]
    public IActionResult Privacy() => null;
}
```

Improved Kestrel connection metrics

We've made a significant improvement to Kestrel's connection metrics by including metadata about why a connection failed. The kestrel.connection.duration metric now includes the connection close reason in the `error.type` attribute.

Here is a small sample of the `error.type` values:

- `tls_handshake_failed` - The connection requires TLS, and the TLS handshake failed.
- `connection_reset` - The connection was unexpectedly closed by the client while requests were in progress.
- `request_headers_timeout` - Kestrel closed the connection because it didn't receive request headers in time.
- `max_request_body_size_exceeded` - Kestrel closed the connection because uploaded data exceeded max size.

Previously, diagnosing Kestrel connection issues required a server to record detailed, low-level logging. However, logs can be expensive to generate and store, and it can be difficult to find the right information among the noise.

Metrics are a much cheaper alternative that can be left on in a production environment with minimal impact. Collected metrics can drive dashboards and alerts. Once a problem is identified at a high-level with metrics, further investigation using logging and other tooling can begin.

We expect improved connection metrics to be useful in many scenarios:

- Investigating performance issues caused by short connection lifetimes.
- Observing ongoing external attacks on Kestrel that impact performance and stability.
- Recording attempted external attacks on Kestrel that Kestrel's built-in security hardening prevented.

For more information, see ASP.NET Core metrics.

Customize Kestrel named pipe endpoints

Kestrel's named pipe support has been improved with advanced customization options. The new `CreateNamedPipeServerStream` method on the named pipe options allows pipes to be customized per-endpoint.

An example of where this is useful is a Kestrel app that requires two pipe endpoints with different access security. The `CreateNamedPipeServerStream` option can be used to create pipes with custom security settings, depending on the pipe name.

```
C#
```

```
var builder = WebApplication.CreateBuilder();

builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenNamedPipe("pipe1");
    options.ListenNamedPipe("pipe2");
});

builder.WebHost.UseNamedPipes(options =>
{
    options.CreateNamedPipeServerStream = (context) =>
    {
        var pipeSecurity =
CreatePipeSecurity(context.NamedPipeEndpoint.PipeName);

        return
NamedPipeServerStreamAcl.Create(context.NamedPipeEndPoint.PipeName,
PipeDirection.InOut,
            NamedPipeServerStream.MaxAllowedServerInstances,
PipeTransmissionMode.Byte,
            context.PipeOptions, inBufferSize: 0, outBufferSize: 0,
pipeSecurity);
    };
});
```

`ExceptionHandlerMiddleware` option to choose the status code based on the exception type

A new option when configuring the `ExceptionHandlerMiddleware` enables app developers to choose what status code to return when an exception occurs during request handling. The new option changes the status code being set in the `ProblemDetails` response from the `ExceptionHandlerMiddleware`.

C#

```
app.UseExceptionHandler(new ExceptionHandlerOptions
{
    StatusCodeSelector = ex => ex is TimeoutException
        ? StatusCodes.Status503ServiceUnavailable
        : StatusCodes.Status500InternalServerError,
});
```

Opt-out of HTTP metrics on certain endpoints and requests

.NET 9 introduces the ability to opt-out of HTTP metrics for specific endpoints and requests. Opting out of recording metrics is beneficial for endpoints frequently called by automated systems, such as health checks. Recording metrics for these requests is generally unnecessary.

HTTP requests to an endpoint can be excluded from metrics by adding metadata. Either:

- Add the [DisableHttpMetrics] attribute to the Web API controller, SignalR hub or gRPC service.
- Call DisableHttpMetrics when mapping endpoints in app startup:

```
C#

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddHealthChecks();

var app = builder.Build();
app.MapHealthChecks("/healthz").DisableHttpMetrics();
app.Run();
```

The `MetricsDisabled` property has been added to `IHttpMetricsTagsFeature` for:

- Advanced scenarios where a request doesn't map to an endpoint.
- Dynamically disabling metrics collection for specific HTTP requests.

```
C#

// Middleware that conditionally opts-out HTTP requests.
app.Use(async (context, next) =>
{
    var metricsFeature = context.Features.Get<IHttpMetricsTagsFeature>();
    if (metricsFeature != null &&
        context.Request.Headers.ContainsKey("x-disable-metrics"))
    {
        metricsFeature.MetricsDisabled = true;
    }

    await next(context);
});
```

Data Protection support for deleting keys

Prior to .NET 9, data protection keys were ***not*** deletable by design, to prevent data loss. Deleting a key renders its protected data irretrievable. Given their small size, the accumulation of these keys generally posed minimal impact. However, to accommodate extremely long-running services, we have introduced the option to delete keys.

Generally, only old keys should be deleted. Only delete keys when you can accept the risk of data loss in exchange for storage savings. We recommend data protection keys should ***not*** be deleted.

C#

```
using Microsoft.AspNetCore.DataProtection.KeyManagement;

var services = new ServiceCollection();
services.AddDataProtection();

var serviceProvider = services.BuildServiceProvider();

var keyManager = serviceProvider.GetService<IKeyManager>();

if (keyManager is IDeletableKeyManager deletableKeyManager)
{
    var utcNow = DateTimeOffset.UtcNow;
    var yearAgo = utcNow.AddYears(-1);

    if (!deletableKeyManager.DeleteKeys(key => key.ExpirationDate <
yearAgo))
    {
        Console.WriteLine("Failed to delete keys.");
    }
    else
    {
        Console.WriteLine("Old keys deleted successfully.");
    }
}
else
{
    Console.WriteLine("Key manager does not support deletion.");
}
```

Middleware supports Keyed DI

Middleware now supports Keyed DI in both the constructor and the `Invoke`/`InvokeAsync` method:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddKeyedSingleton<MySingletonClass>("test");
builder.Services.AddKeyedScoped<MyScopedClass>("test2");

var app = builder.Build();
app.UseMiddleware<MyMiddleware>();
app.Run();
```

```
internal class MyMiddleware
{
    private readonly RequestDelegate _next;

    public MyMiddleware(RequestDelegate next,
        [FromKeyedServices("test")] MySingletonClass service)
    {
        _next = next;
    }

    public Task Invoke(HttpContext context,
        [FromKeyedServices("test2")]
            MyScopedClass scopedService) => _next(context);
}
```

Trust the ASP.NET Core HTTPS development certificate on Linux

On Ubuntu and Fedora based Linux distros, `dotnet dev-certs https --trust` now configures ASP.NET Core HTTPS development certificate as a trusted certificate for:

- Chromium browsers, for example, Google Chrome, Microsoft Edge, and Chromium.
- Mozilla Firefox and Mozilla derived browsers.
- .NET APIs, for example, HttpClient

Previously, `--trust` only worked on Windows and macOS. Certificate trust is applied per-user.

To establish trust in OpenSSL, the `dev-certs` tool:

- Puts the certificate in `~/.aspnet/dev-certs/trust`
- Runs a simplified version of OpenSSL's c_rehash tool on the directory.
- Asks the user to update the `SSL_CERT_DIR` environment variable.

To establish trust in dotnet, the tool puts the certificate in the `My/Root` certificate store.

To establish trust in NSS databases, if any, the tool searches the home directory for Firefox profiles, `~/.pki/nssdb`, and `~/snap/chromium/current/.pki/nssdb`. For each directory found, the tool adds an entry to the `nssdb`.

Templates updated to latest Bootstrap, jQuery, and jQuery Validation versions

The ASP.NET Core project templates and libraries have been updated to use the latest versions of Bootstrap, jQuery, and jQuery Validation, specifically:

- Bootstrap 5.3.3
- jQuery 3.7.1
- jQuery Validation 1.21.0

What's new in ASP.NET Core 8.0

Article • 11/06/2024

This article highlights the most significant changes in ASP.NET Core 8.0 with links to relevant documentation.

Blazor

Full-stack web UI

With the release of .NET 8, Blazor is a full-stack web UI framework for developing apps that render content at either the component or page level with:

- Static Server rendering (also called *static server-side rendering*, static SSR) to generate static HTML on the server.
- Interactive Server rendering (also called *interactive server-side rendering*, interactive SSR) to generate interactive components with prerendering on the server.
- Interactive WebAssembly rendering (also called *client-side rendering*, CSR, which is always assumed to be interactive) to generate interactive components on the client with prerendering on the server.
- Interactive Auto (automatic) rendering to initially use the server-side ASP.NET Core runtime for content rendering and interactivity. The .NET WebAssembly runtime on the client is used for subsequent rendering and interactivity after the Blazor bundle is downloaded and the WebAssembly runtime activates. Interactive Auto rendering usually provides the fastest app startup experience.

Interactive render modes also prerender content by default.

For more information, see the following articles:

- ASP.NET Core Blazor fundamentals: New sections on rendering and static/interactive concepts appear at the top of the article.
- ASP.NET Core Blazor render modes
- **Migration coverage**: Migrate from ASP.NET Core 7.0 to 8.0

Examples throughout the Blazor documentation have been updated for use in Blazor Web Apps. Blazor Server examples remain in content versioned for .NET 7 or earlier.

New article on class libraries with static server-side rendering (static SSR)

We've added a new article that discusses component library authorship in Razor class libraries (RCLs) with static server-side rendering (static SSR).

For more information, see ASP.NET Core Razor class libraries (RCLs) with static server-side rendering (static SSR).

New article on HTTP caching issues

We've added a new article that discusses some of the common HTTP caching issues that can occur when upgrading Blazor apps across major versions and how to address HTTP caching issues.

For more information, see Avoid HTTP caching issues when upgrading ASP.NET Core Blazor apps.

New Blazor Web App template

We've introduced a new Blazor project template: the *Blazor Web App* template. The new template provides a single starting point for using Blazor components to build any style of web UI. The template combines the strengths of the existing Blazor Server and Blazor WebAssembly hosting models with the new Blazor capabilities added in .NET 8: static server-side rendering (static SSR), streaming rendering, enhanced navigation and form handling, and the ability to add interactivity using either Blazor Server or Blazor WebAssembly on a per-component basis.

As part of unifying the various Blazor hosting models into a single model in .NET 8, we're also consolidating the number of Blazor project templates. We removed the Blazor Server template, and the ASP.NET Core Hosted option has been removed from the Blazor WebAssembly template. Both of these scenarios are represented by options when using the Blazor Web App template.

> **Note**
>
> Existing Blazor Server and Blazor WebAssembly apps remain supported in .NET 8. Optionally, these apps can be updated to use the new full-stack web UI Blazor features.

For more information on the new Blazor Web App template, see the following articles:

- Tooling for ASP.NET Core Blazor
- ASP.NET Core Blazor project structure

New JS initializers for Blazor Web Apps

For Blazor Server, Blazor WebAssembly, and Blazor Hybrid apps:

- `beforeStart` is used for tasks such as customizing the loading process, logging level, and other options.
- `afterStarted` is used for tasks such as registering Blazor event listeners and custom event types.

The preceding legacy JS initializers aren't invoked by default in a Blazor Web App. For Blazor Web Apps, a new set of JS initializers are used: `beforeWebStart`, `afterWebStarted`, `beforeServerStart`, `afterServerStarted`, `beforeWebAssemblyStart`, and `afterWebAssemblyStarted`.

For more information, see ASP.NET Core Blazor startup.

Split of prerendering and integration guidance

For prior releases of .NET, we covered prerendering and integration in a single article. To simplify and focus our coverage, we've split the subjects into the following new articles, which have been updated for .NET 8:

- Prerender ASP.NET Core Razor components
- Integrate ASP.NET Core Razor components with MVC or Razor Pages

Persist component state in a Blazor Web App

You can persist and read component state in a Blazor Web App using the existing PersistentComponentState service. This is useful for persisting component state during prerendering.

Blazor Web Apps automatically persist any registered app-level state created during prerendering, removing the need for the Persist Component State Tag Helper.

Form handling and model binding

Blazor components can now handle submitted form requests, including model binding and validating the request data. Components can implement forms with separate form handlers using the standard HTML `<form>` tag or using the existing `EditForm` component.

Form model binding in Blazor honors the data contract attributes (for example, `[DataMember]` and `[IgnoreDataMember]`) for customizing how the form data is bound to the model.

New antiforgery support is included in .NET 8. A new `AntiforgeryToken` component renders an antiforgery token as a hidden field, and the new `[RequireAntiforgeryToken]` attribute enables antiforgery protection. If an antiforgery check fails, a 400 (Bad Request) response is returned without form processing. The new antiforgery features are enabled by default for forms based on `Editform` and can be applied manually to standard HTML forms.

For more information, see ASP.NET Core Blazor forms overview.

Enhanced navigation and form handling

Static server-side rendering (static SSR) typically performs a full page refresh whenever the user navigates to a new page or submits a form. In .NET 8, Blazor can enhance page navigation and form handling by intercepting the request and performing a fetch request instead. Blazor then handles the rendered response content by patching it into the browser DOM. Enhanced navigation and form handling avoids the need for a full page refresh and preserves more of the page state, so pages load faster and more smoothly. Enhanced navigation is enabled by default when the Blazor script (`blazor.web.js`) is loaded. Enhanced form handling can be optionally enabled for specific forms.

New enhanced navigation API allows you to refresh the current page by calling `NavigationManager.Refresh(bool forceLoad = false)`.

For more information, see the following sections of the Blazor *Routing* article:

- Enhanced navigation and form handling
- Location changes

New article on static rendering with enhanced navigation for JS interop

Some apps depend on JS interop to perform initialization tasks that are specific to each page. When using Blazor's enhanced navigation feature with statically-rendered pages that perform JS interop initialization tasks, page-specific JS may not be executed again as expected each time an enhanced page navigation occurs. A new article explains how to address this scenario in Blazor Web Apps:

ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)

Streaming rendering

You can now stream content updates on the response stream when using static server-side rendering (static SSR) with Blazor. Streaming rendering can improve the user experience for pages that perform long-running asynchronous tasks in order to fully render by rendering content as soon as it's available.

For example, to render a page you might need to make a long running database query or an API call. Normally, asynchronous tasks executed as part of rendering a page must complete before the rendered response is sent, which can delay loading the page. Streaming rendering initially renders the entire page with placeholder content while asynchronous operations execute. After the asynchronous operations are complete, the updated content is sent to the client on the same response connection and patched by into the DOM. The benefit of this approach is that the main layout of the app renders as quickly as possible and the page is updated as soon as the content is ready.

For more information, see ASP.NET Core Razor component rendering.

Inject keyed services into components

Blazor now supports injecting keyed services using the `[Inject]` attribute. Keys allow for scoping of registration and consumption of services when using dependency injection. Use the new `InjectAttribute.Key` property to specify the key for the service to inject:

C#

```
[Inject(Key = "my-service")]
public IMyService MyService { get; set; }
```

The `@inject` Razor directive doesn't support keyed services for this release, but work is tracked by Update @inject to support keyed services (dotnet/razor #9286) for a future .NET release.

For more information, see ASP.NET Core Blazor dependency injection.

Access `HttpContext` as a cascading parameter

You can now access the current HttpContext as a cascading parameter from a static server component:

C#

```
[CascadingParameter]
public HttpContext? HttpContext { get; set; }
```

Accessing the HttpContext from a static server component might be useful for inspecting and modifying headers or other properties.

For an example that passes HttpContext state, access and refresh tokens, to components, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Render Razor components outside of ASP.NET Core

You can now render Razor components outside the context of an HTTP request. You can render Razor components as HTML directly to a string or stream independently of the ASP.NET Core hosting environment. This is convenient for scenarios where you want to generate HTML fragments, such as for a generating email or static site content.

For more information, see Render Razor components outside of ASP.NET Core.

Sections support

The new `SectionOutlet` and `SectionContent` components in Blazor add support for specifying outlets for content that can be filled in later. Sections are often used to define placeholders in layouts that are then filled in by specific pages. Sections are referenced either by a unique name or using a unique object ID.

For more information, see ASP.NET Core Blazor sections.

Error page support

Blazor Web Apps can define a custom error page for use with the ASP.NET Core exception handling middleware. The Blazor Web App project template includes a default error page (`Components/Pages/Error.razor`) with similar content to the one used in MVC and Razor Pages apps. When the error page is rendered in response to a request from Exception Handling Middleware, the error page always renders as a static server component, even if interactivity is otherwise enabled.

Error.razor in 8.0 reference source

QuickGrid

The Blazor QuickGrid component is no longer experimental and is now part of the Blazor framework in .NET 8.

QuickGrid is a high performance grid component for displaying data in tabular form. QuickGrid is built to be a simple and convenient way to display your data, while still providing powerful features, such as sorting, filtering, paging, and virtualization.

For more information, see ASP.NET Core Blazor QuickGrid component.

Route to named elements

Blazor now supports using client-side routing to navigate to a specific HTML element on a page using standard URL fragments. If you specify an identifier for an HTML element using the standard `id` attribute, Blazor correctly scrolls to that element when the URL fragment matches the element identifier.

For more information, see ASP.NET Core Blazor routing and navigation.

Root-level cascading values

Root-level cascading values can be registered for the entire component hierarchy. Named cascading values and subscriptions for update notifications are supported.

For more information, see ASP.NET Core Blazor cascading values and parameters.

Virtualize empty content

Use the new `EmptyContent` parameter on the `Virtualize` component to supply content when the component has loaded and either `Items` is empty or `ItemsProviderResult<T>.TotalItemCount` is zero.

For more information, see ASP.NET Core Razor component virtualization.

Close circuits when there are no remaining interactive server components

Interactive server components handle web UI events using a real-time connection with the browser called a circuit. A circuit and its associated state are set up when a root interactive server component is rendered. The circuit is closed when there are no remaining interactive server components on the page, which frees up server resources.

Monitor SignalR circuit activity

You can now monitor inbound circuit activity in server-side apps using the new `CreateInboundActivityHandler` method on `CircuitHandler`. Inbound circuit activity is any activity sent from the browser to the server, such as UI events or JavaScript-to-.NET interop calls.

For more information, see ASP.NET Core Blazor SignalR guidance.

Faster runtime performance with the Jiterpreter

The *Jiterpreter* is a new runtime feature in .NET 8 that enables partial Just-in-Time (JIT) compilation support when running on WebAssembly to achieve improved runtime performance.

For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

Ahead-of-time (AOT) SIMD and exception handling

Blazor WebAssembly ahead-of-time (AOT) compilation now uses WebAssembly Fixed-width SIMD and WebAssembly Exception handling by default to improve runtime performance.

For more information, see the following articles:

- AOT: Single Instruction, Multiple Data (SIMD)
- AOT: Exception handling

Web-friendly Webcil packaging

Webcil is web-friendly packaging of .NET assemblies that removes content specific to native Windows execution to avoid issues when deploying to environments that block the download or use of `.dll` files. Webcil is enabled by default for Blazor WebAssembly apps.

For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

> **Note**
>
> Prior to the release of .NET 8, guidance in Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps addresses environments that block clients from downloading and executing DLLs with a multipart bundling approach. In .NET 8 or

later, Blazor uses the Webcil file format to address this problem. Multipart bundling using the experimental NuGet package described by the *WebAssembly deployment layout* article isn't supported for Blazor apps in .NET 8 or later. For more information, see **Enhance Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle package to define a custom bundle format (dotnet/aspnetcore #36978)**. If you desire to continue using the multipart bundle package in .NET 8 or later apps, you can use the guidance in the article to create your own multipart bundling NuGet package, but it won't be supported by Microsoft.

Blazor WebAssembly debugging improvements

When debugging .NET on WebAssembly, the debugger now downloads symbol data from symbol locations that are configured in Visual Studio preferences. This improves the debugging experience for apps that use NuGet packages.

You can now debug Blazor WebAssembly apps using Firefox. Debugging Blazor WebAssembly apps requires configuring the browser for remote debugging and then connecting to the browser using the browser developer tools through the .NET WebAssembly debugging proxy. Debugging Firefox from Visual Studio isn't supported at this time.

For more information, see Debug ASP.NET Core Blazor apps.

Content Security Policy (CSP) compatibility

Blazor WebAssembly no longer requires enabling the `unsafe-eval` script source when specifying a Content Security Policy (CSP).

For more information, see Enforce a Content Security Policy for ASP.NET Core Blazor.

Handle caught exceptions outside of a Razor component's lifecycle

Use `ComponentBase.DispatchExceptionAsync` in a Razor component to process exceptions thrown outside of the component's lifecycle call stack. This permits the component's code to treat exceptions as though they're lifecycle method exceptions. Thereafter, Blazor's error handling mechanisms, such as error boundaries, can process exceptions.

For more information, see Handle errors in ASP.NET Core Blazor apps.

Configure the .NET WebAssembly runtime

The .NET WebAssembly runtime can now be configured for Blazor startup.

For more information, see ASP.NET Core Blazor startup.

Configuration of connection timeouts in `HubConnectionBuilder`

Prior workarounds for configuring hub connection timeouts can be replaced with formal SignalR hub connection builder timeout configuration.

For more information, see the following:

- ASP.NET Core Blazor SignalR guidance
- Host and deploy ASP.NET Core Blazor WebAssembly
- Host and deploy ASP.NET Core server-side Blazor apps

Project templates shed Open Iconic

The Blazor project templates no longer depend on Open Iconic for icons.

Support for dialog cancel and close events

Blazor now supports the cancel and close events on the `dialog` HTML element.

In the following example:

- `OnClose` is called when the `my-dialog` dialog is closed with the **Close** button.
- `OnCancel` is called when the dialog is canceled with the `Esc` key. When an HTML dialog is dismissed with the `Esc` key, both the `cancel` and `close` events are triggered.

razor

```
<div>
    <p>Output: @message</p>

    <button onclick="document.getElementById('my-dialog').showModal()">
        Show modal dialog
    </button>

    <dialog id="my-dialog" @onclose="OnClose" @oncancel="OnCancel">
        <p>Hi there!</p>
```

```
        <form method="dialog">
            <button>Close</button>
        </form>
    </dialog>
</div>

@code {
    private string? message;

    private void OnClose(EventArgs e) => message += "onclose, ";

    private void OnCancel(EventArgs e) => message += "oncancel, ";
}
```

Blazor Identity UI

Blazor supports generating a full Blazor-based Identity UI when you choose the authentication option for *Individual Accounts*. You can either select the option for Individual Accounts in the new project dialog for Blazor Web Apps from Visual Studio or pass the `-au|--auth` option set to `Individual` from the command line when you create a new project.

For more information, see the following resources:

- ASP.NET Core Blazor authentication and authorization
- What's new with identity in .NET 8 (blog post)

Secure Blazor WebAssembly with ASP.NET Core Identity

The Blazor documentation hosts a new article and sample app to cover securing a standalone Blazor WebAssembly app with ASP.NET Core Identity.

For more information, see the following resources:

- Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity
- What's new with identity in .NET 8 (blog post)

Blazor Server with Yarp routing

Routing and deep linking for Blazor Server with Yarp work correctly in .NET 8.

For more information, see Migrate from ASP.NET Core 7.0 to 8.0.

Multiple Blazor Web Apps per server project

Support for multiple Blazor Web Apps per server project will be considered for .NET 10 (November, 2025).

For more information, see Support for multiple Blazor Web apps per server project (dotnet/aspnetcore #52216).

Blazor Hybrid

The following articles document changes for Blazor Hybrid in .NET 8:

- Troubleshoot ASP.NET Core Blazor Hybrid: A new article explains how to use BlazorWebView logging.
- Build a .NET MAUI Blazor Hybrid app: The project template name **.NET MAUI Blazor** has changed to **.NET MAUI Blazor Hybrid**.
- ASP.NET Core Blazor Hybrid: `BlazorWebView` gains a `TryDispatchAsync` method that calls a specified `Action<ServiceProvider>` asynchronously and passes in the scoped services available in Razor components. This enables code from the native UI to access scoped services such as `NavigationManager`.
- ASP.NET Core Blazor Hybrid routing and navigation: Use the `BlazorWebView.StartPath` property to get or set the path for initial navigation within the Blazor navigation context when the Razor component is finished loading.

`[Parameter]` attribute is no longer required when supplied from the query string

The `[Parameter]` attribute is no longer required when supplying a parameter from the query string:

```diff
- [Parameter]
  [SupplyParameterFromQuery]
```

SignalR

New approach to set the server timeout and Keep-Alive interval

ServerTimeout (default: 30 seconds) and KeepAliveInterval (default: 15 seconds) can be set directly on HubConnectionBuilder.

Prior approach for JavaScript clients

The following example shows the assignment of values that are double the default values in ASP.NET Core 7.0 or earlier:

JavaScript

```
var connection = new signalR.HubConnectionBuilder()
  .withUrl("/chatHub")
  .build();

connection.serverTimeoutInMilliseconds = 60000;
connection.keepAliveIntervalInMilliseconds = 30000;
```

New approach for JavaScript clients

The following example shows the ***new approach*** for assigning values that are double the default values in ASP.NET Core 8.0 or later:

JavaScript

```
var connection = new signalR.HubConnectionBuilder()
  .withUrl("/chatHub")
  .withServerTimeout(60000)
  .withKeepAlive(30000)
  .build();
```

Prior approach for the JavaScript client of a Blazor Server app

The following example shows the assignment of values that are double the default values in ASP.NET Core 7.0 or earlier:

JavaScript

```
Blazor.start({
  configureSignalR: function (builder) {
    let c = builder.build();
    c.serverTimeoutInMilliseconds = 60000;
    c.keepAliveIntervalInMilliseconds = 30000;
    builder.build = () => {
      return c;
    };
```

```
  }
});
```

New approach for the JavaScript client of server-side Blazor app

The following example shows the ***new approach*** for assigning values that are double the default values in ASP.NET Core 8.0 or later for Blazor Web Apps and Blazor Server.

Blazor Web App:

JavaScript

```
Blazor.start({
  circuit: {
    configureSignalR: function (builder) {
      builder.withServerTimeout(60000).withKeepAliveInterval(30000);
    }
  }
});
```

Blazor Server:

JavaScript

```
Blazor.start({
  configureSignalR: function (builder) {
    builder.withServerTimeout(60000).withKeepAliveInterval(30000);
  }
});
```

Prior approach for .NET clients

The following example shows the assignment of values that are double the default values in ASP.NET Core 7.0 or earlier:

C#

```
var builder = new HubConnectionBuilder()
    .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
    .Build();

builder.ServerTimeout = TimeSpan.FromSeconds(60);
builder.KeepAliveInterval = TimeSpan.FromSeconds(30);

builder.On<string, string>("ReceiveMessage", (user, message) => ...
```

```
await builder.StartAsync();
```

New approach for .NET clients

The following example shows the ***new approach*** for assigning values that are double the default values in ASP.NET Core 8.0 or later:

```
C#

var builder = new HubConnectionBuilder()
    .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
    .WithServerTimeout(TimeSpan.FromSeconds(60))
    .WithKeepAliveInterval(TimeSpan.FromSeconds(30))
    .Build();

builder.On<string, string>("ReceiveMessage", (user, message) => ...

await builder.StartAsync();
```

SignalR stateful reconnect

SignalR stateful reconnect reduces the perceived downtime of clients that have a temporary disconnect in their network connection, such as when switching network connections or a short temporary loss in access.

Stateful reconnect achieves this by:

- Temporarily buffering data on the server and client.
- Acknowledging messages received (ACK-ing) by both the server and client.
- Recognizing when a connection is returning and replaying messages that might have been sent while the connection was down.

Stateful reconnect is available in ASP.NET Core 8.0 and later.

Opt in to stateful reconnect at both the server hub endpoint and the client:

- Update the server hub endpoint configuration to enable the `AllowStatefulReconnects` option:

  ```
  C#

  app.MapHub<MyHub>("/hubName", options =>
  {
  ```

```
    options.AllowStatefulReconnects = true;
});
```

Optionally, the maximum buffer size in bytes allowed by the server can be set globally or for a specific hub with the `StatefulReconnectBufferSize` option:

The `StatefulReconnectBufferSize` option set globally:

C#

```
builder.AddSignalR(o => o.StatefulReconnectBufferSize = 1000);
```

The `StatefulReconnectBufferSize` option set for a specific hub:

C#

```
builder.AddSignalR().AddHubOptions<MyHub>(o =>
o.StatefulReconnectBufferSize = 1000);
```

The `StatefulReconnectBufferSize` option is optional with a default of 100,000 bytes.

- Update JavaScript or TypeScript client code to enable the `withStatefulReconnect` option:

JavaScript

```
const builder = new signalR.HubConnectionBuilder()
  .withUrl("/hubname")
  .withStatefulReconnect({ bufferSize: 1000 });  // Optional, defaults
to 100,000
const connection = builder.build();
```

The `bufferSize` option is optional with a default of 100,000 bytes.

- Update .NET client code to enable the `WithStatefulReconnect` option:

C#

```
  var builder = new HubConnectionBuilder()
      .WithUrl("<hub url>")
      .WithStatefulReconnect();
  builder.Services.Configure<HubConnectionOptions>(o =>
o.StatefulReconnectBufferSize = 1000);
  var hubConnection = builder.Build();
```

The `StatefulReconnectBufferSize` option is optional with a default of 100,000 bytes.

For more information, see Configure stateful reconnect.

Minimal APIs

This section describes new features for minimal APIs. See also the section on Native AOT for more information relevant to minimal APIs.

User override culture

Starting in ASP.NET Core 8.0, the RequestLocalizationOptions.CultureInfoUseUserOverride property allows the application to decide whether or not to use nondefault Windows settings for the CultureInfo DateTimeFormat and NumberFormat properties. This has no impact on Linux. This directly corresponds to UseUserOverride.

C#

```
app.UseRequestLocalization(options =>
{
    options.CultureInfoUseUserOverride = false;
});
```

Binding to forms

Explicit binding to form values using the [FromForm] attribute is now supported. Parameters bound to the request with `[FromForm]` include an antiforgery token. The antiforgery token is validated when the request is processed.

Inferred binding to forms using the IFormCollection, IFormFile, and IFormFileCollection types is also supported. OpenAPI metadata is inferred for form parameters to support integration with Swagger UI.

For more information, see:

- Explicit binding from form values.
- Binding to forms with IFormCollection, IFormFile, and IFormFileCollection.
- Form binding in minimal APIs

Binding from forms is now supported for:

- Collections, for example List and Dictionary
- Complex types, for example, `Todo` or `Project`

For more information, see Bind to collections and complex types from forms.

Antiforgery with minimal APIs

This release adds a middleware for validating antiforgery tokens, which are used to mitigate cross-site request forgery attacks. Call AddAntiforgery to register antiforgery services in DI. `WebApplicationBuilder` automatically adds the middleware when the antiforgery services have been registered in the DI container. Antiforgery tokens are used to mitigate cross-site request forgery attacks.

```
C#

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The antiforgery middleware:

- Does ***not*** short-circuit the execution of the rest of the request pipeline.
- Sets the IAntiforgeryValidationFeature in the HttpContext.Features of the current request.

The antiforgery token is only validated if:

- The endpoint contains metadata implementing IAntiforgeryMetadata where `RequiresValidation=true`.
- The HTTP method associated with the endpoint is a relevant HTTP method. The relevant methods are all HTTP methods except for TRACE, OPTIONS, HEAD, and GET.
- The request is associated with a valid endpoint.

For more information, see Antiforgery with Minimal APIs.

New `IResettable` interface in `ObjectPool`

Microsoft.Extensions.ObjectPool provides support for pooling object instances in memory. Apps can use an object pool if the values are expensive to allocate or initialize.

In this release, we've made the object pool easier to use by adding the IResettable interface. Reusable types often need to be reset back to a default state between uses. `IResettable` types are automatically reset when returned to an object pool.

For more information, see the ObjectPool sample.

Native AOT

Support for .NET native ahead-of-time (AOT) has been added. Apps that are published using AOT can have substantially better performance: smaller app size, less memory usage, and faster startup time. Native AOT is currently supported by gRPC, minimal API, and worker service apps. For more information, see ASP.NET Core support for Native AOT and Tutorial: Publish an ASP.NET Core app using Native AOT. For information about known issues with ASP.NET Core and Native AOT compatibility, see GitHub issue dotnet/core #8288 .

Libraries and Native AOT

Many of the popular libraries used in ASP.NET Core projects currently have some compatibility issues when used in a project targeting Native AOT, such as:

- Use of reflection to inspect and discover types.
- Conditionally loading libraries at runtime.
- Generating code on the fly to implement functionality.

Libraries using these dynamic features need to be updated in order to work with Native AOT. They can be updated using tools like Roslyn source generators.

Library authors hoping to support Native AOT are encouraged to:

- Read about Native AOT compatibility requirements.
- Prepare the library for trimming.

New project template

The new **ASP.NET Core Web API (Native AOT)** project template (short name `webapiaot`) creates a project with AOT publish enabled. For more information, see The Web API (Native AOT) template.

New `CreateSlimBuilder` method

The CreateSlimBuilder() method used in the Web API (Native AOT) template initializes the WebApplicationBuilder with the minimum ASP.NET Core features necessary to run an app. The `CreateSlimBuilder` method includes the following features that are typically needed for an efficient development experience:

- JSON file configuration for `appsettings.json` and `appsettings.{EnvironmentName}.json`.
- User secrets configuration.
- Console logging.
- Logging configuration.

For more information, see The CreateSlimBuilder method.

New `CreateEmptyBuilder` method

There's another new WebApplicationBuilder factory method for building small apps that only contain necessary features: `WebApplication.CreateEmptyBuilder(WebApplicationOptions options)`. This `WebApplicationBuilder` is created with no built-in behavior. The app it builds contains only the services and middleware that are explicitly configured.

Here's an example of using this API to create a small web application:

C#

```
var builder = WebApplication.CreateEmptyBuilder(new
WebApplicationOptions());
builder.WebHost.UseKestrelCore();

var app = builder.Build();

app.Use(async (context, next) =>
{
    await context.Response.WriteAsync("Hello, World!");
    await next(context);
});

Console.WriteLine("Running...");
app.Run();
```

Publishing this code with Native AOT using .NET 8 Preview 7 on a linux-x64 machine results in a self-contained native executable of about 8.5 MB.

Reduced app size with configurable HTTPS support

We've further reduced Native AOT binary size for apps that don't need HTTPS or HTTP/3 support. Not using HTTPS or HTTP/3 is common for apps that run behind a TLS termination proxy (for example, hosted on Azure). The new `WebApplication.CreateSlimBuilder` method omits this functionality by default. It can be added by calling `builder.WebHost.UseKestrelHttpsConfiguration()` for HTTPS or `builder.WebHost.UseQuic()` for HTTP/3. For more information, see The CreateSlimBuilder method.

JSON serialization of compiler-generated `IAsyncEnumerable<T>` types

New features were added to System.Text.Json to better support Native AOT. These new features add capabilities for the source generation mode of `System.Text.Json`, because reflection isn't supported by AOT.

One of the new features is support for JSON serialization of IAsyncEnumerable<T> implementations implemented by the C# compiler. This support opens up their use in ASP.NET Core projects configured to publish Native AOT.

This API is useful in scenarios where a route handler uses `yield return` to asynchronously return an enumeration. For example, to materialize rows from a database query. For more information, see Unspeakable type support in the .NET 8 Preview 4 announcement.

For information abut other improvements in `System.Text.Json` source generation, see Serialization improvements in .NET 8.

Top-level APIs annotated for trim warnings

The main entry points to subsystems that don't work reliably with Native AOT are now annotated. When these methods are called from an application with Native AOT enabled, a warning is provided. For example, the following code produces a warning at the invocation of `AddControllers` because this API isn't trim-safe and isn't supported by Native AOT.

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddControllers();
var app = builder.Build();

app.Run();
```

IL2026: Using member 'Microsoft.Extensions.DependencyInjection.MvcServiceCollectionExtensions.AddControllers(IServiceCollection)' which has 'RequiresUnreferencedCodeAttribute' can break functionality when trimming application code. MVC does not currently support native AOT. https://aka.ms/aspnet/nativeaot

Request delegate generator

In order to make Minimal APIs compatible with Native AOT, we're introducing the Request Delegate Generator (RDG). The RDG is a source generator that does what the RequestDelegateFactory (RDF) does. That is, it turns the various `MapGet()`, `MapPost()`, and calls like them into RequestDelegate instances associated with the specified routes. But rather than doing it in-memory in an application when it starts, the RDG does it at compile time and generates C# code directly into the project. The RDG:

- Removes the runtime generation of this code.
- Ensures that the types used in APIs are statically analyzable by the Native AOT tool-chain.
- Ensures that required code isn't trimmed away.

We're working to ensure that as many as possible of the Minimal API features are supported by the RDG and thus compatible with Native AOT.

The RDG is enabled automatically in a project when publishing with Native AOT is enabled. RDG can be manually enabled even when not using Native AOT by setting `<EnableRequestDelegateGenerator>true</EnableRequestDelegateGenerator>` in the project file. This can be useful when initially evaluating a project's readiness for Native AOT, or to reduce the startup time of an app.

Improved performance using Interceptors

The Request Delegate Generator uses the new C# 12 interceptors compiler feature to support intercepting calls to minimal API Map methods with statically generated

variants at runtime. The use of interceptors results in increased startup performance for apps compiled with `PublishAot`.

Logging and exception handling in compile-time generated minimal APIs

Minimal APIs generated at run time support automatically logging (or throwing exceptions in Development environments) when parameter binding fails. .NET 8 introduces the same support for APIs generated at compile time via the Request Delegate Generator (RDG). For more information, see Logging and exception handling in compile-time generated minimal APIs.

AOT and System.Text.Json

Minimal APIs are optimized for receiving and returning JSON payloads using `System.Text.Json`, so the compatibility requirements for JSON and Native AOT apply too. Native AOT compatibility requires the use of the `System.Text.Json` source generator. All types accepted as parameters to or returned from request delegates in Minimal APIs must be configured on a `JsonSerializerContext` that is registered via ASP.NET Core's dependency injection, for example:

C#

```
// Register the JSON serializer context with DI
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

...

// Add types used in the minimal API app to source generated JSON serializer
content
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

For more information about the TypeInfoResolverChain API, see the following resources:

- JsonSerializerOptions.TypeInfoResolverChain
- Chain source generators

- Changes to support source generation

Libraries and Native AOT

Many of the common libraries available for ASP.NET Core projects today have some compatibility issues if used in a project targeting Native AOT. Popular libraries often rely on the dynamic capabilities of .NET reflection to inspect and discover types, conditionally load libraries at runtime, and generate code on the fly to implement their functionality. These libraries need to be updated in order to work with Native AOT by using tools like Roslyn source generators.

Library authors wishing to learn more about preparing their libraries for Native AOT are encouraged to start by preparing their library for trimming and learning more about the Native AOT compatibility requirements.

Kestrel and HTTP.sys servers

There are several new features for Kestrel and HTTP.sys.

Support for named pipes in Kestrel

Named pipes is a popular technology for building inter-process communication (IPC) between Windows apps. You can now build an IPC server using .NET, Kestrel, and named pipes.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenNamedPipe("MyPipeName");
});
```

For more information about this feature and how to use .NET and gRPC to create an IPC server and client, see Inter-process communication with gRPC.

Performance improvements to named pipes transport

We've improved named pipe connection performance. Kestrel's named pipe transport now accepts connections in parallel, and reuses NamedPipeServerStream instances.

Time to create 100,000 connections:

- Before : 5.916 seconds
- After : 2.374 seconds

HTTP/2 over TLS (HTTPS) support on macOS in Kestrel

.NET 8 adds support for Application-Layer Protocol Negotiation (ALPN) to macOS. ALPN is a TLS feature used to negotiate which HTTP protocol a connection will use. For example, ALPN allows browsers and other HTTP clients to request an HTTP/2 connection. This feature is especially useful for gRPC apps, which require HTTP/2. For more information, see Use HTTP/2 with the ASP.NET Core Kestrel web server.

Certificate file watching in Kestrel

TLS certificates configured by path are now monitored for changes when `reloadOnChange` is passed to KestrelServerOptions.Configure(). A change to the certificate file is treated the same way as a change to the configured path (that is, endpoints are reloaded).

Note that file deletions are specifically not tracked since they arise transiently and would crash the server if non-transient.

Warning when specified HTTP protocols won't be used

If TLS is disabled and HTTP/1.x is available, HTTP/2 and HTTP/3 will be disabled, even if they've been specified. This can cause some nasty surprises, so we've added warning output to let you know when it happens.

`HTTP_PORTS` and `HTTPS_PORTS` config keys

Applications and containers are often only given a port to listen on, like 80, without additional constraints like host or path. `HTTP_PORTS` and `HTTPS_PORTS` are new config keys that allow specifying the listening ports for the Kestrel and HTTP.sys servers. These can be defined with the `DOTNET_` or `ASPNETCORE_` environment variable prefixes, or specified directly through any other config input like appsettings.json. Each is a semicolon delimited list of port values. For example:

cli

```
ASPNETCORE_HTTP_PORTS=80;8080
ASPNETCORE_HTTPS_PORTS=443;8081
```

This is shorthand for the following, which specifies the scheme (HTTP or HTTPS) and any host or IP:

```
cli
```

```
ASPNETCORE_URLS=http://*:80/;http://*:8080/;https://*:443/;https://*:8081/
```

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server and HTTP.sys web server implementation in ASP.NET Core.

SNI host name in ITlsHandshakeFeature

The Server Name Indication (SNI) host name is now exposed in the HostName property of the ITlsHandshakeFeature interface.

SNI is part of the TLS handshake process. It allows clients to specify the host name they're attempting to connect to when the server hosts multiple virtual hosts or domains. To present the correct security certificate during the handshake process, the server needs to know the host name selected for each request.

Normally the host name is only handled within the TLS stack and is used to select the matching certificate. But by exposing it, other components in an app can use that information for purposes such as diagnostics, rate limiting, routing, and billing.

Exposing the host name is useful for large-scale services managing thousands of SNI bindings. This feature can significantly improve debugging efficiency during customer escalations. The increased transparency allows for faster problem resolution and enhanced service reliability.

For more information, see ITlsHandshakeFeature.HostName.

IHttpSysRequestTimingFeature

IHttpSysRequestTimingFeature provides detailed timing information for requests when using the HTTP.sys server and In-process hosting with IIS:

- Timestamps are obtained using QueryPerformanceCounter.
- The timestamp frequency can be obtained via QueryPerformanceFrequency.
- The index of the timing can be cast to HttpSysRequestTimingType to know what the timing represents.
- The value might be 0 if the timing isn't available for the current request.

IHttpSysRequestTimingFeature.TryGetTimestamp retrieves the timestamp for the provided timing type:

C#

```
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys();

var app = builder.Build();

app.Use((context, next) =>
{
    var feature =
context.Features.GetRequiredFeature<IHttpSysRequestTimingFeature>();

    var loggerFactory =
context.RequestServices.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    var timingType = HttpSysRequestTimingType.RequestRoutingEnd;

    if (feature.TryGetTimestamp(timingType, out var timestamp))
    {
        logger.LogInformation("Timestamp {timingType}: {timestamp}",
                                            timingType, timestamp);
    }
    else
    {
        logger.LogInformation("Timestamp {timingType}: not available for the
"
                                             + "current request",
timingType);
    }

    return next(context);
});

app.MapGet("/", () => Results.Ok());

app.Run();
```

For more information, see Get detailed timing information with IHttpSysRequestTimingFeature and Timing information and In-process hosting with IIS.

HTTP.sys: opt-in support for kernel-mode response buffering

In some scenarios, high volumes of small writes with high latency can cause significant performance impact to `HTTP.sys`. This impact is due to the lack of a Pipe buffer in the `HTTP.sys` implementation. To improve performance in these scenarios, support for response buffering has been added to `HTTP.sys`. Enable buffering by setting HttpSysOptions.EnableKernelResponseBuffering to `true`.

Response buffering should be enabled by an app that does synchronous I/O, or asynchronous I/O with no more than one outstanding write at a time. In these scenarios, response buffering can significantly improve throughput over high-latency connections.

Apps that use asynchronous I/O and that can have more than one write outstanding at a time should ***not*** use this flag. Enabling this flag can result in higher CPU and memory usage by HTTP.Sys.

Authentication and authorization

ASP.NET Core 8 adds new features to authentication and authorization.

Identity API endpoints

MapIdentityApi<TUser> is a new extension method that adds two API endpoints (`/register` and `/login`). The main goal of the `MapIdentityApi` is to make it easy for developers to use ASP.NET Core Identity for authentication in JavaScript-based single page apps (SPA) or Blazor apps. Instead of using the default UI provided by ASP.NET Core Identity, which is based on Razor Pages, MapIdentityApi adds JSON API endpoints that are more suitable for SPA apps and nonbrowser apps. For more information, see Identity API endpoints.

IAuthorizationRequirementData

Prior to ASP.NET Core 8, adding a parameterized authorization policy to an endpoint required implementing an:

- `AuthorizeAttribute` for each policy.
- `AuthorizationPolicyProvider` to process a custom policy from a string-based contract.
- `AuthorizationRequirement` for the policy.
- `AuthorizationHandler` for each requirement.

For example, consider the following sample written for ASP.NET Core 7.0:

```
```

C#

```
using AuthRequirementsData.Authorization;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
builder.Services.AddControllers();
builder.Services.AddSingleton<IAuthorizationPolicyProvider,
MinimumAgePolicyProvider>();
builder.Services.AddSingleton<IAuthorizationHandler,
MinimumAgeAuthorizationHandler>();

var app = builder.Build();

app.MapControllers();

app.Run();
```

C#

```
using Microsoft.AspNetCore.Mvc;

namespace AuthRequirementsData.Controllers;

[ApiController]
[Route("api/[controller]")]
public class GreetingsController : Controller
{
    [MinimumAgeAuthorize(16)]
    [HttpGet("hello")]
    public string Hello() => $"Hello {(HttpContext.User.Identity?.Name ??
"world")}!";
}
```

C#

```
using Microsoft.AspNetCore.Authorization;
using System.Globalization;
using System.Security.Claims;

namespace AuthRequirementsData.Authorization;

class MinimumAgeAuthorizationHandler :
AuthorizationHandler<MinimumAgeRequirement>
{
    private readonly ILogger<MinimumAgeAuthorizationHandler> _logger;

    public
MinimumAgeAuthorizationHandler(ILogger<MinimumAgeAuthorizationHandler>
```

```
logger)
    {
        _logger = logger;
    }

    // Check whether a given MinimumAgeRequirement is satisfied or not for a
particular
    // context.
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
                                                MinimumAgeRequirement
requirement)
    {
        // Log as a warning so that it's very clear in sample output which
authorization
        // policies(and requirements/handlers) are in use.
        _logger.LogWarning("Evaluating authorization requirement for age >=
{age}",

requirement.Age);

        // Check the user's age
        var dateOfBirthClaim = context.User.FindFirst(c => c.Type ==

ClaimTypes.DateOfBirth);
        if (dateOfBirthClaim != null)
        {
            // If the user has a date of birth claim, check their age
            var dateOfBirth = Convert.ToDateTime(dateOfBirthClaim.Value,
CultureInfo.InvariantCulture);
            var age = DateTime.Now.Year - dateOfBirth.Year;
            if (dateOfBirth > DateTime.Now.AddYears(-age))
            {
                // Adjust age if the user hasn't had a birthday yet this
year.
                age--;
            }

            // If the user meets the age criterion, mark the authorization
requirement
            // succeeded.
            if (age >= requirement.Age)
            {
                _logger.LogInformation("Minimum age authorization
requirement {age} satisfied",
                                       requirement.Age);
                context.Succeed(requirement);
            }
            else
            {
                _logger.LogInformation("Current user's DateOfBirth claim
({dateOfBirth})" +
                    " does not satisfy the minimum age authorization
requirement {age}",
                    dateOfBirthClaim.Value,
```

```
                    requirement.Age);
            }
        }
        else
        {
            _logger.LogInformation("No DateOfBirth claim present");
        }

        return Task.CompletedTask;
    }
}
```

The complete sample is here in the AspNetCore.Docs.Samples repository.

ASP.NET Core 8 introduces the IAuthorizationRequirementData interface. The `IAuthorizationRequirementData` interface allows the attribute definition to specify the requirements associated with the authorization policy. Using `IAuthorizationRequirementData`, the preceding custom authorization policy code can be written with fewer lines of code. The updated `Program.cs` file:

diff

```
  using AuthRequirementsData.Authorization;
  using Microsoft.AspNetCore.Authorization;

  var builder = WebApplication.CreateBuilder();

  builder.Services.AddAuthentication().AddJwtBearer();
  builder.Services.AddAuthorization();
  builder.Services.AddControllers();
- builder.Services.AddSingleton<IAuthorizationPolicyProvider,
MinimumAgePolicyProvider>();
  builder.Services.AddSingleton<IAuthorizationHandler,
MinimumAgeAuthorizationHandler>();

  var app = builder.Build();

  app.MapControllers();

  app.Run();
```

The updated `MinimumAgeAuthorizationHandler`:

diff

```
using Microsoft.AspNetCore.Authorization;
using System.Globalization;
using System.Security.Claims;

namespace AuthRequirementsData.Authorization;
```

```
- class MinimumAgeAuthorizationHandler :
AuthorizationHandler<MinimumAgeRequirement>
+ class MinimumAgeAuthorizationHandler :
AuthorizationHandler<MinimumAgeAuthorizeAttribute>
{
    private readonly ILogger<MinimumAgeAuthorizationHandler> _logger;

    public
MinimumAgeAuthorizationHandler(ILogger<MinimumAgeAuthorizationHandler>
logger)
    {
        _logger = logger;
    }

    // Check whether a given MinimumAgeRequirement is satisfied or not for a
particular
    // context
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
-                                           MinimumAgeRequirement
requirement)
+                                           MinimumAgeAuthorizeAttribute
requirement)
    {
        // Remaining code omitted for brevity.
```

The complete updated sample can be found here.

See Custom authorization policies with IAuthorizationRequirementData for a detailed examination of the new sample.

Securing Swagger UI endpoints

Swagger UI endpoints can now be secured in production environments by calling MapSwagger().RequireAuthorization. For more information, see Securing Swagger UI endpoints

Miscellaneous

The following sections describe miscellaneous new features in ASP.NET Core 8.

Keyed services support in Dependency Injection

Keyed services refers to a mechanism for registering and retrieving Dependency Injection (DI) services using keys. A service is associated with a key by calling AddKeyedSingleton (or `AddKeyedScoped` or `AddKeyedTransient`) to register it. Access a registered service by

specifying the key with the [FromKeyedServices] attribute. The following code shows how to use keyed services:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.SignalR;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
builder.Services.AddControllers();

var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));
app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>

smallCache.Get("date"));

app.MapControllers();

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

[ApiController]
[Route("/cache")]
public class CustomServicesApiController : Controller
{
    [HttpGet("big-cache")]
    public ActionResult<object> GetOk([FromKeyedServices("big")] ICache
cache)
    {
        return cache.Get("data-mvc");
    }
}

public class MyHub : Hub
{
```

```
    public void Method([FromKeyedServices("small")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }
}
```

Visual Studio project templates for SPA apps with ASP.NET Core backend

Visual Studio project templates are now the recommended way to create single-page apps (SPAs) that have an ASP.NET Core backend. Templates are provided that create apps based on the JavaScript frameworks Angular, React, and Vue. These templates:

- Create a Visual Studio solution with a frontend project and a backend project.
- Use the Visual Studio project type for JavaScript and TypeScript (*.esproj*) for the frontend.
- Use an ASP.NET Core project for the backend.

For more information about the Visual Studio templates and how to access the legacy templates, see Overview of Single Page Apps (SPAs) in ASP.NET Core

Support for generic attributes

Attributes that previously required a Type parameter are now available in cleaner generic variants. This is made possible by support for generic attributes in C# 11. For example, the syntax for annotating the response type of an action can be modified as follows:

diff

```
[ApiController]
[Route("api/[controller]")]
public class TodosController : Controller
{
  [HttpGet("/")]
- [ProducesResponseType(typeof(Todo), StatusCodes.Status200OK)]
+ [ProducesResponseType<Todo>(StatusCodes.Status200OK)]
  public Todo Get() => new Todo(1, "Write a sample", DateTime.Now, false);
}
```

Generic variants are supported for the following attributes:

- `[ProducesResponseType<T>]`

- `[Produces<T>]`
- `[MiddlewareFilter<T>]`
- `[ModelBinder<T>]`
- `[ModelMetadataType<T>]`
- `[ServiceFilter<T>]`
- `[TypeFilter<T>]`

Code analysis in ASP.NET Core apps

The new analyzers shown in the following table are available in ASP.NET Core 8.0.

Expand table

Diagnostic ID	Breaking or nonbreaking	Description
ASP0016	Nonbreaking	Don't return a value from RequestDelegate
ASP0019	Nonbreaking	Suggest using IHeaderDictionary.Append or the indexer
ASP0020	Nonbreaking	Complex types referenced by route parameters must be parsable
ASP0021	Nonbreaking	The return type of the BindAsync method must be `ValueTask<T>`
ASP0022	Nonbreaking	Route conflict detected between route handlers
ASP0023	Nonbreaking	MVC: Route conflict detected between route handlers
ASP0024	Nonbreaking	Route handler has multiple parameters with the `[FromBody]` attribute
ASP0025	Nonbreaking	Use AddAuthorizationBuilder

Route tooling

ASP.NET Core is built on routing. Minimal APIs, Web APIs, Razor Pages, and Blazor all use routes to customize how HTTP requests map to code.

In .NET 8 we've invested in a suite of new features to make routing easier to learn and use. These new features include:

- Route syntax highlighting
- Autocomplete of parameter and route names

- Autocomplete of route constraints
- Route analyzers and fixers
  - Route syntax analyzer
  - Mismatched parameter optionality analyzer and fixer
  - Ambiguous Minimal API and Web API route analyzer
- Support for Minimal APIs, Web APIs, and Blazor

For more information, see Route tooling in .NET 8.

ASP.NET Core metrics

Metrics are measurements reported over time and are most often used to monitor the health of an app and to generate alerts. For example, a counter that reports failed HTTP requests could be displayed in dashboards or generate alerts when failures pass a threshold.

This preview adds new metrics throughout ASP.NET Core using System.Diagnostics.Metrics. `Metrics` is a modern API for reporting and collecting information about apps.

Metrics offers many improvements compared to existing event counters:

- New kinds of measurements with counters, gauges and histograms.
- Powerful reporting with multi-dimensional values.
- Integration into the wider cloud native ecosystem by aligning with OpenTelemetry standards.

Metrics have been added for ASP.NET Core hosting, Kestrel, and SignalR. For more information, see System.Diagnostics.Metrics.

IExceptionHandler

IExceptionHandler is a new interface that gives the developer a callback for handling known exceptions in a central location.

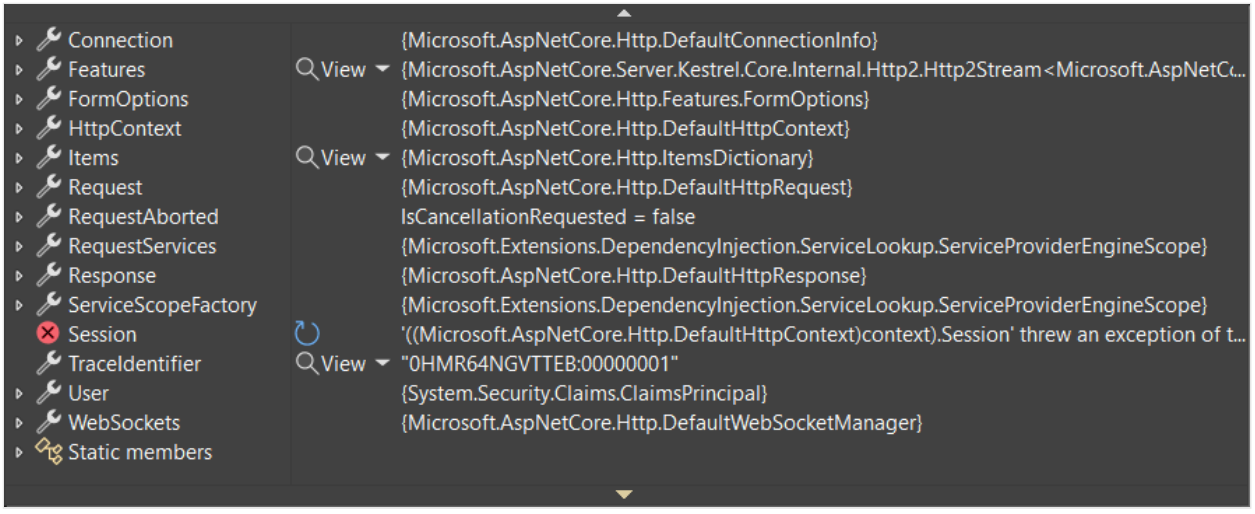
`IExceptionHandler` implementations are registered by calling IServiceCollection.AddExceptionHandler<T>. Multiple implementations can be added, and they're called in the order registered. If an exception handler handles a request, it can return `true` to stop processing. If an exception isn't handled by any exception handler, then control falls back to the default behavior and options from the middleware.

For more information, see IExceptionHandler.

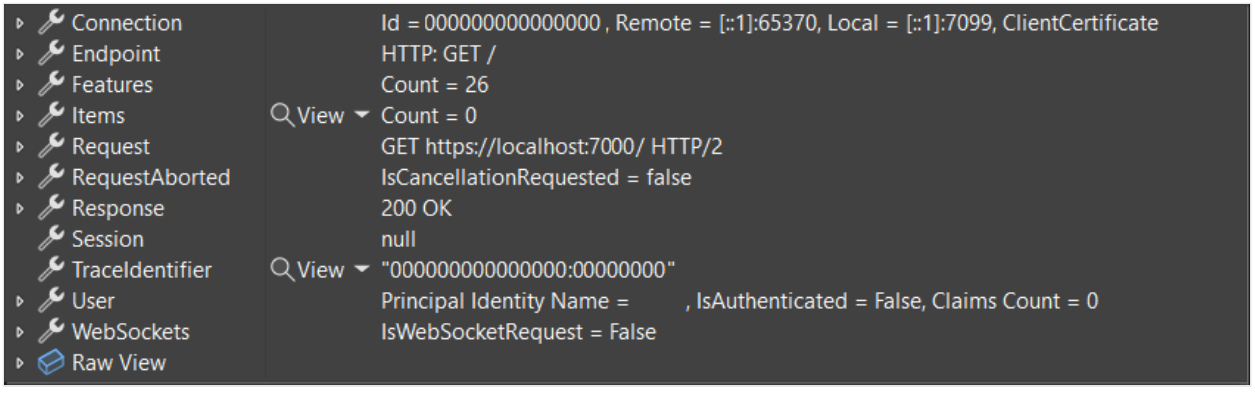
Improved debugging experience

Debug customization attributes have been added to types like `HttpContext`, `HttpRequest`, `HttpResponse`, `ClaimsPrincipal`, and `WebApplication`. The enhanced debugger displays for these types make finding important information easier in an IDE's debugger. The following screenshots show the difference that these attributes make in the debugger's display of `HttpContext`.

.NET 7:

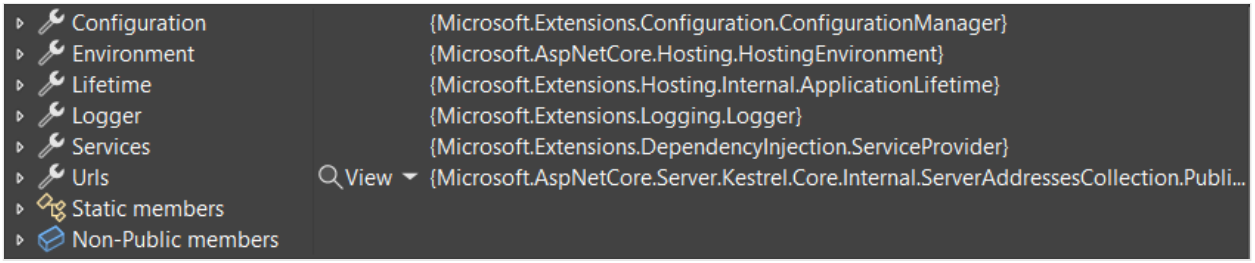


.NET 8:

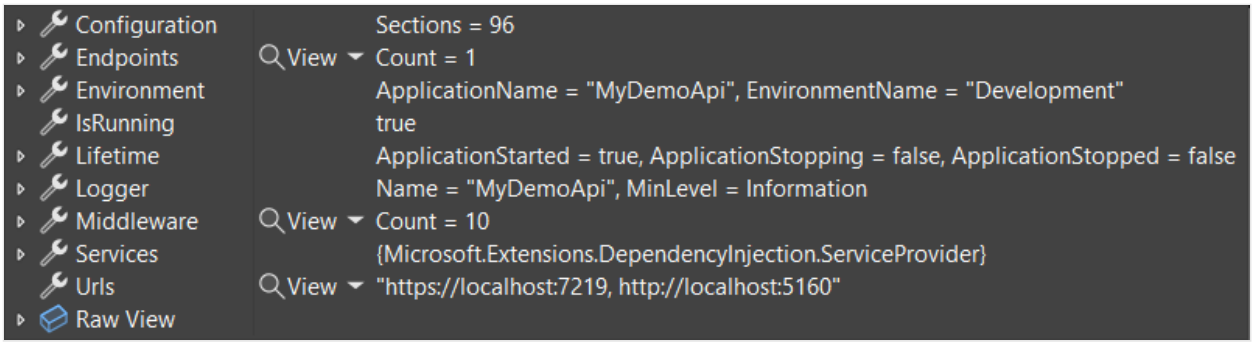


The debugger display for `WebApplication` highlights important information such as configured endpoints, middleware, and `IConfiguration` values.

.NET 7:



.NET 8:



For more information about debugging improvements in .NET 8, see:

- Debugging Enhancements in .NET 8
- GitHub issue dotnet/aspnetcore 48205

`IPNetwork.Parse` and `TryParse`

The new Parse and TryParse methods on IPNetwork add support for creating an `IPNetwork` by using an input string in CIDR notation or "slash notation".

Here are IPv4 examples:

C#

```
// Using Parse
var network = IPNetwork.Parse("192.168.0.1/32");
```

C#

```
// Using TryParse
bool success = IPNetwork.TryParse("192.168.0.1/32", out var network);
```

C#

```
// Constructor equivalent
var network = new IPNetwork(IPAddress.Parse("192.168.0.1"), 32);
```

And here are examples for IPv6:

C#

```
// Using Parse
var network = IPNetwork.Parse("2001:db8:3c4d::1/128");
```

C#

```
// Using TryParse
bool success = IPNetwork.TryParse("2001:db8:3c4d::1/128", out var network);
```

C#

```
// Constructor equivalent
var network = new IPNetwork(IPAddress.Parse("2001:db8:3c4d::1"), 128);
```

Redis-based output caching

ASP.NET Core 8 adds support for using Redis as a distributed cache for output caching. Output caching is a feature that enables an app to cache the output of a minimal API endpoint, controller action, or Razor Page. For more information, see Output caching.

Short-circuit middleware after routing

When routing matches an endpoint, it typically lets the rest of the middleware pipeline run before invoking the endpoint logic. Services can reduce resource usage by filtering out known requests early in the pipeline. Use the ShortCircuit extension method to cause routing to invoke the endpoint logic immediately and then end the request. For example, a given route might not need to go through authentication or CORS middleware. The following example short-circuits requests that match the `/short-circuit` route:

C#

```
app.MapGet("/short-circuit", () => "Short circuiting!").ShortCircuit();
```

Use the MapShortCircuit method to set up short-circuiting for multiple routes at once, by passing to it a params array of URL prefixes. For example, browsers and bots often probe servers for well known paths like `robots.txt` and `favicon.ico`. If the app doesn't have those files, one line of code can configure both routes:

C#

```
app.MapShortCircuit(404, "robots.txt", "favicon.ico");
```

For more information, see Short-circuit middleware after routing.

HTTP logging middleware extensibility

The HTTP logging middleware has several new capabilities:

- HttpLoggingFields.Duration: When enabled, the middleware emits a new log at the end of the request and response that measures the total time taken for processing. This new field has been added to the HttpLoggingFields.All set.
- HttpLoggingOptions.CombineLogs: When enabled, the middleware consolidates all of its enabled logs for a request and response into one log at the end. A single log message includes the request, request body, response, response body, and duration.
- IHttpLoggingInterceptor: A new interface for a service that can be implemented and registered (using AddHttpLoggingInterceptor) to receive per-request and per-response callbacks for customizing what details get logged. Any endpoint-specific log settings are applied first and can then be overridden in these callbacks. An implementation can:
  - Inspect a request and response.
  - Enable or disable any HttpLoggingFields.
  - Adjust how much of the request or response body is logged.
  - Add custom fields to the logs.

For more information, see HTTP logging in .NET Core and ASP.NET Core.

New APIs in ProblemDetails to support more resilient integrations

In .NET 7, the ProblemDetails service was introduced to improve the experience for generating error responses that comply with the ProblemDetails specification. In .NET 8, a new API was added to make it easier to implement fallback behavior if IProblemDetailsService isn't able to generate ProblemDetails. The following example illustrates use of the new TryWriteAsync API:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler(exceptionHandlerApp =>
{
    exceptionHandlerApp.Run(async httpContext =>
    {
        var pds =
```

```
httpContext.RequestServices.GetService<IProblemDetailsService>();
        if (pds == null
            || !await pds.TryWriteAsync(new() { HttpContext = httpContext
}))
        {
            // Fallback behavior
            await httpContext.Response.WriteAsync("Fallback: An error
occurred.");
        }
    });
});

app.MapGet("/exception", () =>
{
    throw new InvalidOperationException("Sample Exception");
});

app.MapGet("/", () => "Test by calling /exception");

app.Run();
```

For more information, see IProblemDetailsService fallback

Additional resources

- Announcing ASP.NET Core in .NET 8 (blog post)
- ASP.NET Core announcements and breaking changes (aspnet/Announcements GitHub repository)
- .NET announcements and breaking changes (dotnet/Announcements GitHub repository)

What's new in ASP.NET Core 7.0

Article • 09/27/2024

This article highlights the most significant changes in ASP.NET Core 7.0 with links to relevant documentation.

Rate limiting middleware in ASP.NET Core

The `Microsoft.AspNetCore.RateLimiting` middleware provides rate limiting middleware. Apps configure rate limiting policies and then attach the policies to endpoints. For more information, see Rate limiting middleware in ASP.NET Core.

Authentication uses single scheme as DefaultScheme

As part of the work to simplify authentication, when there's only a single authentication scheme registered, it's automatically used as the DefaultScheme and doesn't need to be specified. For more information, see DefaultScheme.

MVC and Razor pages

Support for nullable models in MVC views and Razor Pages

Nullable page or view models are supported to improve the experience when using null state checking with ASP.NET Core apps:

```
C#

@model Product?
```

Bind with `IParsable<T>.TryParse` in MVC and API Controllers

The IParsable<TSelf>.TryParse API supports binding controller action parameter values. For more information, see Bind with IParsable<T>.TryParse.

Customize the cookie consent value

In ASP.NET Core versions earlier than 7, the cookie consent validation uses the cookie value `yes` to indicate consent. Now you can specify the value that represents consent. For example, you could use `true` instead of `yes`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.CheckConsentNeeded = context => true;
    options.MinimumSameSitePolicy = SameSiteMode.None;
    options.ConsentCookieValue = "true";
});

var app = builder.Build();
```

For more information, see Customize the cookie consent value.

API controllers

Parameter binding with DI in API controllers

Parameter binding for API controller actions binds parameters through dependency injection when the type is configured as a service. This means it's no longer required to explicitly apply the [FromServices] attribute to a parameter. In the following code, both actions return the time:

C#

```
[Route("[controller]")]
[ApiController]
public class MyController : ControllerBase
{
    public ActionResult GetWithAttribute([FromServices] IDateTime dateTime)
                                                        => Ok(dateTime.Now);

    [Route("noAttribute")]
    public ActionResult Get(IDateTime dateTime) => Ok(dateTime.Now);
}
```

In rare cases, automatic DI can break apps that have a type in DI that is also accepted in an API controllers action method. It's not common to have a type in DI and as an argument in an API controller action. To disable automatic binding of parameters, set DisableImplicitFromServicesParameters

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSingleton<IDateTime, SystemDateTime>();

builder.Services.Configure<ApiBehaviorOptions>(options =>
{
    options.DisableImplicitFromServicesParameters = true;
});

var app = builder.Build();

app.MapControllers();

app.Run();
```

In ASP.NET Core 7.0, types in DI are checked at app startup with IServiceProviderIsService to determine if an argument in an API controller action comes from DI or from the other sources.

The new mechanism to infer binding source of API Controller action parameters uses the following rules:

1. A previously specified BindingInfo.BindingSource is never overwritten.
2. A complex type parameter, registered in the DI container, is assigned BindingSource.Services.
3. A complex type parameter, not registered in the DI container, is assigned BindingSource.Body.
4. A parameter with a name that appears as a route value in ***any*** route template is assigned BindingSource.Path.
5. All other parameters are BindingSource.Query.

JSON property names in validation errors

By default, when a validation error occurs, model validation produces a ModelStateDictionary with the property name as the error key. Some apps, such as single page apps, benefit from using JSON property names for validation errors

generated from Web APIs. The following code configures validation to use the SystemTextJsonValidationMetadataProvider to use JSON property names:

C#

```
using Microsoft.AspNetCore.Mvc.ModelBinding.Metadata;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
SystemTextJsonValidationMetadataProvider());
});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The following code configures validation to use the NewtonsoftJsonValidationMetadataProvider to use JSON property name when using Json.NET:

C#

```
using Microsoft.AspNetCore.Mvc.NewtonsoftJson;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
NewtonsoftJsonValidationMetadataProvider());
}).AddNewtonsoftJson();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

For more information, see Use JSON property names in validation errors

Minimal APIs

Filters in Minimal API apps

Minimal API filters allow developers to implement business logic that supports:

- Running code before and after the route handler.
- Inspecting and modifying parameters provided during a route handler invocation.
- Intercepting the response behavior of a route handler.

Filters can be helpful in the following scenarios:

- Validating the request parameters and body that are sent to an endpoint.
- Logging information about the request and response.
- Validating that a request is targeting a supported API version.

For more information, see Filters in Minimal API apps

Bind arrays and string values from headers and query strings

In ASP.NET 7, binding query strings to an array of primitive types, string arrays, and StringValues is supported:

C#

```
// Bind query string values to a primitive type array.
// GET  /tags?q=1&q=2&q=3
app.MapGet("/tags", (int[] q) =>
                      $"tag1: {q[0]} , tag2: {q[1]}, tag3: {q[2]}");

// Bind to a string array.
// GET /tags2?names=john&names=jack&names=jane
app.MapGet("/tags2", (string[] names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");

// Bind to StringValues.
// GET /tags3?names=john&names=jack&names=jane
app.MapGet("/tags3", (StringValues names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");
```

Binding query strings or header values to an array of complex types is supported when the type has `TryParse` implemented. For more information, see Bind arrays and string

values from headers and query strings.

For more information, see Add endpoint summary or description.

Bind the request body as a `Stream` or `PipeReader`

The request body can bind as a Stream or PipeReader to efficiently support scenarios where the user has to process data and:

- Store the data to blob storage or enqueue the data to a queue provider.
- Process the stored data with a worker process or cloud function.

For example, the data might be enqueued to Azure Queue storage or stored in Azure Blob storage.

For more information, see Bind the request body as a Stream or PipeReader

New Results.Stream overloads

We introduced new Results.Stream overloads to accommodate scenarios that need access to the underlying HTTP response stream without buffering. These overloads also improve cases where an API streams data to the HTTP response stream, like from Azure Blob Storage. The following example uses ImageSharp to return a reduced size of the specified image:

C#

```
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats.Jpeg;
using SixLabors.ImageSharp.Processing;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/process-image/{strImage}", (string strImage, HttpContext http,
CancellationToken token) =>
{
    http.Response.Headers.CacheControl = $"public,max-age=
{TimeSpan.FromHours(24).TotalSeconds}";
    return Results.Stream(stream => ResizeImageAsync(strImage, stream,
token), "image/jpeg");
});

async Task ResizeImageAsync(string strImage, Stream stream,
CancellationToken token)
{
    var strPath = $"wwwroot/img/{strImage}";
```

```
    using var image = await Image.LoadAsync(strPath, token);
    int width = image.Width / 2;
    int height = image.Height / 2;
    image.Mutate(x =>x.Resize(width, height));
    await image.SaveAsync(stream, JpegFormat.Instance, cancellationToken:
token);
}
```

For more information, see Stream examples

Typed results for minimal APIs

In .NET 6, the IResult interface was introduced to represent values returned from minimal APIs that don't utilize the implicit support for JSON serializing the returned object to the HTTP response. The static Results class is used to create varying `IResult` objects that represent different types of responses. For example, setting the response status code or redirecting to another URL. The `IResult` implementing framework types returned from these methods were internal however, making it difficult to verify the specific `IResult` type being returned from methods in a unit test.

In .NET 7 the types implementing `IResult` are public, allowing for type assertions when testing. For example:

C#

```
[TestClass()]
public class WeatherApiTests
{
    [TestMethod()]
    public void MapWeatherApiTest()
    {
        var result = WeatherApi.GetAllWeathers();
        Assert.IsInstanceOfType(result, typeof(Ok<WeatherForecast[]>));
    }
}
```

Improved unit testability for minimal route handlers

IResult implementation types are now publicly available in the Microsoft.AspNetCore.Http.HttpResults namespace. The `IResult` implementation types can be used to unit test minimal route handlers when using named methods instead of lambdas.

The following code uses the Ok<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsTodoFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title",
        Description = "Test description",
        IsDone = false
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var okResult = (Ok<Todo>)result.Result;

    Assert.NotNull(okResult.Value);
    Assert.Equal(1, okResult.Value.Id);
}
```

For more information, see IResult implementation types.

New HttpResult interfaces

The following interfaces in the Microsoft.AspNetCore.Http namespace provide a way to detect the `IResult` type at runtime, which is a common pattern in filter implementations:

- IContentTypeHttpResult
- IFileHttpResult
- INestedHttpResult
- IStatusCodeHttpResult
- IValueHttpResult
- IValueHttpResult<TValue>

For more information, see IHttpResult interfaces.

OpenAPI improvements for minimal APIs

`Microsoft.AspNetCore.OpenApi` NuGet package

The Microsoft.AspNetCore.OpenApi package allows interactions with OpenAPI specifications for endpoints. The package acts as a link between the OpenAPI models that are defined in the `Microsoft.AspNetCore.OpenApi` package and the endpoints that are defined in Minimal APIs. The package provides an API that examines an endpoint's parameters, responses, and metadata to construct an OpenAPI annotation type that is used to describe an endpoint.

C#

```
app.MapPost("/todoitems/{id}", async (int id, Todo todo, TodoDb db) =>
{
    todo.Id = id;
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
})
.WithOpenApi();
```

Call `WithOpenApi` with parameters

The WithOpenApi method accepts a function that can be used to modify the OpenAPI annotation. For example, in the following code, a description is added to the first parameter of the endpoint:

C#

```
app.MapPost("/todo2/{id}", async (int id, Todo todo, TodoDb db) =>
{
    todo.Id = id;
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
})
.WithOpenApi(generatedOperation =>
{
    var parameter = generatedOperation.Parameters[0];
    parameter.Description = "The ID associated with the created Todo";
    return generatedOperation;
});
```

Provide endpoint descriptions and summaries

Minimal APIs now support annotating operations with descriptions and summaries for OpenAPI spec generation. You can call extension methods WithDescription and WithSummary or use attributes [EndpointDescription] and [EndpointSummary]).

For more information, see OpenAPI in minimal API apps

File uploads using IFormFile and IFormFileCollection

Minimal APIs now support file upload with `IFormFile` and `IFormFileCollection`. The following code uses IFormFile and IFormFileCollection to upload file:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapPost("/upload", async (IFormFile file) =>
{
    var tempFile = Path.GetTempFileName();
    app.Logger.LogInformation(tempFile);
    using var stream = File.OpenWrite(tempFile);
    await file.CopyToAsync(stream);
});

app.MapPost("/upload_many", async (IFormFileCollection myFiles) =>
{
    foreach (var file in myFiles)
    {
        var tempFile = Path.GetTempFileName();
        app.Logger.LogInformation(tempFile);
        using var stream = File.OpenWrite(tempFile);
        await file.CopyToAsync(stream);
    }
});

app.Run();
```

Authenticated file upload requests are supported using an Authorization header, a client certificate, or a cookie header.

There is no built-in support for antiforgery. However, it can be implemented using the IAntiforgery service.

`[AsParameters]` attribute enables parameter binding for argument lists

The [AsParameters] attribute enables parameter binding for argument lists. For more information, see Parameter binding for argument lists with [AsParameters].

Minimal APIs and API controllers

New problem details service

The problem details service implements the IProblemDetailsService interface, which supports creating Problem Details for HTTP APIs.

For more information, see Problem details service.

Route groups

The MapGroup extension method helps organize groups of endpoints with a common prefix. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata which add endpoint metadata.

For example, the following code creates two similar groups of endpoints:

C#

```
app.MapGroup("/public/todos")
    .MapTodosApi()
    .WithTags("Public");

app.MapGroup("/private/todos")
    .MapTodosApi()
    .WithTags("Private")
    .AddEndpointFilterFactory(QueryPrivateTodos)
    .RequireAuthorization();


EndpointFilterDelegate QueryPrivateTodos(EndpointFilterFactoryContext
factoryContext, EndpointFilterDelegate next)
{
    var dbContextIndex = -1;

    foreach (var argument in factoryContext.MethodInfo.GetParameters())
    {
        if (argument.ParameterType == typeof(TodoDb))
        {
            dbContextIndex = argument.Position;
            break;
        }
    }
```

```
    // Skip filter if the method doesn't have a TodoDb parameter.
    if (dbContextIndex < 0)
    {
        return next;
    }

    return async invocationContext =>
    {
        var dbContext = invocationContext.GetArgument<TodoDb>
(dbContextIndex);
        dbContext.IsPrivate = true;

        try
        {
            return await next(invocationContext);
        }
        finally
        {
            // This should only be relevant if you're pooling or otherwise
reusing the DbContext instance.
            dbContext.IsPrivate = false;
        }
    };
}
```

C#

```
public static RouteGroupBuilder MapTodosApi(this RouteGroupBuilder group)
{
    group.MapGet("/", GetAllTodos);
    group.MapGet("/{id}", GetTodo);
    group.MapPost("/", CreateTodo);
    group.MapPut("/{id}", UpdateTodo);
    group.MapDelete("/{id}", DeleteTodo);

    return group;
}
```

In this scenario, you can use a relative address for the `Location` header in the `201 Created` result:

C#

```
public static async Task<Created<Todo>> CreateTodo(Todo todo, TodoDb
database)
{
    await database.AddAsync(todo);
    await database.SaveChangesAsync();
```

```
    return TypedResults.Created($"{todo.Id}", todo);
}
```

The first group of endpoints will only match requests prefixed with `/public/todos` and are accessible without any authentication. The second group of endpoints will only match requests prefixed with `/private/todos` and require authentication.

The `QueryPrivateTodos` endpoint filter factory is a local function that modifies the route handler's `TodoDb` parameters to allow to access and store private todo data.

Route groups also support nested groups and complex prefix patterns with route parameters and constraints. In the following example, and route handler mapped to the `user` group can capture the `{org}` and `{group}` route parameters defined in the outer group prefixes.

The prefix can also be empty. This can be useful for adding endpoint metadata or filters to a group of endpoints without changing the route pattern.

C#

```
var all = app.MapGroup("").WithOpenApi();
var org = all.MapGroup("{org}");
var user = org.MapGroup("{user}");
user.MapGet("", (string org, string user) => $"{org}/{user}");
```

Adding filters or metadata to a group behaves the same way as adding them individually to each endpoint before adding any extra filters or metadata that may have been added to an inner group or specific endpoint.

C#

```
var outer = app.MapGroup("/outer");
var inner = outer.MapGroup("/inner");

inner.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/inner group filter");
    return next(context);
});

outer.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/outer group filter");
    return next(context);
});

inner.MapGet("/", () => "Hi!").AddEndpointFilter((context, next) =>
```

```
{
    app.Logger.LogInformation("MapGet filter");
    return next(context);
});
```

In the above example, the outer filter will log the incoming request before the inner filter even though it was added second. Because the filters were applied to different groups, the order they were added relative to each other does not matter. The order filters are added does matter if applied to the same group or specific endpoint.

A request to `/outer/inner/` will log the following:

.NET CLI

```
/outer group filter
/inner group filter
MapGet filter
```

gRPC

JSON transcoding

gRPC JSON transcoding is an extension for ASP.NET Core that creates RESTful JSON APIs for gRPC services. gRPC JSON transcoding allows:

- Apps to call gRPC services with familiar HTTP concepts.
- ASP.NET Core gRPC apps to support both gRPC and RESTful JSON APIs without replicating functionality.
- Experimental support for generating OpenAPI from transcoded RESTful APIs by integrating with Swashbuckle.

For more information, see gRPC JSON transcoding in ASP.NET Core gRPC apps and Use OpenAPI with gRPC JSON transcoding ASP.NET Core apps.

gRPC health checks in ASP.NET Core

The gRPC health checking protocol is a standard for reporting the health of gRPC server apps. An app exposes health checks as a gRPC service. They are typically used with an external monitoring service to check the status of an app.

gRPC ASP.NET Core has added built-in support for gRPC health checks with the Grpc.AspNetCore.HealthChecks package. Results from .NET health checks are

reported to callers.

For more information, see gRPC health checks in ASP.NET Core.

Improved call credentials support

Call credentials are the recommended way to configure a gRPC client to send an auth token to the server. gRPC clients support two new features to make call credentials easier to use:

- Support for call credentials with plaintext connections. Previously, a gRPC call only sent call credentials if the connection was secured with TLS. A new setting on `GrpcChannelOptions`, called `UnsafeUseInsecureChannelCallCredentials`, allows this behavior to be customized. There are security implications to not securing a connection with TLS.
- A new method called `AddCallCredentials` is available with the gRPC client factory. `AddCallCredentials` is a quick way to configure call credentials for a gRPC client and integrates well with dependency injection (DI).

The following code configures the gRPC client factory to send `Authorization` metadata:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
       o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials((context, metadata) =>
    {
       if (!string.IsNullOrEmpty(_token))
       {
          metadata.Add("Authorization", $"Bearer {_token}");
       }
       return Task.CompletedTask;
    });
```

For more information, see Configure a bearer token with the gRPC client factory.

SignalR

Client results

The server now supports requesting a result from a client. This requires the server to use `ISingleClientProxy.InvokeAsync` and the client to return a result from its `.On` handler. Strongly-typed hubs can also return values from interface methods.

For more information, see Client results

Dependency injection for SignalR hub methods

SignalR hub methods now support injecting services through dependency injection (DI).

Hub constructors can accept services from DI as parameters, which can be stored in properties on the class for use in a hub method. For more information, see Inject services into a hub

Blazor

Handle location changing events and navigation state

In .NET 7, Blazor supports location changing events and maintaining navigation state. This allows you to warn users about unsaved work or to perform related actions when the user performs a page navigation.

For more information, see the following sections of the *Routing and navigation* article:

- Navigation options
- Handle/prevent location changes

Empty Blazor project templates

Blazor has two new project templates for starting from a blank slate. The new **Blazor Server App Empty** and **Blazor WebAssembly App Empty** project templates are just like their non-empty counterparts but without example code. These empty templates only include a basic home page, and we've removed Bootstrap so that you can start with a different CSS framework.

For more information, see the following articles:

- Tooling for ASP.NET Core Blazor
- ASP.NET Core Blazor project structure

Blazor custom elements

The Microsoft.AspNetCore.Components.CustomElements package enables building standards based custom DOM elements using Blazor.

For more information, see ASP.NET Core Razor components.

Bind modifiers (`@bind:after`, `@bind:get`, `@bind:set`)

> **Important**
>
> The `@bind:after`/`@bind:get`/`@bind:set` features are receiving further updates at this time. To take advantage of the latest updates, confirm that you've installed the latest SDK.
>
> Using an event callback parameter (`[Parameter] public EventCallback<string> ValueChanged { get; set; }`) isn't supported. Instead, pass an Action-returning or Task-returning method to `@bind:set`/`@bind:after`.
>
> For more information, see the following resources:
>
> - Blazor @bind:after not working on .NET 7 RTM release (dotnet/aspnetcore #44957)
> - BindGetSetAfter701 sample app (javiercn/BindGetSetAfter701 GitHub repository)

In .NET 7, you can run asynchronous logic after a binding event has completed using the new `@bind:after` modifier. In the following example, the `PerformSearch` asynchronous method runs automatically after any changes to the search text are detected:

razor

```
<input @bind="searchText" @bind:after="PerformSearch" />

@code {
    private string searchText;

    private async Task PerformSearch()
    {
        ...
    }
}
```

In .NET 7, it's also easier to set up binding for component parameters. Components can support two-way data binding by defining a pair of parameters:

- `@bind:get`: Specifies the value to bind.
- `@bind:set`: Specifies a callback for when the value changes.

The `@bind:get` and `@bind:set` modifiers are always used together.

Examples:

razor

```
@* Elements *@

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind:get="text" @bind:set="(value) => { }" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<input type="text" @bind:get="text" @bind:set="SetAsync" />

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind:get="text" @bind:set="(value) => { }" />

<input type="text" @bind="text" @bind:after="AfterAsync" />

<input type="text" @bind:get="text" @bind:set="SetAsync" />

@* Components *@

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { }" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { }" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

@code {
    private string text = "";

    private void After(){}
    private void Set() {}
    private Task AfterAsync() { return Task.CompletedTask; }
```

```
    private Task SetAsync(string value) { return Task.CompletedTask; }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

Hot Reload improvements

In .NET 7, Hot Reload support includes the following:

- Components reset their parameters to their default values when a value is removed.
- Blazor WebAssembly:
  - Add new types.
  - Add nested classes.
  - Add static and instance methods to existing types.
  - Add static fields and methods to existing types.
  - Add static lambdas to existing methods.
  - Add lambdas that capture `this` to existing methods that already captured `this` previously.

Dynamic authentication requests with MSAL in Blazor WebAssembly

New in .NET 7, Blazor WebAssembly supports creating dynamic authentication requests at runtime with custom parameters to handle advanced authentication scenarios.

For more information, see the following articles:

- Secure ASP.NET Core Blazor WebAssembly
- ASP.NET Core Blazor WebAssembly additional security scenarios

Blazor WebAssembly debugging improvements

Blazor WebAssembly debugging has the following improvements:

- Support for the **Just My Code** setting to show or hide type members that aren't from user code.
- Support for inspecting multidimensional arrays.
- **Call Stack** now shows the correct name for asynchronous methods.
- Improved expression evaluation.
- Correct handling of the `new` keyword on derived members.

- Support for debugger-related attributes in `System.Diagnostics`.

`System.Security.Cryptography` support on WebAssembly

.NET 6 supported the SHA family of hashing algorithms when running on WebAssembly. .NET 7 enables more cryptographic algorithms by taking advantage of SubtleCrypto, when possible, and falling back to a .NET implementation when SubtleCrypto can't be used. The following algorithms are supported on WebAssembly in .NET 7:

- SHA1
- SHA256
- SHA384
- SHA512
- HMACSHA1
- HMACSHA256
- HMACSHA384
- HMACSHA512
- AES-CBC
- PBKDF2
- HKDF

For more information, see Developers targeting browser-wasm can use Web Crypto APIs (dotnet/runtime #40074).

Inject services into custom validation attributes

You can now inject services into custom validation attributes. Blazor sets up the `ValidationContext` so that it can be used as a service provider.

For more information, see ASP.NET Core Blazor forms validation.

`Input*` components outside of an `EditContext`/`EditForm`

The built-in input components are now supported outside of a form in Razor component markup.

For more information, see ASP.NET Core Blazor input components.

Project template changes

When .NET 6 was released last year, the HTML markup of the `_Host` page (`Pages/_Host.chstml`) was split between the `_Host` page and a new `_Layout` page (`Pages/_Layout.chstml`) in the .NET 6 Blazor Server project template.

In .NET 7, the HTML markup has been recombined with the `_Host` page in project templates.

Several additional changes were made to the Blazor project templates. It isn't feasible to list every change to the templates in the documentation. To migrate an app to .NET 7 in order to adopt all of the changes, see Migrate from ASP.NET Core 6.0 to 7.0.

Experimental `QuickGrid` component

The new `QuickGrid` component provides a convenient data grid component for most common requirements and as a reference architecture and performance baseline for anyone building Blazor data grid components.

For more information, see ASP.NET Core Blazor QuickGrid component.

Live demo: QuickGrid for Blazor sample app

Virtualization enhancements

Virtualization enhancements in .NET 7:

- The `Virtualize` component supports using the document itself as the scroll root, as an alternative to having some other element with `overflow-y: scroll` applied.
- If the `Virtualize` component is placed inside an element that requires a specific child tag name, `SpacerElement` allows you to obtain or set the virtualization spacer tag name.

For more information, see the following sections of the *Virtualization* article:

- Root-level virtualization
- Control the spacer element tag name

`MouseEventArgs` updates

`MovementX` and `MovementY` have been added to `MouseEventArgs`.

For more information, see ASP.NET Core Blazor event handling.

New Blazor loading page

The Blazor WebAssembly project template has a new loading UI that shows the progress of loading the app.

For more information, see ASP.NET Core Blazor startup.

Improved diagnostics for authentication in Blazor WebAssembly

To help diagnose authentication issues in Blazor WebAssembly apps, detailed logging is available.

For more information, see ASP.NET Core Blazor logging.

JavaScript interop on WebAssembly

JavaScript `[JSImport]`/`[JSExport]` interop API is a new low-level mechanism for using .NET in Blazor WebAssembly and JavaScript-based apps. With this new JavaScript interop capability, you can invoke .NET code from JavaScript using the .NET WebAssembly runtime and call into JavaScript functionality from .NET without any dependency on the Blazor UI component model.

For more information:

- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor: Pertains only to Blazor WebAssembly apps.
- JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly: Pertains only to JavaScript apps that don't depend on the Blazor UI component model.

Conditional registration of the authentication state provider

Prior to the release of .NET 7, `AuthenticationStateProvider` was registered in the service container with `AddScoped`. This made it difficult to debug apps, as it forced a specific order of service registrations when providing a custom implementation. Due to internal framework changes over time, it's no longer necessary to register `AuthenticationStateProvider` with `AddScoped`.

In developer code, make the following change to the authentication state provider service registration:

```diff
- builder.Services.AddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
+ builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
```

In the preceding example, `ExternalAuthStateProvider` is the developer's service implementation.

Improvements to the .NET WebAssembly build tools

New features in the `wasm-tools` workload for .NET 7 that help improve performance and handle exceptions:

- WebAssembly Single Instruction, Multiple Data (SIMD) support (only with AOT, not supported by Apple Safari)
- WebAssembly exception handling support

For more information, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Blazor Hybrid

External URLs

An option has been added that permits opening external webpages in the browser.

For more information, see ASP.NET Core Blazor Hybrid routing and navigation.

Security

New guidance is available for Blazor Hybrid security scenarios. For more information, see the following articles:

- ASP.NET Core Blazor Hybrid authentication and authorization
- ASP.NET Core Blazor Hybrid security considerations

Performance

Output caching middleware

Output caching is a new middleware that stores responses from a web app and serves them from a cache rather than computing them every time. Output caching differs from response caching in the following ways:

- The caching behavior is configurable on the server.
- Cache entries can be programmatically invalidated.
- Resource locking mitigates the risk of cache stampede and thundering herd.
- Cache revalidation means the server can return a `304 Not Modified` HTTP status code instead of a cached response body.
- The cache storage medium is extensible.

For more information, see Overview of caching and Output caching middleware.

HTTP/3 improvements

This release:

- Makes HTTP/3 fully supported by ASP.NET Core, it's no longer experimental.
- Improves Kestrel's support for HTTP/3. The two main areas of improvement are feature parity with HTTP/1.1 and HTTP/2, and performance.
- Provides full support for UseHttps(ListenOptions, X509Certificate2) with HTTP/3. Kestrel offers advanced options for configuring connection certificates, such as hooking into Server Name Indication (SNI).
- Adds support for HTTP/3 on HTTP.sys and IIS.

The following example shows how to use an SNI callback to resolve TLS options:

C#

```
using Microsoft.AspNetCore.Server.Kestrel.Core;
using Microsoft.AspNetCore.Server.Kestrel.Https;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(options =>
{
    options.ListenAnyIP(8080, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2AndHttp3;
        listenOptions.UseHttps(new TlsHandshakeCallbackOptions
        {
            OnConnection = context =>
            {
                var options = new SslServerAuthenticationOptions
                {
                    ServerCertificate =
```

```
MyResolveCertForHost(context.ClientHelloInfo.ServerName)
                };
                return new ValueTask<SslServerAuthenticationOptions>
(options);
            },
        });
    });
});
```

Significant work was done in .NET 7 to reduce HTTP/3 allocations. You can see some of those improvements in the following GitHub PR's:

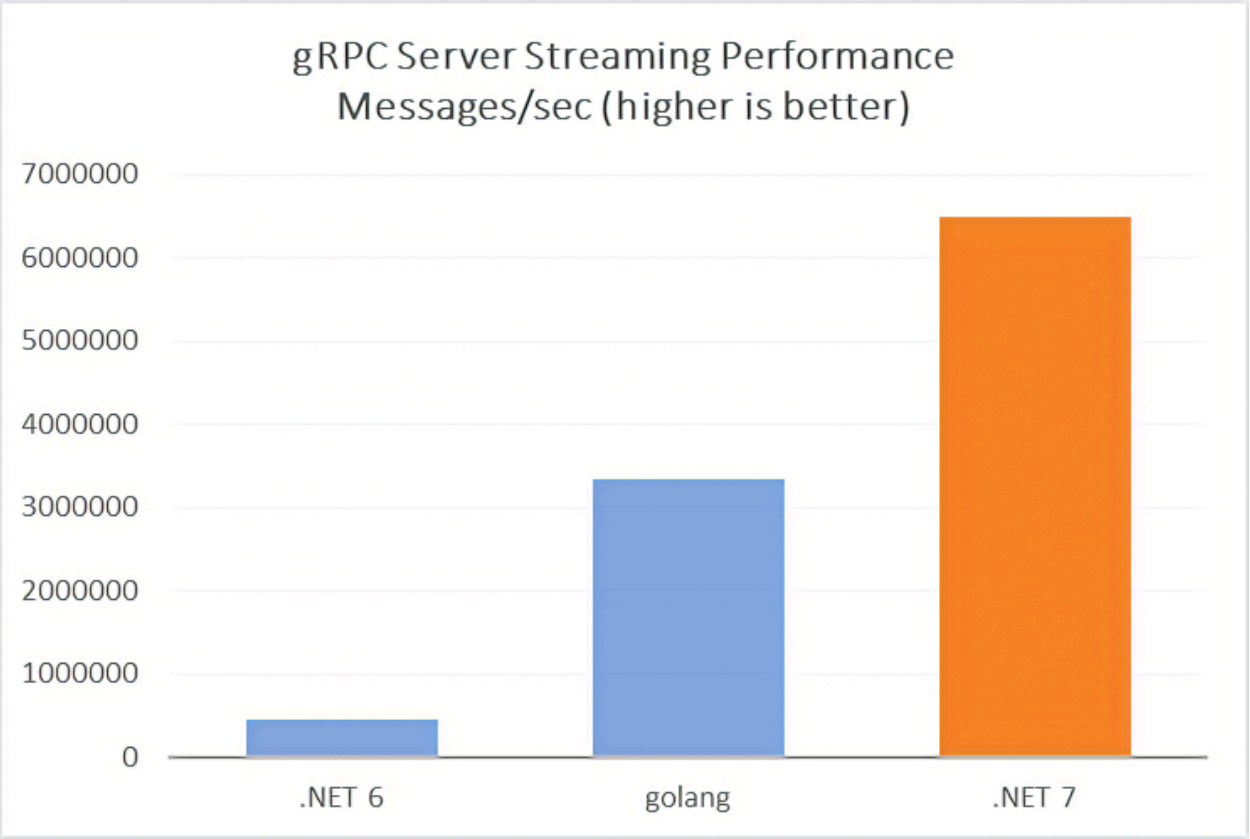
- HTTP/3: Avoid per-request cancellation token allocations
- HTTP/3: Avoid ConnectionAbortedException allocations
- HTTP/3: ValueTask pooling

HTTP/2 Performance improvements

.NET 7 introduces a significant re-architecture of how Kestrel processes HTTP/2 requests. ASP.NET Core apps with busy HTTP/2 connections will experience reduced CPU usage and higher throughput.

Previously, the HTTP/2 multiplexing implementation relied on a lock controlling which request can write to the underlying TCP connection. A thread-safe queue replaces the write lock. Now, rather than fighting over which thread gets to use the write lock, requests now queue up and a dedicated consumer processes them. Previously wasted CPU resources are available to the rest of the app.

One place where these improvements can be noticed is in gRPC, a popular RPC framework that uses HTTP/2. Kestrel + gRPC benchmarks show a dramatic improvement:



Changes were made in the HTTP/2 frame writing code that improves performance when there are multiple streams trying to write data on a single HTTP/2 connection. We now dispatch TLS work to the thread pool and more quickly release a write lock that other streams can acquire to write their data. The reduction in wait times can yield significant performance improvements in cases where there is contention for this write lock. A gRPC benchmark with 70 streams on a single connection (with TLS) showed a ~15% improvement in requests per second (RPS) with this change.

Http/2 WebSockets support

.NET 7 introduces WebSockets over HTTP/2 support for Kestrel, the SignalR JavaScript client, and SignalR with Blazor WebAssembly.

Using WebSockets over HTTP/2 takes advantage of new features such as:

- Header compression.
- Multiplexing, which reduces the time and resources needed when making multiple requests to the server.

These supported features are available in Kestrel on all HTTP/2 enabled platforms. The version negotiation is automatic in browsers and Kestrel, so no new APIs are needed.

For more information, see Http/2 WebSockets support.

Kestrel performance improvements on high core machines

Kestrel uses ConcurrentQueue<T> for many purposes. One purpose is scheduling I/O operations in Kestrel's default Socket transport. Partitioning the `ConcurrentQueue` based on the associated socket reduces contention and increases throughput on machines with many CPU cores.

Profiling on high core machines on .NET 6 showed significant contention in one of Kestrel's other `ConcurrentQueue` instances, the `PinnedMemoryPool` that Kestrel uses to cache byte buffers.

In .NET 7, Kestrel's memory pool is partitioned the same way as its I/O queue, which leads to much lower contention and higher throughput on high core machines. On the 80 core ARM64 VMs, we're seeing over 500% improvement in responses per second (RPS) in the TechEmpower plaintext benchmark. On 48 Core AMD VMs, the improvement is nearly 100% in our HTTPS JSON benchmark.

`ServerReady` event to measure startup time

Apps using EventSource can measure the startup time to understand and optimize startup performance. The new ServerReady event in Microsoft.AspNetCore.Hosting represents the point where the server is ready to respond to requests.

Server

New ServerReady event for measuring startup time

The ServerReady event has been added to measure startup time of ASP.NET Core apps.

IIS

Shadow copying in IIS

Shadow copying app assemblies to the ASP.NET Core Module (ANCM) for IIS can provide a better end user experience than stopping the app by deploying an app offline file.

For more information, see Shadow copying in IIS.

Miscellaneous

Kestrel full certificate chain improvements

HttpsConnectionAdapterOptions has a new ServerCertificateChain property of type X509Certificate2Collection, which makes it easier to validate certificate chains by allowing a full chain including intermediate certificates to be specified. See dotnet/aspnetcore#21513 for more details.

dotnet watch

Improved console output for dotnet watch

The console output from dotnet watch has been improved to better align with the logging of ASP.NET Core and to stand out with 😮emojis😍.

Here's an example of what the new output looks like:

```
dotnet watch 🔥 Hot reload enabled. For a list of supported edits, see https://aka.ms/dotnet/hot-reload.
  💡 Press "Ctrl + R" to restart.
dotnet watch 🔧 Building...
  Determining projects to restore...
  All projects are up-to-date for restore.
  WebApplication18 -> C:\tmp\WebApplication18\bin\Debug\net6.0\WebApplication18.dll
dotnet watch 🚀 Started
dotnet watch ⌚ Files changed: .\Pages\Index.cshtml, .\Pages\Index.cshtml.cs
dotnet watch 🔥 Hot reload of changes succeeded.
```

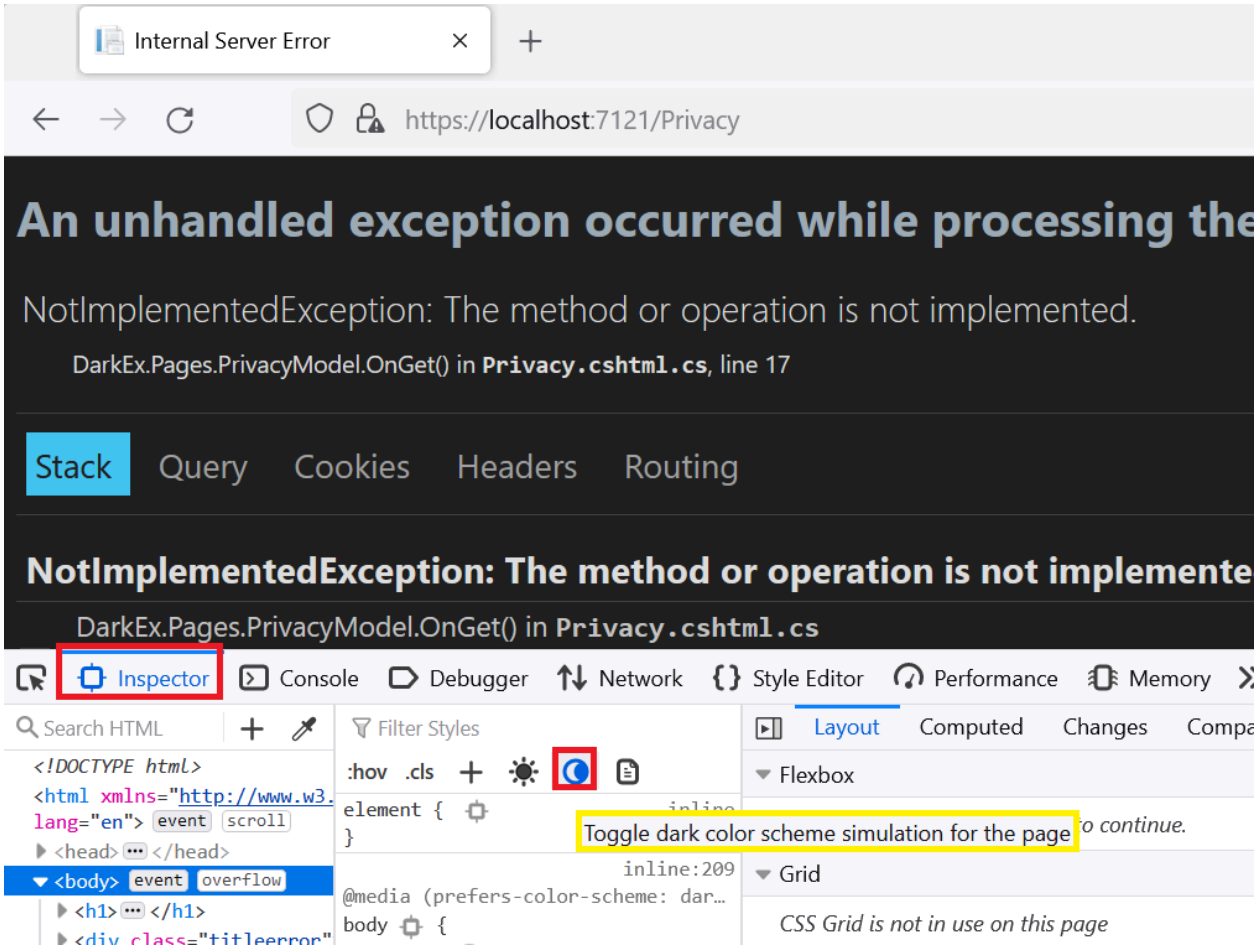
For more information, see this GitHub pull request.

Configure dotnet watch to always restart for rude edits

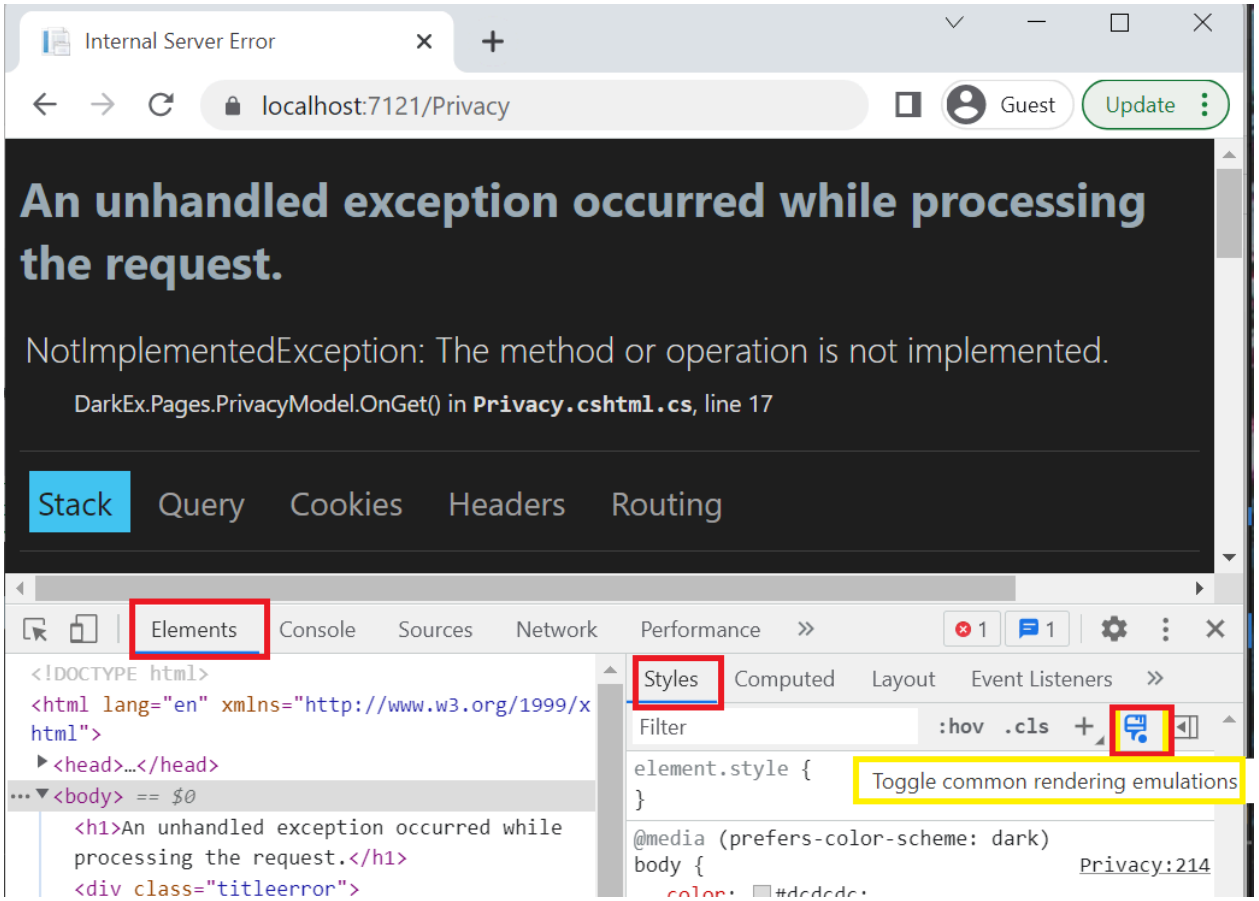
Rude edits are edits that can't be hot reloaded. To configure dotnet watch to always restart without a prompt for rude edits, set the `DOTNET_WATCH_RESTART_ON_RUDE_EDIT` environment variable to `true`.

Developer exception page dark mode

Dark mode support has been added to the developer exception page, thanks to a contribution by Patrick Westerhoff. To test dark mode in a browser, from the developer tools page, set the mode to dark. For example, in Firefox:


Internal Server Error
https://localhost:7121/Privacy
An unhandled exception occurred while processing the
NotImplementedException: The method or operation is not implemented.
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs, line 17
Stack
Query
Cookies
Headers
Routing
NotImplementedException: The method or operation is not implemente
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Search HTML
Filter Styles
Layout
Computed
Changes
:hov .cls
Flexbox
Toggle dark color scheme simulation for the page
Grid
CSS Grid is not in use on this page

In Chrome:


Internal Server Error
localhost:7121/Privacy
Guest
Update
An unhandled exception occurred while processing the request.
NotImplementedException: The method or operation is not implemented.
DarkEx.Pages.PrivacyModel.OnGet() in Privacy.cshtml.cs, line 17
Stack
Query
Cookies
Headers
Routing
Elements
Console
Sources
Network
Performance
Styles
Computed
Layout
Event Listeners
Filter
:hov .cls
element.style {
}
Toggle common rendering emulations
@media (prefers-color-scheme: dark)
body {
Privacy:214

Project template option to use Program.Main method instead of top-level statements

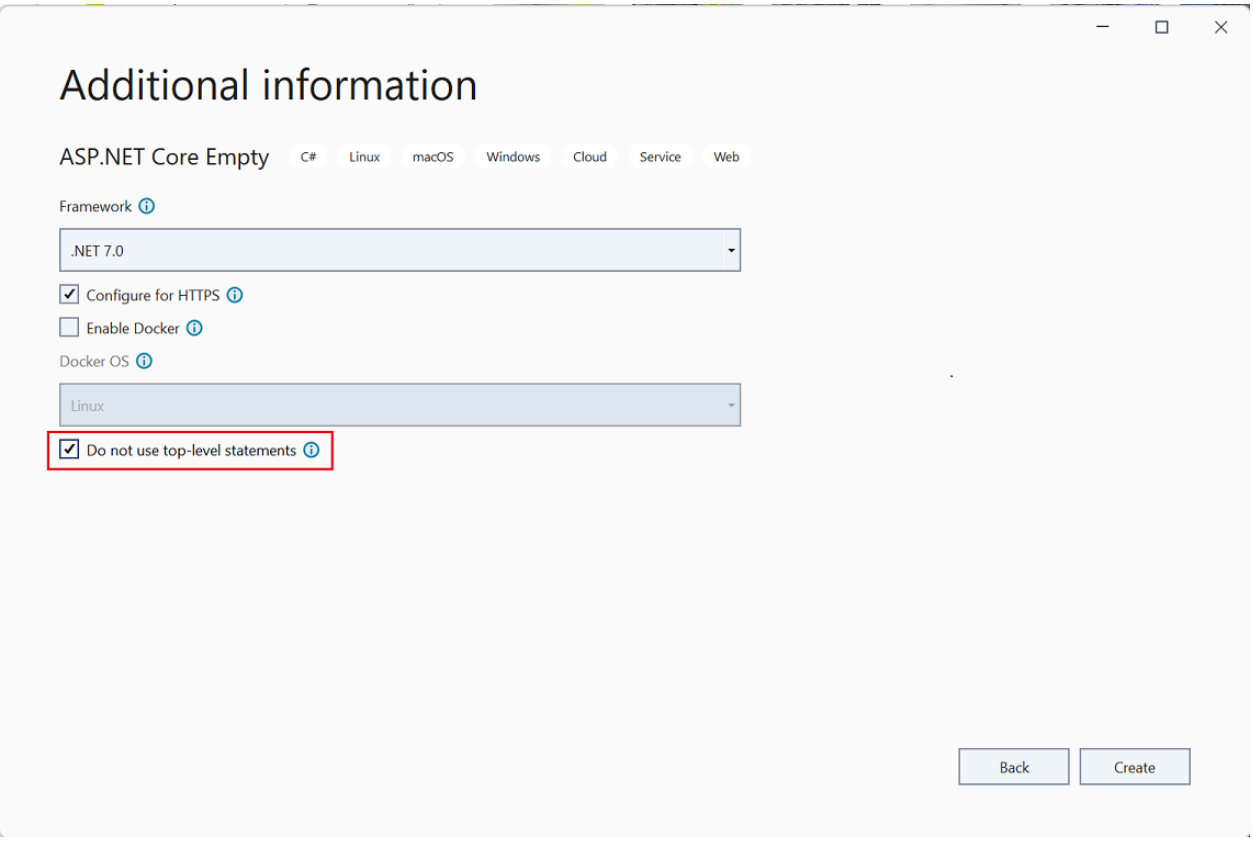
The .NET 7 templates include an option to not use top-level statements and generate a `namespace` and a `Main` method declared on a `Program` class.

Using the .NET CLI, use the `--use-program-main` option:

.NET CLI

```
dotnet new web --use-program-main
```

With Visual Studio, select the new **Do not use top-level statements** checkbox during project creation:



Updated Angular and React templates

The Angular project template has been updated to Angular 14. The React project template has been updated to React 18.2.

Manage JSON Web Tokens in development with dotnet user-jwts

The new `dotnet user-jwts` command line tool can create and manage app specific local JSON Web Tokens (JWTs). For more information, see Manage JSON Web Tokens in development with dotnet user-jwts.

Support for additional request headers in W3CLogger

You can now specify additional request headers to log when using the W3C logger by calling `AdditionalRequestHeaders()` on W3CLoggerOptions:

C#

```
services.AddW3CLogging(logging =>
{
    logging.AdditionalRequestHeaders.Add("x-forwarded-for");
    logging.AdditionalRequestHeaders.Add("x-client-ssl-protocol");
});
```

For more information, see W3CLogger options.

Request decompression

The new Request decompression middleware:

- Enables API endpoints to accept requests with compressed content.
- Uses the Content-Encoding HTTP header to automatically identify and decompress requests which contain compressed content.
- Eliminates the need to write code to handle compressed requests.

For more information, see Request decompression middleware.

What's new in ASP.NET Core 6.0

Article • 10/18/2024

This article highlights the most significant changes in ASP.NET Core 6.0 with links to relevant documentation.

ASP.NET Core MVC and Razor improvements

Minimal APIs

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They are ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core. For more information, see:

- Tutorial: Create a minimal API with ASP.NET Core
- Differences between minimal APIs and APIs with controllers
- Minimal APIs quick reference
- Code samples migrated to the new minimal hosting model in 6.0

SignalR

Long running activity tag for SignalR connections

SignalR uses the new Microsoft.AspNetCore.Http.Features.IHttpActivityFeature.Activity to add an `http.long_running` tag to the request activity. `IHttpActivityFeature.Activity` is used by APM services like Azure Monitor Application Insights to filter SignalR requests from creating long running request alerts.

SignalR performance improvements

- Allocate HubCallerClients once per connection instead of every hub method call.
- Avoid closure allocation in SignalR `DefaultHubDispatcher.Invoke`. State is passed to a local static function via parameters to avoid a closure allocation. For more information, see this GitHub pull request.
- Allocate a single StreamItemMessage per stream instead of per stream item in server-to-client streaming. For more information, see this GitHub pull request.

Razor compiler

Razor compiler updated to use source generators

The Razor compiler is now based on C# source generators. Source generators run during compilation and inspect what is being compiled to produce additional files that are compiled along with the rest of the project. Using source generators simplifies the Razor compiler and significantly speeds up build times.

Razor compiler no longer produces a separate Views assembly

The Razor compiler previously utilized a two-step compilation process that produced a separate Views assembly that contained the generated views and pages (`.cshtml` files) defined in the app. The generated types were public and under the `AspNetCore` namespace.

The updated Razor compiler builds the views and pages types into the main project assembly. These types are now generated by default as internal sealed in the `AspNetCoreGeneratedDocument` namespace. This change improves build performance, enables single file deployment, and enables these types to participate in Hot Reload.

For more information about this change, see the related announcement issue on GitHub.

ASP.NET Core performance and API improvements

Many changes were made to reduce allocations and improve performance across the stack:

- Non-allocating app.Use extension method. The new overload of `app.Use` requires passing the context to `next` which saves two internal per-request allocations that are required when using the other overload.
- Reduced memory allocations when accessing HttpRequest.Cookies. For more information, see this GitHub issue.
- Use LoggerMessage.Define for the windows only HTTP.sys web server. The ILogger extension methods calls have been replaced with calls to `LoggerMessage.Define`.
- Reduce the per connection overhead in SocketConnection by ~30%. For more information, see this GitHub pull request.

- Reduce allocations by removing logging delegates in generic types. For more information, see this GitHub pull request.
- Faster GET access (about 50%) to commonly-used features such as IHttpRequestFeature, IHttpResponseFeature, IHttpResponseBodyFeature, IRouteValuesFeature, and IEndpointFeature. For more information, see this GitHub pull request.
- Use single instance strings for known header names, even if they aren't in the preserved header block. Using single instance string helps prevent multiple duplicates of the same string in long lived connections, for example, in Microsoft.AspNetCore.WebSockets. For more information, see this GitHub issue.
- Reuse HttpProtocol CancellationTokenSource in Kestrel. Use the new CancellationTokenSource.TryReset method on `CancellationTokenSource` to reuse tokens if they haven't been canceled. For more information, see this GitHub issue and this video.
- Implement and use an AdaptiveCapacityDictionary in Microsoft.AspNetCore.Http RequestCookieCollection for more efficient access to dictionaries. For more information, see this GitHub pull request.

Reduced memory footprint for idle TLS connections

For long running TLS connections where data is only occasionally sent back and forth, we've significantly reduced the memory footprint of ASP.NET Core apps in .NET 6. This should help improve the scalability of scenarios such as WebSocket servers. This was possible due to numerous improvements in System.IO.Pipelines, SslStream, and Kestrel. The following sections detail some of the improvements that have contributed to the reduced memory footprint:

Reduce the size of `System.IO.Pipelines.Pipe`

For every connection that is established, two pipes are allocated in Kestrel:

- The transport layer to the app for the request.
- The application layer to the transport for the response.

By shrinking the size of System.IO.Pipelines.Pipe from 368 bytes to 264 bytes (about a 28.2% reduction), 208 bytes per connection are saved (104 bytes per Pipe).

Pool SocketSender

`SocketSender` objects (that subclass SocketAsyncEventArgs) are around 350 bytes at runtime. Instead of allocating a new `SocketSender` object per connection, they can be

pooled. `SocketSender` objects can be pooled because sends are usually very fast. Pooling reduces the per connection overhead. Instead of allocating 350 bytes per connection, only pay 350 bytes per `IOQueue` are allocated. Allocation is done per queue to avoid contention. Our WebSocket server with 5000 idle connections went from allocating ~1.75 MB (350 bytes * 5000) to allocating ~2.8 kb (350 bytes * 8) for `SocketSender` objects.

Zero bytes reads with SslStream

Bufferless reads are a technique employed in ASP.NET Core to avoid renting memory from the memory pool if there's no data available on the socket. Prior to this change, our WebSocket server with 5000 idle connections required ~200 MB without TLS compared to ~800 MB with TLS. Some of these allocations (4k per connection) were from Kestrel having to hold on to an ArrayPool<T> buffer while waiting for the reads on SslStream to complete. Given that these connections were idle, none of reads completed and returned their buffers to the `ArrayPool`, forcing the `ArrayPool` to allocate more memory. The remaining allocations were in `SslStream` itself: 4k buffer for TLS handshakes and 32k buffer for normal reads. In .NET 6, when the user performs a zero byte read on `SslStream` and it has no data available, `SslStream` internally performs a zero-byte read on the underlying wrapped stream. In the best case (idle connection), these changes result in a savings of 40 Kb per connection while still allowing the consumer (Kestrel) to be notified when data is available without holding on to any unused buffers.

Zero byte reads with PipeReader

With bufferless reads supported on `SslStream`, an option was added to perform zero byte reads to `StreamPipeReader`, the internal type that adapts a `Stream` into a `PipeReader`. In Kestrel, a `StreamPipeReader` is used to adapt the underlying `SslStream` into a `PipeReader`. Therefore it was necessary to expose these zero byte read semantics on the `PipeReader`.

A `PipeReader` can now be created that supports zero bytes reads over any underlying `Stream` that supports zero byte read semantics (e.g,. `SslStream`, NetworkStream, etc) using the following API:

.NET CLI

```
var reader = PipeReader.Create(stream, new
StreamPipeReaderOptions(useZeroByteReads: true));
```

Remove slabs from the `SlabMemoryPool`

To reduce fragmentation of the heap, Kestrel employed a technique where it allocated slabs of memory of 128 KB as part of its memory pool. The slabs were then further divided into 4 KB blocks that were used by Kestrel internally. The slabs had to be larger than 85 KB to force allocation on the large object heap to try and prevent the GC from relocating this array. However, with the introduction of the new GC generation, Pinned Object Heap (POH), it no longer makes sense to allocate blocks on slab. Kestrel now directly allocates blocks on the POH, reducing the complexity involved in managing the memory pool. This change should make easier to perform future improvements such as making it easier to shrink the memory pool used by Kestrel.

IAsyncDisposable supported

IAsyncDisposable is now available for controllers, Razor Pages, and View Components. Asynchronous versions have been added to the relevant interfaces in factories and activators:

- The new methods offer a default interface implementation that delegates to the synchronous version and calls Dispose.
- The implementations override the default implementation and handle disposing `IAsyncDisposable` implementations.
- The implementations favor `IAsyncDisposable` over `IDisposable` when both interfaces are implemented.
- Extenders must override the new methods included to support `IAsyncDisposable` instances.

`IAsyncDisposable` is beneficial when working with:

- Asynchronous enumerators, for example, in asynchronous streams.
- Unmanaged resources that have resource-intensive I/O operations to release.

When implementing this interface, use the `DisposeAsync` method to release resources.

Consider a controller that creates and uses a Utf8JsonWriter. `Utf8JsonWriter` is an `IAsyncDisposable` resource:

```
C#

public class HomeController : Controller, IAsyncDisposable
{
    private Utf8JsonWriter? _jsonWriter;
    private readonly ILogger<HomeController> _logger;
```

```
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
        _jsonWriter = new Utf8JsonWriter(new MemoryStream());
    }
```

`IAsyncDisposable` must implement `DisposeAsync`:

C#

```
public async ValueTask DisposeAsync()
{
    if (_jsonWriter is not null)
    {
        await _jsonWriter.DisposeAsync();
    }

    _jsonWriter = null;
}
```

Vcpkg port for SignalR C++ client

Vcpkg is a cross-platform command-line package manager for C and C++ libraries. We've recently added a port to `vcpkg` to add `CMake` native support for the SignalR C++ client. `vcpkg` also works with MSBuild.

The SignalR client can be added to a CMake project with the following snippet when the vcpkg is included in the toolchain file:

.NET CLI

```
find_package(microsoft-signalr CONFIG REQUIRED)
link_libraries(microsoft-signalr::microsoft-signalr)
```

With the preceding snippet, the SignalR C++ client is ready to use #include and used in a project without any additional configuration. For a complete example of a C++ application that utilizes the SignalR C++ client, see the halter73/SignalR-Client-Cpp-Sample repository.

Blazor

Project template changes

Several project template changes were made for Blazor apps, including the use of the `Pages/_Layout.cshtml` file for layout content that appeared in the `_Host.cshtml` file for earlier Blazor Server apps. Study the changes by creating an app from a 6.0 project template or accessing the ASP.NET Core reference source for the project templates:

- Blazor Server
- Blazor WebAssembly

Blazor WebAssembly native dependencies support

Blazor WebAssembly apps can use native dependencies built to run on WebAssembly. For more information, see ASP.NET Core Blazor WebAssembly native dependencies.

WebAssembly Ahead-of-time (AOT) compilation and runtime relinking

Blazor WebAssembly supports ahead-of-time (AOT) compilation, where you can compile your .NET code directly into WebAssembly. AOT compilation results in runtime performance improvements at the expense of a larger app size. Relinking the .NET WebAssembly runtime trims unused runtime code and thus improves download speed. For more information, see Ahead-of-time (AOT) compilation and Runtime relinking.

Persist prerendered state

Blazor supports persisting state in a prerendered page so that the state doesn't need to be recreated when the app is fully loaded. For more information, see Integrate ASP.NET Core Razor components.

Error boundaries

Error boundaries provide a convenient approach for handling exceptions on the UI level. For more information, see Handle errors in ASP.NET Core Blazor apps.

SVG support

The <foreignObject> element element is supported to display arbitrary HTML within an SVG. For more information, see ASP.NET Core Razor components.

Blazor Server support for byte array transfer in JS Interop

Blazor supports optimized byte array JS interop that avoids encoding and decoding byte arrays into Base64. For more information, see the following resources:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Query string enhancements

Support for working with query strings is improved. For more information, see ASP.NET Core Blazor routing and navigation.

Binding to select multiple

Binding supports multiple option selection with `<input>` elements. For more information, see the following resources:

- ASP.NET Core Blazor data binding
- ASP.NET Core Blazor input components

Head (`<head>`) content control

Razor components can modify the HTML `<head>` element content of a page, including setting the page's title (`<title>` element) and modifying metadata (`<meta>` elements). For more information, see Control <head> content in ASP.NET Core Blazor apps.

Generate Angular and React components

Generate framework-specific JavaScript components from Razor components for web frameworks, such as Angular or React. For more information, see ASP.NET Core Razor components.

Render components from JavaScript

Render Razor components dynamically from JavaScript for existing JavaScript apps. For more information, see ASP.NET Core Razor components.

Custom elements

Experimental support is available for building custom elements, which use standard HTML interfaces. For more information, see ASP.NET Core Razor components.

Infer component generic types from ancestor components

An ancestor component can cascade a type parameter by name to descendants using the new `[CascadingTypeParameter]` attribute. For more information, see ASP.NET Core Razor components.

Dynamically rendered components

Use the new built-in `DynamicComponent` component to render components by type. For more information, see Dynamically-rendered ASP.NET Core Razor components.

Improved Blazor accessibility

Use the new `FocusOnNavigate` component to set the UI focus to an element based on a CSS selector after navigating from one page to another. For more information, see ASP.NET Core Blazor routing and navigation.

Custom event argument support

Blazor supports custom event arguments, which enable you to pass arbitrary data to .NET event handlers with custom events. For more information, see ASP.NET Core Blazor event handling.

Required parameters

Apply the new `[EditorRequired]` attribute to specify a required component parameter. For more information, see ASP.NET Core Razor components.

Collocation of JavaScript files with pages, views, and components

Collocate JavaScript files for pages, views, and Razor components as a convenient way to organize scripts in an app. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

JavaScript initializers

JavaScript initializers execute logic before and after a Blazor app loads. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

Streaming JavaScript interop

Blazor now supports streaming data directly between .NET and JavaScript. For more information, see the following resources:

- Stream from .NET to JavaScript
- Stream from JavaScript to .NET

Generic type constraints

Generic type parameters are now supported. For more information, see ASP.NET Core Razor components.

WebAssembly deployment layout

Use a deployment layout to enable Blazor WebAssembly app downloads in restricted security environments. For more information, see Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps.

New Blazor articles

In addition to the Blazor features described in the preceding sections, new Blazor articles are available on the following subjects:

- ASP.NET Core Blazor file downloads: Learn how to download a file using native `byte[]` streaming interop to ensure efficient transfer to the client.
- Display images and documents in ASP.NET Core Blazor: Discover how to work with images and documents in Blazor apps, including how to stream image and document data.

Build Blazor Hybrid apps with .NET MAUI, WPF, and Windows Forms

Use Blazor Hybrid to blend desktop and mobile native client frameworks with .NET and Blazor:

- .NET Multi-platform App UI (.NET MAUI) is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.
- Blazor Hybrid apps can be built with Windows Presentation Foundation (WPF) and Windows Forms frameworks.

> **ⓘ Important**
>
> Blazor Hybrid is in preview and shouldn't be used in production apps until final release.

For more information, see the following resources:

- Preview ASP.NET Core Blazor Hybrid documentation
- What is .NET MAUI?
- Microsoft .NET Blog (category: ".NET MAUI")

Kestrel

HTTP/3 is currently in draft and therefore subject to change. HTTP/3 support in ASP.NET Core is not released, it's a preview feature included in .NET 6.

Kestrel now supports HTTP/3. For more information, see Use HTTP/3 with the ASP.NET Core Kestrel web server and the blog entry HTTP/3 support in .NET 6.

New Kestrel logging categories for selected logging

Prior to this change, enabling verbose logging for Kestrel was prohibitively expensive as all of Kestrel shared the `Microsoft.AspNetCore.Server.Kestrel` logging category name. `Microsoft.AspNetCore.Server.Kestrel` is still available, but the following new subcategories allow for more control of logging:

- `Microsoft.AspNetCore.Server.Kestrel` (current category): `ApplicationError`, `ConnectionHeadResponseBodyWrite`, `ApplicationNeverCompleted`, `RequestBodyStart`, `RequestBodyDone`, `RequestBodyNotEntirelyRead`, `RequestBodyDrainTimedOut`, `ResponseMinimumDataRateNotSatisfied`, `InvalidResponseHeaderRemoved`, `HeartbeatSlow`.
- `Microsoft.AspNetCore.Server.Kestrel.BadRequests`: `ConnectionBadRequest`, `RequestProcessingError`, `RequestBodyMinimumDataRateNotSatisfied`.
- `Microsoft.AspNetCore.Server.Kestrel.Connections`: `ConnectionAccepted`, `ConnectionStart`, `ConnectionStop`, `ConnectionPause`, `ConnectionResume`,

`ConnectionKeepAlive`, `ConnectionRejected`, `ConnectionDisconnect`, `NotAllConnectionsClosedGracefully`, `NotAllConnectionsAborted`, `ApplicationAbortedConnection`.

- `Microsoft.AspNetCore.Server.Kestrel.Http2`: `Http2ConnectionError`, `Http2ConnectionClosing`, `Http2ConnectionClosed`, `Http2StreamError`, `Http2StreamResetAbort`, `HPackDecodingError`, `HPackEncodingError`, `Http2FrameReceived`, `Http2FrameSending`, `Http2MaxConcurrentStreamsReached`.
- `Microsoft.AspNetCore.Server.Kestrel.Http3`: `Http3ConnectionError`, `Http3ConnectionClosing`, `Http3ConnectionClosed`, `Http3StreamAbort`, `Http3FrameReceived`, `Http3FrameSending`.

Existing rules continue to work, but you can now be more selective on which rules you enable. For example, the observability overhead of enabling `Debug` logging for just bad requests is greatly reduced and can be enabled with the following configuration:

XML

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.Kestrel.BadRequests": "Debug"
    }
  }
```

Log filtering applies rules with the longest matching category prefix. For more information, see How filtering rules are applied

Emit KestrelServerOptions via EventSource event

The KestrelEventSource emits a new event containing the JSON-serialized KestrelServerOptions when enabled with verbosity `EventLevel.LogAlways`. This event makes it easier to reason about the server behavior when analyzing collected traces. The following JSON is an example of the event payload:

JSON

```
{
  "AllowSynchronousIO": false,
  "AddServerHeader": true,
  "AllowAlternateSchemes": false,
  "AllowResponseHeaderCompression": true,
  "EnableAltSvc": false,
```

```
  "IsDevCertLoaded": true,
  "RequestHeaderEncodingSelector": "default",
  "ResponseHeaderEncodingSelector": "default",
  "Limits": {
    "KeepAliveTimeout": "00:02:10",
    "MaxConcurrentConnections": null,
    "MaxConcurrentUpgradedConnections": null,
    "MaxRequestBodySize": 30000000,
    "MaxRequestBufferSize": 1048576,
    "MaxRequestHeaderCount": 100,
    "MaxRequestHeadersTotalSize": 32768,
    "MaxRequestLineSize": 8192,
    "MaxResponseBufferSize": 65536,
    "MinRequestBodyDataRate": "Bytes per second: 240, Grace Period:
00:00:05",
    "MinResponseDataRate": "Bytes per second: 240, Grace Period: 00:00:05",
    "RequestHeadersTimeout": "00:00:30",
    "Http2": {
      "MaxStreamsPerConnection": 100,
      "HeaderTableSize": 4096,
      "MaxFrameSize": 16384,
      "MaxRequestHeaderFieldSize": 16384,
      "InitialConnectionWindowSize": 131072,
      "InitialStreamWindowSize": 98304,
      "KeepAlivePingDelay": "10675199.02:48:05.4775807",
      "KeepAlivePingTimeout": "00:00:20"
    },
    "Http3": {
      "HeaderTableSize": 0,
      "MaxRequestHeaderFieldSize": 16384
    }
  },
  "ListenOptions": [
    {
      "Address": "https://127.0.0.1:7030",
      "IsTls": true,
      "Protocols": "Http1AndHttp2"
    },
    {
      "Address": "https://[::1]:7030",
      "IsTls": true,
      "Protocols": "Http1AndHttp2"
    },
    {
      "Address": "http://127.0.0.1:5030",
      "IsTls": false,
      "Protocols": "Http1AndHttp2"
    },
    {
      "Address": "http://[::1]:5030",
      "IsTls": false,
      "Protocols": "Http1AndHttp2"
    }
  ]
}
```

New DiagnosticSource event for rejected HTTP requests

Kestrel now emits a new `DiagnosticSource` event for HTTP requests rejected at the server layer. Prior to this change, there was no way to observe these rejected requests. The new `DiagnosticSource` event `Microsoft.AspNetCore.Server.Kestrel.BadRequest` contains a IBadRequestExceptionFeature that can be used to introspect the reason for rejecting the request.

C#

```
using Microsoft.AspNetCore.Http.Features;
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
var diagnosticSource = app.Services.GetRequiredService<DiagnosticListener>
();
using var badRequestListener = new BadRequestEventListener(diagnosticSource,
    (badRequestExceptionFeature) =>
{
    app.Logger.LogError(badRequestExceptionFeature.Error, "Bad request
received");
});
app.MapGet("/", () => "Hello world");

app.Run();

class BadRequestEventListener : IObserver<KeyValuePair<string, object>>,
IDisposable
{
    private readonly IDisposable _subscription;
    private readonly Action<IBadRequestExceptionFeature> _callback;

    public BadRequestEventListener(DiagnosticListener diagnosticListener,
                                   Action<IBadRequestExceptionFeature>
callback)
    {
        _subscription = diagnosticListener.Subscribe(this!, IsEnabled);
        _callback = callback;
    }
    private static readonly Predicate<string> IsEnabled = (provider) =>
provider switch
    {
        "Microsoft.AspNetCore.Server.Kestrel.BadRequest" => true,
        _ => false
    };
    public void OnNext(KeyValuePair<string, object> pair)
    {
        if (pair.Value is IFeatureCollection featureCollection)
```

```
        {
            var badRequestFeature =
featureCollection.Get<IBadRequestExceptionFeature>();

            if (badRequestFeature is not null)
            {
                _callback(badRequestFeature);
            }
        }
    }
    public void OnError(Exception error) { }
    public void OnCompleted() { }
    public virtual void Dispose() => _subscription.Dispose();
}
```

For more information, see Logging and diagnostics in Kestrel.

Create a ConnectionContext from an Accept Socket

The new SocketConnectionContextFactory makes it possible to create a ConnectionContext from an accepted socket. This makes it possible to build a custom socket-based IConnectionListenerFactory without losing out on all the performance work and pooling happening in SocketConnection.

See this example of a custom IConnectionListenerFactory which shows how to use this `SocketConnectionContextFactory`.

Kestrel is the default launch profile for Visual Studio

The default launch profile for all new dotnet web projects is Kestrel. Starting Kestrel is significantly faster and results in a more responsive experience while developing apps.

IIS Express is still available as a launch profile for scenarios such as Windows Authentication or port sharing.

Localhost ports for Kestrel are random

See Template generated ports for Kestrel in this document for more information.

Authentication and authorization

Authentication servers

.NET 3 to .NET 5 used IdentityServer4 as part of our template to support the issuing of JWT tokens for SPA and Blazor applications. The templates now use the Duende Identity Server.

If you are extending the identity models and are updating existing projects, update the namespaces in your code from `IdentityServer4.IdentityServer` to `Duende.IdentityServer` and follow their migration instructions.

The license model for Duende Identity Server has changed to a reciprocal license, which may require license fees when it's used commercially in production. See the Duende license page for more details.

Delayed client certificate negotiation

Developers can now opt-in to using delayed client certificate negotiation by specifying ClientCertificateMode.DelayCertificate on the HttpsConnectionAdapterOptions. This only works with HTTP/1.1 connections because HTTP/2 forbids delayed certificate renegotiation. The caller of this API must buffer the request body before requesting the client certificate:

C#

```
using Microsoft.AspNetCore.Server.Kestrel.Https;
using Microsoft.AspNetCore.WebUtilities;

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseKestrel(options =>
{
    options.ConfigureHttpsDefaults(adapterOptions =>
    {
        adapterOptions.ClientCertificateMode =
ClientCertificateMode.DelayCertificate;
    });
});

var app = builder.Build();
app.Use(async (context, next) =>
{
    bool desiredState = GetDesiredState();
    // Check if your desired criteria is met
    if (desiredState)
    {
        // Buffer the request body
        context.Request.EnableBuffering();
        var body = context.Request.Body;
        await body.DrainAsync(context.RequestAborted);
        body.Position = 0;
```

```
        // Request client certificate
        var cert = await context.Connection.GetClientCertificateAsync();

        //  Disable buffering on future requests if the client doesn't
provide a cert
    }
    await next(context);
});


app.MapGet("/", () => "Hello World!");
app.Run();
```

`OnCheckSlidingExpiration` event for controlling cookie renewal

Cookie authentication sliding expiration can now be customized or suppressed using the new OnCheckSlidingExpiration. For example, this event can be used by a single-page app that needs to periodically ping the server without affecting the authentication session.

Miscellaneous

Hot Reload

Quickly make UI and code updates to running apps without losing app state for faster and more productive developer experience using Hot Reload. For more information, see .NET Hot Reload support for ASP.NET Core and Update on .NET Hot Reload progress and Visual Studio 2022 Highlights.

Improved single-page app (SPA) templates

The ASP.NET Core project templates have been updated for Angular and React to use an improved pattern for single-page apps that is more flexible and more closely aligns with common patterns for modern front-end web development.

Previously, the ASP.NET Core template for Angular and React used specialized middleware during development to launch the development server for the front-end framework and then proxy requests from ASP.NET Core to the development server. The logic for launching the front-end development server was specific to the command-line interface for the corresponding front-end framework. Supporting additional front-end frameworks using this pattern meant adding additional logic to ASP.NET Core.

The updated ASP.NET Core templates for Angular and React in .NET 6 flips this arrangement around and take advantage of the built-in proxying support in the development servers of most modern front-end frameworks. When the ASP.NET Core app is launched, the front-end development server is launched just as before, but the development server is configured to proxy requests to the backend ASP.NET Core process. All of the front-end specific configuration to setup proxying is part of the app, not ASP.NET Core. Setting up ASP.NET Core projects to work with other front-end frameworks is now straight-forward: setup the front-end development server for the chosen framework to proxy to the ASP.NET Core backend using the pattern established in the Angular and React templates.

The startup code for the ASP.NET Core app no longer needs any single-page app-specific logic. The logic for starting the front-end development server during development is injecting into the app at runtime by the new Microsoft.AspNetCore.SpaProxy package. Fallback routing is handled using endpoint routing instead of SPA-specific middleware.

Templates that follow this pattern can still be run as a single project in Visual Studio or using `dotnet run` from the command-line. When the app is published, the front-end code in the *ClientApp* folder is built and collected as before into the web root of the host ASP.NET Core app and served as static files. Scripts included in the template configure the front-end development server to use HTTPS using the ASP.NET Core development certificate.

Draft HTTP/3 support in .NET 6

HTTP/3 is currently in draft and therefore subject to change. HTTP/3 support in ASP.NET Core is not released, it's a preview feature included in .NET 6.

See the blog entry HTTP/3 support in .NET 6.

Nullable Reference Type Annotations

Portions of the ASP.NET Core 6.0 source code has had nullability annotations applied.

By utilizing the new Nullable feature in C# 8, ASP.NET Core can provide additional compile-time safety in the handling of reference types. For example, protecting against `null` reference exceptions. Projects that have opted in to using nullable annotations may see new build-time warnings from ASP.NET Core APIs.

To enable nullable reference types, add the following property to project files:

```
```

XML

```xml
<PropertyGroup>
    <Nullable>enable</Nullable>
</PropertyGroup>
```

For more information, see Nullable reference types.

Source Code Analysis

Several .NET compiler platform analyzers were added that inspect application code for problems such as incorrect middleware configuration or order, routing conflicts, etc. For more information, see Code analysis in ASP.NET Core apps.

Web app template improvements

The web app templates:

- Use the new minimal hosting model.
- Significantly reduces the number of files and lines of code required to create an app. For example, the ASP.NET Core empty web app creates one C# file with four lines of code and is a complete app.
- Unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file.
- Uses top-level statements to minimize the code required for an app.
- Uses global using directives to eliminate or minimize the number of using statement lines required.

Template generated ports for Kestrel

Random ports are assigned during project creation for use by the Kestrel web server. Random ports help minimize a port conflict when multiple projects are run on the same machine.

When a project is created, a random HTTP port between 5000-5300 and a random HTTPS port between 7000-7300 is specified in the generated `Properties/launchSettings.json` file. The ports can be changed in the `Properties/launchSettings.json` file. If no port is specified, Kestrel defaults to the HTTP 5000 and HTTPS 5001 ports. For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

New logging defaults

The following changes were made to both `appsettings.json` and `appsettings.Development.json`:

```diff
- "Microsoft": "Warning",
- "Microsoft.Hosting.Lifetime": "Information"
+ "Microsoft.AspNetCore": "Warning"
```

The change from `"Microsoft": "Warning"` to `"Microsoft.AspNetCore": "Warning"` results in logging all informational messages from the `Microsoft` namespace ***except*** `Microsoft.AspNetCore`. For example, `Microsoft.EntityFrameworkCore` is now logged at the informational level.

Developer exception page Middleware added automatically

In the development environment, the DeveloperExceptionPageMiddleware is added by default. It's no longer necessary to add the following code to web UI apps:

```C#
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
```

Support for Latin1 encoded request headers in HttpSysServer

`HttpSysServer` now supports decoding request headers that are `Latin1` encoded by setting the UseLatin1RequestHeaders property on HttpSysOptions to `true`:

```C#
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseHttpSys(o => o.UseLatin1RequestHeaders = true);

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The ASP.NET Core Module logs include timestamps and PID

The ASP.NET Core Module (ANCM) for IIS (ANCM) enhanced diagnostic logs include timestamps and PID of the process emitting the logs. Logging timestamps and PID makes it easier to diagnose issues with overlapping process restarts in IIS when multiple IIS worker processes are running.

The resulting logs now resemble the sample output show below:

.NET CLI

```
[2021-07-28T19:23:44.076Z, PID: 11020] [aspnetcorev2.dll] Initializing logs
for 'C:\<path>\aspnetcorev2.dll'. Process Id: 11020. File Version:
16.0.21209.0. Description: IIS ASP.NET Core Module V2. Commit:
96475a2acdf50d7599ba8e96583fa73efbe27912.
[2021-07-28T19:23:44.079Z, PID: 11020] [aspnetcorev2.dll] Resolving hostfxr
parameters for application: '.\InProcessWebSite.exe' arguments: '' path:
'C:\Temp\e86ac4e9ced24bb6bacf1a9415e70753\'
[2021-07-28T19:23:44.080Z, PID: 11020] [aspnetcorev2.dll] Known dotnet.exe
location: ''
```

Configurable unconsumed incoming buffer size for IIS

The IIS server previously only buffered 64 KiB of unconsumed request bodies. The 64 KiB buffering resulted in reads being constrained to that maximum size, which impacts the performance with large incoming bodies such as uploads. In .NET 6 , the default buffer size changes from 64 KiB to 1 MiB which should improve throughput for large uploads. In our tests, a 700 MiB upload that used to take 9 seconds now only takes 2.5 seconds.

The downside of a larger buffer size is an increased per-request memory consumption when the app isn't quickly reading from the request body. So, in addition to changing the default buffer size, the buffer size configurable, allowing apps to configure the buffer size based on workload.

View Components Tag Helpers

Consider a view component with an optional parameter, as shown in the following code:

C#

```
class MyViewComponent
{
```

```
    IViewComponentResult Invoke(bool showSomething = false) { ... }
}
```

With ASP.NET Core 6, the tag helper can be invoked without having to specify a value for the `showSomething` parameter:

razor

```
<vc:my />
```

Angular template updated to Angular 12

The ASP.NET Core 6.0 template for Angular now uses Angular 12.

The React template has been updated to React 17.

Configurable buffer threshold before writing to disk in Json.NET output formatter

Note: We recommend using the System.Text.Json output formatter except when the `Newtonsoft.Json` serializer is required for compatibility reasons. The `System.Text.Json` serializer is fully `async` and works efficiently for larger payloads.

The `Newtonsoft.Json` output formatter by default buffers responses up to 32 KiB in memory before buffering to disk. This is to avoid performing synchronous IO, which can result in other side-effects such as thread starvation and application deadlocks. However, if the response is larger than 32 KiB, considerable disk I/O occurs. The memory threshold is now configurable via the MvcNewtonsoftJsonOptions.OutputFormatterMemoryBufferThreshold property before buffering to disk:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages()
            .AddNewtonsoftJson(options =>
            {
                options.OutputFormatterMemoryBufferThreshold = 48 * 1024;
            });

var app = builder.Build();
```

For more information, see this GitHub pull request and the NewtonsoftJsonOutputFormatterTest.cs file.

Faster get and set for HTTP headers

New APIs were added to expose all common headers available on Microsoft.Net.Http.Headers.HeaderNames as properties on the IHeaderDictionary resulting in an easier to use API. For example, the in-line middleware in the following code gets and sets both request and response headers using the new APIs:

```C#
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Use(async (context, next) =>
{
    var hostHeader = context.Request.Headers.Host;
    app.Logger.LogInformation("Host header: {host}", hostHeader);
    context.Response.Headers.XPoweredBy = "ASP.NET Core 6.0";
    await next.Invoke(context);
    var dateHeader = context.Response.Headers.Date;
    app.Logger.LogInformation("Response date: {date}", dateHeader);
});

app.Run();
```

For implemented headers the get and set accessors are implemented by going directly to the field and bypassing the lookup. For non-implemented headers, the accessors can bypass the initial lookup against implemented headers and directly perform the `Dictionary<string, StringValues>` lookup. Avoiding the lookup results in faster access for both scenarios.

Async streaming

ASP.NET Core now supports asynchronous streaming from controller actions and responses from the JSON formatter. Returning an `IAsyncEnumerable` from an action no longer buffers the response content in memory before it gets sent. Not buffering helps reduce memory usage when returning large datasets that can be asynchronously enumerated.

Note that Entity Framework Core provides implementations of `IAsyncEnumerable` for querying the database. The improved support for `IAsyncEnumerable` in ASP.NET Core in .NET 6 can make using EF Core with ASP.NET Core more efficient. For example, the following code no longer buffers the product data into memory before sending the response:

C#

```
public IActionResult GetMovies()
{
    return Ok(_context.Movie);
}
```

However, when using lazy loading in EF Core, this new behavior may result in errors due to concurrent query execution while the data is being enumerated. Apps can revert back to the previous behavior by buffering the data:

C#

```
public async Task<IActionResult> GetMovies2()
{
    return Ok(await _context.Movie.ToListAsync());
}
```

See the related announcement for additional details about this change in behavior.

HTTP logging middleware

HTTP logging is a new built-in middleware that logs information about HTTP requests and HTTP responses including the headers and entire body:

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();
app.UseHttpLogging();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Navigating to `/` with the previous code logs information similar to the following output:

.NET CLI

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/2
      Method: GET
      Scheme: https
      PathBase:
      Path: /
      Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,
*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
      Accept-Encoding: gzip, deflate, br
      Accept-Language: en-US,en;q=0.9
      Cache-Control: max-age=0
      Connection: close
      Cookie: [Redacted]
      Host: localhost:44372
      User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/95.0.4638.54 Safari/537.36
Edg/95.0.1020.30
      sec-ch-ua: [Redacted]
      sec-ch-ua-mobile: [Redacted]
      sec-ch-ua-platform: [Redacted]
      upgrade-insecure-requests: [Redacted]
      sec-fetch-site: [Redacted]
      sec-fetch-mode: [Redacted]
      sec-fetch-user: [Redacted]
      sec-fetch-dest: [Redacted]
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: text/plain; charset=utf-8
```

The preceding output was enabled with the following `appsettings.Development.json` file:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware":
"Information"
    }
  }
}
```

HTTP logging provides logs of:

- HTTP Request information

- Common properties
- Headers
- Body
- HTTP Response information

To configure the HTTP logging middleware, specify HttpLoggingOptions:

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddHttpLogging(logging =>
{
    // Customize HTTP logging.
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("My-Request-Header");
    logging.ResponseHeaders.Add("My-Response-Header");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
});

var app = builder.Build();
app.UseHttpLogging();

app.MapGet("/", () => "Hello World!");

app.Run();
```

IConnectionSocketFeature

The IConnectionSocketFeature request feature provides access to the underlying accept socket associated with the current request. It can be accessed via the FeatureCollection on `HttpContext`.

For example, the following app sets the LingerState property on the accepted socket:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions =>
listenOptions.Use((connection, next) =>
    {
        var socketFeature =
connection.Features.Get<IConnectionSocketFeature>();
        socketFeature.Socket.LingerState = new LingerOption(true, seconds:
```

```
10);
        return next();
    }));
});
var app = builder.Build();
app.MapGet("/", (Func<string>)(() => "Hello world"));
await app.RunAsync();
```

Generic type constraints in Razor

When defining generic type parameters in Razor using the `@typeparam` directive, generic type constraints can now be specified using the standard C# syntax:

Smaller SignalR, Blazor Server, and MessagePack scripts

The SignalR, MessagePack, and Blazor Server scripts are now significantly smaller, enabling smaller downloads, less JavaScript parsing and compiling by the browser, and faster start-up. The size reductions:

- `signalr.js`: 70%
- `blazor.server.js`: 45%

The smaller scripts are a result of a community contribution from Ben Adams. For more information on the details of the size reduction, see Ben's GitHub pull request.

Enable Redis profiling sessions

A community contribution from Gabriel Lucaci enables Redis profiling session with Microsoft.Extensions.Caching.StackExchangeRedis:

C#

```
using StackExchange.Redis.Profiling;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddStackExchangeRedisCache(options =>
{
    options.ProfilingSession = () => new ProfilingSession();
});
```

For more information, see StackExchange.Redis Profiling.

Shadow copying in IIS

An experimental feature has been added to the ASP.NET Core Module (ANCM) for IIS to add support for shadow copying application assemblies. Currently .NET locks application binaries when running on Windows making it impossible to replace binaries when the app is running. While our recommendation remains to use an app offline file, we recognize there are certain scenarios (for example FTP deployments) where it isn't possible to do so.

In such scenarios, enable shadow copying by customizing the ASP.NET Core module handler settings. In most cases, ASP.NET Core apps do not have a `web.config` checked into source control that you can modify. In ASP.NET Core, `web.config` is ordinarily generated by the SDK. The following sample `web.config` can be used to get started:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <!-- To customize the asp.net core module uncomment and edit the following
section.
  For more info see https://go.microsoft.com/fwlink/?linkid=838655 -->

  <system.webServer>
    <handlers>
      <remove name="aspNetCore"/>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%"
stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout">
      <handlerSettings>
        <handlerSetting name="experimentalEnableShadowCopy" value="true" />
        <handlerSetting name="shadowCopyDirectory"
value="../ShadowCopyDirectory/" />
        <!-- Only enable handler logging if you encounter issues-->
        <!--<handlerSetting name="debugFile" value=".\logs\aspnetcore-
debug.log" />-->
        <!--<handlerSetting name="debugLevel" value="FILE,TRACE" />-->
      </handlerSettings>
    </aspNetCore>
  </system.webServer>
</configuration>
```

Shadow copying in IIS is an experimental feature that is not guaranteed to be part of ASP.NET Core. Please leave feedback on IIS Shadow copying in this GitHub issue.

Additional resources

- Code samples migrated to the new minimal hosting model in 6.0

- What's new in .NET 6

What's new in ASP.NET Core 5.0

Article • 11/01/2023

This article highlights the most significant changes in ASP.NET Core 5.0 with links to relevant documentation.

ASP.NET Core MVC and Razor improvements

Model binding DateTime as UTC

Model binding now supports binding UTC time strings to `DateTime`. If the request contains a UTC time string, model binding binds it to a UTC `DateTime`. For example, the following time string is bound the UTC `DateTime`:

`https://example.com/mycontroller/myaction?time=2019-06-14T02%3A30%3A04.0576719Z`

Model binding and validation with C# 9 record types

C# 9 record types can be used with model binding in an MVC controller or a Razor Page. Record types are a good way to model data being transmitted over the network.

For example, the following `PersonController` uses the `Person` record type with model binding and form validation:

C#

```
public record Person([Required] string Name, [Range(0, 150)] int Age);

public class PersonController
{
   public IActionResult Index() => View();

   [HttpPost]
   public IActionResult Index(Person person)
   {
          // ...
   }
}
```

The `Person/Index.cshtml` file:

CSHTML

```
@model Person

<label>Name: <input asp-for="Model.Name" /></label>
<span asp-validation-for="Model.Name" />

<label>Age: <input asp-for="Model.Age" /></label>
<span asp-validation-for="Model.Age" />
```

Improvements to DynamicRouteValueTransformer

ASP.NET Core 3.1 introduced DynamicRouteValueTransformer as a way to use custom endpoint to dynamically select an MVC controller action or a Razor page. ASP.NET Core 5.0 apps can pass state to a `DynamicRouteValueTransformer` and filter the set of endpoints chosen.

Miscellaneous

- The [Compare] attribute can be applied to properties on a Razor Page model.
- Parameters and properties bound from the body are considered required by default.

Web API

OpenAPI Specification on by default

OpenAPI Specification is an industry standard for describing HTTP APIs and integrating them into complex business processes or with third parties. OpenAPI is widely supported by all cloud providers and many API registries. Apps that emit OpenAPI documents from web APIs have a variety of new opportunities in which those APIs can be used. In partnership with the maintainers of the open-source project Swashbuckle.AspNetCore, the ASP.NET Core API template contains a NuGet dependency on Swashbuckle. Swashbuckle is a popular open-source NuGet package that emits OpenAPI documents dynamically. Swashbuckle does this by introspecting over the API controllers and generating the OpenAPI document at run-time, or at build time using the Swashbuckle CLI.

In ASP.NET Core 5.0, the web API templates enable the OpenAPI support by default. To disable OpenAPI:

- From the command line:

.NET CLI

```
dotnet new webapi --no-openapi true
```

- From Visual Studio: Uncheck **Enable OpenAPI support**.

All `.csproj` files created for web API projects contain the Swashbuckle.AspNetCore NuGet package reference.

XML

```
<ItemGroup>
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.5.1" />
</ItemGroup>
```

The template generated code contains code in `Startup.ConfigureServices` that activates OpenAPI document generation:

C#

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "WebApp1", Version =
"v1" });
    });
}
```

The `Startup.Configure` method adds the Swashbuckle middleware, which enables the:

- Document generation process.
- Swagger UI page by default in development mode.

The template generated code won't accidentally expose the API's description when publishing to production.

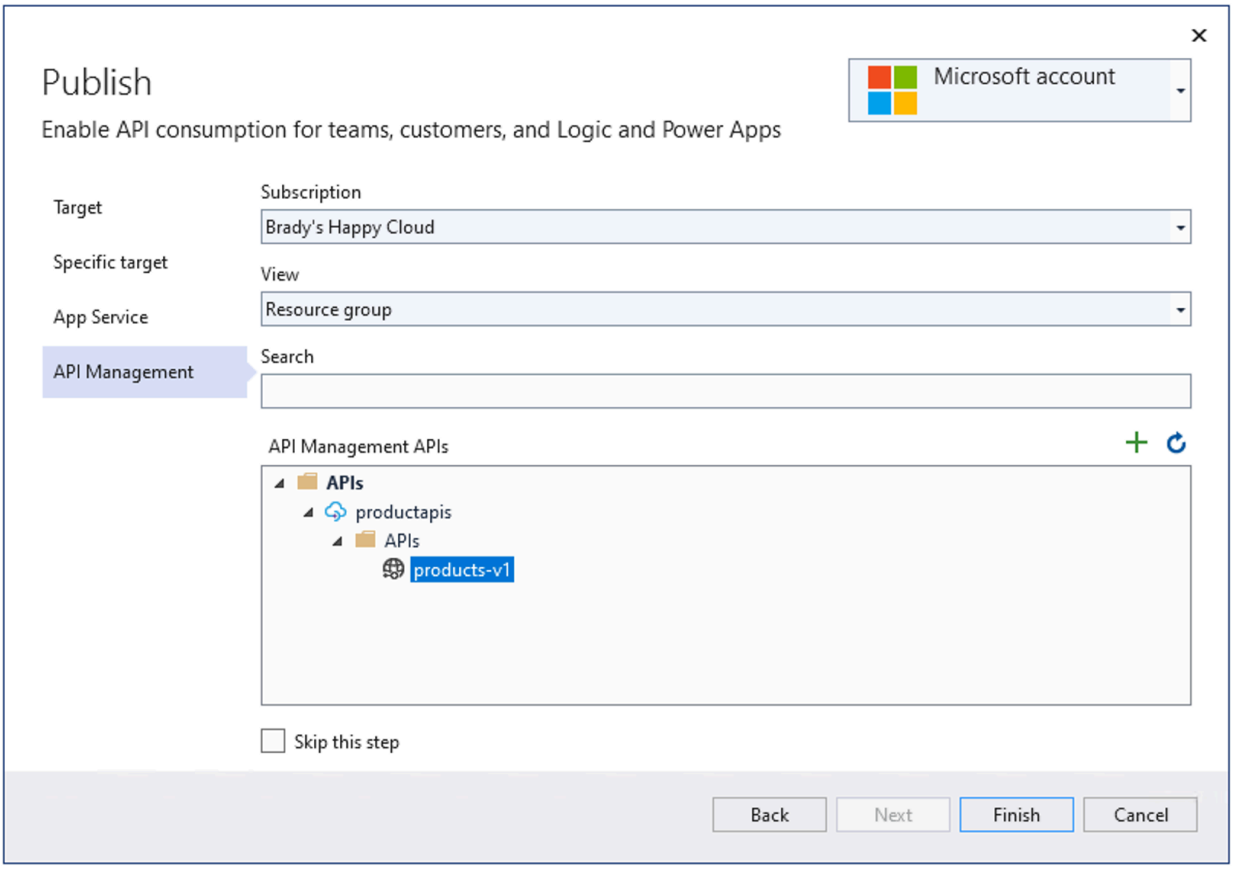
C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();  // UseSwaggerUI Protected by if
```

```
(env.IsDevelopment())
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
                            "WebApp1 v1"));
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

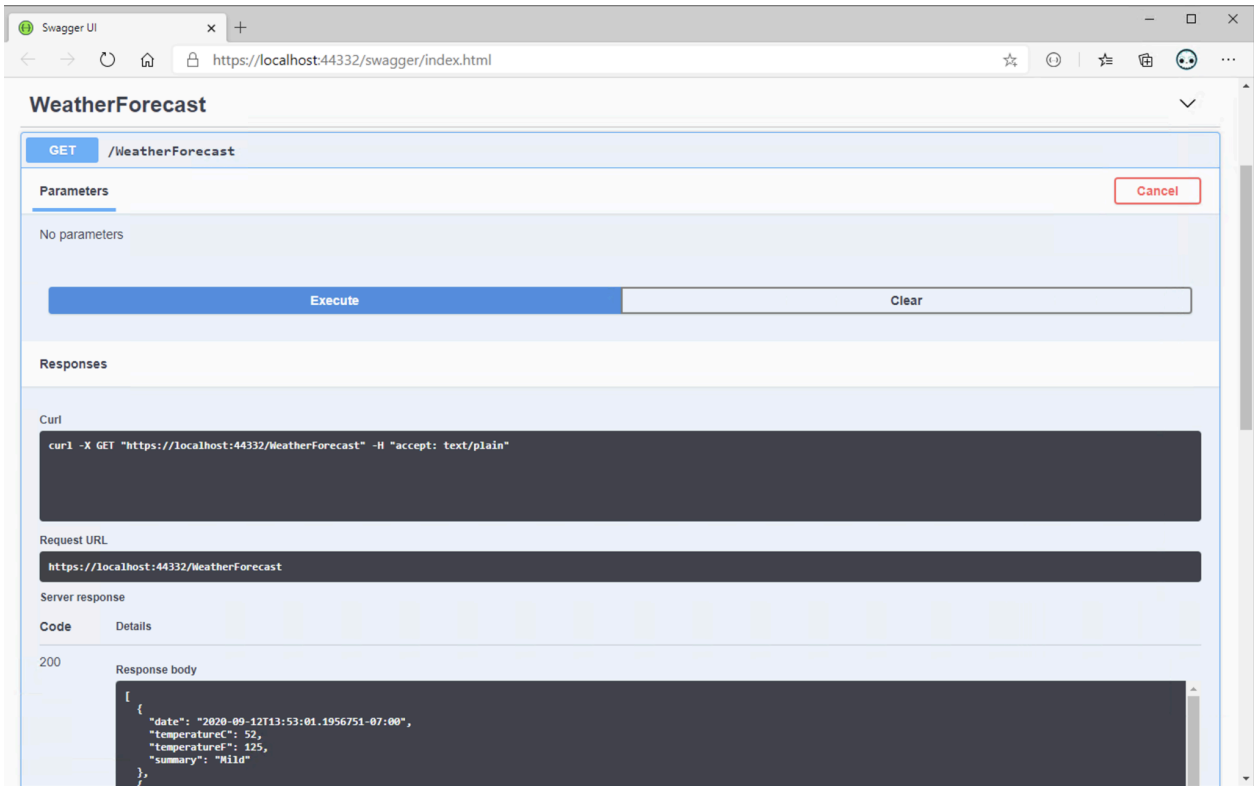
Azure API Management Import

When ASP.NET Core API projects enable OpenAPI, the Visual Studio 2019 version 16.8 and later publishing automatically offer an additional step in the publishing flow. Developers who use Azure API Management have an opportunity to automatically import the APIs into Azure API Management during the publish flow:



Better launch experience for web API projects

With OpenAPI enabled by default, the app launching experience (F5) for web API developers significantly improves. With ASP.NET Core 5.0, the web API template comes pre-configured to load up the Swagger UI page. The Swagger UI page provides both the documentation added for the published API, and enables testing the APIs with a single click.



Blazor

Performance improvements

For .NET 5, we made significant improvements to .NET WebAssembly runtime performance with a specific focus on complex UI rendering and JSON serialization. In our performance tests, Blazor WebAssembly in .NET 5 is two to three times faster for most scenarios. For more information, see ASP.NET Blog: ASP.NET Core updates in .NET 5 Release Candidate 1.

CSS isolation

Blazor now supports defining CSS styles that are scoped to a given component. Component-specific CSS styles make it easier to reason about the styles in an app and to avoid unintentional side effects of global styles. For more information, see ASP.NET Core Blazor CSS isolation.

New `InputFile` component

The `InputFile` component allows reading one or more files selected by a user for upload. For more information, see ASP.NET Core Blazor file uploads.

New `InputRadio` and `InputRadioGroup` components

Blazor has built-in `InputRadio` and `InputRadioGroup` components that simplify data binding to radio button groups with integrated validation. For more information, see ASP.NET Core Blazor input components.

Component virtualization

Improve the perceived performance of component rendering using the Blazor framework's built-in virtualization support. For more information, see ASP.NET Core Razor component virtualization.

`ontoggle` event support

Blazor events now support the `ontoggle` DOM event. For more information, see ASP.NET Core Blazor event handling.

Set UI focus in Blazor apps

Use the `FocusAsync` convenience method on element references to set the UI focus to that element. For more information, see ASP.NET Core Blazor event handling.

Custom validation CSS class attributes

Custom validation CSS class attributes are useful when integrating with CSS frameworks, such as Bootstrap. For more information, see ASP.NET Core Blazor forms validation.

IAsyncDisposable support

Razor components now support the IAsyncDisposable interface for the asynchronous release of allocated resources.

JavaScript isolation and object references

Blazor enables JavaScript isolation in standard JavaScript modules. For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Form components support display name

The following built-in components support display names with the `DisplayName` parameter:

- `InputDate`
- `InputNumber`
- `InputSelect`

For more information, see ASP.NET Core Blazor forms overview.

Catch-all route parameters

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components. For more information, see ASP.NET Core Blazor routing and navigation.

Debugging improvements

Debugging Blazor WebAssembly apps is improved in ASP.NET Core 5.0. Additionally, debugging is now supported on Visual Studio for Mac. For more information, see Debug ASP.NET Core Blazor apps.

Microsoft Identity v2.0 and MSAL v2.0

Blazor security now uses Microsoft Identity v2.0 (Microsoft.Identity.Web and Microsoft.Identity.Web.UI) and MSAL v2.0. For more information, see the topics in the Blazor Security and Identity node.

Protected Browser Storage for Blazor Server

Blazor Server apps can now use built-in support for storing app state in the browser that has been protected from tampering using ASP.NET Core data protection. Data can be stored in either local browser storage or session storage. For more information, see ASP.NET Core Blazor state management.

Blazor WebAssembly prerendering

Component integration is improved across hosting models, and Blazor WebAssembly apps can now prerender output on the server.

Trimming/linking improvements

Blazor WebAssembly performs Intermediate Language (IL) trimming/linking during a build to trim unnecessary IL from the app's output assemblies. With the release of ASP.NET Core 5.0, Blazor WebAssembly performs improved trimming with additional configuration options. For more information, see Configure the Trimmer for ASP.NET Core Blazor and Trimming options.

Browser compatibility analyzer

Blazor WebAssembly apps target the full .NET API surface area, but not all .NET APIs are supported on WebAssembly due to browser sandbox constraints. Unsupported APIs throw PlatformNotSupportedException when running on WebAssembly. A platform compatibility analyzer warns the developer when the app uses APIs that aren't supported by the app's target platforms. For more information, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Lazy load assemblies

Blazor WebAssembly app startup performance can be improved by deferring the loading of some application assemblies until they're required. For more information, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Updated globalization support

Globalization support is available for Blazor WebAssembly based on International Components for Unicode (ICU). For more information, see ASP.NET Core Blazor globalization and localization.

gRPC

Many preformance improvements have been made in gRPC. For more information, see gRPC performance improvements in .NET 5.

For more gRPC information, see Overview for gRPC on .NET.

SignalR

SignalR Hub filters

SignalR Hub filters, called Hub pipelines in ASP.NET SignalR, is a feature that allows code to run before and after Hub methods are called. Running code before and after Hub methods are called is similar to how middleware has the ability to run code before and after an HTTP request. Common uses include logging, error handling, and argument validation.

For more information, see Use hub filters in ASP.NET Core SignalR.

SignalR parallel hub invocations

ASP.NET Core SignalR is now capable of handling parallel hub invocations. The default behavior can be changed to allow clients to invoke more than one hub method at a time:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(options =>
    {
        options.MaximumParallelInvocationsPerClient = 5;
    });
}
```

Added Messagepack support in SignalR Java client

A new package, com.microsoft.signalr.messagepack, adds MessagePack support to the SignalR Java client. To use the MessagePack hub protocol, add `.withHubProtocol(new MessagePackHubProtocol())` to the connection builder:

Java

```
HubConnection hubConnection = HubConnectionBuilder.create(
                              "http://localhost:53353/MyHub")
                .withHubProtocol(new MessagePackHubProtocol())
                .build();
```

Kestrel

- Reloadable endpoints via configuration: Kestrel can detect changes to configuration passed to KestrelServerOptions.Configure and unbind from existing

endpoints and bind to new endpoints without requiring an application restart when the `reloadOnChange` parameter is `true`. By default when using ConfigureWebHostDefaults or CreateDefaultBuilder, Kestrel binds to the "Kestrel" configuration subsection with `reloadOnChange` enabled. Apps must pass `reloadOnChange: true` when calling `KestrelServerOptions.Configure` manually to get reloadable endpoints.

- HTTP/2 response headers improvements. For more information, see Performance improvements in the next section.

- Support for additional endpoints types in the sockets transport: Adding to the new API introduced in System.Net.Sockets, the sockets default transport in Kestrel allows binding to both existing file handles and Unix domain sockets. Support for binding to existing file handles enables using the existing `Systemd` integration without requiring the `libuv` transport.

- Custom header decoding in Kestrel: Apps can specify which Encoding to use to interpret incoming headers based on the header name instead of defaulting to UTF-8. Set the `Microsoft.AspNetCore.Server.Kestrel.KestrelServerOptions.RequestHeaderEncodingSelector` property to specify which encoding to use:

```
C#
```

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
          webBuilder.ConfigureKestrel(options =>
          {
              options.RequestHeaderEncodingSelector = encoding =>
              {
                    return encoding switch
                    {
                        "Host" => System.Text.Encoding.Latin1,
                        _ => System.Text.Encoding.UTF8,
                    };
              };
          });
          webBuilder.UseStartup<Startup>();
      });
```

Kestrel endpoint-specific options via configuration

Support has been added for configuring Kestrel's endpoint-specific options via configuration. The endpoint-specific configurations includes the:

- HTTP protocols used
- TLS protocols used
- Certificate selected
- Client certificate mode

Configuration allows specifying which certificate is selected based on the specified server name. The server name is part of the Server Name Indication (SNI) extension to the TLS protocol as indicated by the client. Kestrel's configuration also supports a wildcard prefix in the host name.

The following example shows how to specify endpoint-specific using a configuration file:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "EndpointName": {
        "Url": "https://*",
        "Sni": {
          "a.example.org": {
            "Protocols": "Http1AndHttp2",
            "SslProtocols": [ "Tls11", "Tls12"],
            "Certificate": {
              "Path": "testCert.pfx",
              "Password": "testPassword"
            },
            "ClientCertificateMode" : "NoCertificate"
          },
          "*.example.org": {
            "Certificate": {
              "Path": "testCert2.pfx",
              "Password": "testPassword"
            }
          },
          "*": {
            // At least one sub-property needs to exist per
            // SNI section or it cannot be discovered via
            // IConfiguration
            "Protocols": "Http1",
          }
        }
      }
    }
  }
}
```

Server Name Indication (SNI) is a TLS extension to include a virtual domain as a part of SSL negotiation. What this effectively means is that the virtual domain name, or a hostname, can be used to identify the network end point.

Performance improvements

HTTP/2

- Significant reductions in allocations in the HTTP/2 code path.
- Support for HPack dynamic compression of HTTP/2 response headers in Kestrel. For more information, see Header table size and HPACK: the silent killer (feature) of HTTP/2.
- Sending HTTP/2 PING frames: HTTP/2 has a mechanism for sending PING frames to ensure an idle connection is still functional. Ensuring a viable connection is especially useful when working with long-lived streams that are often idle but only intermittently see activity, for example, gRPC streams. Apps can send periodic PING frames in Kestrel by setting limits on KestrelServerOptions:

```
C#
```

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
      .ConfigureWebHostDefaults(webBuilder =>
      {
          webBuilder.ConfigureKestrel(options =>
          {
              options.Limits.Http2.KeepAlivePingInterval =
                                              TimeSpan.FromSeconds(10);
              options.Limits.Http2.KeepAlivePingTimeout =
                                              TimeSpan.FromSeconds(1);
          });
          webBuilder.UseStartup<Startup>();
      });
```

Containers

Prior to .NET 5.0, building and publishing a *Dockerfile* for an ASP.NET Core app required pulling the entire .NET Core SDK and the ASP.NET Core image. With this release, pulling the SDK images bytes is reduced and the bytes pulled for the ASP.NET Core image is largely eliminated. For more information, see this GitHub issue comment.

Authentication and authorization

Microsoft Entra ID authentication with Microsoft.Identity.Web

The ASP.NET Core project templates now integrate with Microsoft.Identity.Web to handle authentication with Microsoft Entra ID. The Microsoft.Identity.Web package provides:

- A better experience for authentication through Microsoft Entra ID.
- An easier way to access Azure resources on behalf of your users, including Microsoft Graph. See the Microsoft.Identity.Web sample, which starts with a basic login and advances through multi-tenancy, using Azure APIs, using Microsoft Graph, and protecting your own APIs. `Microsoft.Identity.Web` is available alongside .NET 5.

Allow anonymous access to an endpoint

The `AllowAnonymous` extension method allows anonymous access to an endpoint:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        })
        .AllowAnonymous();
    });
}
```

Custom handling of authorization failures

Custom handling of authorization failures is now easier with the new IAuthorizationMiddlewareResultHandler interface that is invoked by the authorization Middleware. The default implementation remains the same, but a custom handler can

be registered in [Dependency injection, which allows custom HTTP responses based on why authorization failed. See this sample that demonstrates usage of the `IAuthorizationMiddlewareResultHandler`.

Authorization when using endpoint routing

Authorization when using endpoint routing now receives the `HttpContext` rather than the endpoint instance. This allows the authorization middleware to access the `RouteData` and other properties of the `HttpContext` that were not accessible though the Endpoint class. The endpoint can be fetched from the context using context.GetEndpoint.

Role-based access control with Kerberos authentication and LDAP on Linux

See Kerberos authentication and role-based access control (RBAC)

API improvements

JSON extension methods for HttpRequest and HttpResponse

JSON data can be read and written to from an `HttpRequest` and `HttpResponse` using the new ReadFromJsonAsync and `WriteAsJsonAsync` extension methods. These extension methods use the System.Text.Json serializer to handle the JSON data. The new `HasJsonContentType` extension method can also check if a request has a JSON content type.

The JSON extension methods can be combined with endpoint routing to create JSON APIs in a style of programming we call ***route to code***. It is a new option for developers who want to create basic JSON APIs in a lightweight way. For example, a web app that has only a handful of endpoints might choose to use route to code rather than the full functionality of ASP.NET Core MVC:

C#

```
endpoints.MapGet("/weather/{city:alpha}", async context =>
{
    var city = (string)context.Request.RouteValues["city"];
    var weather = GetFromDatabase(city);
```

```
    await context.Response.WriteAsJsonAsync(weather);
});
```

System.Diagnostics.Activity

The default format for System.Diagnostics.Activity now defaults to the W3C format. This makes distributed tracing support in ASP.NET Core interoperable with more frameworks by default.

FromBodyAttribute

FromBodyAttribute now supports configuring an option that allows these parameters or properties to be considered optional:

C#

```
public IActionResult Post([FromBody(EmptyBodyBehavior =
EmptyBodyBehavior.Allow)]
                          MyModel model)
{
    ...
}
```

Miscellaneous improvements

We've started applying nullable annotations to ASP.NET Core assemblies. We plan to annotate most of the common public API surface of the .NET 5 framework.

Control Startup class activation

An additional UseStartup overload has been added that lets an app provide a factory method for controlling `Startup` class activation. Controlling `Startup` class activation is useful to pass additional parameters to `Startup` that are initialized along with the host:

C#

```
public class Program
{
    public static async Task Main(string[] args)
    {
        var logger = CreateLogger();
        var host = Host.CreateDefaultBuilder()
            .ConfigureWebHost(builder =>
```

```
            {
                builder.UseStartup(context => new Startup(logger));
            })
            .Build();

        await host.RunAsync();
    }
}
```

Auto refresh with dotnet watch

In .NET 5, running dotnet watch on an ASP.NET Core project both launches the default browser and auto refreshes the browser as changes are made to the code. This means you can:

- Open an ASP.NET Core project in a text editor.
- Run `dotnet watch`.
- Focus on the code changes while the tooling handles rebuilding, restarting, and reloading the app.

Console Logger Formatter

Improvements have been made to the console log provider in the `Microsoft.Extensions.Logging` library. Developers can now implement a custom `ConsoleFormatter` to exercise complete control over formatting and colorization of the console output. The formatter APIs allow for rich formatting by implementing a subset of the VT-100 escape sequences. VT-100 is supported by most modern terminals. The console logger can parse out escape sequences on unsupported terminals allowing developers to author a single formatter for all terminals.

JSON Console Logger

In addition to support for custom formatters, we've also added a built-in JSON formatter that emits structured JSON logs to the console. The following code shows how to switch from the default logger to JSON:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
           Host.CreateDefaultBuilder(args)
  .ConfigureLogging(logging =>
  {
     logging.AddJsonConsole(options =>
     {
```

```
            options.JsonWriterOptions = new JsonWriterOptions()
            { Indented = true };
        });
    })
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
    });
```

Log messages emitted to the console are JSON formatted:

JSON

```
{
  "EventId": 0,
  "LogLevel": "Information",
  "Category": "Microsoft.Hosting.Lifetime",
  "Message": "Now listening on: https://localhost:5001",
  "State": {
    "Message": "Now listening on: https://localhost:5001",
    "address": "https://localhost:5001",
    "{OriginalFormat}": "Now listening on: {address}"
  }
}
```

What's new in ASP.NET Core 3.1

Article • 09/25/2023

This article highlights the most significant changes in ASP.NET Core 3.1 with links to relevant documentation.

Partial class support for Razor components

Razor components are now generated as partial classes. Code for a Razor component can be written using a code-behind file defined as a partial class rather than defining all the code for the component in a single file. For more information, see Partial class support.

Component Tag Helper and pass parameters to top-level components

In Blazor with ASP.NET Core 3.0, components were rendered into pages and views using an HTML Helper (`Html.RenderComponentAsync`). In ASP.NET Core 3.1, render a component from a page or view with the new Component Tag Helper:

CSHTML

```
<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

The HTML Helper remains supported in ASP.NET Core 3.1, but the Component Tag Helper is recommended.

Blazor Server apps can now pass parameters to top-level components during the initial render. Previously you could only pass parameters to a top-level component with RenderMode.Static. With this release, both RenderMode.Server and RenderMode.ServerPrerendered are supported. Any specified parameter values are serialized as JSON and included in the initial response.

For example, prerender a `Counter` component with an increment amount (`IncrementAmount`):

CSHTML

```
<component type="typeof(Counter)" render-mode="ServerPrerendered"
    param-IncrementAmount="10" />
```

For more information, see Integrate components into Razor Pages and MVC apps.

Support for shared queues in HTTP.sys

HTTP.sys supports creating anonymous request queues. In ASP.NET Core 3.1, we've added the ability to create or attach to an existing named HTTP.sys request queue. Creating or attaching to an existing named HTTP.sys request queue enables scenarios where the HTTP.sys controller process that owns the queue is independent of the listener process. This independence makes it possible to preserve existing connections and enqueued requests between listener process restarts:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
     Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            // ...
            webBuilder.UseHttpSys(options =>
            {
                options.RequestQueueName = "MyExistingQueue";
                options.RequestQueueMode = RequestQueueMode.CreateOrAttach;
            });
        });
```

Breaking changes for SameSite cookies

The behavior of SameSite cookies has changed to reflect upcoming browser changes. This may affect authentication scenarios like AzureAd, OpenIdConnect, or WsFederation. For more information, see Work with SameSite cookies in ASP.NET Core.

Prevent default actions for events in Blazor apps

Use the `@on{EVENT}:preventDefault` directive attribute to prevent the default action for an event. In the following example, the default action of displaying the key's character in the text box is prevented:

razor

```
<input value="@_count" @onkeypress="KeyHandler" @onkeypress:preventDefault
/>
```

For more information, see Prevent default actions.

Stop event propagation in Blazor apps

Use the `@on{EVENT}:stopPropagation` directive attribute to stop event propagation. In the following example, selecting the checkbox prevents click events from the child `<div>` from propagating to the parent `<div>`:

razor

```
<input @bind="_stopPropagation" type="checkbox" />

<div @onclick="OnSelectParentDiv">
    <div @onclick="OnSelectChildDiv"
@onclick:stopPropagation="_stopPropagation">
        ...
    </div>
</div>

@code {
    private bool _stopPropagation = false;
}
```

For more information, see Stop event propagation.

Detailed errors during Blazor app development

When a Blazor app isn't functioning properly during development, receiving detailed error information from the app assists in troubleshooting and fixing the issue. When an error occurs, Blazor apps display a gold bar at the bottom of the screen:

- During development, the gold bar directs you to the browser console, where you can see the exception.
- In production, the gold bar notifies the user that an error has occurred and recommends refreshing the browser.

For more information, see Handle errors in ASP.NET Core Blazor apps.

What's new in ASP.NET Core 3.0

Article • 09/27/2023

This article highlights the most significant changes in ASP.NET Core 3.0 with links to relevant documentation.

Blazor

Blazor is a new framework in ASP.NET Core for building interactive client-side web UI with .NET:

- Create rich interactive UIs using C#.
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.

Blazor framework supported scenarios:

- Reusable UI components (Razor components)
- Client-side routing
- Component layouts
- Support for dependency injection
- Forms and validation
- Supply Razor components in Razor class libraries
- JavaScript interop

For more information, see ASP.NET Core Blazor.

Blazor Server

Blazor decouples component rendering logic from how UI updates are applied. Blazor Server provides support for hosting Razor components on the server in an ASP.NET Core app. UI updates are handled over a SignalR connection. Blazor Server is supported in ASP.NET Core 3.0.

Blazor WebAssembly (Preview)

Blazor apps can also be run directly in the browser using a WebAssembly-based .NET runtime. Blazor WebAssembly is in preview and *not* supported in ASP.NET Core 3.0. Blazor WebAssembly will be supported in a future release of ASP.NET Core.

Razor components

Blazor apps are built from components. Components are self-contained chunks of user interface (UI), such as a page, dialog, or form. Components are normal .NET classes that define UI rendering logic and client-side event handlers. You can create rich interactive web apps without JavaScript.

Components in Blazor are typically authored using Razor syntax, a natural blend of HTML and C#. Razor components are similar to Razor Pages and MVC views in that they both use Razor. Unlike pages and views, which are based on a request-response model, components are used specifically for handling UI composition.

gRPC

gRPC:

- Is a popular, high-performance RPC (remote procedure call) framework.
- Offers an opinionated contract-first approach to API development.
- Uses modern technologies such as:
  - HTTP/2 for transport.
  - Protocol Buffers as the interface description language.
  - Binary serialization format.
- Provides features such as:
  - Authentication
  - Bidirectional streaming and flow control.
  - Cancellation and timeouts.

gRPC functionality in ASP.NET Core 3.0 includes:

- Grpc.AspNetCore: An ASP.NET Core framework for hosting gRPC services. gRPC on ASP.NET Core integrates with standard ASP.NET Core features like logging, dependency injection (DI), authentication, and authorization.
- Grpc.Net.Client: A gRPC client for .NET Core that builds upon the familiar `HttpClient`.
- Grpc.Net.ClientFactory: gRPC client integration with `HttpClientFactory`.

For more information, see Overview for gRPC on .NET.

SignalR

See Update SignalR code for migration instructions. SignalR now uses `System.Text.Json` to serialize/deserialize JSON messages. See Switch to Newtonsoft.Json for instructions to restore the `Newtonsoft.Json`-based serializer.

In the JavaScript and .NET Clients for SignalR, support was added for automatic reconnection. By default, the client tries to reconnect immediately and retry after 2, 10, and 30 seconds if necessary. If the client successfully reconnects, it receives a new connection ID. Automatic reconnect is opt-in:

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

The reconnection intervals can be specified by passing an array of millisecond-based durations:

JavaScript

```
.withAutomaticReconnect([0, 3000, 5000, 10000, 15000, 30000])
//.withAutomaticReconnect([0, 2000, 10000, 30000]) The default intervals.
```

A custom implementation can be passed in for full control of the reconnection intervals.

If the reconnection fails after the last reconnect interval:

- The client considers the connection is offline.
- The client stops trying to reconnect.

During reconnection attempts, update the app UI to notify the user that the reconnection is being attempted.

To provide UI feedback when the connection is interrupted, the SignalR client API has been expanded to include the following event handlers:

- `onreconnecting`: Gives developers an opportunity to disable UI or to let users know the app is offline.
- `onreconnected`: Gives developers an opportunity to update the UI once the connection is reestablished.

The following code uses `onreconnecting` to update the UI while trying to connect:

JavaScript

```
connection.onreconnecting((error) => {
    const status = `Connection lost due to error "${error}". Reconnecting.`;
    document.getElementById("messageInput").disabled = true;
    document.getElementById("sendButton").disabled = true;
    document.getElementById("connectionStatus").innerText = status;
});
```

The following code uses `onreconnected` to update the UI on connection:

JavaScript

```
connection.onreconnected((connectionId) => {
    const status = `Connection reestablished. Connected.`;
    document.getElementById("messageInput").disabled = false;
    document.getElementById("sendButton").disabled = false;
    document.getElementById("connectionStatus").innerText = status;
});
```

SignalR 3.0 and later provides a custom resource to authorization handlers when a hub method requires authorization. The resource is an instance of `HubInvocationContext`. The `HubInvocationContext` includes the:

- `HubCallerContext`
- Name of the hub method being invoked.
- Arguments to the hub method.

Consider the following example of a chat room app allowing multiple organization sign-in via Azure Active Directory. Anyone with a Microsoft account can sign in to chat, but only members of the owning organization can ban users or view users' chat histories. The app could restrict certain functionality from specific users.

C#

```
public class DomainRestrictedRequirement :
    AuthorizationHandler<DomainRestrictedRequirement, HubInvocationContext>,
    IAuthorizationRequirement
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
        DomainRestrictedRequirement requirement,
        HubInvocationContext resource)
    {
        if (context.User?.Identity?.Name == null)
        {
            return Task.CompletedTask;
        }
```

```
        if (IsUserAllowedToDoThis(resource.HubMethodName,
context.User.Identity.Name))
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }

    private bool IsUserAllowedToDoThis(string hubMethodName, string
currentUsername)
    {
        if (hubMethodName.Equals("banUser",
StringComparison.OrdinalIgnoreCase))
        {
            return currentUsername.Equals("bob42@jabbr.net",
StringComparison.OrdinalIgnoreCase);
        }

        return currentUsername.EndsWith("@jabbr.net",
StringComparison.OrdinalIgnoreCase));
    }
}
```

In the preceding code, `DomainRestrictedRequirement` serves as a custom `IAuthorizationRequirement`. Because the `HubInvocationContext` resource parameter is being passed in, the internal logic can:

- Inspect the context in which the Hub is being called.
- Make decisions on allowing the user to execute individual Hub methods.

Individual Hub methods can be marked with the name of the policy the code checks at run-time. As clients attempt to call individual Hub methods, the `DomainRestrictedRequirement` handler runs and controls access to the methods. Based on the way the `DomainRestrictedRequirement` controls access:

- All logged-in users can call the `SendMessage` method.
- Only users who have logged in with a `@jabbr.net` email address can view users' histories.
- Only `bob42@jabbr.net` can ban users from the chat room.

C#

```
[Authorize]
public class ChatHub : Hub
{
    public void SendMessage(string message)
    {
    }
```

```
    [Authorize("DomainRestricted")]
    public void BanUser(string username)
    {
    }

    [Authorize("DomainRestricted")]
    public void ViewUserHistory(string username)
    {
    }
}
```

Creating the `DomainRestricted` policy might involve:

- In `Startup.cs`, adding the new policy.
- Provide the custom `DomainRestrictedRequirement` requirement as a parameter.
- Registering `DomainRestricted` with the authorization middleware.

C#

```
services
    .AddAuthorization(options =>
    {
        options.AddPolicy("DomainRestricted", policy =>
        {
            policy.Requirements.Add(new DomainRestrictedRequirement());
        });
    });
```

SignalR hubs use Endpoint Routing. SignalR hub connection was previously done explicitly:

C#

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
});
```

In the previous version, developers needed to wire up controllers, Razor pages, and hubs in a variety of places. Explicit connection results in a series of nearly-identical routing segments:

C#

```
app.UseSignalR(routes =>
{
    routes.MapHub<ChatHub>("hubs/chat");
```

```
});

app.UseRouting(routes =>
{
    routes.MapRazorPages();
});
```

SignalR 3.0 hubs can be routed via endpoint routing. With endpoint routing, typically all routing can be configured in `UseRouting`:

C#

```
app.UseRouting(routes =>
{
    routes.MapRazorPages();
    routes.MapHub<ChatHub>("hubs/chat");
});
```

ASP.NET Core 3.0 SignalR added:

Client-to-server streaming. With client-to-server streaming, server-side methods can take instances of either an `IAsyncEnumerable<T>` or `ChannelReader<T>`. In the following C# sample, the `UploadStream` method on the Hub will receive a stream of strings from the client:

C#

```
public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        // process content
    }
}
```

.NET client apps can pass either an `IAsyncEnumerable<T>` or `ChannelReader<T>` instance as the `stream` argument of the `UploadStream` Hub method above.

After the `for` loop has completed and the local function exits, the stream completion is sent:

C#

```
async IAsyncEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
```

```
        var data = await FetchSomeData();
        yield return data;
    }
}

await connection.SendAsync("UploadStream", clientStreamData());
```

JavaScript client apps use the SignalR `Subject` (or an RxJS Subject) for the `stream` argument of the `UploadStream` Hub method above.

JavaScript

```
let subject = new signalR.Subject();
await connection.send("StartStream", "MyAsciiArtStream", subject);
```

The JavaScript code could use the `subject.next` method to handle strings as they are captured and ready to be sent to the server.

JavaScript

```
subject.next("example");
subject.complete();
```

Using code like the two preceding snippets, real-time streaming experiences can be created.

New JSON serialization

ASP.NET Core 3.0 now uses System.Text.Json by default for JSON serialization:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically higher performance than `Newtonsoft.Json`.

To add Json.NET to ASP.NET Core 3.0, see Add Newtonsoft.Json-based JSON format support.

New Razor directives

The following list contains new Razor directives:

- @attribute: The `@attribute` directive applies the given attribute to the class of the generated page or view. For example, `@attribute [Authorize]`.

- @implements: The `@implements` directive implements an interface for the generated class. For example, `@implements IDisposable`.

IdentityServer4 supports authentication and authorization for web APIs and SPAs

ASP.NET Core 3.0 offers authentication in Single Page Apps (SPAs) using the support for web API authorization. ASP.NET Core Identity for authenticating and storing users is combined with IdentityServer4 for implementing OpenID Connect.

IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core 3.0. It enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

For more information, see the IdentityServer4 documentation or Authentication and authorization for SPAs.

Certificate and Kerberos authentication

Certificate authentication requires:

- Configuring the server to accept certificates.
- Adding the authentication middleware in `Startup.Configure`.
- Adding the certificate authentication service in `Startup.ConfigureServices`.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
            .AddCertificate();
    // Other service configuration removed.
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Options for certificate authentication include the ability to:

- Accept self-signed certificates.
- Check for certificate revocation.
- Check that the proffered certificate has the right usage flags in it.

A default user principal is constructed from the certificate properties. The user principal contains an event that enables supplementing or replacing the principal. For more information, see Configure certificate authentication in ASP.NET Core.

Windows Authentication has been extended onto Linux and macOS. In previous versions, Windows Authentication was limited to IIS and HTTP.sys. In ASP.NET Core 3.0, Kestrel has the ability to use Negotiate, Kerberos, and NTLM on Windows, Linux, and macOS for Windows domain-joined hosts. Kestrel support of these authentication schemes is provided by the Microsoft.AspNetCore.Authentication.Negotiate NuGet package. As with the other authentication services, configure authentication app wide, then configure the service:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
        .AddNegotiate();
    // Other service configuration removed.
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
    // Other app configuration removed.
}
```

Host requirements:

- Windows hosts must have Service Principal Names (SPNs) added to the user account hosting the app.
- Linux and macOS machines must be joined to the domain.
  - SPNs must be created for the web process.
  - Keytab files must be generated and configured on the host machine.

For more information, see Configure Windows Authentication in ASP.NET Core.

Template changes

The web UI templates (Razor Pages, MVC with controller and views) have the following removed:

- The cookie consent UI is no longer included. To enable the cookie consent feature in an ASP.NET Core 3.0 template-generated app, see General Data Protection Regulation (GDPR) support in ASP.NET Core.
- Scripts and related static assets are now referenced as local files instead of using CDNs. For more information, see Scripts and related static assets are now referenced as local files instead of using CDNs based on the current environment (dotnet/AspNetCore.Docs #14350).

The Angular template updated to use Angular 8.

The Razor class library (RCL) template defaults to Razor component development by default. A new template option in Visual Studio provides template support for pages and views. When creating an RCL from the template in a command shell, pass the `--support-pages-and-views` option (`dotnet new razorclasslib --support-pages-and-views`).

Generic Host

The ASP.NET Core 3.0 templates use .NET Generic Host in ASP.NET Core. Previous versions used WebHostBuilder. Using the .NET Core Generic Host (HostBuilder) provides better integration of ASP.NET Core apps with other server scenarios that aren't web-specific. For more information, see HostBuilder replaces WebHostBuilder.

Host configuration

Prior to the release of ASP.NET Core 3.0, environment variables prefixed with `ASPNETCORE_` were loaded for host configuration of the Web Host. In 3.0, `AddEnvironmentVariables` is used to load environment variables prefixed with `DOTNET_` for host configuration with `CreateDefaultBuilder`.

Changes to Startup constructor injection

The Generic Host only supports the following types for `Startup` constructor injection:

- IHostEnvironment
- `IWebHostEnvironment`
- IConfiguration

All services can still be injected directly as arguments to the `Startup.Configure` method. For more information, see Generic Host restricts Startup constructor injection (aspnet/Announcements #353).

Kestrel

- Kestrel configuration has been updated for the migration to the Generic Host. In 3.0, Kestrel is configured on the web host builder provided by `ConfigureWebHostDefaults`.
- Connection Adapters have been removed from Kestrel and replaced with Connection Middleware, which is similar to HTTP Middleware in the ASP.NET Core pipeline but for lower-level connections.
- The Kestrel transport layer has been exposed as a public interface in `Connections.Abstractions`.
- Ambiguity between headers and trailers has been resolved by moving trailing headers to a new collection.
- Synchronous I/O APIs, such as `HttpRequest.Body.Read`, are a common source of thread starvation leading to app crashes. In 3.0, `AllowSynchronousIO` is disabled by default.

For more information, see Migrate from ASP.NET Core 2.2 to 3.0.

HTTP/2 enabled by default

HTTP/2 is enabled by default in Kestrel for HTTPS endpoints. HTTP/2 support for IIS or HTTP.sys is enabled when supported by the operating system.

EventCounters on request

The Hosting EventSource, `Microsoft.AspNetCore.Hosting`, emits the following new EventCounter types related to incoming requests:

- `requests-per-second`
- `total-requests`
- `current-requests`
- `failed-requests`

Endpoint routing

Endpoint Routing, which allows frameworks (for example, MVC) to work well with middleware, is enhanced:

- The order of middleware and endpoints is configurable in the request processing pipeline of `Startup.Configure`.
- Endpoints and middleware compose well with other ASP.NET Core-based technologies, such as Health Checks.
- Endpoints can implement a policy, such as CORS or authorization, in both middleware and MVC.
- Filters and attributes can be placed on methods in controllers.

For more information, see Routing in ASP.NET Core.

Health Checks

Health Checks use endpoint routing with the Generic Host. In `Startup.Configure`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

```
C#

app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
});
```

Health Checks endpoints can:

- Specify one or more permitted hosts/ports.
- Require authorization.
- Require CORS.

For more information, see the following articles:

- Migrate from ASP.NET Core 2.2 to 3.0
- Health checks in ASP.NET Core

Pipes on HttpContext

It's now possible to read the request body and write the response body using the System.IO.Pipelines API. The `HttpRequest.BodyReader` property provides a PipeReader that can be used to read the request body. The `HttpResponse.BodyWriter` property provides a PipeWriter that can be used to write the response body.

`HttpRequest.BodyReader` is an analogue of the `HttpRequest.Body` stream.
`HttpResponse.BodyWriter` is an analogue of the `HttpResponse.Body` stream.

Improved error reporting in IIS

Startup errors when hosting ASP.NET Core apps in IIS now produce richer diagnostic data. These errors are reported to the Windows Event Log with stack traces wherever applicable. In addition, all warnings, errors, and unhandled exceptions are logged to the Windows Event Log.

Worker Service and Worker SDK

.NET Core 3.0 introduces the new Worker Service app template. This template provides a starting point for writing long running services in .NET Core.

For more information, see:

- .NET Core Workers as Windows Services
- Background tasks with hosted services in ASP.NET Core
- Host ASP.NET Core in a Windows Service

Forwarded Headers Middleware improvements

In previous versions of ASP.NET Core, calling UseHsts and UseHttpsRedirection were problematic when deployed to an Azure Linux or behind any reverse proxy other than IIS. The fix for previous versions is documented in Forward the scheme for Linux and non-IIS reverse proxies.

This scenario is fixed in ASP.NET Core 3.0. The host enables the Forwarded Headers Middleware when the `ASPNETCORE_FORWARDEDHEADERS_ENABLED` environment variable is set to `true`. `ASPNETCORE_FORWARDEDHEADERS_ENABLED` is set to `true` in our container images.

Performance improvements

ASP.NET Core 3.0 includes many improvements that reduce memory usage and improve throughput:

- Reduction in memory usage when using the built-in dependency injection container for scoped services.

- Reduction in allocations across the framework, including middleware scenarios and routing.
- Reduction in memory usage for WebSocket connections.
- Memory reduction and throughput improvements for HTTPS connections.
- New optimized and fully asynchronous JSON serializer.
- Reduction in memory usage and throughput improvements in form parsing.

ASP.NET Core 3.0 only runs on .NET Core 3.0

As of ASP.NET Core 3.0, .NET Framework is no longer a supported target framework. Projects targeting .NET Framework can continue in a fully supported fashion using the .NET Core 2.1 LTS release. Most ASP.NET Core 2.1.x related packages will be supported indefinitely, beyond the three-year LTS period for .NET Core 2.1.

For migration information, see Port your code from .NET Framework to .NET Core.

Use the ASP.NET Core shared framework

The ASP.NET Core 3.0 shared framework, contained in the Microsoft.AspNetCore.App metapackage, no longer requires an explicit `<PackageReference />` element in the project file. The shared framework is automatically referenced when using the `Microsoft.NET.Sdk.Web` SDK in the project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

Assemblies removed from the ASP.NET Core shared framework

The most notable assemblies removed from the ASP.NET Core 3.0 shared framework are:

- Newtonsoft.Json (Json.NET). To add Json.NET to ASP.NET Core 3.0, see Add Newtonsoft.Json-based JSON format support. ASP.NET Core 3.0 introduces `System.Text.Json` for reading and writing JSON. For more information, see New JSON serialization in this document.
- Entity Framework Core

For a complete list of assemblies removed from the shared framework, see Assemblies being removed from Microsoft.AspNetCore.App 3.0. For more information on the motivation for this change, see Breaking changes to Microsoft.AspNetCore.App in 3.0 and A first look at changes coming in ASP.NET Core 3.0.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

What's new in ASP.NET Core 2.2

Article • 06/03/2022

This article highlights the most significant changes in ASP.NET Core 2.2, with links to relevant documentation.

OpenAPI Analyzers & Conventions

OpenAPI (formerly known as Swagger) is a language-agnostic specification for describing REST APIs. The OpenAPI ecosystem has tools that allow for discovering, testing, and producing client code using the specification. Support for generating and visualizing OpenAPI documents in ASP.NET Core MVC is provided via community driven projects such as NSwag and Swashbuckle.AspNetCore. ASP.NET Core 2.2 provides improved tooling and runtime experiences for creating OpenAPI documents.

For more information, see the following resources:

- Use web API analyzers
- Use web API conventions
- ASP.NET Core 2.2.0-preview1: OpenAPI Analyzers & Conventions

Problem details support

ASP.NET Core 2.1 introduced `ProblemDetails`, based on the RFC 7807 specification for carrying details of an error with an HTTP Response. In 2.2, `ProblemDetails` is the standard response for client error codes in controllers attributed with `ApiControllerAttribute`. An `IActionResult` returning a client error status code (4xx) now returns a `ProblemDetails` body. The result also includes a correlation ID that can be used to correlate the error using request logs. For client errors, `ProducesResponseType` defaults to using `ProblemDetails` as the response type. This is documented in OpenAPI / Swagger output generated using NSwag or Swashbuckle.AspNetCore.

Endpoint Routing

ASP.NET Core 2.2 uses a new *endpoint routing* system for improved dispatching of requests. The changes include new link generation API members and route parameter transformers.

For more information, see the following resources:

- Endpoint routing in 2.2
- Route parameter transformers (see **Routing** section)
- Differences between IRouter- and endpoint-based routing

Health checks

A new health checks service makes it easier to use ASP.NET Core in environments that require health checks, such as Kubernetes. Health checks includes middleware and a set of libraries that define an `IHealthCheck` abstraction and service.

Health checks are used by a container orchestrator or load balancer to quickly determine if a system is responding to requests normally. A container orchestrator might respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might respond to a health check by routing traffic away from the failing instance of the service.

Health checks are exposed by an application as an HTTP endpoint used by monitoring systems. Health checks can be configured for a variety of real-time monitoring scenarios and monitoring systems. The health checks service integrates with the BeatPulse project . which makes it easier to add checks for dozens of popular systems and dependencies.

For more information, see Health checks in ASP.NET Core.

HTTP/2 in Kestrel

ASP.NET Core 2.2 adds support for HTTP/2.

HTTP/2 is a major revision of the HTTP protocol. Notable features of HTTP/2 include:

- Support for header compression.
- Fully multiplexed streams over a single connection.

While HTTP/2 preserves HTTP's semantics (for example, HTTP headers and methods), it's a breaking change from HTTP/1.x on how data is framed and sent between the client and server.

As a consequence of this change in framing, servers and clients need to negotiate the protocol version used. Application-Layer Protocol Negotiation (ALPN) is a TLS extension that allows the server and client to negotiate the protocol version used as part of their TLS handshake. While it is possible to have prior knowledge between the server and the

client on the protocol, all major browsers support ALPN as the only way to establish an HTTP/2 connection.

For more information, see HTTP/2 support.

Kestrel configuration

In earlier versions of ASP.NET Core, Kestrel options are configured by calling `UseKestrel`. In 2.2, Kestrel options are configured by calling `ConfigureKestrel` on the host builder. This change resolves an issue with the order of `IServer` registrations for in-process hosting. For more information, see the following resources:

- Mitigate UseIIS conflict
- Configure Kestrel server options with ConfigureKestrel

IIS in-process hosting

In earlier versions of ASP.NET Core, IIS serves as a reverse proxy. In 2.2, the ASP.NET Core Module can boot the CoreCLR and host an app inside the IIS worker process (*w3wp.exe*). In-process hosting provides performance and diagnostic gains when running with IIS.

For more information, see in-process hosting for IIS.

SignalR Java client

ASP.NET Core 2.2 introduces a Java Client for SignalR. This client supports connecting to an ASP.NET Core SignalR Server from Java code, including Android apps.

For more information, see ASP.NET Core SignalR Java client.

CORS improvements

In earlier versions of ASP.NET Core, CORS Middleware allows `Accept`, `Accept-Language`, `Content-Language`, and `Origin` headers to be sent regardless of the values configured in `CorsPolicy.Headers`. In 2.2, a CORS Middleware policy match is only possible when the headers sent in `Access-Control-Request-Headers` exactly match the headers stated in `WithHeaders`.

For more information, see CORS Middleware.

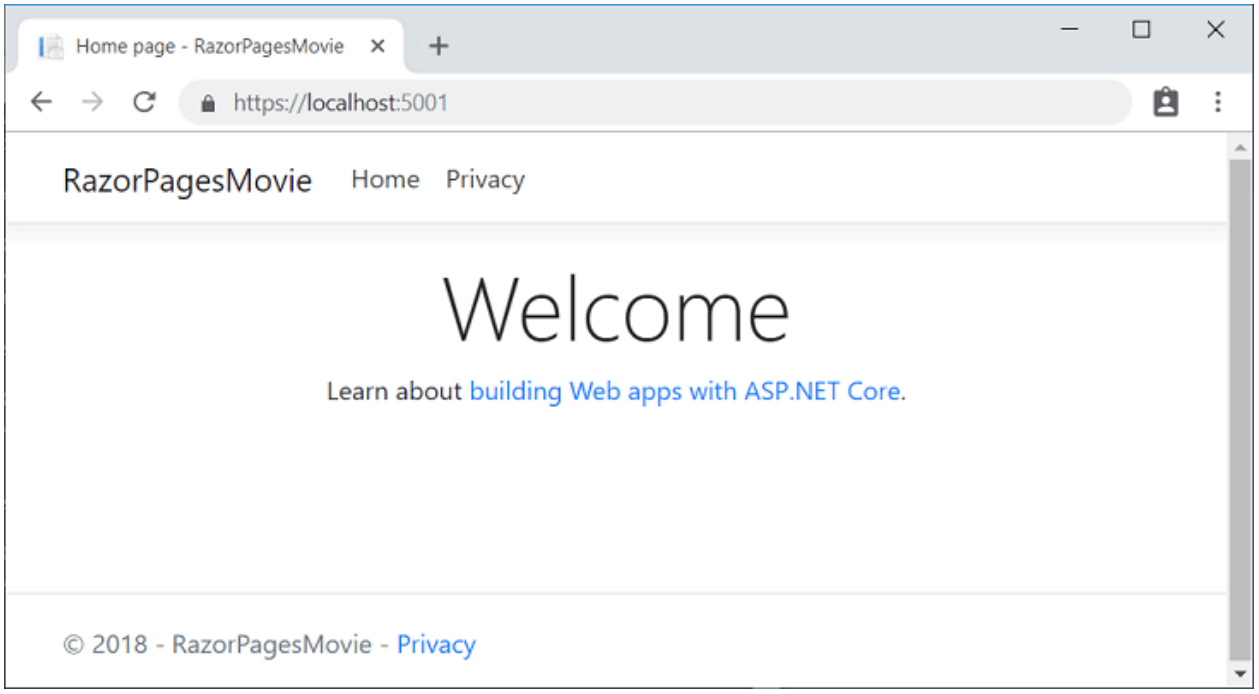
Response compression

ASP.NET Core 2.2 can compress responses with the Brotli compression format.

For more information, see Response Compression Middleware supports Brotli compression.

Project templates

ASP.NET Core web project templates were updated to Bootstrap 4 and Angular 6. The new look is visually simpler and makes it easier to see the important structures of the app.



Validation performance

MVC's validation system is designed to be extensible and flexible, allowing you to determine on a per request basis which validators apply to a given model. This is great for authoring complex validation providers. However, in the most common case an application only uses the built-in validators and don't require this extra flexibility. Built-in validators include DataAnnotations such as [Required] and [StringLength], and `IValidatableObject`.

In ASP.NET Core 2.2, MVC can short-circuit validation if it determines that a given model graph doesn't require validation. Skipping validation results in significant improvements when validating models that can't or don't have any validators. This includes objects

such as collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`), or complex object graphs without many validators.

HTTP Client performance

In ASP.NET Core 2.2, the performance of `SocketsHttpHandler` was improved by reducing connection pool locking contention. For apps that make many outgoing HTTP requests, such as some microservices architectures, throughput is improved. Under load, `HttpClient` throughput can be improved by up to 60% on Linux and 20% on Windows.

For more information, see the pull request that made this improvement.

Additional information

For the complete list of changes, see the ASP.NET Core 2.2 Release Notes.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

What's new in ASP.NET Core 2.1

Article • 02/07/2023

This article highlights the most significant changes in ASP.NET Core 2.1, with links to relevant documentation.

SignalR

SignalR has been rewritten for ASP.NET Core 2.1.

ASP.NET Core SignalR includes a number of improvements:

- A simplified scale-out model.
- A new JavaScript client with no jQuery dependency.
- A new compact binary protocol based on MessagePack.
- Support for custom protocols.
- A new streaming response model.
- Support for clients based on bare WebSockets.

For more information, see ASP.NET Core SignalR.

Razor class libraries

ASP.NET Core 2.1 makes it easier to build and include Razor-based UI in a library and share it across multiple projects. The new Razor SDK enables building Razor files into a class library project that can be packaged into a NuGet package. Views and pages in libraries are automatically discovered and can be overridden by the app. By integrating Razor compilation into the build:

- The app startup time is significantly faster.
- Fast updates to Razor views and pages at runtime are still available as part of an iterative development workflow.

For more information, see Reusable Razor UI in class libraries with ASP.NET Core.

Identity UI library & scaffolding

ASP.NET Core 2.1 provides ASP.NET Core Identity as a Razor class library. Apps that include Identity can apply the new Identity scaffolder to selectively add the source code contained in the Identity Razor class library (RCL). You might want to generate source

code so you can modify the code and change the behavior. For example, you could instruct the scaffolder to generate the code used in registration. Generated code takes precedence over the same code in the Identity RCL.

Apps that do **not** include authentication can apply the Identity scaffolder to add the RCL Identity package. You have the option of selecting Identity code to be generated.

For more information, see Scaffold Identity in ASP.NET Core projects.

HTTPS

With the increased focus on security and privacy, enabling HTTPS for web apps is important. HTTPS enforcement is becoming increasingly strict on the web. Sites that don't use HTTPS are considered insecure. Browsers (Chromium, Mozilla) are starting to enforce that web features must be used from a secure context. GDPR requires the use of HTTPS to protect user privacy. While using HTTPS in production is critical, using HTTPS in development can help prevent issues in deployment (for example, insecure links). ASP.NET Core 2.1 includes a number of improvements that make it easier to use HTTPS in development and to configure HTTPS in production. For more information, see Enforce HTTPS.

On by default

To facilitate secure website development, HTTPS is now enabled by default. Starting in 2.1, Kestrel listens on `https://localhost:5001` when a local development certificate is present. A development certificate is created:

- As part of the .NET Core SDK first-run experience, when you use the SDK for the first time.
- Manually using the new `dev-certs` tool.

Run `dotnet dev-certs https --trust` to trust the certificate.

HTTPS redirection and enforcement

Web apps typically need to listen on both HTTP and HTTPS, but then redirect all HTTP traffic to HTTPS. In 2.1, specialized HTTPS redirection middleware that intelligently redirects based on the presence of configuration or bound server ports has been introduced.

Use of HTTPS can be further enforced using HTTP Strict Transport Security Protocol (HSTS). HSTS instructs browsers to always access the site via HTTPS. ASP.NET Core 2.1

adds HSTS middleware that supports options for max age, subdomains, and the HSTS preload list.

Configuration for production

In production, HTTPS must be explicitly configured. In 2.1, default configuration schema for configuring HTTPS for Kestrel has been added. Apps can be configured to use:

- Multiple endpoints including the URLs. For more information, see Kestrel web server implementation: Endpoint configuration.
- The certificate to use for HTTPS either from a file on disk or from a certificate store.

GDPR

ASP.NET Core provides APIs and templates to help meet some of the EU General Data Protection Regulation (GDPR) requirements. For more information, see GDPR support in ASP.NET Core. A sample app shows how to use and lets you test most of the GDPR extension points and APIs added to the ASP.NET Core 2.1 templates.

Integration tests

A new package is introduced that streamlines test creation and execution. The Microsoft.AspNetCore.Mvc.Testing package handles the following tasks:

- Copies the dependency file (*.*deps*) from the tested app into the test project's *bin* folder.
- Sets the content root to the tested app's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory<TEntryPoint> class to streamline bootstrapping the tested app with TestServer.

The following test uses xUnit to check that the Index page loads with a success status code and with the correct Content-Type header:

C#

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup>
factory)
```

```
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task GetHomePage()
    {
        // Act
        var response = await _client.GetAsync("/");

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

For more information, see the Integration tests topic.

[ApiController], ActionResult<T>

ASP.NET Core 2.1 adds new programming conventions that make it easier to build clean and descriptive web APIs. `ActionResult<T>` is a new type added to allow an app to return either a response type or any other action result (similar to IActionResult), while still indicating the response type. The `[ApiController]` attribute has also been added as the way to opt in to Web API-specific conventions and behaviors.

For more information, see Build Web APIs with ASP.NET Core.

IHttpClientFactory

ASP.NET Core 2.1 includes a new `IHttpClientFactory` service that makes it easier to configure and consume instances of `HttpClient` in apps. `HttpClient` already has the concept of delegating handlers that could be linked together for outgoing HTTP requests. The factory:

- Makes registering of instances of `HttpClient` per named client more intuitive.
- Implements a Polly handler that allows Polly policies to be used for Retry, CircuitBreakers, etc.

For more information, see Initiate HTTP Requests.

Kestrel libuv transport configuration

With the release of ASP.NET Core 2.1, Kestrel's default transport is no longer based on Libuv but instead based on managed sockets. For more information, see Kestrel web server implementation: Libuv transport configuration.

Generic host builder

The Generic Host Builder (`HostBuilder`) has been introduced. This builder can be used for apps that don't process HTTP requests (Messaging, background tasks, etc.).

For more information, see .NET Generic Host.

Updated SPA templates

The Single Page Application templates for Angular and React are updated to use the standard project structures and build systems for each framework.

The Angular template is based on the Angular CLI, and the React template is based on create-react-app.

For more information, see:

- Use Angular with ASP.NET Core
- Use React with ASP.NET Core

Razor Pages search for Razor assets

In 2.1, Razor Pages search for Razor assets (such as layouts and partials) in the following directories in the listed order:

1. Current Pages folder.
2. */Pages/Shared/*
3. */Views/Shared/*

Razor Pages in an area

Razor Pages now support areas. To see an example of areas, create a new Razor Pages web app with individual user accounts. A Razor Pages web app with individual user accounts includes */Areas/Identity/Pages*.

MVC compatibility version

The SetCompatibilityVersion method allows an app to opt-in or opt-out of potentially breaking behavior changes introduced in ASP.NET Core MVC 2.1 or later.

For more information, see Compatibility version for ASP.NET Core MVC.

Migrate from 2.0 to 2.1

See Migrate from ASP.NET Core 2.0 to 2.1.

Additional information

For the complete list of changes, see the ASP.NET Core 2.1 Release Notes.

What's new in ASP.NET Core 2.0

Article • 06/03/2022

This article highlights the most significant changes in ASP.NET Core 2.0, with links to relevant documentation.

Razor Pages

Razor Pages is a new feature of ASP.NET Core MVC that makes coding page-focused scenarios easier and more productive.

For more information, see the introduction and tutorial:

- Introduction to Razor Pages
- Get started with Razor Pages

ASP.NET Core metapackage

A new ASP.NET Core metapackage includes all of the packages made and supported by the ASP.NET Core and Entity Framework Core teams, along with their internal and 3rd-party dependencies. You no longer need to choose individual ASP.NET Core features by package. All features are included in the Microsoft.AspNetCore.All package. The default templates use this package.

For more information, see Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0.

Runtime Store

Applications that use the `Microsoft.AspNetCore.All` metapackage automatically take advantage of the new .NET Core Runtime Store. The Store contains all the runtime assets needed to run ASP.NET Core 2.0 applications. When you use the `Microsoft.AspNetCore.All` metapackage, no assets from the referenced ASP.NET Core NuGet packages are deployed with the application because they already reside on the target system. The assets in the Runtime Store are also precompiled to improve application startup time.

For more information, see Runtime store

.NET Standard 2.0

The ASP.NET Core 2.0 packages target .NET Standard 2.0. The packages can be referenced by other .NET Standard 2.0 libraries, and they can run on .NET Standard 2.0-compliant implementations of .NET, including .NET Core 2.0 and .NET Framework 4.6.1.

The `Microsoft.AspNetCore.All` metapackage targets .NET Core 2.0 only, because it's intended to be used with the .NET Core 2.0 Runtime Store.

Configuration update

An `IConfiguration` instance is added to the services container by default in ASP.NET Core 2.0. `IConfiguration` in the services container makes it easier for applications to retrieve configuration values from the container.

For information about the status of planned documentation, see the GitHub issue.

Logging update

In ASP.NET Core 2.0, logging is incorporated into the dependency injection (DI) system by default. You add providers and configure filtering in the `Program.cs` file instead of in the `Startup.cs` file. And the default `ILoggerFactory` supports filtering in a way that lets you use one flexible approach for both cross-provider filtering and specific-provider filtering.

For more information, see Introduction to Logging.

Authentication update

A new authentication model makes it easier to configure authentication for an application using DI.

New templates are available for configuring authentication for web apps and web APIs using Azure AD B2C.

For information about the status of planned documentation, see the GitHub issue.

Identity update

We've made it easier to build secure web APIs using Identity in ASP.NET Core 2.0. You can acquire access tokens for accessing your web APIs using the Microsoft Authentication Library (MSAL).

For more information on authentication changes in 2.0, see the following resources:

- Account confirmation and password recovery in ASP.NET Core
- Enable QR Code generation for authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core 2.0

SPA templates

Single Page Application (SPA) project templates for Angular, Aurelia, Knockout.js, React.js, and React.js with Redux are available. The Angular template has been updated to Angular 4. The Angular and React templates are available by default; for information about how to get the other templates, see Create a new SPA project. For information about how to build a SPA in ASP.NET Core, see The features described in this article are obsolete as of ASP.NET Core 3.0.

Kestrel improvements

The Kestrel web server has new features that make it more suitable as an Internet-facing server. A number of server constraint configuration options are added in the `KestrelServerOptions` class's new `Limits` property. Add limits for the following:

- Maximum client connections
- Maximum request body size
- Minimum request body data rate

For more information, see Kestrel web server implementation in ASP.NET Core.

WebListener renamed to HTTP.sys

The packages `Microsoft.AspNetCore.Server.WebListener` and `Microsoft.Net.Http.Server` have been merged into a new package `Microsoft.AspNetCore.Server.HttpSys`. The namespaces have been updated to match.

For more information, see HTTP.sys web server implementation in ASP.NET Core.

Enhanced HTTP header support

When using MVC to transmit a `FileStreamResult` or a `FileContentResult`, you now have the option to set an `ETag` or a `LastModified` date on the content you transmit. You can set these values on the returned content with code similar to the following:

C#

```
var data = Encoding.UTF8.GetBytes("This is a sample text from a binary
array");
var entityTag = new EntityTagHeaderValue("\"MyCalculatedEtagValue\"");
return File(data, "text/plain", "downloadName.txt", lastModified:
DateTime.UtcNow.AddSeconds(-5), entityTag: entityTag);
```

The file returned to your visitors has the appropriate HTTP headers for the `ETag` and `LastModified` values.

If an application visitor requests content with a Range Request header, ASP.NET Core recognizes the request and handles the header. If the requested content can be partially delivered, ASP.NET Core appropriately skips and returns just the requested set of bytes. You don't need to write any special handlers into your methods to adapt or handle this feature; it's automatically handled for you.

Hosting startup and Application Insights

Hosting environments can now inject extra package dependencies and execute code during application startup, without the application needing to explicitly take a dependency or call any methods. This feature can be used to enable certain environments to "light-up" features unique to that environment without the application needing to know ahead of time.

In ASP.NET Core 2.0, this feature is used to automatically enable Application Insights diagnostics when debugging in Visual Studio and (after opting in) when running in Azure App Services. As a result, the project templates no longer add Application Insights packages and code by default.

For information about the status of planned documentation, see the GitHub issue.

Automatic use of antiforgery tokens

ASP.NET Core has always helped HTML-encode content by default, but with the new version an extra step is taken to help prevent cross-site request forgery (XSRF) attacks. ASP.NET Core will now emit antiforgery tokens by default and validate them on form POST actions and pages without extra configuration.

For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Automatic precompilation

Razor view pre-compilation is enabled during publish by default, reducing the publish output size and application startup time.

For more information, see Razor view compilation and precompilation in ASP.NET Core.

Razor support for C# 7.1

The Razor view engine has been updated to work with the new Roslyn compiler. That includes support for C# 7.1 features like Default Expressions, Inferred Tuple Names, and Pattern-Matching with Generics. To use C# 7.1 in your project, add the following property in your project file and then reload the solution:

XML

```
<LangVersion>latest</LangVersion>
```

For information about the status of C# 7.1 features, see the Roslyn GitHub repository.

Other documentation updates for 2.0

- Visual Studio publish profiles for ASP.NET Core app deployment
- Key Management
- Configure Facebook authentication
- Configure Twitter authentication
- Configure Google authentication
- Configure Microsoft Account authentication

Migration guidance

For guidance on how to migrate ASP.NET Core 1.x applications to ASP.NET Core 2.0, see the following resources:

- Migrate from ASP.NET Core 1.x to ASP.NET Core 2.0
- Migrate Authentication and Identity to ASP.NET Core 2.0

Additional Information

For the complete list of changes, see the ASP.NET Core 2.0 Release Notes.

To connect with the ASP.NET Core development team's progress and plans, tune in to the ASP.NET Community Standup.

What's new in ASP.NET Core 1.1

Article • 06/03/2022

ASP.NET Core 1.1 includes the following new features:

- URL Rewriting Middleware
- Response Caching Middleware
- View Components as Tag Helpers
- Middleware as MVC filters
- Cookie-based TempData provider
- Azure App Service logging provider
- Azure Key Vault configuration provider
- Azure and Redis Storage Data Protection Key Repositories
- WebListener Server for Windows
- WebSockets support

Choosing between versions 1.0 and 1.1 of ASP.NET Core

ASP.NET Core 1.1 has more features than ASP.NET Core 1.0. In general, we recommend you use the latest version.

Additional Information

- ASP.NET Core 1.1.0 Release Notes
- To connect with the ASP.NET Core development team's progress and plans, tune in to the ASP.NET Community Standup .

ASP.NET Core documentation - what's new?

Welcome to what's new in ASP.NET Core docs. Use this page to quickly find the latest changes.

Find ASP.NET Core docs updates

WHAT'S NEW

June 2024

May 2024

April 2024

March 2024

February 2024

January 2024

Get involved - contribute to ASP.NET Core docs

OVERVIEW

ASP.NET Core docs repository

Project structure and labels for issues and pull requests

CONCEPT

Contributor guide

ASP.NET Core docs contributor guide

ASP.NET Core API reference docs contributor guide

Community

WHAT'S NEW

Community

Related what's new pages

WHAT'S NEW

.NET Multi-platform App UI (.NET MAUI) docs updates

.NET Core release notes

ASP.NET Core release notes

C# compiler (Roslyn) release notes

Visual Studio release notes

Visual Studio for Mac release notes

Visual Studio Code release notes

Choose an ASP.NET Core web UI

Article • 12/05/2023

ASP.NET Core is a complete UI framework. Choose which functionalities to combine that fit the app's web UI needs.

For new project development, we recommend ASP.NET Core Blazor.

ASP.NET Core Blazor

Blazor is a full-stack web UI framework and is recommended for most web UI scenarios.

Benefits of using Blazor:

- Reusable component model.
- Efficient diff-based component rendering.
- Flexibly render components from the server or client via WebAssembly.
- Build rich interactive web UI components in C#.
- Render components statically from the server.
- Progressively enhance server rendered components for smoother navigation and form handling and to enable streaming rendering.
- Share code for common logic on the client and server.
- Interop with JavaScript.
- Integrate components with existing MVC, Razor Pages, or JavaScript based apps.

For a complete overview of Blazor, its architecture and benefits, see ASP.NET Core Blazor and ASP.NET Core Blazor hosting models. To get started with your first Blazor app, see Build your first Blazor app.

ASP.NET Core Razor Pages

Razor Pages is a page-based model for building server rendered web UI. Razor pages UI are dynamically rendered on the server to generate the page's HTML and CSS in response to a browser request. The page arrives at the client ready to display. Support for Razor Pages is built on ASP.NET Core MVC.

Razor Pages benefits:

- Quickly build and update UI. Code for the page is kept with the page, while keeping UI and business logic concerns separate.
- Testable and scales to large apps.

- Keep your ASP.NET Core pages organized in a simpler way than ASP.NET MVC:
  - View specific logic and view models can be kept together in their own namespace and directory.
  - Groups of related pages can be kept in their own namespace and directory.

To get started with your first ASP.NET Core Razor Pages app, see Tutorial: Get started with Razor Pages in ASP.NET Core. For a complete overview of ASP.NET Core Razor Pages, its architecture and benefits, see: Introduction to Razor Pages in ASP.NET Core.

ASP.NET Core MVC

ASP.NET Core MVC renders UI on the server and uses a Model-View-Controller (MVC) architectural pattern. The MVC pattern separates an app into three main groups of components: models, views, and controllers. User requests are routed to a controller. The controller is responsible for working with the model to perform user actions or retrieve results of queries. The controller chooses the view to display to the user and provides it with any model data it requires.

ASP.NET Core MVC benefits:

- Based on a scalable and mature model for building large web apps.
- Clear separation of concerns for maximum flexibility.
- The Model-View-Controller separation of responsibilities ensures that the business model can evolve without being tightly coupled to low-level implementation details.

To get started with ASP.NET Core MVC, see Get started with ASP.NET Core MVC. For an overview of ASP.NET Core MVC's architecture and benefits, see Overview of ASP.NET Core MVC.

ASP.NET Core Single Page Applications (SPA) with frontend JavaScript frameworks

Build client-side logic for ASP.NET Core apps using popular JavaScript frameworks, like Angular, React, and Vue. ASP.NET Core provides project templates for Angular, React, and Vue, and it can be used with other JavaScript frameworks as well.

Benefits of ASP.NET Core SPA with JavaScript Frameworks, in addition to the client rendering benefits previously listed:

- The JavaScript runtime environment is already provided with the browser.
- Large community and mature ecosystem.

- Build client-side logic for ASP.NET Core apps using popular JS frameworks, like Angular, React, and Vue.

Downsides:

- More coding languages, frameworks, and tools required.
- Difficult to share code so some logic may be duplicated.

To get started, see:

- Create an ASP.NET Core app with Angular
- Create an ASP.NET Core app with React
- Create an ASP.NET Core app with Vue
- JavaScript and TypeScript in Visual Studio

Combine multiple web UI solutions: ASP.NET Core MVC or Razor Pages plus Blazor

MVC, Razor Pages, and Blazor are part of the ASP.NET Core framework and are designed to be used together. Razor components can be integrated into Razor Pages and MVC apps. When a view or page is rendered, components can be prerendered at the same time.

Benefits for MVC or Razor Pages plus Blazor, in addition to MVC or Razor Pages benefits:

- Prerendering executes Razor components on the server and renders them into a view or page, which improves the perceived load time of the app.
- Add interactivity to existing views or pages with the Component Tag Helper.

To get started with ASP.NET Core MVC or Razor Pages plus Blazor, see Integrate ASP.NET Core Razor components into ASP.NET Core apps.

Next steps

For more information, see:

- ASP.NET Core Blazor
- ASP.NET Core Blazor hosting models
- Integrate ASP.NET Core Razor components into ASP.NET Core apps
- Compare gRPC services with HTTP APIs

Tutorial: Create a Razor Pages web app with ASP.NET Core

Article • 06/23/2024

This series of tutorials explains the basics of building a Razor Pages web app.

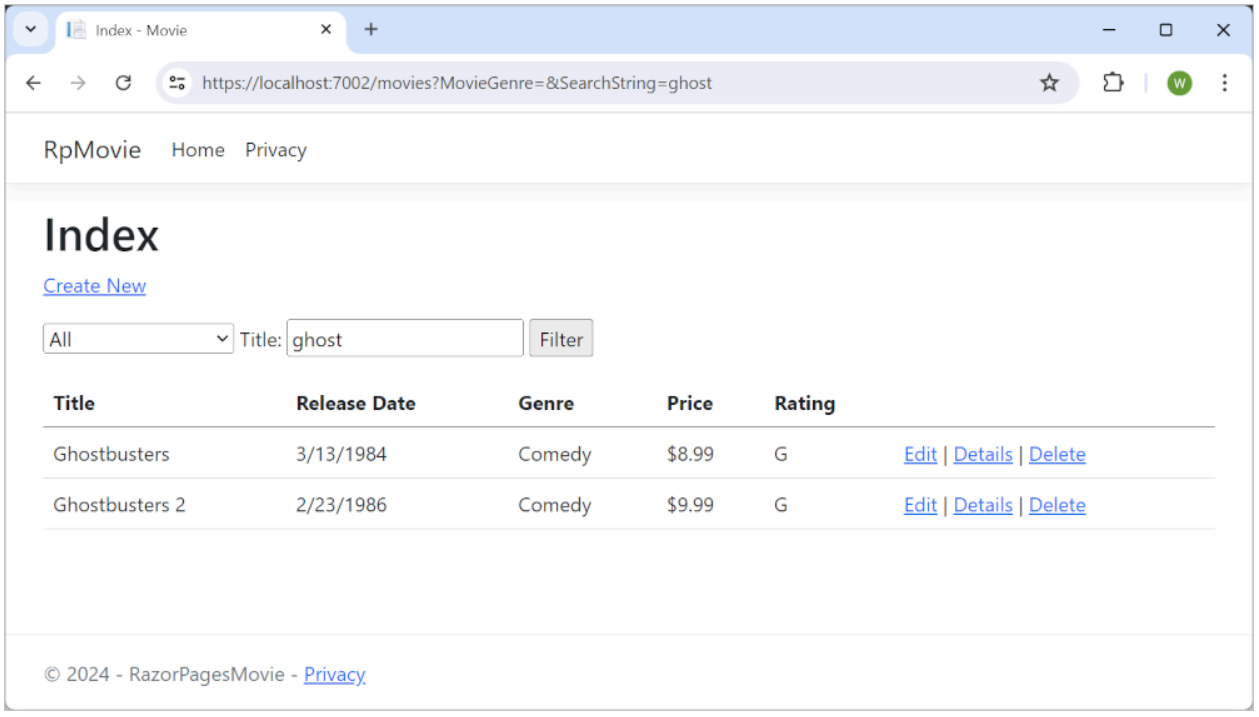
For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages in ASP.NET Core.

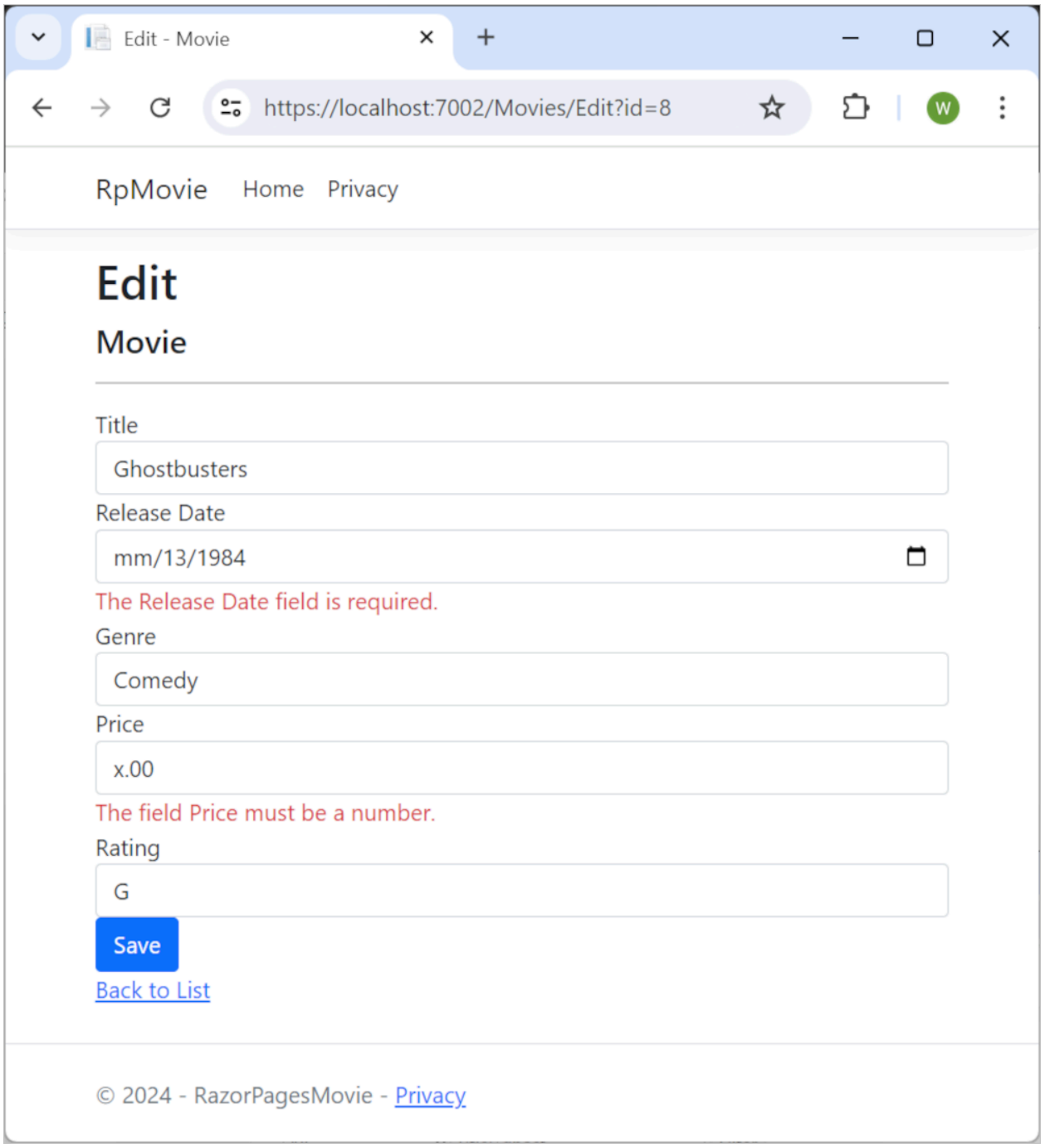
If you're new to ASP.NET Core development and are unsure of which ASP.NET Core web UI solution will best fit your needs, see Choose an ASP.NET Core UI.

This series includes the following tutorials:

1. Create a Razor Pages web app
2. Add a model to a Razor Pages app
3. Scaffold (generate) Razor pages
4. Work with a database
5. Update Razor pages
6. Add search
7. Add a new field
8. Add validation

At the end, you'll have an app that can display and manage a database of movies.





Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Get started with Razor Pages in ASP.NET Core

Article • 08/05/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

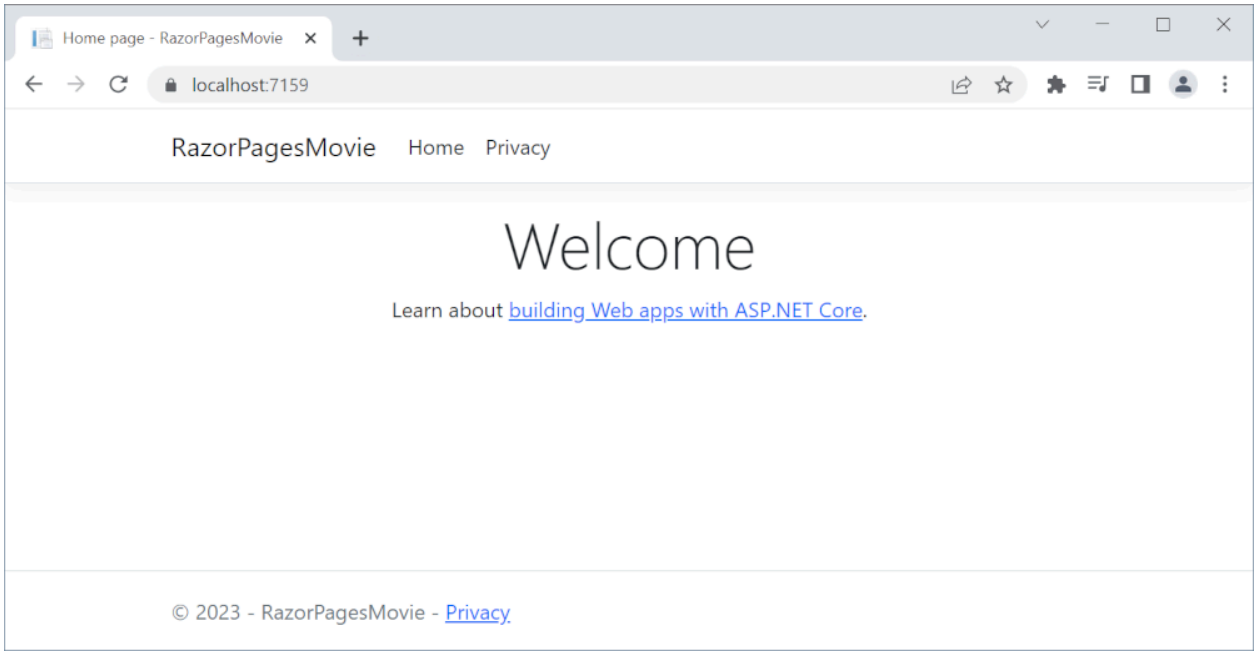
By Rick Anderson

This is the first tutorial of a series that teaches the basics of building an ASP.NET Core Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages. For a video introduction, see Entity Framework Core for Beginners.

If you're new to ASP.NET Core development and are unsure of which ASP.NET Core web UI solution will best fit your needs, see Choose an ASP.NET Core UI.

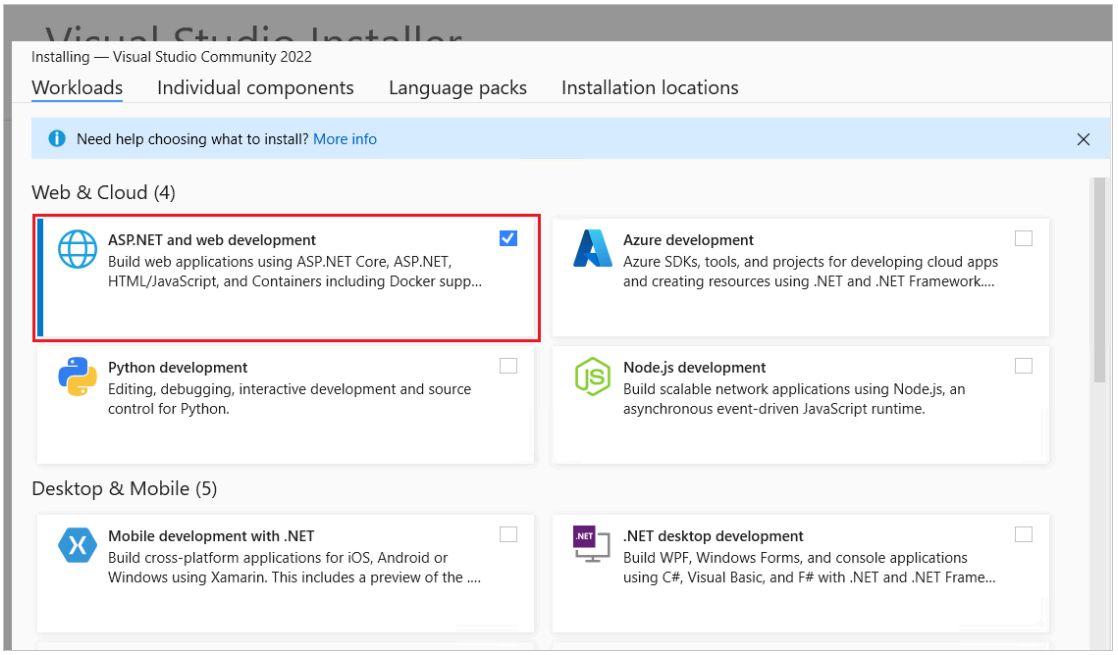
At the end of this tutorial, you'll have a Razor Pages web app that manages a database of movies.



Prerequisites

Visual Studio

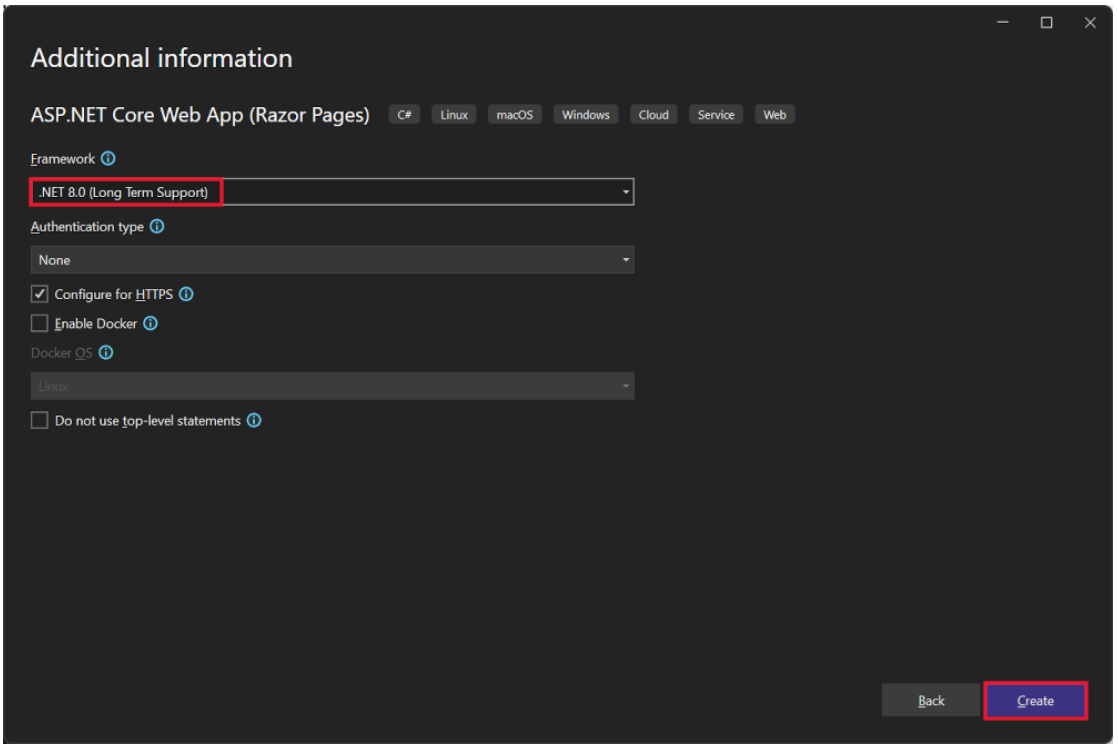
- Visual Studio 2022 with the **ASP.NET and web development** workload.



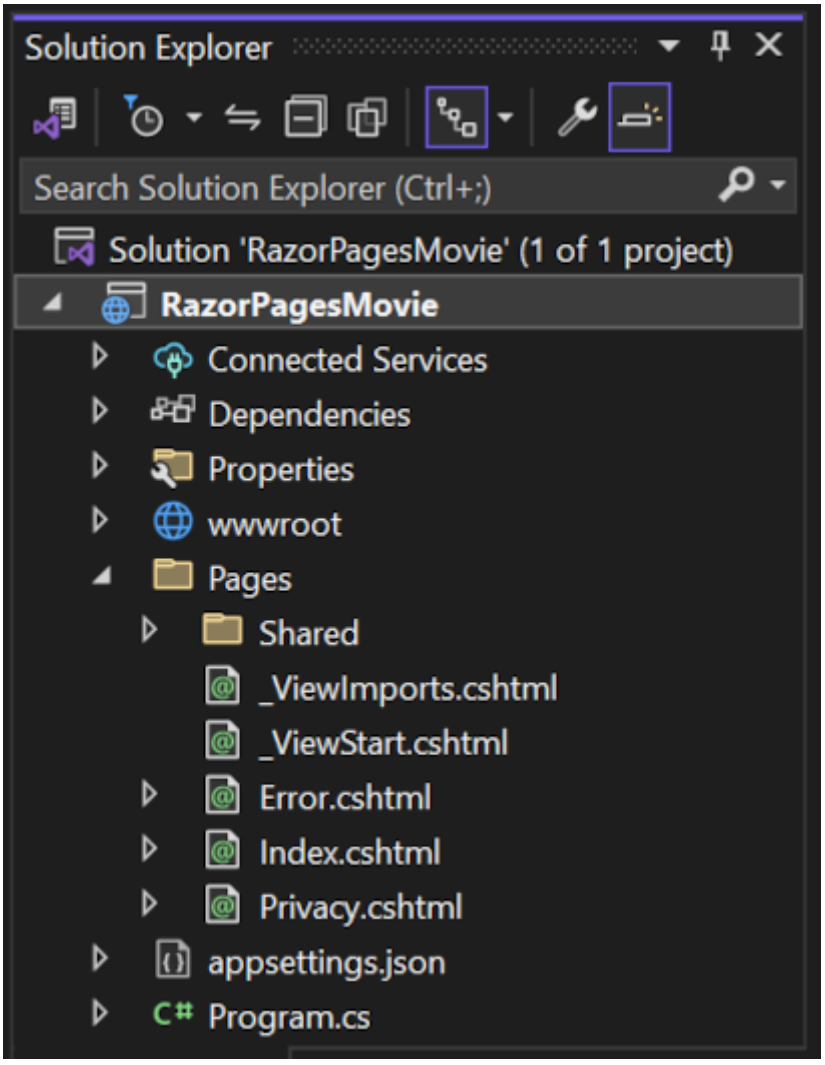
Create a Razor Pages web app

Visual Studio

- Start Visual Studio and select **New project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web App (Razor Pages)** > **Next**.
- In the **Configure your new project** dialog, enter `RazorPagesMovie` for **Project name**. It's important to name the project **RazorPagesMovie**, including matching the capitalization, so the namespaces will match when you copy and paste example code.
- Select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 8.0 (Long Term Support)**.
  - Verify: **Do not use top-level statements** is unchecked.
- Select **Create**.



The following starter project is created:



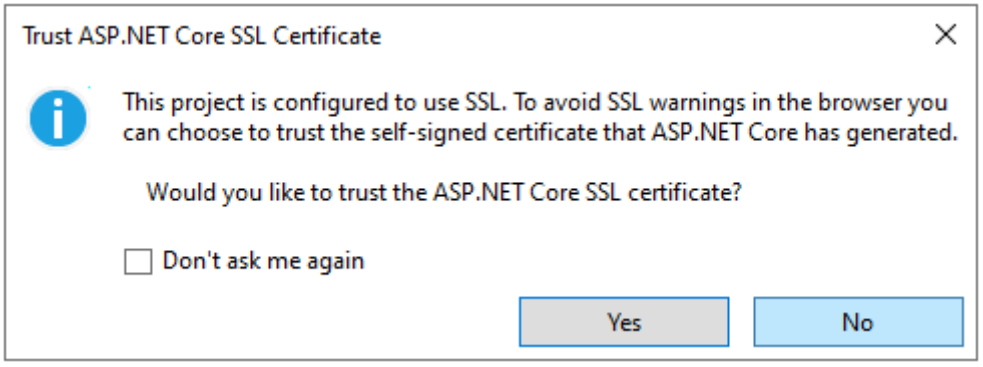
For alternative approaches to create the project, see Create a new project in Visual Studio.

Run the app

Visual Studio

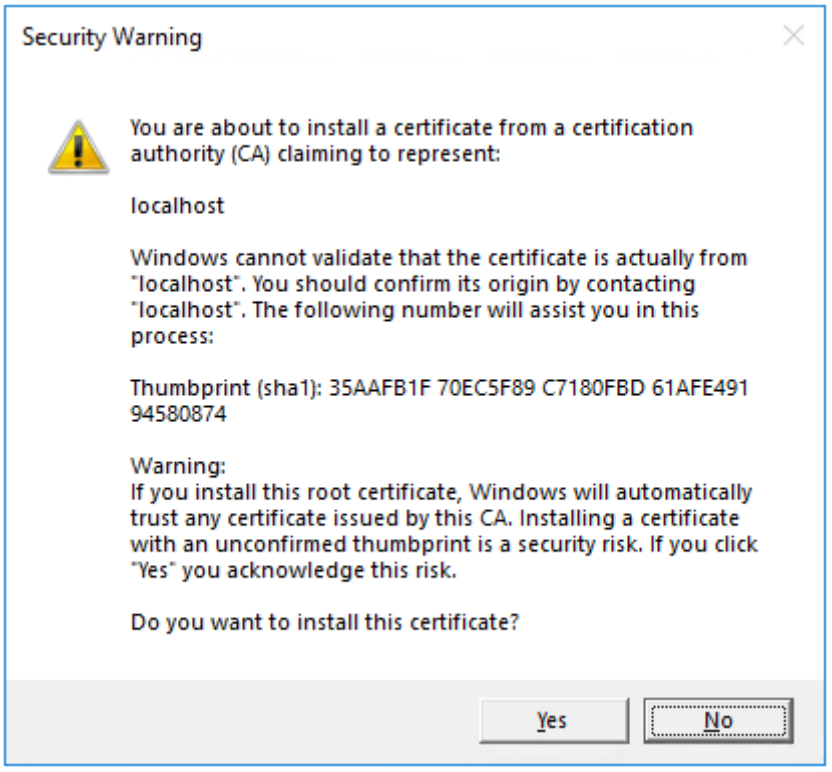
Select **RazorPagesMovie** in **Solution Explorer**, and then press `Ctrl`+`F5` to run the app without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio:

- Runs the app, which launches the Kestrel server.
- Launches the default browser at `https://localhost:<port>`, which displays the apps UI. `<port>` is the random port that is assigned when the app was created.

Close the browser window.

Examine the project files

The following sections contain an overview of the main project folders and files that you'll work with in later tutorials.

Pages folder

Contains Razor pages and supporting files. Each Razor page is a pair of files:

- A `.cshtml` file that has HTML markup with C# code using Razor syntax.
- A `.cshtml.cs` file that has C# code that handles page events.

Supporting files have names that begin with an underscore. For example, the `_Layout.cshtml` file configures UI elements common to all pages. `_Layout.cshtml` sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page. For more information, see Layout in ASP.NET Core.

wwwroot folder

Contains static assets, like HTML files, JavaScript files, and CSS files. For more information, see Static files in ASP.NET Core.

`appsettings.json`

Contains configuration data, like connection strings. For more information, see Configuration in ASP.NET Core.

Program.cs

Contains the following code:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The following lines of code in this file create a `WebApplicationBuilder` with preconfigured defaults, add Razor Pages support to the Dependency Injection (DI) container, and builds the app:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();
```

The developer exception page is enabled by default and provides helpful information on exceptions. Production apps should not be run in development mode because the developer exception page can leak sensitive information.

The following code sets the exception endpoint to `/Error` and enables HTTP Strict Transport Security Protocol (HSTS) when the app is ***not*** running in development mode:

C#

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
```

For example, the preceding code runs when the app is in production or test mode. For more information, see Use multiple environments in ASP.NET Core.

The following code enables various Middleware:

- `app.UseHttpsRedirection();` : Redirects HTTP requests to HTTPS.
- `app.UseStaticFiles();` : Enables static files, such as HTML, CSS, images, and JavaScript to be served. For more information, see Static files in ASP.NET Core.
- `app.UseRouting();` : Adds route matching to the middleware pipeline. For more information, see Routing in ASP.NET Core
- `app.MapRazorPages();`: Configures endpoint routing for Razor Pages.
- `app.UseAuthorization();` : Authorizes a user to access secure resources. This app doesn't use authorization, therefore this line could be removed.
- `app.Run();` : Runs the app.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

Next: Add a model

Part 2, add a model to a Razor Pages app in ASP.NET Core

Article • 08/04/2024

In this tutorial, classes are added for managing movies in a database. The app's model classes use Entity Framework Core (EF Core) to work with the database. EF Core is an object-relational mapper (O/RM) that simplifies data access. You write the model classes first, and EF Core creates the database.

The model classes are known as POCO classes (from "**P**lain-**O**ld **C**LR **O**bjects") because they don't have a dependency on EF Core. They define the properties of the data that are stored in the database.

Add a data model

Visual Studio

1. In **Solution Explorer**, right-click the *RazorPagesMovie* project > **Add** > **New Folder**. Name the folder `Models`.
2. Right-click the `Models` folder. Select **Add** > **Class**. Name the class *Movie*.
3. Add the following properties to the `Movie` class:

C#

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    public decimal Price { get; set; }
}
```

The `Movie` class contains:

- The `ID` field is required by the database for the primary key.
- A [DataType] attribute that specifies the type of data in the `ReleaseDate` property. With this attribute:
  - The user isn't required to enter time information in the date field.
  - Only the date is displayed, not time information.
- The question mark after `string` indicates that the property is nullable. For more information, see Nullable reference types.

DataAnnotations are covered in a later tutorial.

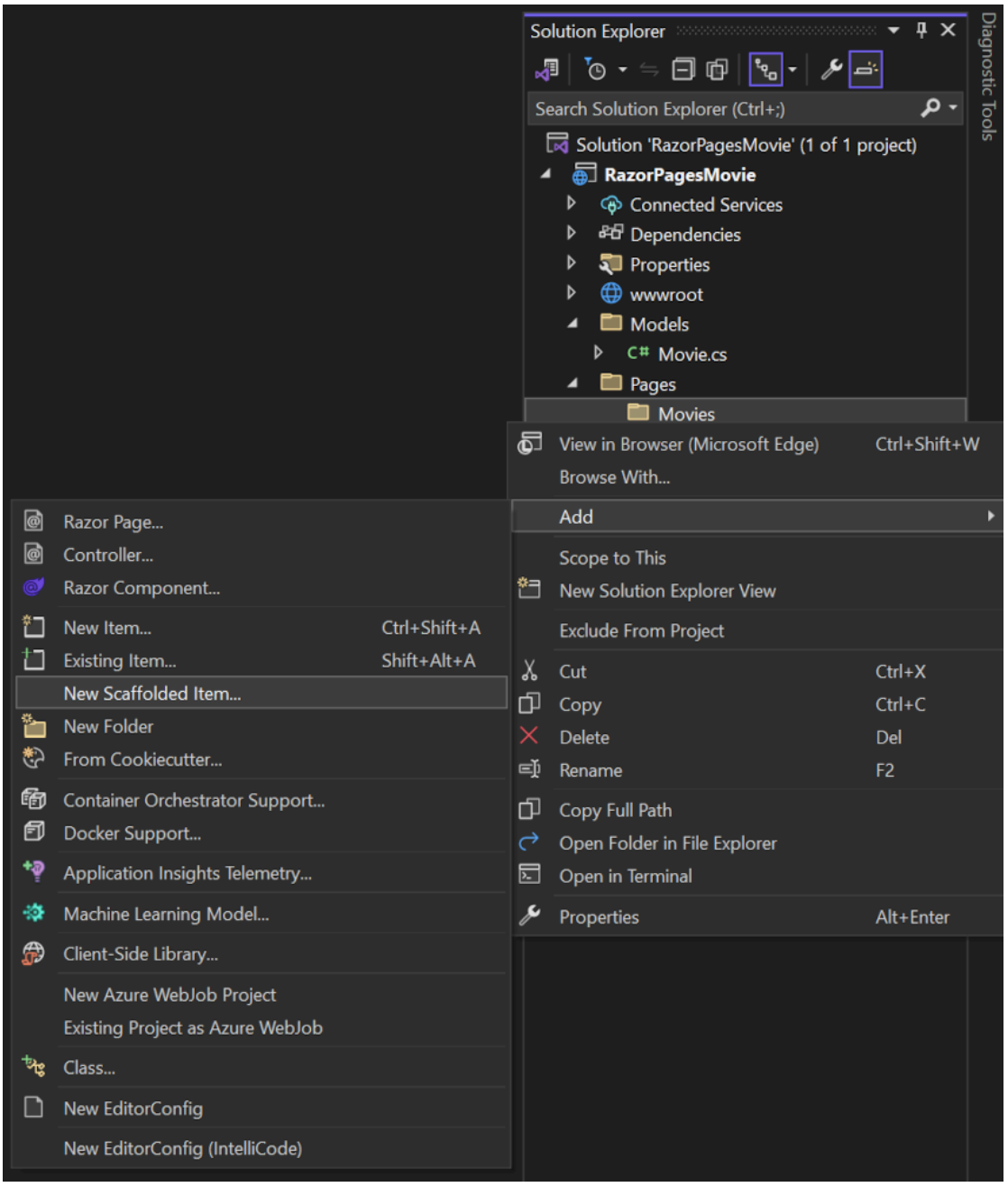
Build the project to verify there are no compilation errors.

Scaffold the movie model

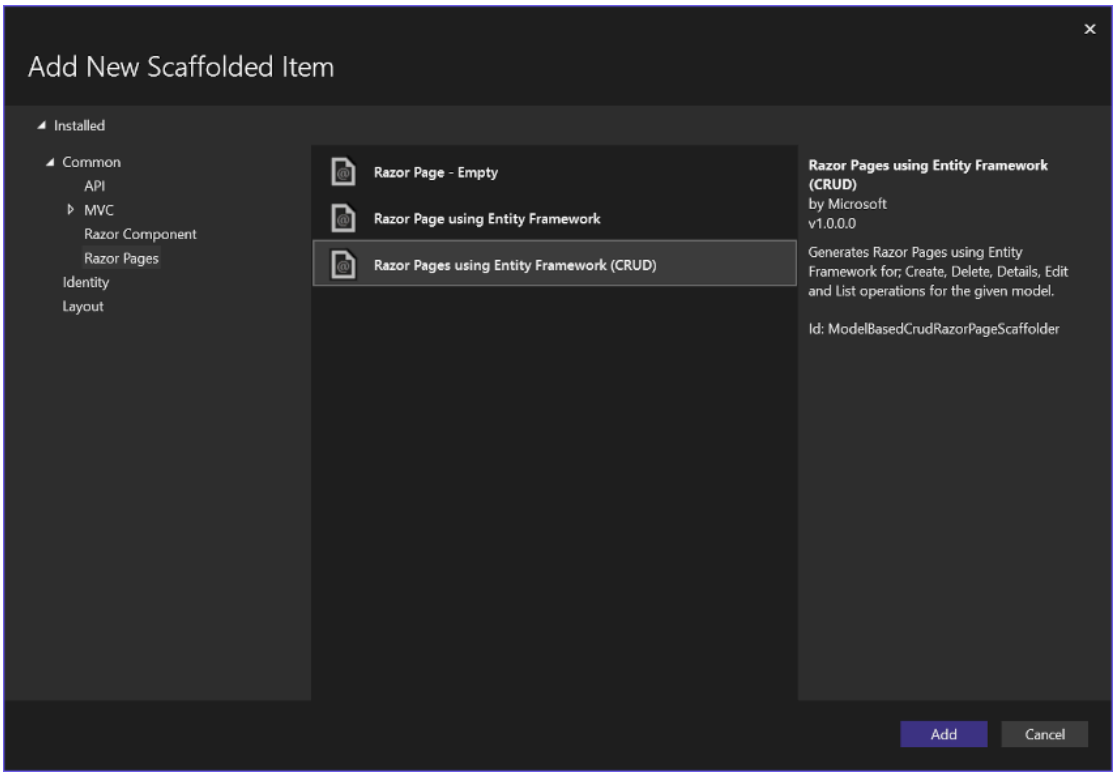
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

Visual Studio

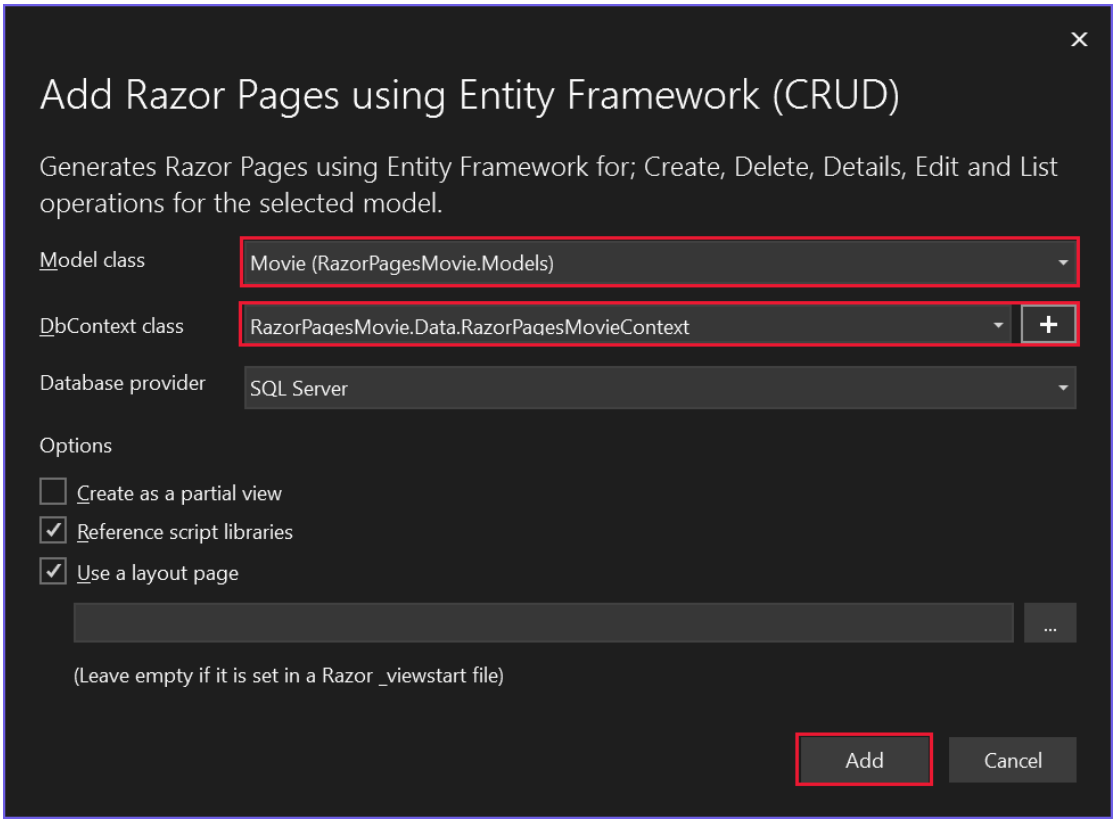
1. Create the *Pages/Movies* folder:
   a. Right-click on the *Pages* folder > **Add** > **New Folder**.
   b. Name the folder *Movies*.
2. Right-click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.



3. In the **Add New Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **Add**.



4. Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:
   a. In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.
   b. In the **Data context class** row, select the + (plus) sign.
      i. In the **Add Data Context** dialog, the class name `RazorPagesMovie.Data.RazorPagesMovieContext` is generated.
      ii. In the **Database provider** drop down, select **SQL Server**.
   c. Select **Add**.



The `appsettings.json` file is updated with the connection string used to connect to a local database.

> ⚠ **Warning**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Files created and updated

The scaffold process creates the following files:

- *Pages/Movies*: Create, Delete, Details, Edit, and Index.
- `Data/RazorPagesMovieContext.cs`

The created files are explained in the next tutorial.

The scaffold process adds the following highlighted code to the `Program.cs` file:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
```

```
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The `Program.cs` changes are explained later in this tutorial.

Create the initial database schema using EF's migration feature

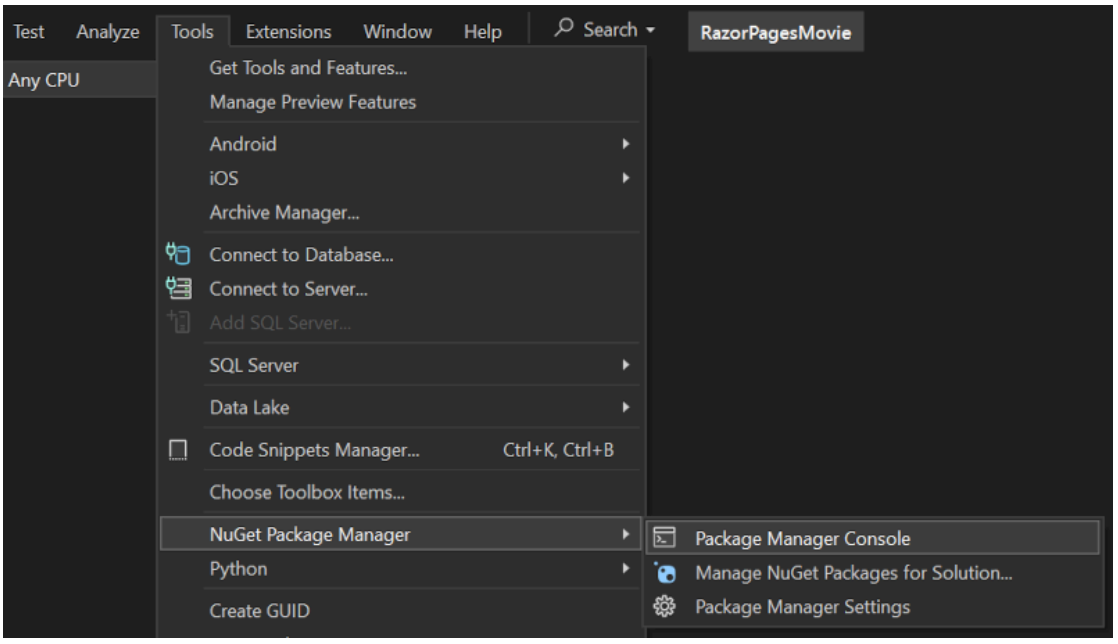
The migrations feature in Entity Framework Core provides a way to:

- Create the initial database schema.
- Incrementally update the database schema to keep it in sync with the app's data model. Existing data in the database is preserved.

Visual Studio

In this section, the **Package Manager Console** (PMC) window is used to:

- Add an initial migration.
- Update the database with the initial migration.
- From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



- In the PMC, enter the following command:

```
PowerShell

Add-Migration InitialCreate
```

- The `Add-Migration` command generates code to create the initial database schema. The schema is based on the model specified in `DbContext`. The `InitialCreate` argument is used to name the migration. Any name can be used, but by convention a name is selected that describes the migration.

The following warning is displayed, which is addressed in a later step:

> No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.

- In the PMC, enter the following command:

```
PowerShell

Update-Database
```

The `Update-Database` command runs the `Up` method in migrations that have not been applied. In this case, the command runs the `Up` method in the `Migrations/<time-stamp>_InitialCreate.cs` file, which creates the database.

The data context `RazorPagesMovieContext`:

- Derives from Microsoft.EntityFrameworkCore.DbContext.
- Specifies which entities are included in the data model.
- Coordinates EF Core functionality, such as Create, Read, Update and Delete, for the `Movie` model.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Data
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext
(DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; } =
default!;
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the Configuration system reads the connection string from the `appsettings.json` file.

Test the app

1. Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

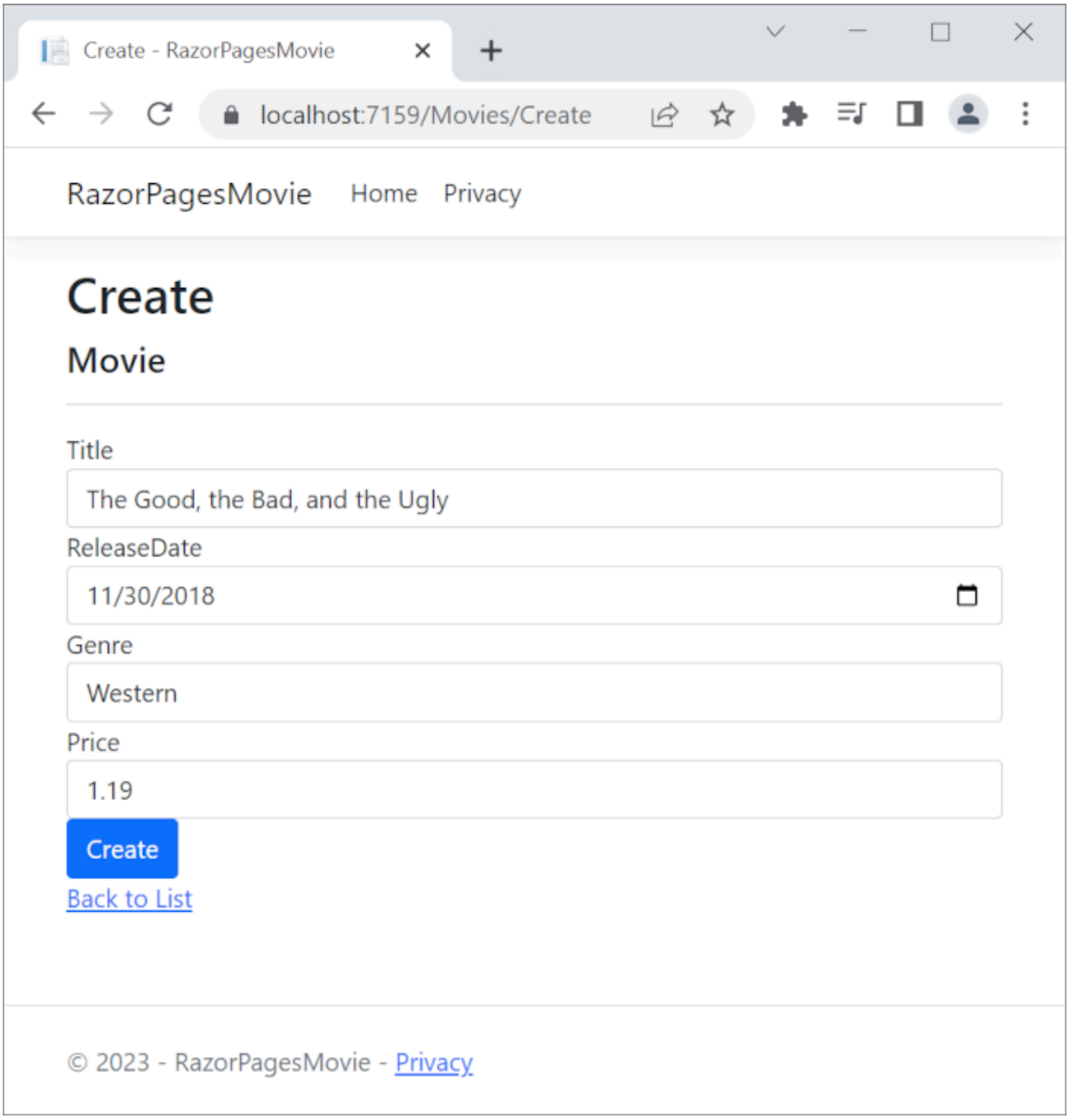
   If you receive the following error:

   Console

```
SqlException: Cannot open database "RazorPagesMovieContext-GUID"
requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

2. Test the **Create New** link.



> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

3. Test the **Edit**, **Details**, and **Delete** links.

The next tutorial explains the files created by scaffolding.

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services, such as the EF Core database context, are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically created a database context and registered it with the dependency injection container. The following highlighted code is added to the `Program.cs` file by the scaffolder:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));
var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();
```

```
app.MapRazorPages();

app.Run();
```

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

Previous: Get Started | Next: Scaffolded Razor Pages

Part 3, scaffolded Razor Pages in ASP.NET Core

Article • 06/23/2024

By Rick Anderson

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial.

The Create, Delete, Details, and Edit pages

Examine the `Pages/Movies/Index.cshtml.cs` Page Model:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Movies;

public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get;set; }  = default!;

    public async Task OnGetAsync()
    {
        if (_context.Movie != null)
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor Pages are derived from PageModel. By convention, the `PageModel` derived class is named `PageNameModel`. For example, the Index page is named `IndexModel`.

The constructor uses dependency injection to add the `RazorPagesMovieContext` to the page:

C#

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }
```

See Asynchronous code for more information on asynchronous programming with Entity Framework.

When a `GET` request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. On a Razor Page, `OnGetAsync` or `OnGet` is called to initialize the state of the page. In this case, `OnGetAsync` gets a list of movies and displays them.

When `OnGet` returns `void` or `OnGetAsync` returns `Task`, no return statement is used. For example, examine the Privacy Page:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace RazorPagesMovie.Pages
{
    public class PrivacyModel : PageModel
    {
        private readonly ILogger<PrivacyModel> _logger;

        public PrivacyModel(ILogger<PrivacyModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

When the return type is IActionResult or `Task<IActionResult>`, a return statement must be provided. For example, the `Pages/Movies/Create.cshtml.cs OnPostAsync` method:

C#

```
public async Task<IActionResult> OnPostAsync()
{
  if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Examine the `Pages/Movies/Index.cshtml` Razor Page:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
```

```
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup, otherwise it transitions into C#.

The @page directive

The `@page` Razor directive makes the file an MVC action, which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` and `@model` are examples of transitioning into Razor-specific markup. See Razor syntax for more information.

The @model directive

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel` derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayFor` HTML Helpers on the page.

Examine the lambda expression used in the following HTML Helper:

CSHTML

```
@Html.DisplayNameFor(model => model.Movie[0].Title)
```

The DisplayNameFor HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` is `null` or empty. When the lambda expression is evaluated, for example, with `@Html.DisplayFor(modelItem => item.Title)`, the model's property values are evaluated.

The layout page

Select the menu links **RazorPagesMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the `Pages/Shared/_Layout.cshtml` file.

Open and examine the `Pages/Shared/_Layout.cshtml` file.

Layout templates allow the HTML container layout to be:

- Specified in one place.
- Applied in multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the page-specific views show up, *wrapped* in the layout page. For example, select the **Privacy** link and the `Pages/Privacy.cshtml` view is rendered inside the `RenderBody` method.

ViewData and layout

Consider the following markup from the `Pages/Movies/Index.cshtml` file:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

The preceding highlighted markup is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class contains a `ViewData` dictionary property that can be used to pass data to a View. Objects are added to the `ViewData` dictionary using a ***key value*** pattern. In the preceding sample, the `Title` property is added to the `ViewData` dictionary.

The `Title` property is used in the `Pages/Shared/_Layout.cshtml` file. The following markup shows the first few lines of the `_Layout.cshtml` file.

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css"
/>
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/RazorPagesMovie.styles.css" asp-append-
version="true" />
```

Update the layout

1. Change the `<title>` element in the `Pages/Shared/_Layout.cshtml` file to display **Movie** rather than **RazorPagesMovie**.

   CSHTML

   ```
   <!DOCTYPE html>
   <html lang="en">
   <head>
       <meta charset="utf-8" />
       <meta name="viewport" content="width=device-width, initial-
   scale=1.0" />
       <title>@ViewData["Title"] - Movie</title>
   ```

2. Find the following anchor element in the `Pages/Shared/_Layout.cshtml` file.

   CSHTML

   ```
   <a class="navbar-brand" asp-area="" asp-
   page="/Index">RazorPagesMovie</a>
   ```

3. Replace the preceding element with the following markup:

```cshtml
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

The preceding anchor element is a Tag Helper. In this case, it's the Anchor Tag Helper. The `asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page. The `asp-area` attribute value is empty, so the area isn't used in the link. See Areas for more information.

4. Save the changes and test the app by selecting the **RpMovie** link. See the _Layout.cshtml file in GitHub if you have any problems.

5. Test the **Home**, **RpMovie**, **Create**, **Edit**, and **Delete** links. Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark.

> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize the app. See this GitHub issue 4076 for instructions on adding decimal comma.

The `Layout` property is set in the `Pages/_ViewStart.cshtml` file:

```cshtml
@{
    Layout = "_Layout";
}
```

The preceding markup sets the layout file to `Pages/Shared/_Layout.cshtml` for all Razor files under the *Pages* folder. See Layout for more information.

The Create page model

Examine the `Pages/Movies/Create.cshtml.cs` page model:

```C#
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
```

```
namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext
_context;

        public CreateModel(RazorPagesMovie.Data.RazorPagesMovieContext
context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; } = default!;


        // To protect from overposting attacks, see
https://aka.ms/RazorPagesCRUD
        public async Task<IActionResult> OnPostAsync()
        {
          if (!ModelState.IsValid || _context.Movie == null || Movie ==
null)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize, so `Page` is returned. Later in the tutorial, an example of `OnGet` initializing state is shown. The `Page` method creates a `PageResult` object that renders the `Create.cshtml` page.

The `Movie` property uses the [BindProperty] attribute to opt-in to model binding. When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

C#

```
public async Task<IActionResult> OnPostAsync()
{
  if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date. Client-side validation and model validation are discussed later in the tutorial.

If there are no model errors:

- The data is saved.
- The browser is redirected to the Index page.

The Create Razor Page

Examine the `Pages/Movies/Create.cshtml` Razor Page file:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
```

```
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label">
</label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Visual Studio

Visual Studio displays the following tags in a distinctive bold font used for Tag Helpers:

- `<form method="post">`
- `<div asp-validation-summary="ModelOnly" class="text-danger"></div>`
- `<label asp-for="Movie.Title" class="control-label"></label>`
- `<input asp-for="Movie.Title" class="form-control" />`
- `<span asp-validation-for="Movie.Title" class="text-danger"></span>`

Create.cshtml

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
```

The `<form method="post">` element is a Form Tag Helper. The Form Tag Helper automatically includes an antiforgery token.

The scaffolding engine creates Razor markup for each field in the model, except the ID, similar to the following:

CSHTML

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

The Validation Tag Helpers (`<div asp-validation-summary` and `<span asp-validation-for`) display validation errors. Validation is covered in more detail later in this series.

The Label Tag Helper (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `[for]` attribute for the `Title` property.

The Input Tag Helper (`<input asp-for="Movie.Title" class="form-control">`) uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side.

For more information on Tag Helpers such as `<form method="post">`, see Tag Helpers in ASP.NET Core.

Next steps

Previous: Add a model | Next: Work with a database

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 4 of tutorial series on Razor Pages

Article • 08/26/2024

By Joe Audette

The `RazorPagesMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in `Program.cs`:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();
```

The ASP.NET Core Configuration system reads the `ConnectionString` key. For local development, configuration gets the connection string from the `appsettings.json` file.

Visual Studio

The generated connection string is similar to the following JSON:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
```

```
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=
(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
bc;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

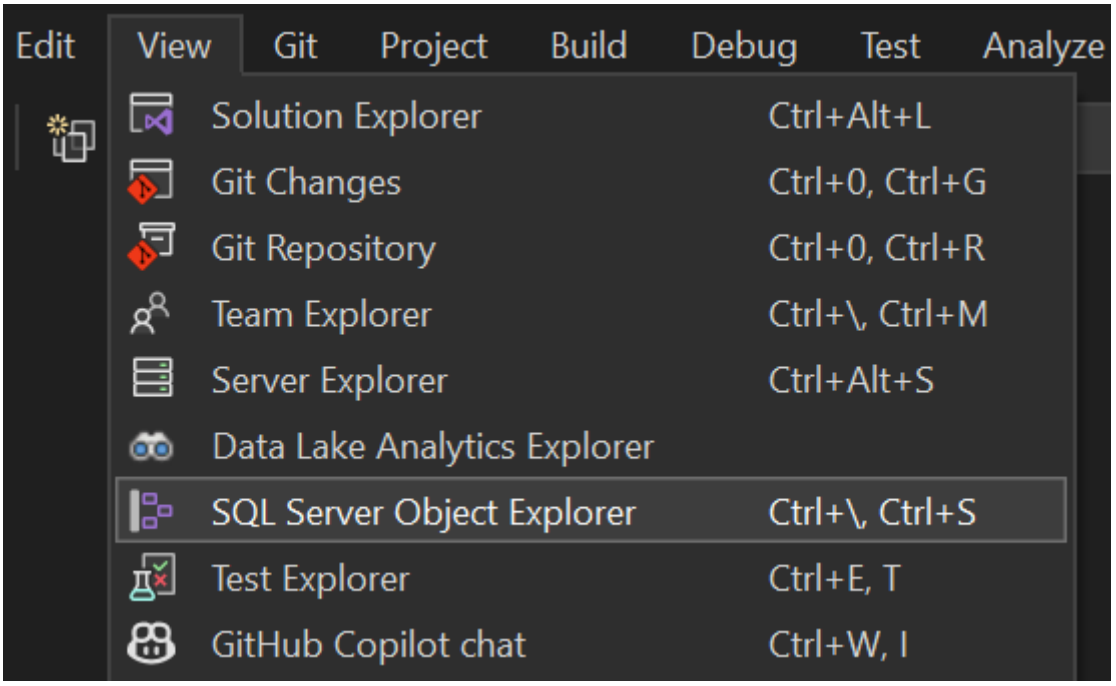
> **Note**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Visual Studio

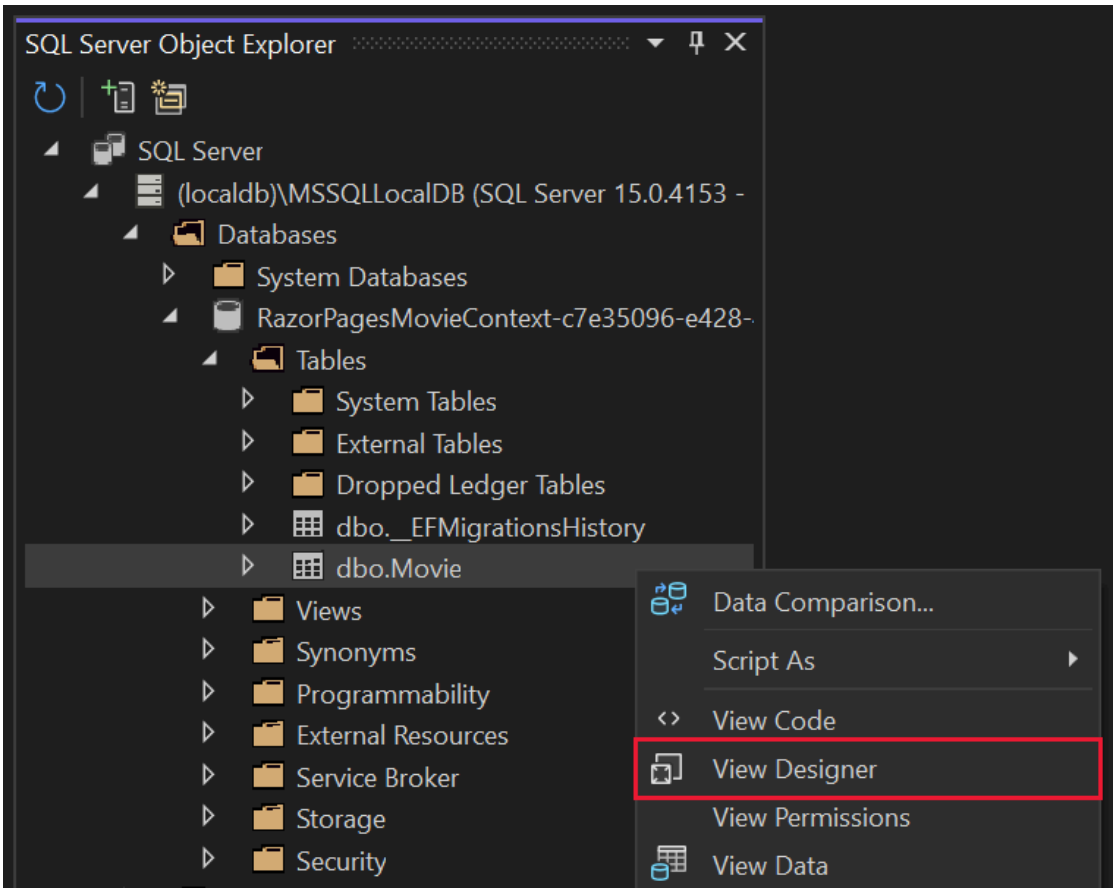
SQL Server Express LocalDB

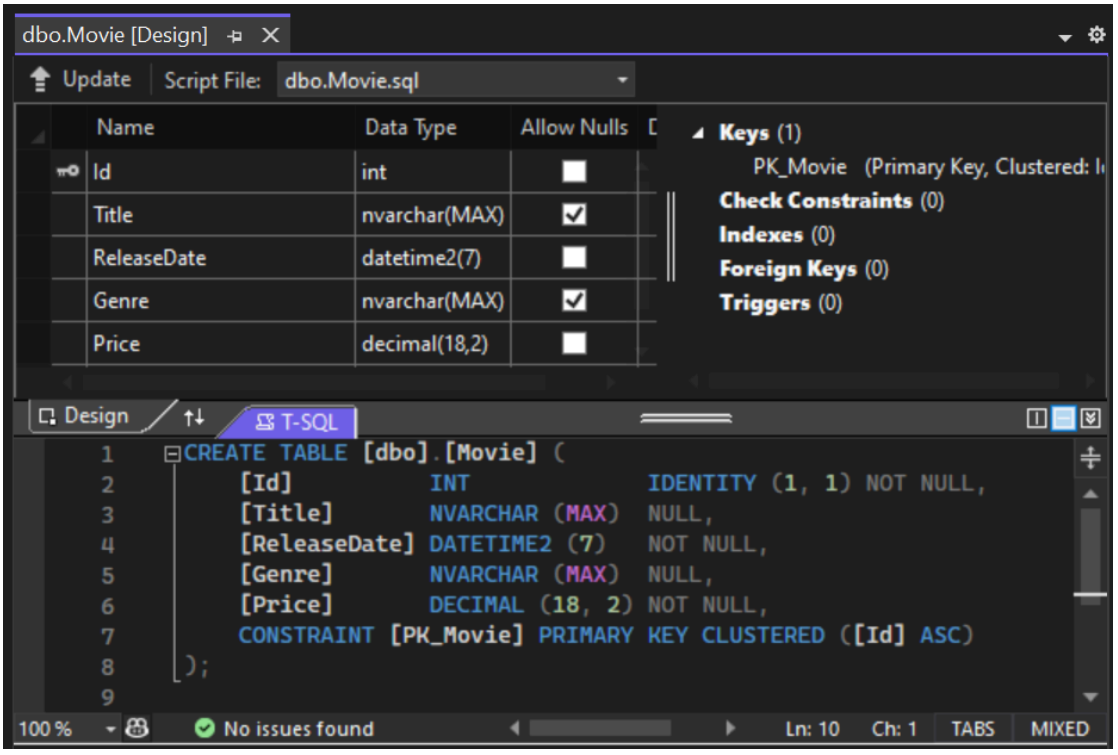
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:\Users\<user>\` directory.

1. From the **View** menu, open **SQL Server Object Explorer** (SSOX).



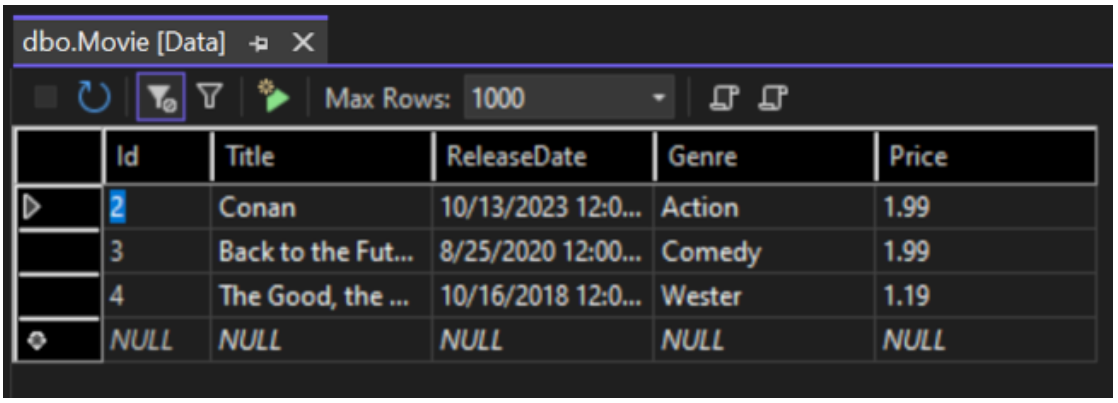
2. Right-click on the `Movie` table and select **View Designer**:





Note the key icon next to `ID`. By default, EF creates a property named `ID` for the primary key.

3. Right-click on the `Movie` table and select **View Data**:



Seed the database

Create a new class named `SeedData` in the *Models* folder with the following code:

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;

namespace RazorPagesMovie.Models;

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new RazorPagesMovieContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<RazorPagesMovieContext>>()))
        {
            if (context == null || context.Movie == null)
            {
                throw new ArgumentNullException("Null
RazorPagesMovieContext");
            }

            // Look for any movies.
            if (context.Movie.Any())
            {
                return;   // DB has been seeded
            }

            context.Movie.AddRange(
                new Movie
                {
                    Title = "When Harry Met Sally",
                    ReleaseDate = DateTime.Parse("1989-2-12"),
                    Genre = "Romantic Comedy",
                    Price = 7.99M
                },
```

```
                new Movie
                {
                    Title = "Ghostbusters ",
                    ReleaseDate = DateTime.Parse("1984-3-13"),
                    Genre = "Comedy",
                    Price = 8.99M
                },

                new Movie
                {
                    Title = "Ghostbusters 2",
                    ReleaseDate = DateTime.Parse("1986-2-23"),
                    Genre = "Comedy",
                    Price = 9.99M
                },

                new Movie
                {
                    Title = "Rio Bravo",
                    ReleaseDate = DateTime.Parse("1959-4-15"),
                    Genre = "Western",
                    Price = 3.99M
                }
            );
            context.SaveChanges();
        }
    }
}
```

If there are any movies in the database, the seed initializer returns and no movies are added.

C#

```
if (context.Movie.Any())
{
    return;
}
```

Add the seed initializer

Update the `Program.cs` with the following highlighted code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;
using RazorPagesMovie.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In the previous code, `Program.cs` has been modified to do the following:

- Get a database context instance from the dependency injection (DI) container.
- Call the `seedData.Initialize` method, passing to it the database context instance.
- Dispose the context when the seed method completes. The using statement ensures the context is disposed.

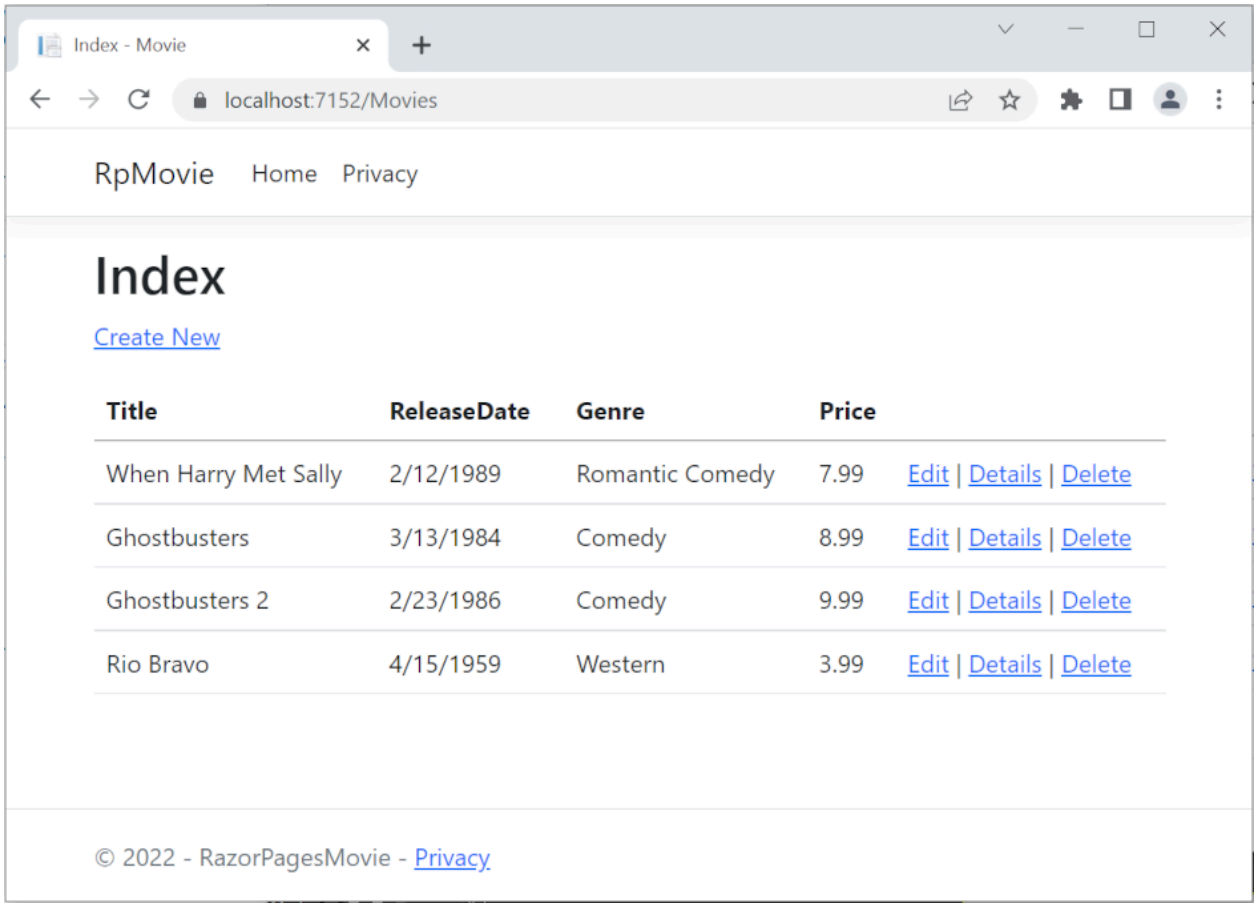
The following exception occurs when `Update-Database` has not been run:

> `SqlException: Cannot open database "RazorPagesMovieContext-" requested by the login. The login failed.` `Login failed for user 'user name'.`

Test the app

Delete all the records in the database so the seed method will run. Stop and start the app to seed the database. If the database isn't seeded, put a breakpoint on `if (context.Movie.Any())` and step through the code.

The app shows the seeded data:



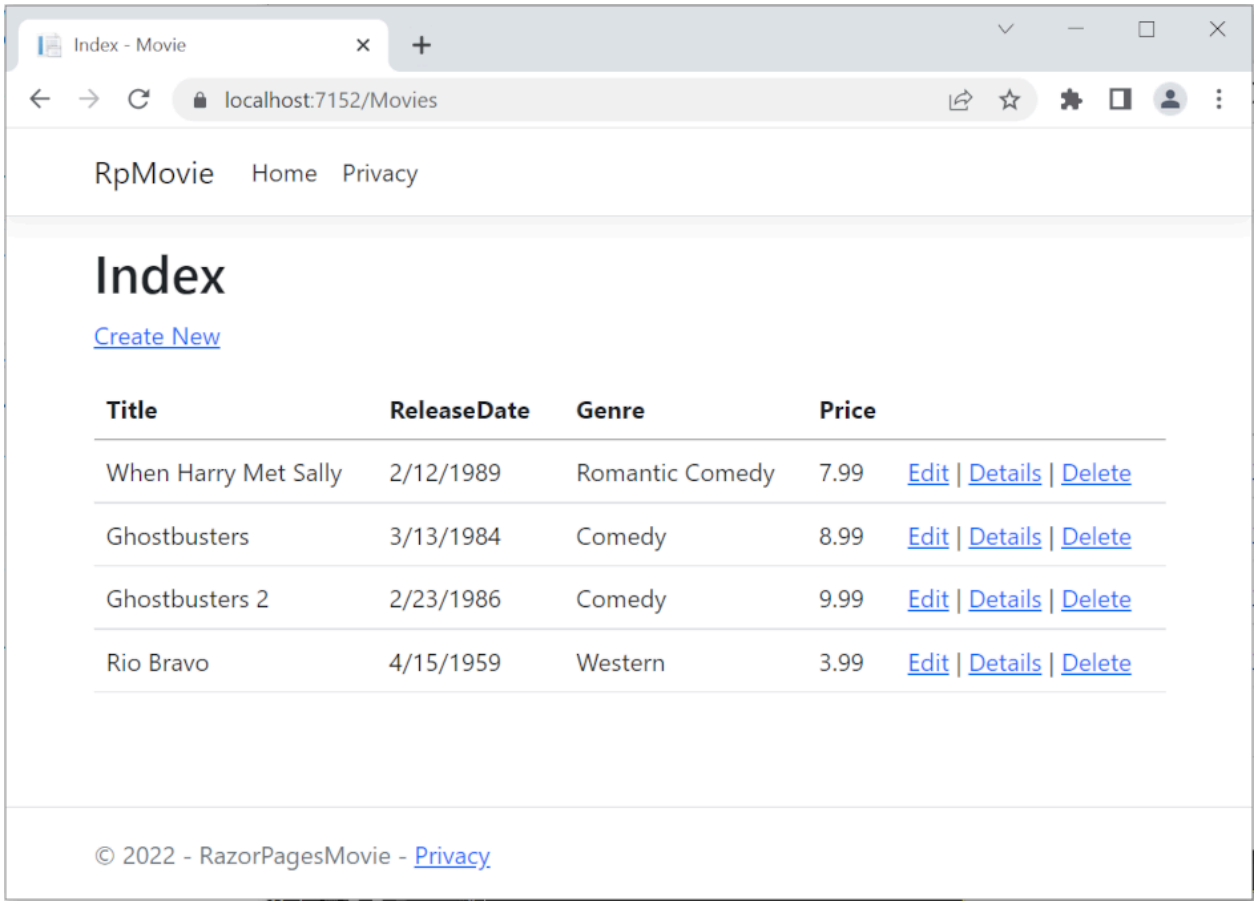
Next steps

Previous: Scaffolded Razor Pages | Next: Update the pages

Part 5, update the generated pages in an ASP.NET Core app

Article • 06/23/2024

The scaffolded movie app has a good start, but the presentation isn't ideal. **ReleaseDate** should be two words, **Release Date**.



Update the model

Update `Models/Movie.cs` with the following highlighted code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
```

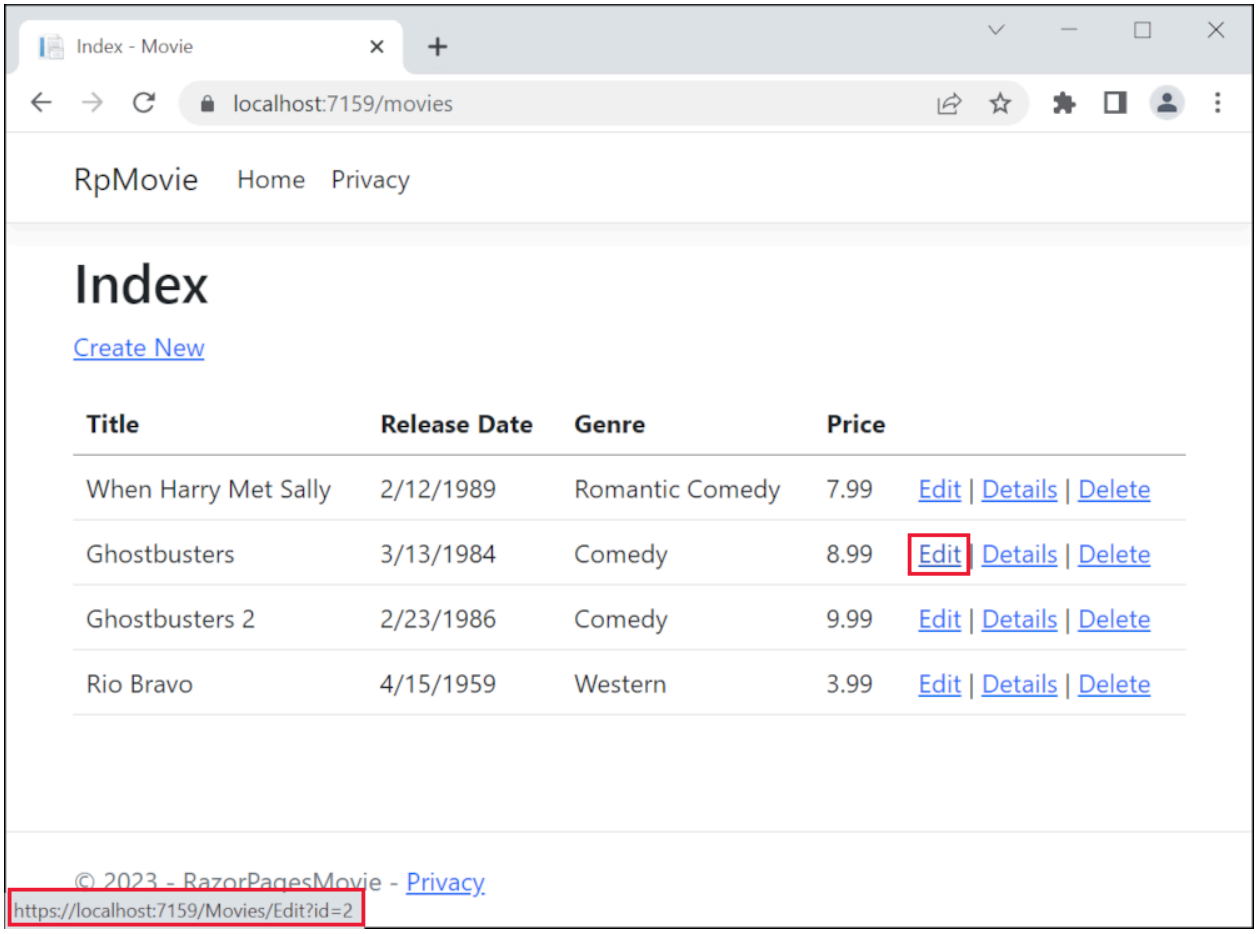
```
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; } = string.Empty;

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
```

In the previous code:

- The `[Column(TypeName = "decimal(18, 2)")]` data annotation enables Entity Framework Core to correctly map `Price` to currency in the database. For more information, see Data Types.
- The [Display] attribute specifies the display name of a field. In the preceding code, `Release Date` instead of `ReleaseDate`.
- The [DataType] attribute specifies the type of the data (`Date`). The time information stored in the field isn't displayed.

DataAnnotations is covered in the next tutorial.

Browse to *Pages/Movies* and hover over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the `Pages/Movies/Index.cshtml` file.

CSHTML

```
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.

In the preceding code, the Anchor Tag Helper dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route identifier (`asp-route-id`). For more information, see URL generation for Pages.

Use **View Source** from a browser to examine the generated markup. A portion of the generated HTML is shown below:

HTML

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically generated links pass the movie ID with a query string. For example, the `?id=1` in `https://localhost:5001/Movies/Details?id=1`.

Add route template

Update the Edit, Details, and Delete Razor Pages to use the `{id:int}` route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`. Run the app and then view source.

The generated HTML adds the ID to the path portion of the URL:

HTML

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the `{id:int}` route template that does **not** include the integer returns an HTTP 404 (not found) error. For example, `https://localhost:5001/Movies/Details` returns a 404 error. To make the ID optional, append `?` to the route constraint:

CSHTML

```
@page "{id:int?}"
```

Test the behavior of `@page "{id:int?}"`:

1. Set the page directive in `Pages/Movies/Details.cshtml` to `@page "{id:int?}"`.
2. Set a break point in `public async Task<IActionResult> OnGetAsync(int? id)`, in `Pages/Movies/Details.cshtml.cs`.
3. Navigate to `https://localhost:5001/Movies/Details/`.

With the `@page "{id:int}"` directive, the break point is never hit. The routing engine returns HTTP 404. Using `@page "{id:int?}"`, the `OnGetAsync` method returns `NotFound` (HTTP 404):

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
```

```
    if (id == null)
    {
        return NotFound();
    }

    Movie = await _context.Movie.FirstOrDefaultAsync(m => m.ID == id);

    if (Movie == null)
    {
        return NotFound();
    }
    return Page();
}
```

Review concurrency exception handling

Review the `OnPostAsync` method in the `Pages/Movies/Edit.cshtml.cs` file:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.Id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
```

```
    return _context.Movie.Any(e => e.Id == id);
}
```

The previous code detects concurrency exceptions when one client deletes the movie and the other client posts changes to the movie.

To test the `catch` block:

1. Set a breakpoint on `catch (DbUpdateConcurrencyException)`.
2. Select **Edit** for a movie, make changes, but don't enter **Save**.
3. In another browser window, select the **Delete** link for the same movie, and then delete the movie.
4. In the previous browser window, post changes to the movie.

Production code may want to detect concurrency conflicts. See Handle concurrency conflicts for more information.

Posting and binding review

Examine the `Pages/Movies/Edit.cshtml.cs` file:

C#

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; } = default!;

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null || _context.Movie == null)
        {
            return NotFound();
        }

        var movie =  await _context.Movie.FirstOrDefaultAsync(m => m.Id ==
id);
        if (movie == null)
        {
            return NotFound();
        }
        Movie = movie;
```

```
            return Page();
        }

        // To protect from overposting attacks, enable the specific properties
    you want to bind to.
        // For more details, see https://aka.ms/RazorPagesCRUD.
        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Attach(Movie).State = EntityState.Modified;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!MovieExists(Movie.Id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }

            return RedirectToPage("./Index");
        }

        private bool MovieExists(int id)
        {
          return _context.Movie.Any(e => e.Id == id);
        }
```

When an HTTP GET request is made to the Movies/Edit page, for example, `https://localhost:5001/Movies/Edit/3`:

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the `Pages/Movies/Edit.cshtml` Razor Page. The `Pages/Movies/Edit.cshtml` file contains the model directive `@model RazorPagesMovie.Pages.Movies.EditModel`, which makes the movie model available on the page.
- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables Model binding.

  ```
  C#

  [BindProperty]
  public Movie Movie { get; set; }
  ```

- If there are errors in the model state, for example, `ReleaseDate` cannot be converted to a date, the form is redisplayed with the submitted values.
- If there are no model errors, the movie is saved.

The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Next steps

Previous: Work with a database | Next: Add search

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 6, add search to ASP.NET Core Razor Pages

Article • 06/23/2024

By Rick Anderson

In the following sections, searching movies by *genre* or *name* is added.

Add the following highlighted code to `Pages/Movies/Index.cshtml.cs`:

C#

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get;set; }  = default!;

    [BindProperty(SupportsGet = true)]
    public string? SearchString { get; set; }

    public SelectList? Genres { get; set; }

    [BindProperty(SupportsGet = true)]
    public string? MovieGenre { get; set; }
```

In the previous code:

- `SearchString`: Contains the text users enter in the search text box. `SearchString` has the [BindProperty] attribute. `[BindProperty]` binds form values and query strings with the same name as the property. `[BindProperty(SupportsGet = true)]` is required for binding on HTTP GET requests.
- `Genres`: Contains the list of genres. `Genres` allows the user to select a genre from the list. `SelectList` requires `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`: Contains the specific genre the user selects. For example, "Western".
- `Genres` and `MovieGenre` are used later in this tutorial.

> ⚠ **Warning**

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the [BindProperty] attribute's `SupportsGet` property to `true`:

C#

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Update the Index page's `OnGetAsync` method with the following code:

C#

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a LINQ query to select the movies:

C#

```
// using System.Linq;
var movies = from m in _context.Movie
             select m;
```

The query is only ***defined*** at this point, it has ***not*** been run against the database.

If the `SearchString` property is not `null` or empty, the movies query is modified to filter on the search string:

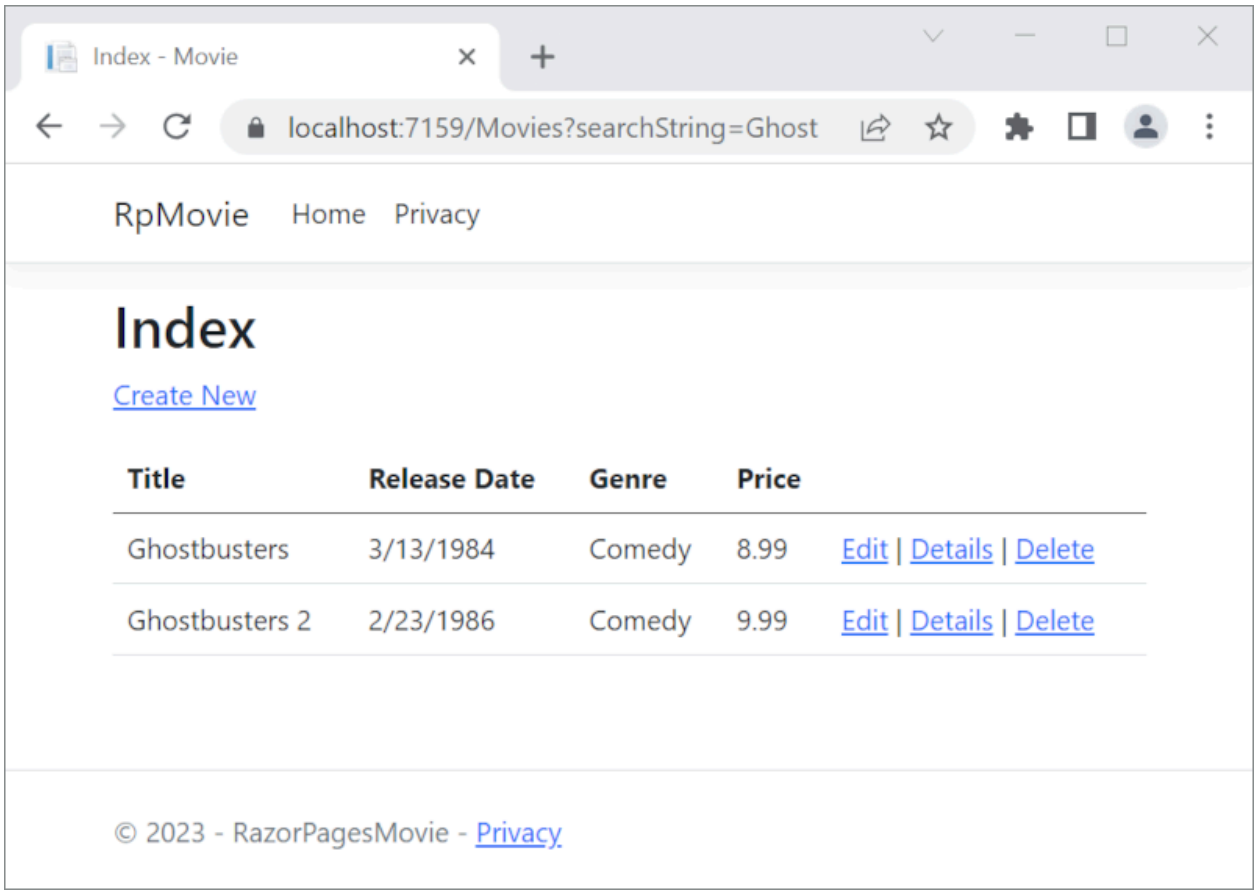
C#

```
if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}
```

The `s => s.Title.Contains()` code is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains`. LINQ queries are not executed when they're defined or when they're modified by calling a method, such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. The evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See Query Execution for more information.

> **Note**
>
> The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. SQLite with the default collation is a mixture of case sensitive and case ***IN***sensitive, depending on the query. For information on making case insensitive SQLite queries, see the following:
>
> - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
> - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
> - Collations and Case Sensitivity

Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL. For example, `https://localhost:5001/Movies?searchString=Ghost`. The filtered movies are displayed.

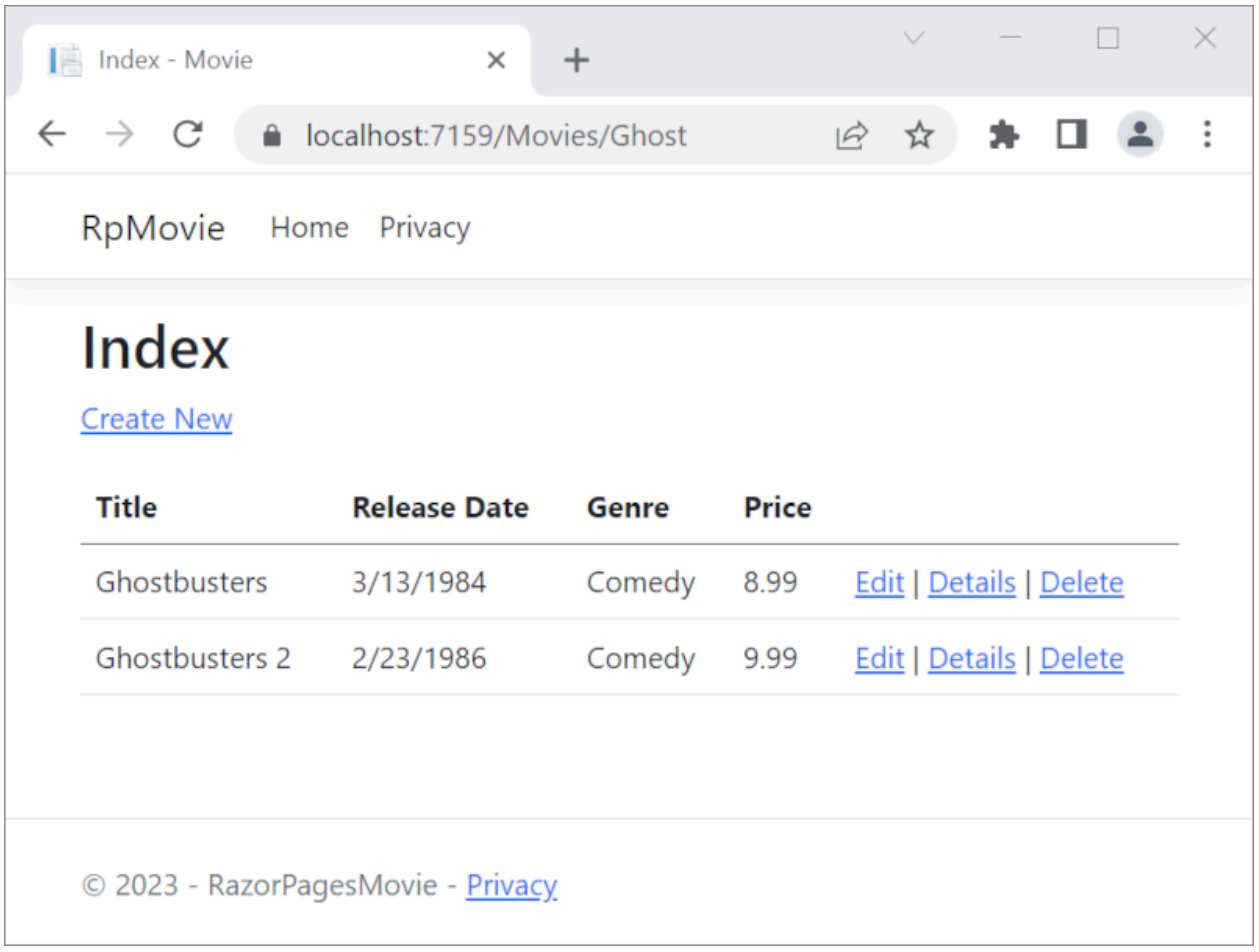


If the following route template is added to the Index page, the search string can be passed as a URL segment. For example, `https://localhost:5001/Movies/Ghost`.

CSHTML

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



The ASP.NET Core runtime uses model binding to set the value of the `SearchString` property from the query string (`?searchString=Ghost`) or route data (`https://localhost:5001/Movies/Ghost`). Model binding is ***not*** case sensitive.

However, users cannot be expected to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the `Pages/Movies/Index.cshtml` file, and add the markup highlighted in the following code:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
```

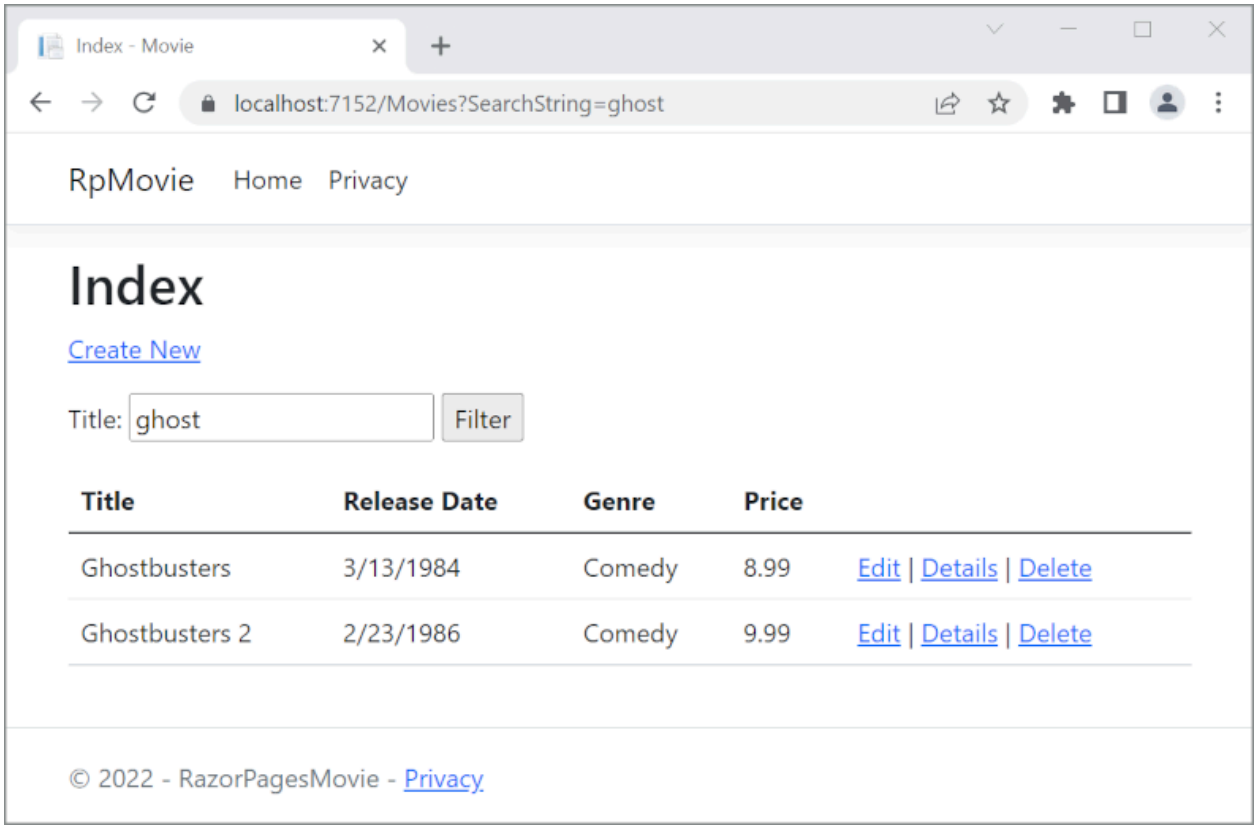
```
<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

```
<table class="table">
    @*Markup removed for brevity.*@
```

The HTML `<form>` tag uses the following Tag Helpers:

- Form Tag Helper. When the form is submitted, the filter string is sent to the *Pages/Movies/Index* page via query string.
- Input Tag Helper

Save the changes and test the filter.



Search by genre

Update the `Movies/Index.cshtml.cs` page `OnGetAsync` method with the following code:

C#

```
public async Task OnGetAsync()
{
```

```
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

The following code is a LINQ query that retrieves all the genres from the database.

C#

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres.

C#

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Add search by genre to the Razor Page

Update the `Index.cshtml` <form> element as highlighted in the following markup:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

```
<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

Test the app by searching by genre, by movie title, and by both.

Next steps

Previous: Update the pages | Next: Add a new field

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 7, add a new field to a Razor Page in ASP.NET Core

Article • 06/23/2024

By Rick Anderson

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field schema change to the database.

When using EF Code First to automatically create and track a database, Code First:

- Adds an __EFMigrationsHistory table to the database to track whether the schema of the database is in sync with the model classes it was generated from.
- Throws an exception if the model classes aren't in sync with the database.

Automatic verification that the schema and model are in sync makes it easier to find inconsistent database code issues.

Adding a Rating Property to the Movie Model

1. Open the `Models/Movie.cs` file and add a `Rating` property:

   C#

   ```
   public class Movie
   {
       public int Id { get; set; }
       public string Title { get; set; } = string.Empty;

       [Display(Name = "Release Date")]
       [DataType(DataType.Date)]
       public DateTime ReleaseDate { get; set; }
       public string Genre { get; set; } = string.Empty;

       [Column(TypeName = "decimal(18, 2)")]
       public decimal Price { get; set; }
       public string Rating { get; set; } = string.Empty;
   }
   ```

2. Edit `Pages/Movies/Index.cshtml`, and add a `Rating` field:

   CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
```

```
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-
id="@item.Id">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

3. Update the following pages with a `Rating` field:

   - *Pages/Movies/Create.cshtml*.
   - *Pages/Movies/Delete.cshtml*.
   - *Pages/Movies/Details.cshtml*.
   - *Pages/Movies/Edit.cshtml*.

The app won't work until the database is updated to include the new field. Running the app without an update to the database throws a `SqlException`:

```
SqlException: Invalid column name 'Rating'.
```

The `SqlException` exception is caused by the updated Movie model class being different than the schema of the Movie table of the database. There's no `Rating` column in the database table.

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle, it allows developers to quickly evolve the model and database schema together. The downside is that existing data in the database is lost. Don't use this approach on a production database! Dropping the database on schema changes

and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is to keep the data. Make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but make this change for each `new Movie` block.

C#

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Price = 7.99M,
        Rating = "R"
    },
```

See the completed SeedData.cs file.

Build the app

Visual Studio

Press `Ctrl`+`Shift`+`B`

Visual Studio

Add a migration for the rating field

1. From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.

2. In the PMC, enter the following commands:

PowerShell

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` database schema.
- Create code to migrate the database schema to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

The `Update-Database` command tells the framework to apply the schema changes to the database and to preserve existing data.

Delete all the records in the database, the initializer will seed the database and include the `Rating` field. Deleting can be done with the delete links in the browser or from Sql Server Object Explorer (SSOX).

Another option is to delete the database and use migrations to re-create the database. To delete the database in SSOX:

1. Select the database in SSOX.
2. Right-click on the database, and select **Delete**.
3. Check **Close existing connections**.
4. Select **OK**.
5. In the PMC, update the database:

```
PowerShell
Update-Database
```

Run the app and verify you can create, edit, and display movies with a `Rating` field. If the database isn't seeded, set a break point in the `SeedData.Initialize` method.

Next steps

Previous: Add Search | Next: Add Validation

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 8 of tutorial series on Razor Pages

Article • 06/23/2024

By Rick Anderson

In this section, validation logic is added to the `Movie` model. The validation rules are enforced any time a user creates or edits a movie.

Validation

A key tenet of software development is called DRY ("**D**on't **R**epeat **Y**ourself"). Razor Pages encourages development where functionality is specified once, and it's reflected throughout the app. DRY can help:

- Reduce the amount of code in an app.
- Make the code less error prone, and easier to test and maintain.

The validation support provided by Razor Pages and Entity Framework is a good example of the DRY principle:

- Validation rules are declaratively specified in one place, in the model class.
- Rules are enforced everywhere in the app.

Add validation rules to the movie model

The System.ComponentModel.DataAnnotations namespace provides:

- A set of built-in validation attributes that are applied declaratively to a class or property.
- Formatting attributes like `[DataType]` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `[Required]`, `[StringLength]`, `[RegularExpression]`, and `[Range]` validation attributes.

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models;

public class Movie
```

```
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; } = string.Empty;

    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; } = string.Empty;

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; } = string.Empty;
}
```

The validation attributes specify behavior to enforce on the model properties they're applied to:

- The `[Required]` and `[MinimumLength]` attributes indicate that a property must have a value. Nothing prevents a user from entering white space to satisfy this validation.
- The `[RegularExpression]` attribute is used to limit what characters can be input. In the preceding code, `Genre`:
  - Must only use letters.
  - The first letter must be uppercase. White spaces are allowed, while numbers and special characters aren't allowed.
- The `RegularExpression` `Rating`:
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a `Genre`.
- The `[Range]` attribute constrains a value to within a specified range.
- The `[StringLength]` attribute can set a maximum length of a string property, and optionally its minimum length.

- Value types, such as `decimal`, `int`, `float`, `DateTime`, are inherently required and don't need the `[Required]` attribute.

The preceding validation rules are used for demonstration, they are not optimal for a production system. For example, the preceding prevents entering a movie with only two chars and doesn't allow special characters in `Genre`.

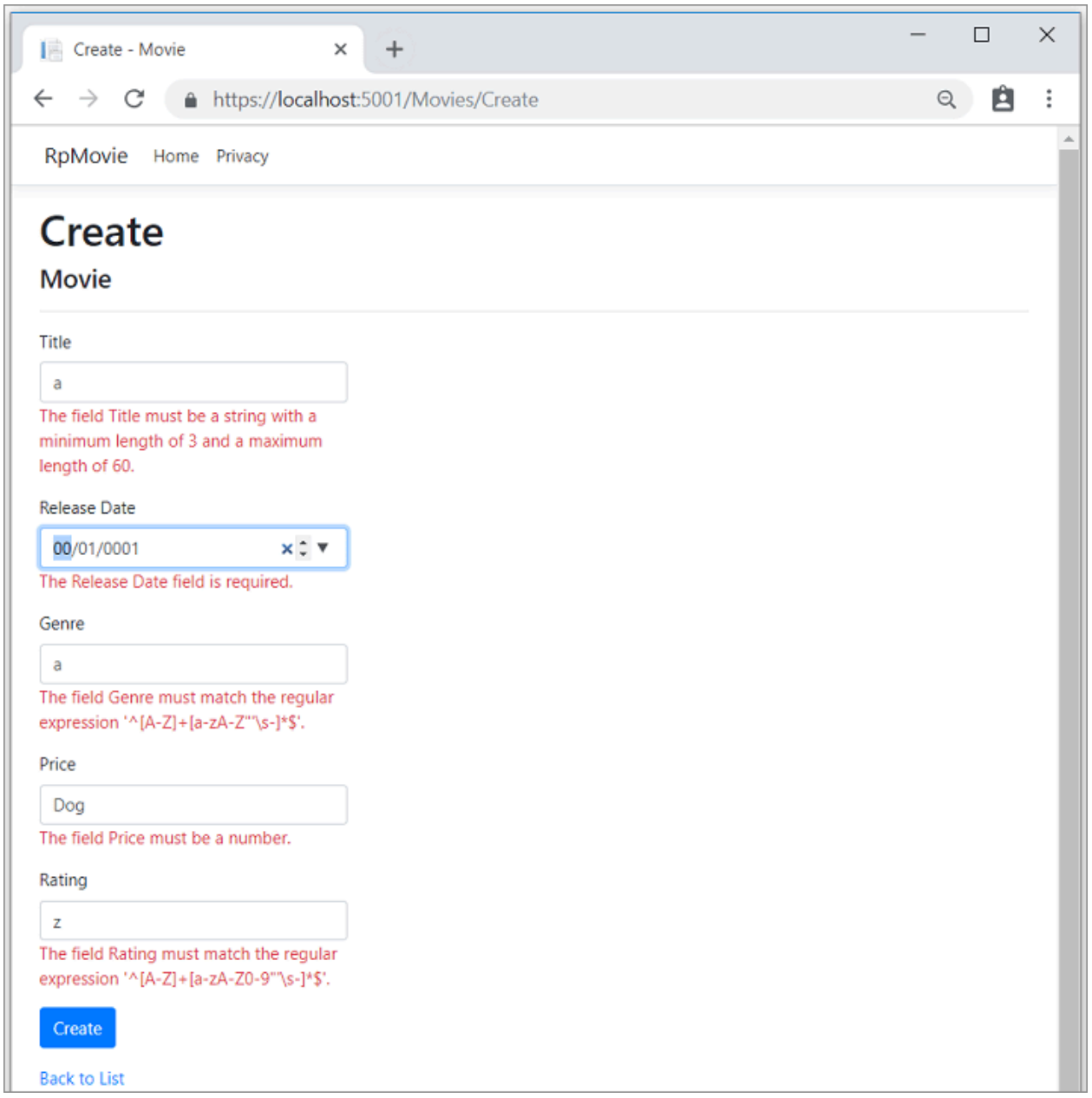
Having validation rules automatically enforced by ASP.NET Core helps:

- Make the app more robust.
- Reduce chances of saving invalid data to the database.

Validation Error UI in Razor Pages

Run the app and navigate to Pages/Movies.

Select the **Create New** link. Complete the form with some invalid values. When jQuery client-side validation detects the error, it displays an error message.



> **Note**
>
> You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered a validation error message in each field containing an invalid value. The errors are enforced both client-side, using JavaScript and jQuery, and server-side, when a user has JavaScript disabled.

A significant benefit is that **no** code changes were necessary in the Create or Edit pages. Once data annotations were applied to the model, the validation UI was enabled. The Razor Pages created in this tutorial automatically picked up the validation rules, using

validation attributes on the properties of the `Movie` model class. Test validation using the Edit page, the same validation is applied.

The form data isn't posted to the server until there are no client-side validation errors. Verify form data isn't posted by one or more of the following approaches:

- Put a break point in the `OnPostAsync` method. Submit the form by selecting **Create** or **Save**. The break point is never hit.
- Use the Fiddler tool.
- Use the browser developer tools to monitor network traffic.

Server-side validation

When JavaScript is disabled in the browser, submitting the form with errors will post to the server.

Optional, test server-side validation:

1. Disable JavaScript in the browser. JavaScript can be disabled using browser's developer tools. If JavaScript cannot be disabled in the browser, try another browser.
2. Set a break point in the `OnPostAsync` method of the Create or Edit page.
3. Submit a form with invalid data.
4. Verify the model state is invalid:

```
C#
```

```
if (!ModelState.IsValid)
{
    return Page();
}
```

Alternatively, Disable client-side validation on the server.

The following code shows a portion of the `Create.cshtml` page scaffolded earlier in the tutorial. It's used by the Create and Edit pages to:

- Display the initial form.
- Redisplay the form in the event of an error.

```
CSHTML
```

```
<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Movie.Title" class="control-label"></label>
        <input asp-for="Movie.Title" class="form-control" />
        <span asp-validation-for="Movie.Title" class="text-danger"></span>
    </div>
```

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side. The Validation Tag Helper displays validation errors. See Validation for more information.

The Create and Edit pages have no validation rules in them. The validation rules and the error strings are specified only in the `Movie` class. These validation rules are automatically applied to Razor Pages that edit the `Movie` model.

When validation logic needs to change, it's done only in the model. Validation is applied consistently throughout the app, validation logic is defined in one place. Validation in one place helps keep the code clean, and makes it easier to maintain and update.

Use DataType Attributes

Examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. The `[DataType]` attribute is applied to the `ReleaseDate` and `Price` properties.

C#

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

The `[DataType]` attributes provide:

- Hints for the view engine to format the data.
- Supplies attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email.

Use the `[RegularExpression]` attribute to validate the format of the data. The `[DataType]` attribute is used to specify a data type that's more specific than the

database intrinsic type. `[DataType]` attributes aren't validation attributes. In the sample app, only the date is displayed, without time.

The `DataType` enumeration provides many data types, such as `Date`, `Time`, `PhoneNumber`, `Currency`, `EmailAddress`, and more.

The `[DataType]` attributes:

- Can enable the app to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`.
- Can provide a date selector `DataType.Date` in browsers that support HTML5.
- Emit HTML 5 `data-`, pronounced "data dash", attributes that HTML 5 browsers consume.
- Do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

The `[DisplayFormat]` attribute is used to explicitly specify the date format:

```
C#

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting will be applied when the value is displayed for editing. That behavior may not be wanted for some fields. For example, in currency values, the currency symbol is usually not wanted in the edit UI.

The `[DisplayFormat]` attribute can be used by itself, but it's generally a good idea to use the `[DataType]` attribute. The `[DataType]` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `[DataType]` attribute provides the following benefits that aren't available with `[DisplayFormat]`:

- The browser can enable HTML5 features, for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.
- By default, the browser renders data using the correct format based on its locale.

- The `[DataType]` attribute can enable the ASP.NET Core framework to choose the right field template to render the data. The `DisplayFormat`, if used by itself, uses the string template.

Note: jQuery validation doesn't work with the `[Range]` attribute and `DateTime`. For example, the following code will always display a client-side validation error, even when the date is in the specified range:

C#

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

It's a best practice to avoid compiling hard dates in models, so using the `[Range]` attribute and `DateTime` is discouraged. Use Configuration for date ranges and other values that are subject to frequent change rather than specifying it in code.

The following code shows combining attributes on one line:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; } = string.Empty;

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$"), Required, StringLength(30)]
    public string Genre { get; set; } = string.Empty;

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; } = string.Empty;
}
```

Get started with Razor Pages and EF Core shows advanced EF Core operations with Razor Pages.

Apply migrations

The DataAnnotations applied to the class changes the schema. For example, the DataAnnotations applied to the `Title` field:

C#

```
[StringLength(60, MinimumLength = 3)]
[Required]
public string Title { get; set; } = string.Empty;
```

- Limits the characters to 60.
- Doesn't allow a `null` value.

The `Movie` table currently has the following schema:

SQL

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (MAX)  NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (MAX)  NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (MAX)  NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

The preceding schema changes don't cause EF to throw an exception. However, create a migration so the schema is consistent with the model.

Visual Studio

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the PMC, enter the following commands:

PowerShell

```
Add-Migration New_DataAnnotations
Update-Database
```

`Update-Database` runs the `Up` method of the `New_DataAnnotations` class.

Examine the `Up` method:

C#

```
public partial class New_DataAnnotations : Migration
{
    /// <inheritdoc />
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "Title",
            table: "Movie",
            type: "nvarchar(60)",
            maxLength: 60,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");

        migrationBuilder.AlterColumn<string>(
            name: "Rating",
            table: "Movie",
            type: "nvarchar(5)",
            maxLength: 5,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");

        migrationBuilder.AlterColumn<string>(
            name: "Genre",
            table: "Movie",
            type: "nvarchar(30)",
            maxLength: 30,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");
    }
```

The updated `Movie` table has the following schema:

SQL

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (60)   NOT NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (30)   NOT NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (5)    NOT NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core app in Azure with SQL Database.

Thanks for completing this introduction to Razor Pages. Get started with Razor Pages and EF Core is an excellent follow up to this tutorial.

Additional resources

- Tag Helpers in forms in ASP.NET Core
- Globalization and localization in ASP.NET Core
- Tag Helpers in ASP.NET Core
- Author Tag Helpers in ASP.NET Core

Next steps

Previous: Add a new field

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Get started with ASP.NET Core MVC

Article • 07/30/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Rick Anderson

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point. See Choose an ASP.NET Core UI, which compares Razor Pages, MVC, and Blazor for UI development.

This is the first tutorial of a series that teaches ASP.NET Core MVC web development with controllers and views.

At the end of the series, you'll have an app that manages and displays movie data. You learn how to:

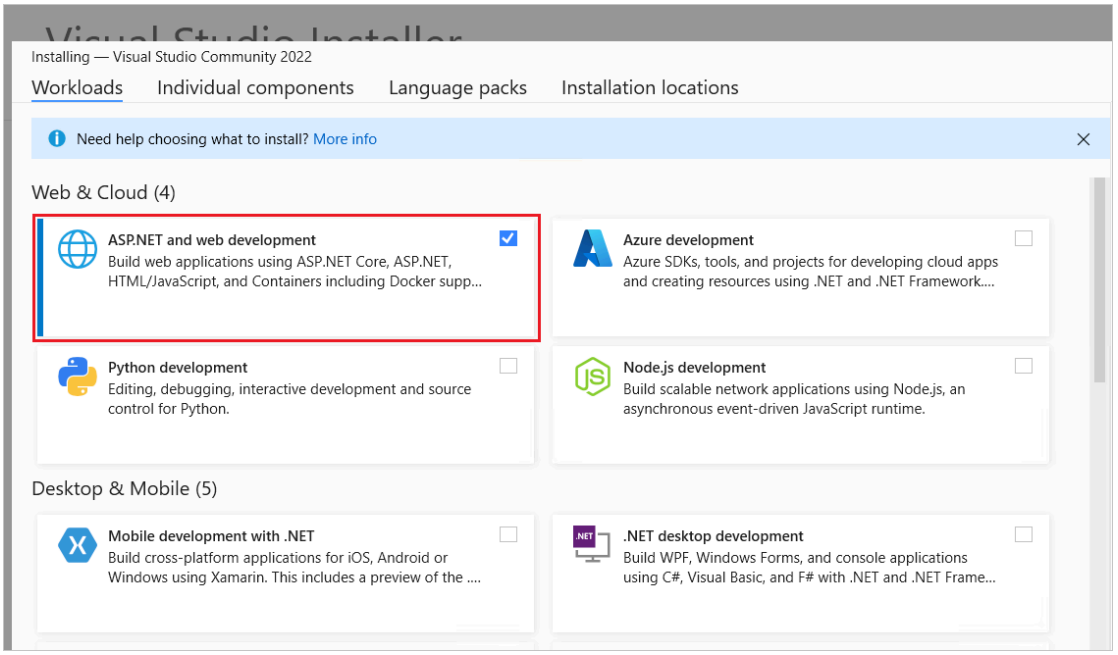
- ✓ Create a web app.
- ✓ Add and scaffold a model.
- ✓ Work with a database.
- ✓ Add search and validation.

View or download sample code (how to download).

Prerequisites

Visual Studio

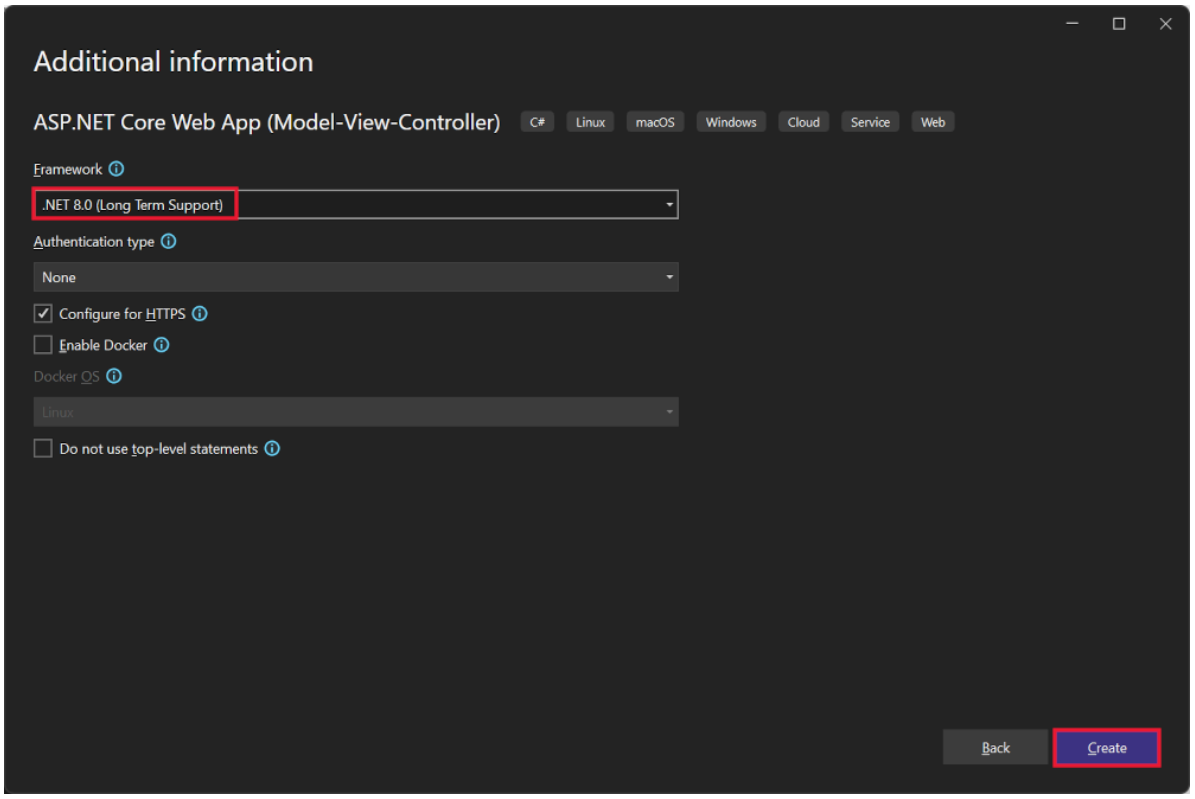
- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a web app

Visual Studio

- Start Visual Studio and select **Create a new project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web App (Model-View-Controller)** > **Next**.
- In the **Configure your new project** dialog:
  - Enter `MvcMovie` for **Project name**. It's important to name the project *MvcMovie*. Capitalization needs to match each `namespace` when code is copied.
  - The **Location** for the project can be set to anywhere.
- Select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 8.0 (Long Term Support)**.
  - Verify that **Do not use top-level statements** is unchecked.
- Select **Create**.



For more information, including alternative approaches to create the project, see Create a new project in Visual Studio.

Visual Studio uses the default project template for the created MVC project. The created project:

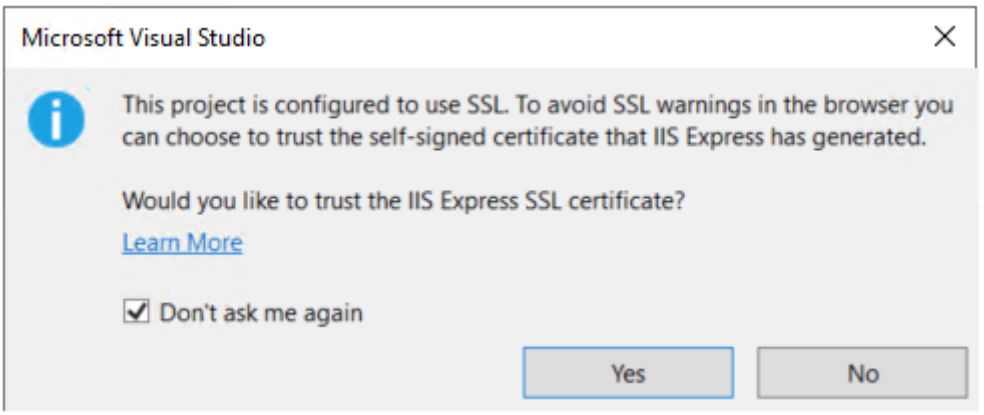
- Is a working app.
- Is a basic starter project.

Run the app

Visual Studio

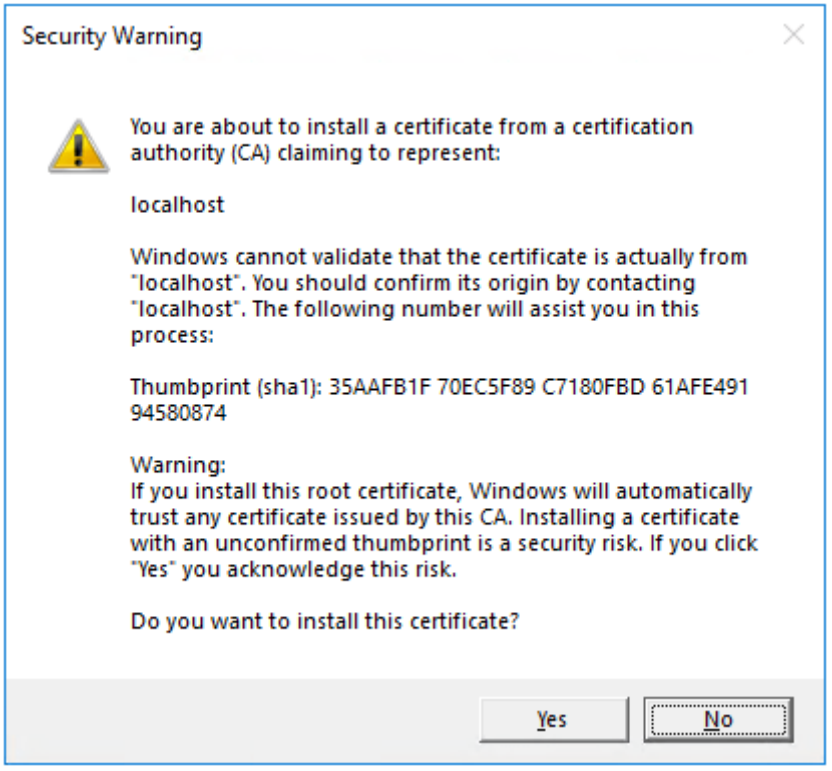
- Press `Ctrl`+`F5` to run the app without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio runs the app and opens the default browser.

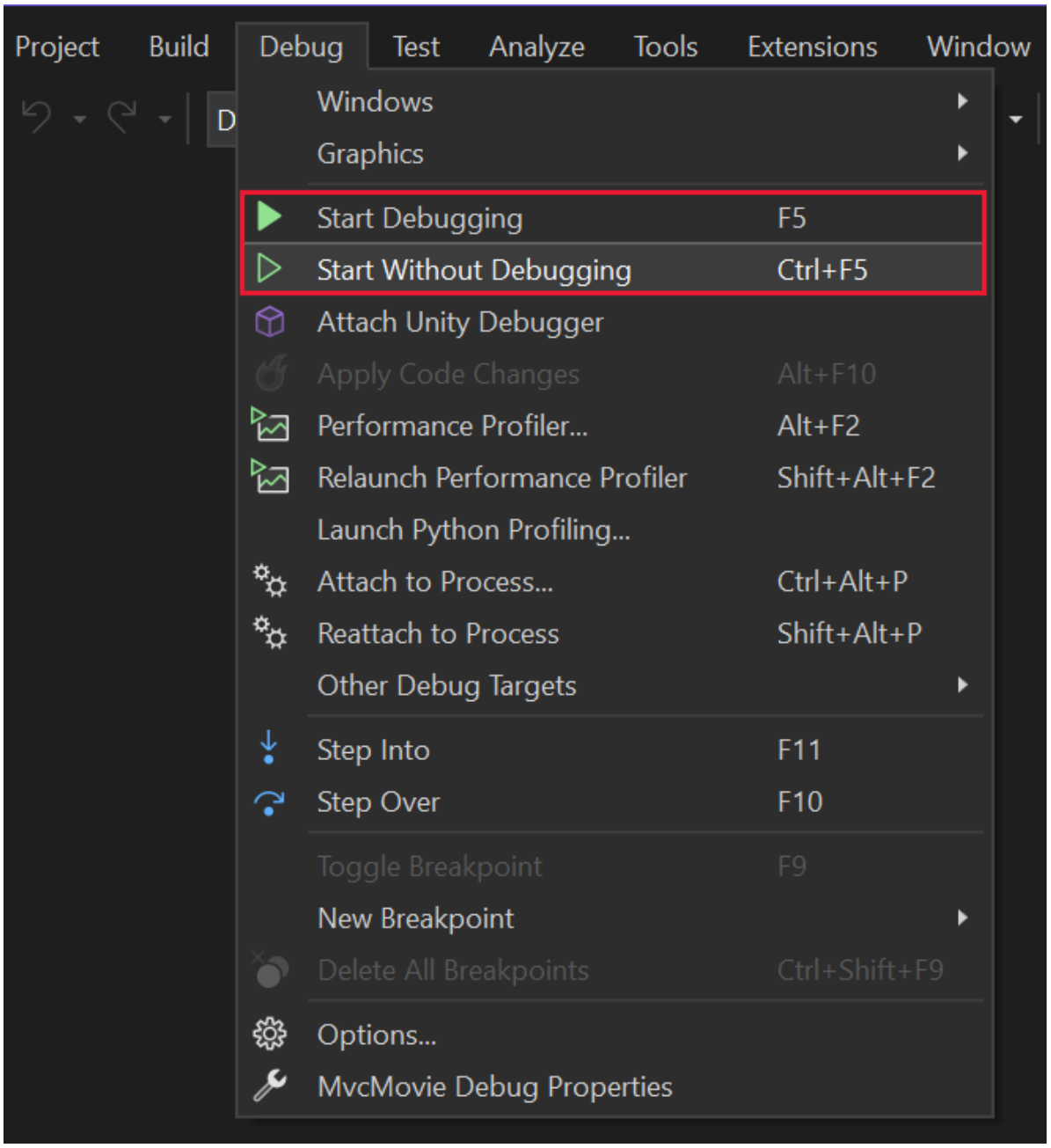
The address bar shows `localhost:<port#>` and not something like `example.com`. The standard hostname for your local computer is `localhost`. When Visual Studio creates a web project, a random port is used for the web server.

Launching the app without debugging by pressing `Ctrl`+`F5` allows you to:

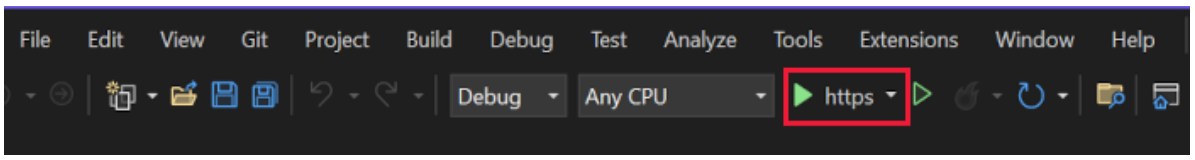
- Make code changes.

- Save the file.
- Quickly refresh the browser and see the code changes.

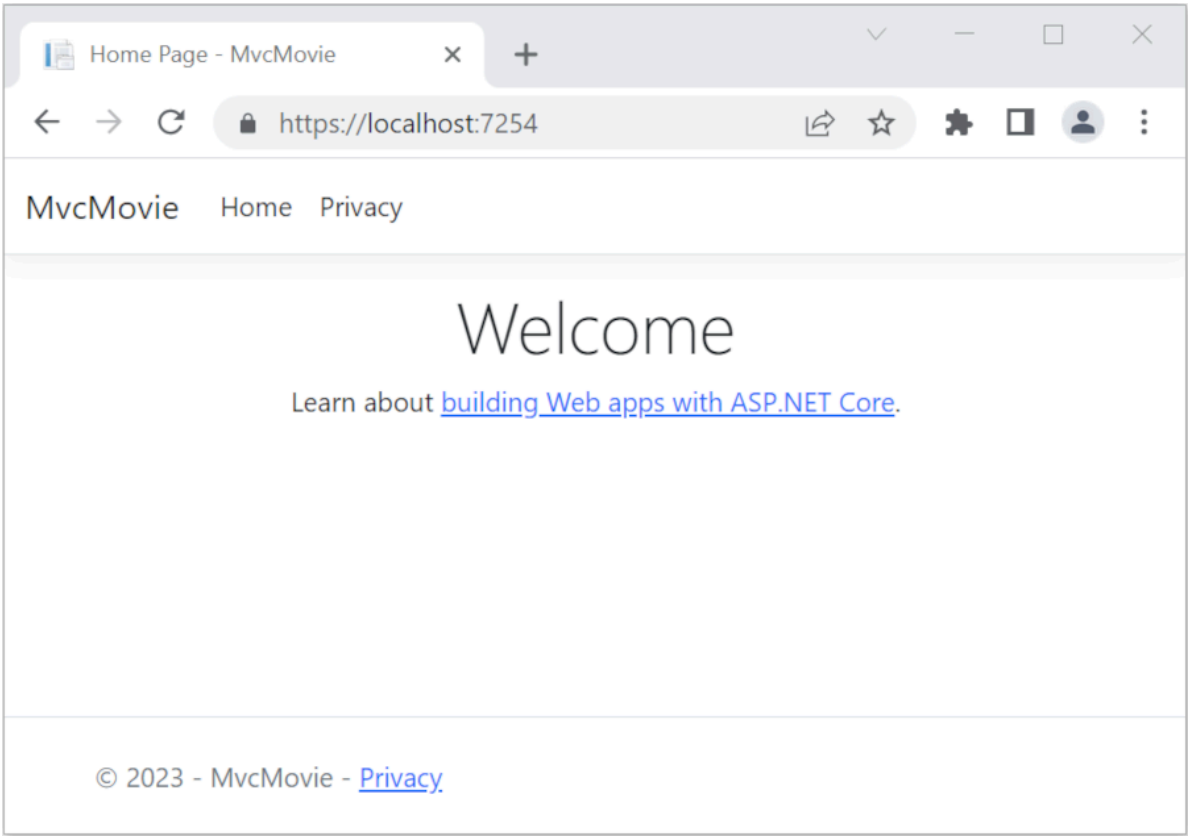
You can launch the app in debug or non-debug mode from the **Debug** menu:



You can debug the app by selecting the **https** button in the toolbar:



The following image shows the app:



- Close the browser window. Visual Studio will stop the application.

Visual Studio

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next tutorial in this series, you learn about MVC and start writing some code.

Next: Add a controller

Part 2, add a controller to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.

MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that:
  - Handle browser requests.
  - Retrieve model data.
  - Call view templates that return a response.

In an MVC app, the view only displays information. The controller handles and responds to user input and interaction. For example, the controller handles URL segments and query-string values, and passes these values to the model. The model might use these values to query the database. For example:

- `https://localhost:5001/Home/Privacy`: specifies the `Home` controller and the `Privacy` action.
- `https://localhost:5001/Movies/Edit/5`: is a request to edit the movie with ID=5 using the `Movies` controller and the `Edit` action, which are detailed later in the tutorial.

Route data is explained later in the tutorial.

The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps manage complexity when building an app,

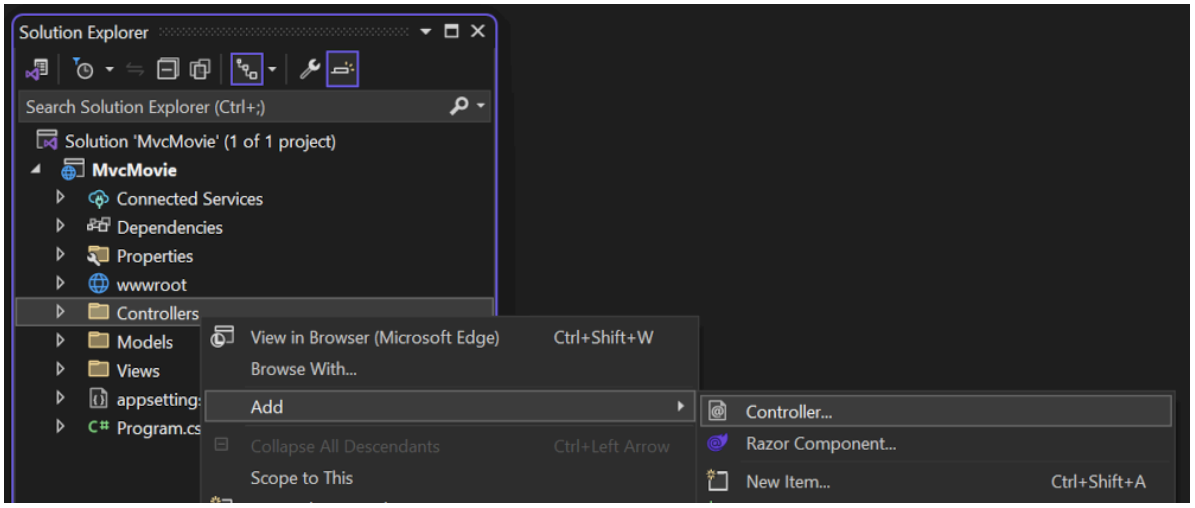
because it enables work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

These concepts are introduced and demonstrated in this tutorial series while building a movie app. The MVC project contains folders for the *Controllers* and *Views*.

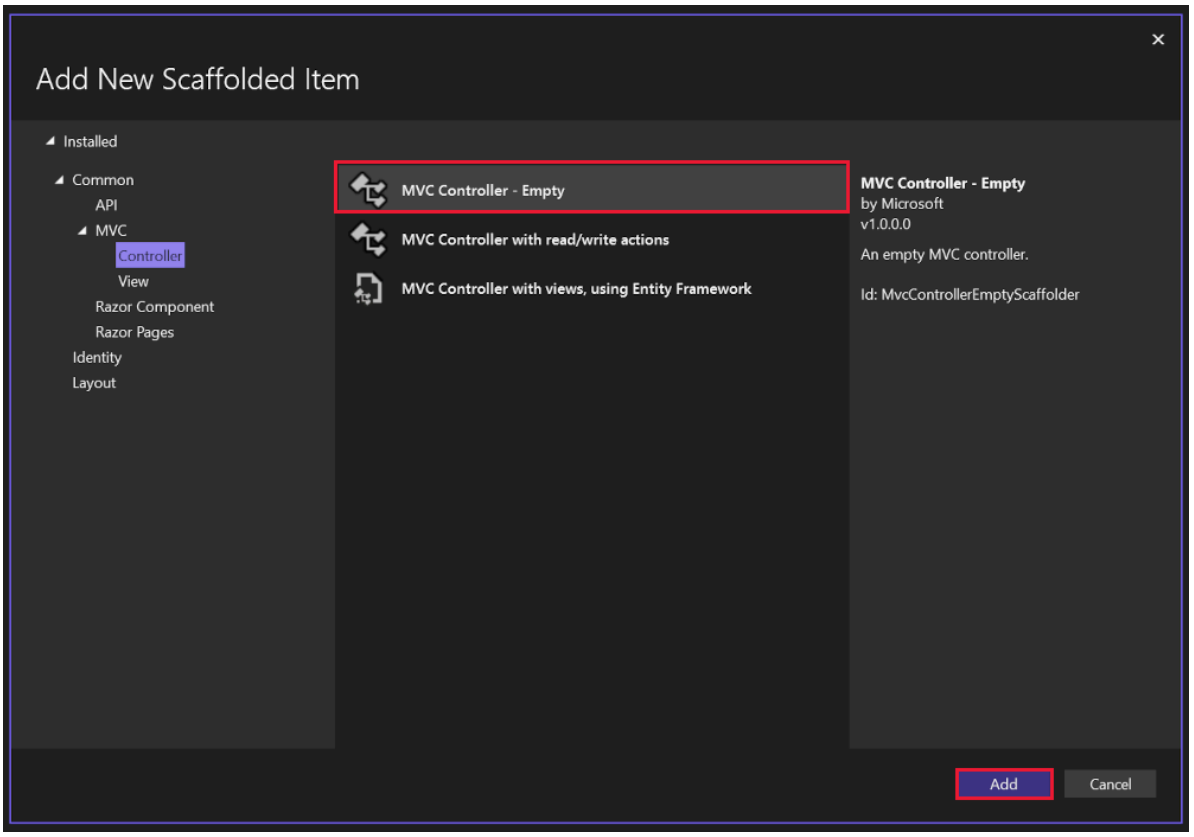
Add a controller

Visual Studio

In **Solution Explorer**, right-click **Controllers** > **Add** > **Controller**.



In the **Add New Scaffolded Item** dialog box, select **MVC Controller - Empty** > **Add**.



In the **Add New Item - MvcMovie** dialog, enter `HelloWorldController.cs` and select **Add**.

Replace the contents of `Controllers/HelloWorldController.cs` with the following code:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    //
    // GET: /HelloWorld/
    public string Index()
    {
        return "This is my default action...";
    }
    //
    // GET: /HelloWorld/Welcome/
    public string Welcome()
    {
        return "This is the Welcome action method...";
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint:

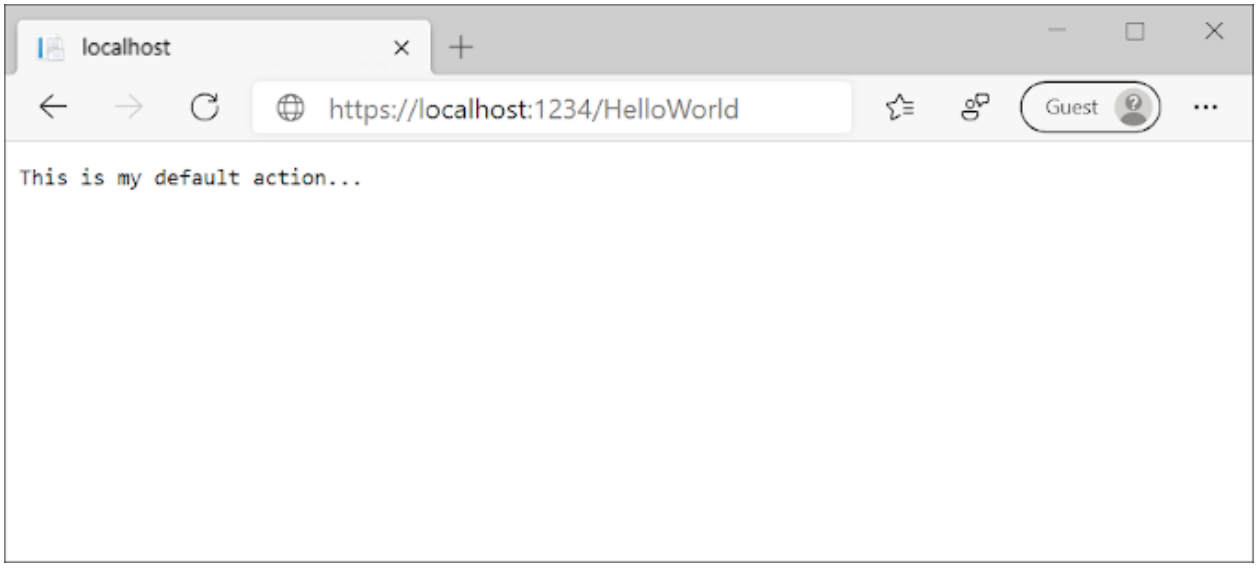
- Is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`.
- Combines:
  - The protocol used: `HTTPS`.
  - The network location of the web server, including the TCP port: `localhost:5001`.
  - The target URI: `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL.

The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial, the scaffolding engine is used to generate `HTTP POST` methods, which update data.

Run the app without the debugger by pressing `Ctrl`+`F5` (Windows) or `⌘`+`F5` (macOS).

Append `/HelloWorld` to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes, and the action methods within them, depending on the incoming URL. The default URL routing logic used by MVC, uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

The routing format is set in the `Program.cs` file.

C#

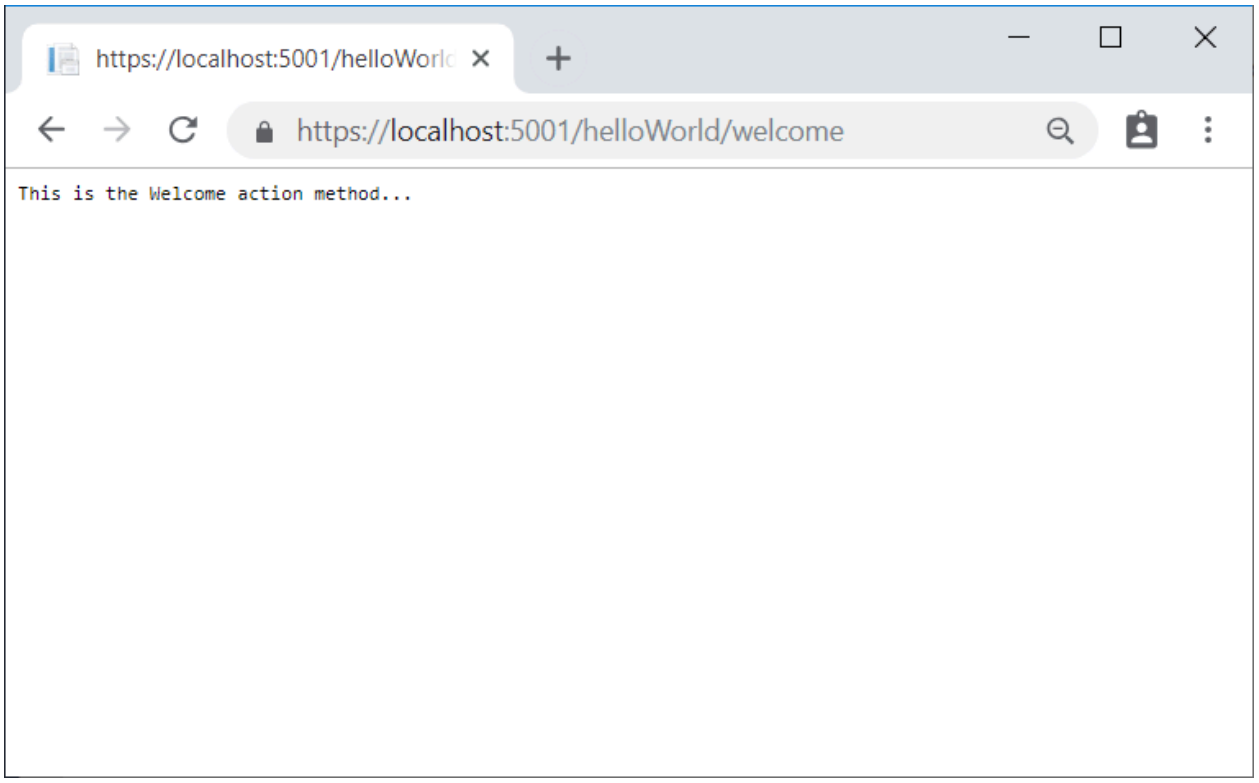
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:

- The first URL segment determines the controller class to run. So `localhost:5001/HelloWorld` maps to the **HelloWorld** Controller class.
- The second part of the URL segment determines the action method on the class. So `localhost:5001/HelloWorld/Index` causes the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:5001/HelloWorld` and the `Index` method was called by default. `Index` is the default method that will be called on a controller if a method name isn't explicitly specified.
- The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to: `https://localhost:{PORT}/HelloWorld/Welcome`. Replace `{PORT}` with your port number.

The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`.

Change the `Welcome` method to include two parameters as shown in the following code.

C#

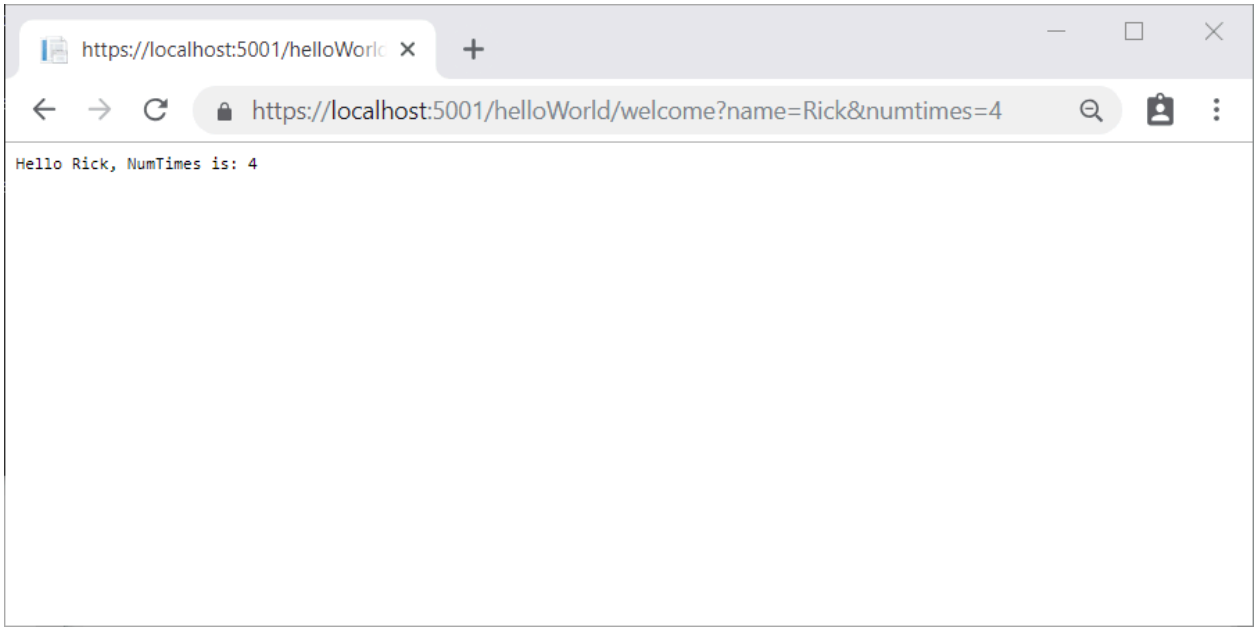
```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is:
{numTimes}");
}
```

The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input, such as through JavaScript.
- Uses Interpolated Strings in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to: `https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`. Replace `{PORT}` with your port number.

Try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string to parameters in the method. See Model Binding for more information.



In the previous image:

- The URL segment `Parameters` isn't used.
- The `name` and `numTimes` parameters are passed in the query string.
- The `?` (question mark) in the above URL is a separator, and the query string follows.
- The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

C#

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick`

In the preceding URL:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` starts the query string.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

In the preceding example:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` (in `id?`) indicates the `id` parameter is optional.

Previous: Get Started Next: Add a View

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 3, add a view to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section, you modify the `HelloWorldController` class to use Razor view files. This cleanly encapsulates the process of generating HTML responses to a client.

View templates are created using Razor. Razor-based view templates:

- Have a `.cshtml` file extension.
- Provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

C#

```
public IActionResult Index()
{
    return View();
}
```

The preceding code:

- Calls the controller's View method.
- Uses a view template to generate an HTML response.

Controller methods:

- Are referred to as *action methods*. For example, the `Index` action method in the preceding code.
- Generally return an IActionResult or a class derived from ActionResult, not a type like `string`.

Add a view

Visual Studio

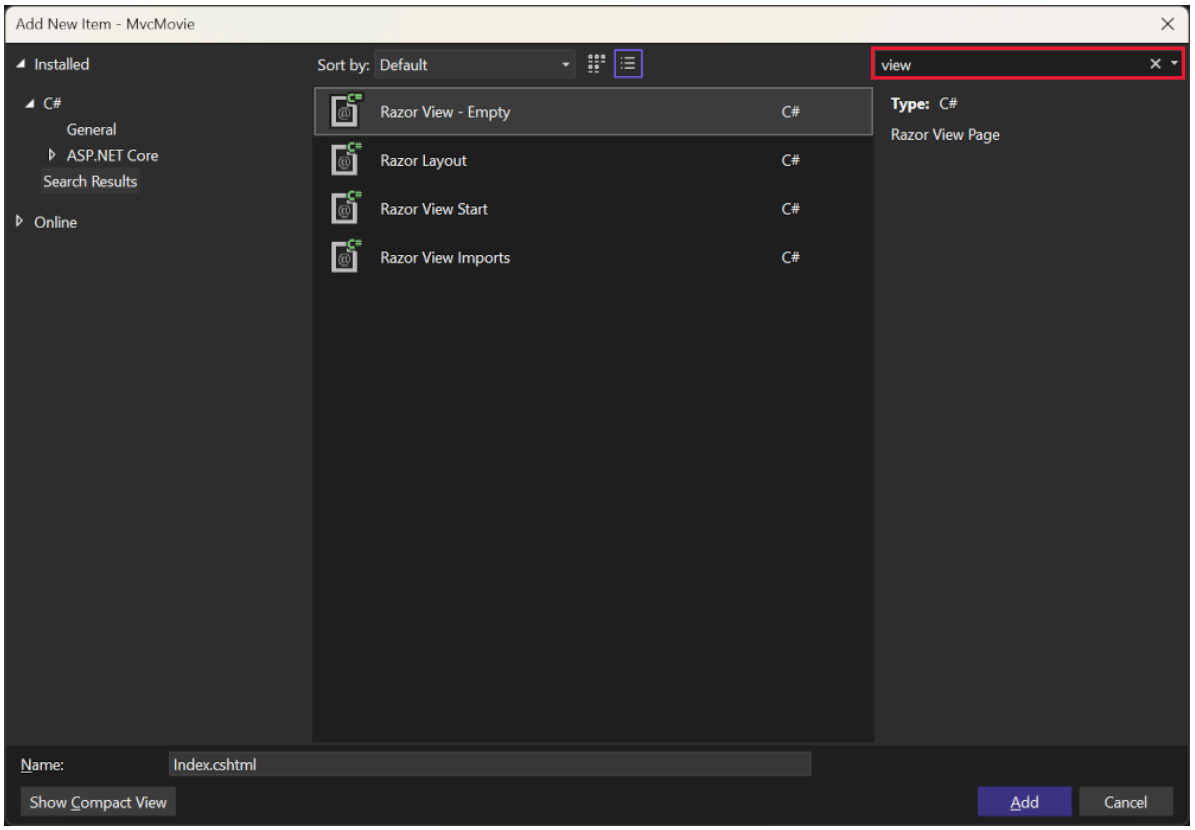
Right-click on the *Views* folder, and then **Add** > **New Folder** and name the folder *HelloWorld*.

Right-click on the *Views/HelloWorld* folder, and then **Add** > **New Item**.

In the **Add New Item** dialog select **Show All Templates**.

In the **Add New Item - MvcMovie** dialog:

- In the search box in the upper-right, enter *view*
- Select **Razor View - Empty**
- Keep the **Name** box value, `Index.cshtml`.
- Select **Add**



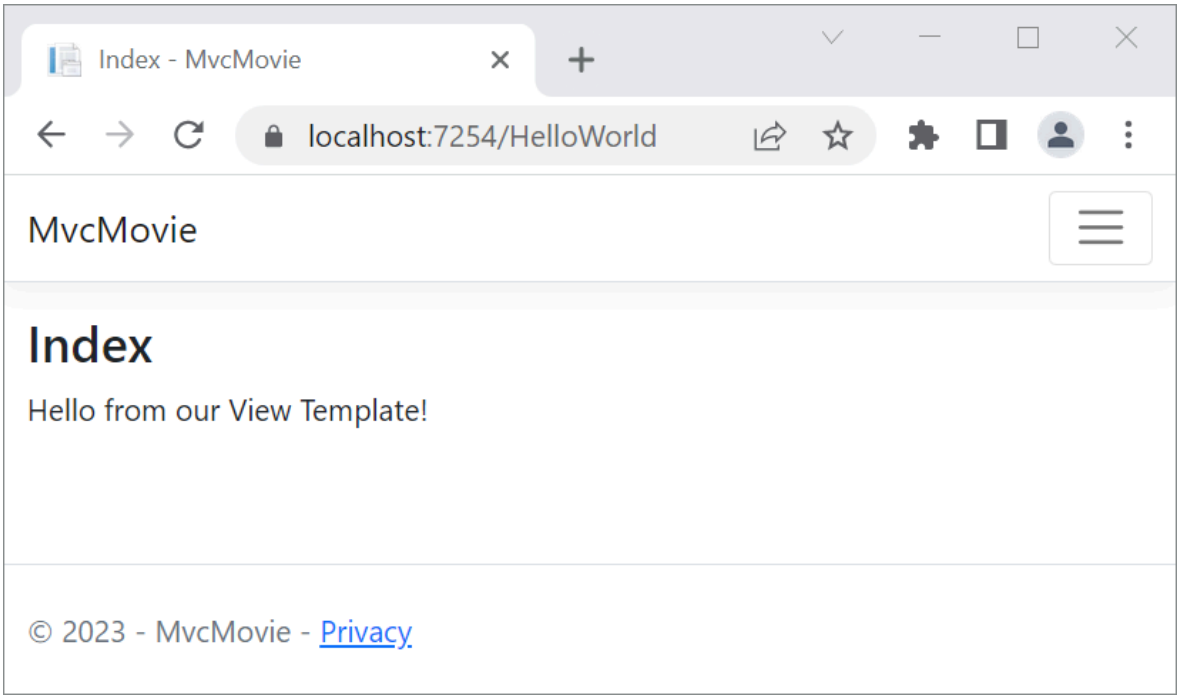
Replace the contents of the `Views/HelloWorld/Index.cshtml` Razor view file with the following:

CSHTML

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://localhost:{PORT}/HelloWorld`:

- The `Index` method in the `HelloWorldController` ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser.
- A view template file name wasn't specified, so MVC defaulted to using the default view file. When the view file name isn't specified, the default view is returned. The default view has the same name as the action method, `Index` in this example. The view template `/Views/HelloWorld/Index.cshtml` is used.
- The following image shows the string "Hello from our View Template!" hard-coded in the view:



Change views and layout pages

Select the menu links **MvcMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the `Views/Shared/_Layout.cshtml` file.

Open the `Views/Shared/_Layout.cshtml` file.

Layout templates allow:

- Specifying the HTML container layout of a site in one place.
- Applying the HTML container layout across multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the *`Views/Home/Privacy.cshtml`* view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

Replace the content of the `Views/Shared/_Layout.cshtml` file with the following markup. The changes are highlighted:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Movies"
asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
```

```
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2023 - Movie App - <a asp-area="" asp-controller="Home"
asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

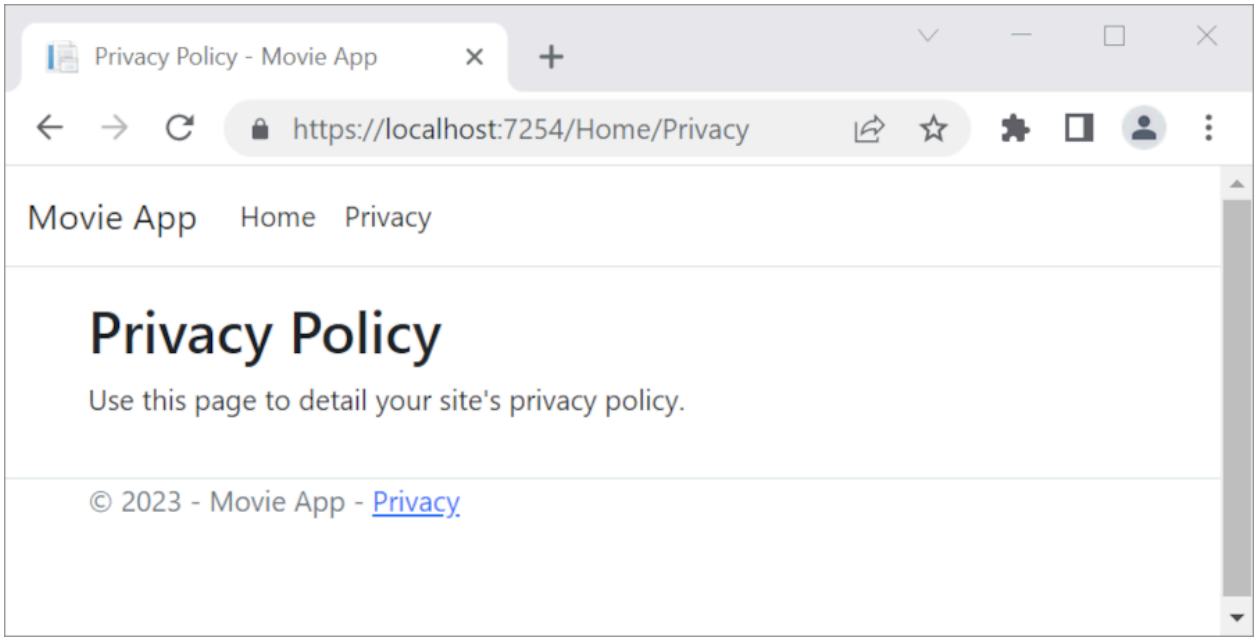
The preceding markup made the following changes:

- Three occurrences of `MvcMovie` to `Movie App`.
- The anchor element `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

In the preceding markup, the `asp-area=""` anchor Tag Helper attribute and attribute value was omitted because this app isn't using Areas.

Note: The `Movies` controller hasn't been implemented. At this point, the `Movie App` link isn't functional.

Save the changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - MvcMovie**



Select the **Home** link.

Notice that the title and anchor text display **Movie App**. The changes were made once in the layout template and all pages on the site reflect the new link text and new title.

Examine the `Views/_ViewStart.cshtml` file:

CSHTML

```
@{
    Layout = "_Layout";
}
```

The `Views/_ViewStart.cshtml` file brings in the `Views/Shared/_Layout.cshtml` file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Open the `Views/HelloWorld/Index.cshtml` view file.

Change the title and `<h2>` element as highlighted in the following:

CSHTML

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

The title and `<h2>` element are slightly different so it's clear which part of the code changes the display.

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

CSHTML

```
<title>@ViewData["Title"] - Movie App</title>
```

Save the change and navigate to `https://localhost:{PORT}/HelloWorld`.

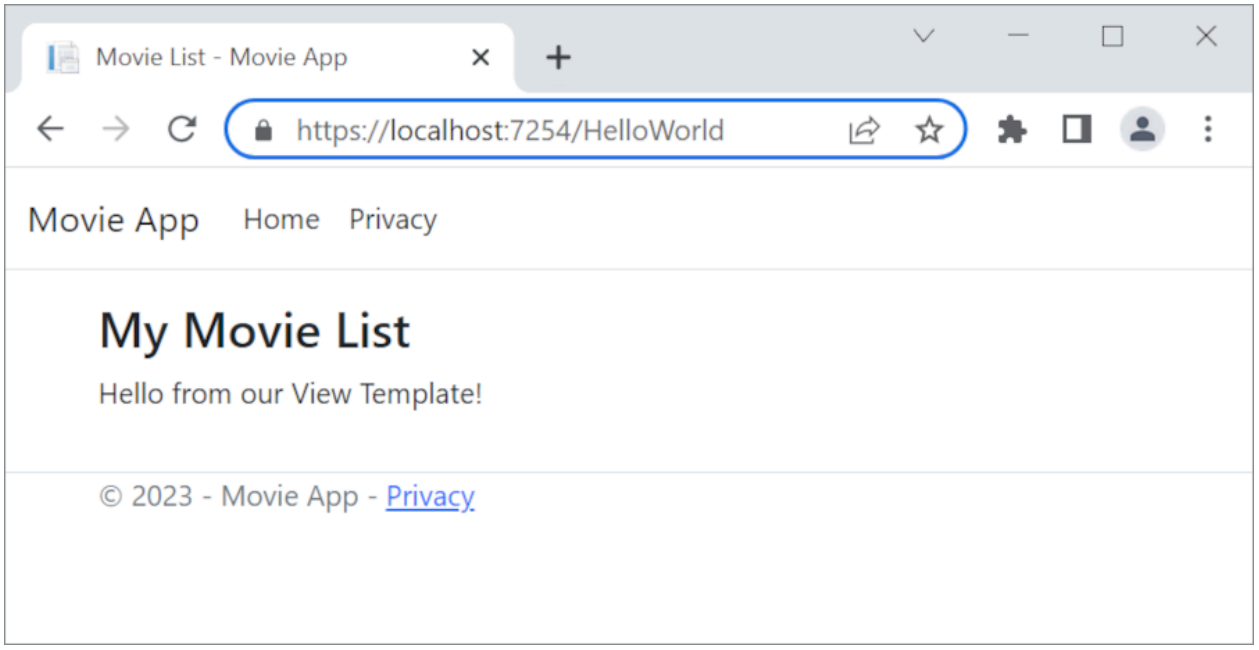
Notice that the following have changed:

- Browser title.

- Primary heading.
- Secondary headings.

If there are no changes in the browser, it could be cached content that is being viewed. Press Ctrl+F5 in the browser to force the response from the server to be loaded. The browser title is created with `ViewData["Title"]` we set in the `Index.cshtml` view template and the additional "- Movie App" added in the layout file.

The content in the `Index.cshtml` view template is merged with the `Views/Shared/_Layout.cshtml` view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app. To learn more, see Layout.



The small bit of "data", the "Hello from our View Template!" message, is hard-coded however. The MVC application has a "V" (view), a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response.

View templates should **not**:

- Do business logic
- Interact with a database directly.

A view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code:

- Clean.
- Testable.
- Maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and an `ID` parameter and then outputs the values directly to the browser.

Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate data must be passed from the controller to the view to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary. The view template can then access the dynamic data.

In `HelloWorldController.cs`, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary.

The `ViewData` dictionary is a dynamic object, which means any type can be used. The `ViewData` object has no defined properties until something is added. The MVC model binding system automatically maps the named parameters `name` and `numTimes` from the query string to parameters in the method. The complete `HelloWorldController`:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
```

```
    public IActionResult Welcome(string name, int numTimes = 1)
    {
        ViewData["Message"] = "Hello " + name;
        ViewData["NumTimes"] = numTimes;
        return View();
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named `Views/HelloWorld/Welcome.cshtml`.

You'll create a loop in the `Welcome.cshtml` view template that displays "Hello" `NumTimes`. Replace the contents of `Views/HelloWorld/Welcome.cshtml` with the following:

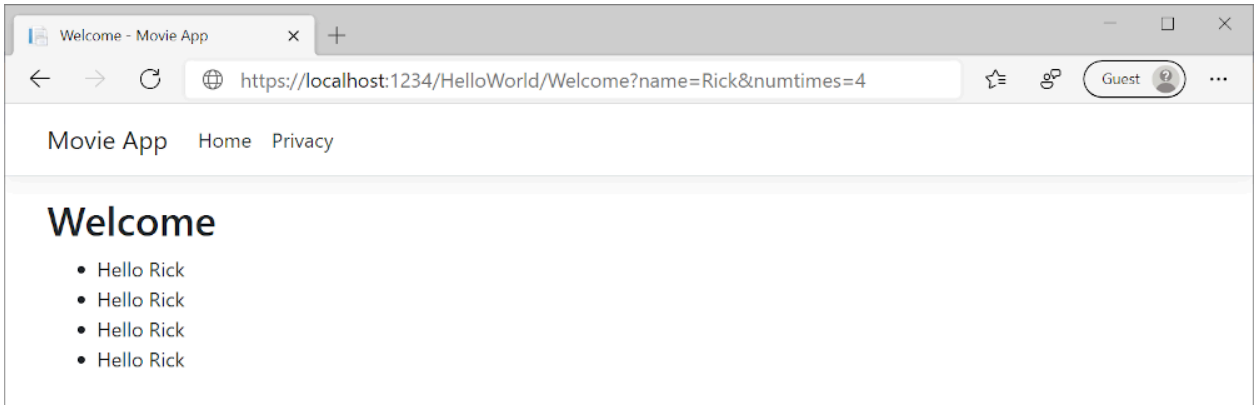
CSHTML

```
@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder. The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the preceding sample, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is preferred over the `ViewData` dictionary approach.

In the next tutorial, a database of movies is created.

Previous: Add a Controller | Next: Add a Model

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 4, add a model to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson and Jon P Smith.

In this tutorial, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.

These model classes are used with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes created are known as ***POCO*** classes, from **P**lain **O**ld **C**LR **O**bjects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.

In this tutorial, model classes are created first, and EF Core creates the database.

Add a data model class

Visual Studio

Right-click the *Models* folder > **Add** > **Class**. Name the file `Movie.cs`.

Update the `Models/Movie.cs` file with the following code:

C#

```
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    public decimal Price { get; set; }
}
```

The `Movie` class contains an `Id` field, which is required by the database for the primary key.

The DataType attribute on `ReleaseDate` specifies the type of the data (`Date`). With this attribute:

- The user isn't required to enter time information in the date field.
- Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

The question mark after `string` indicates that the property is nullable. For more information, see Nullable reference types.

Add NuGet packages

Visual Studio

Visual Studio automatically installs the required packages.

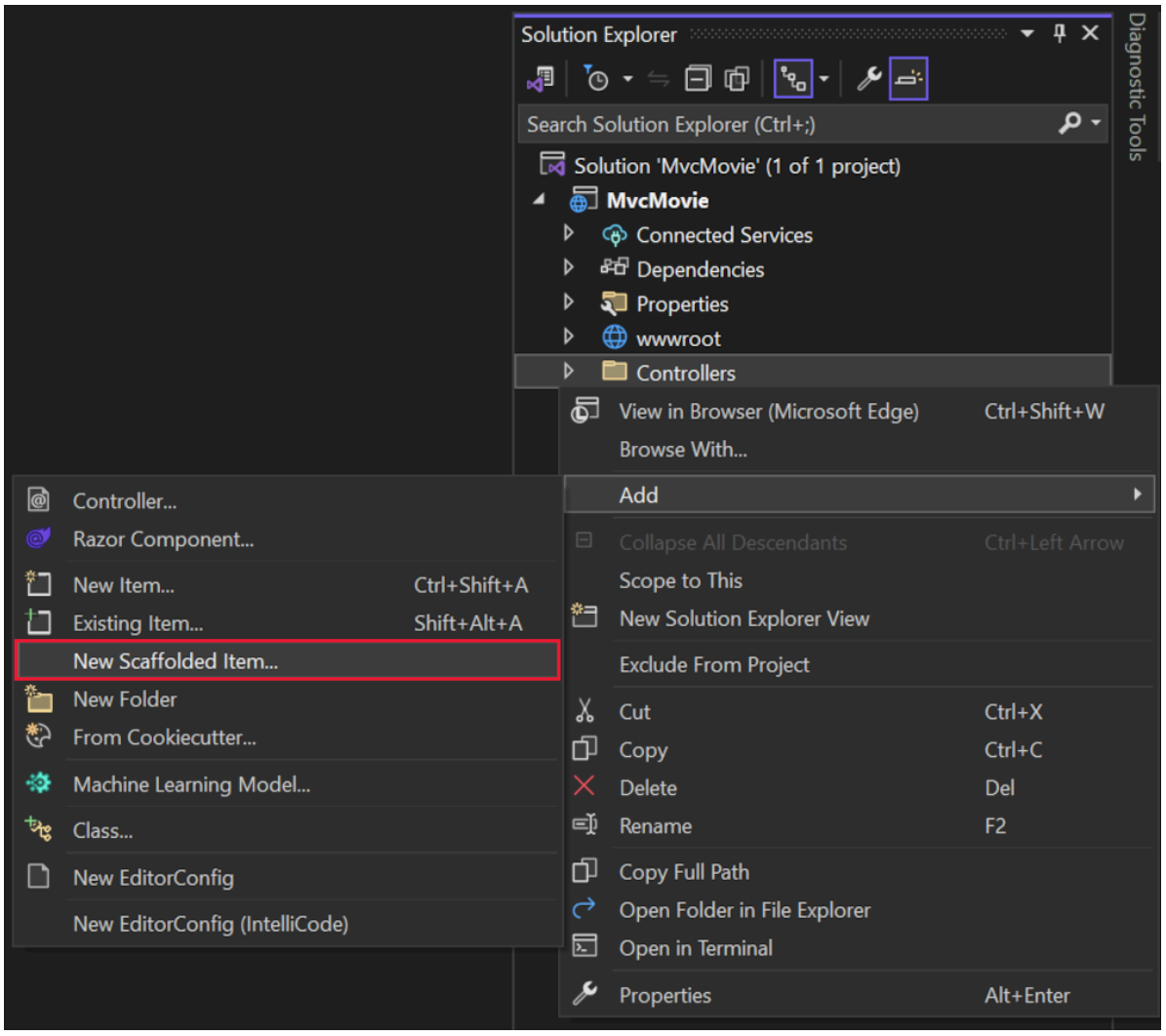
Build the project as a check for compiler errors.

Scaffold movie pages

Use the scaffolding tool to produce `Create`, `Read`, `Update`, and `Delete` (CRUD) pages for the movie model.

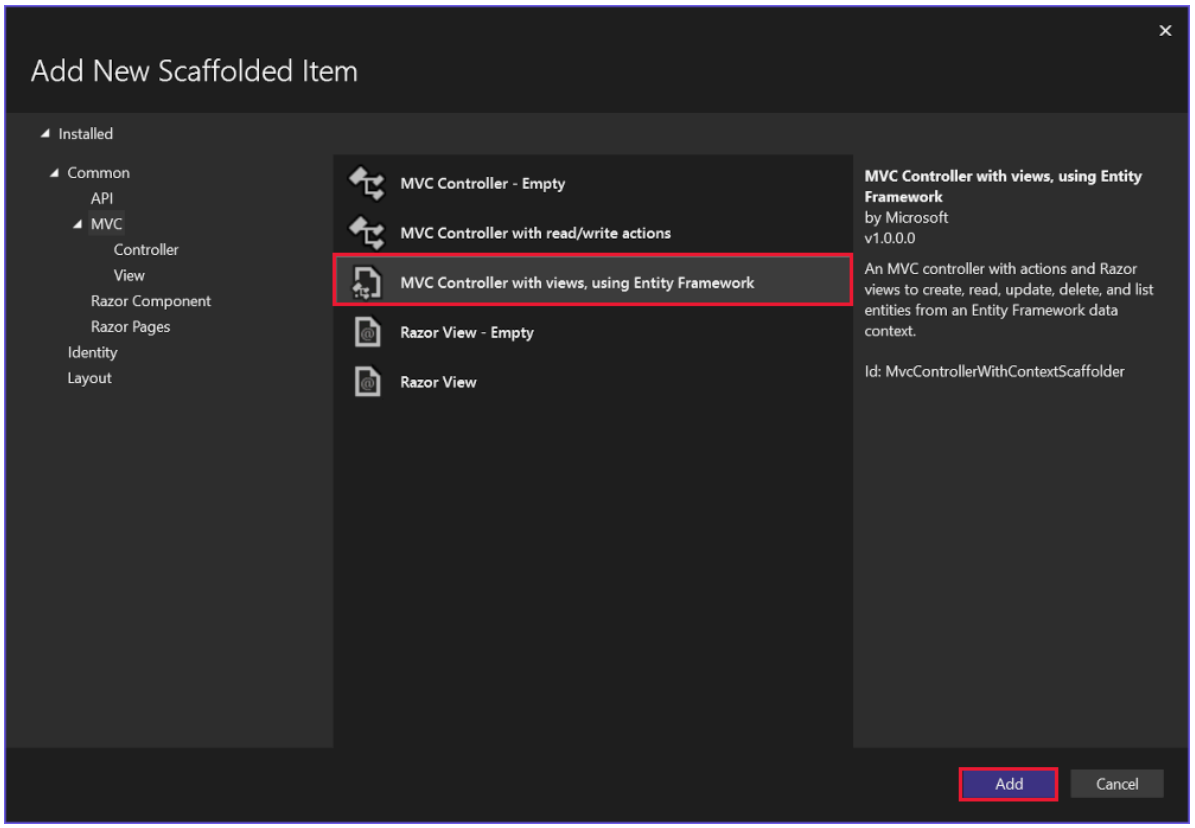
Visual Studio

In **Solution Explorer**, right-click the *Controllers* folder and select **Add** > **New Scaffolded Item**.



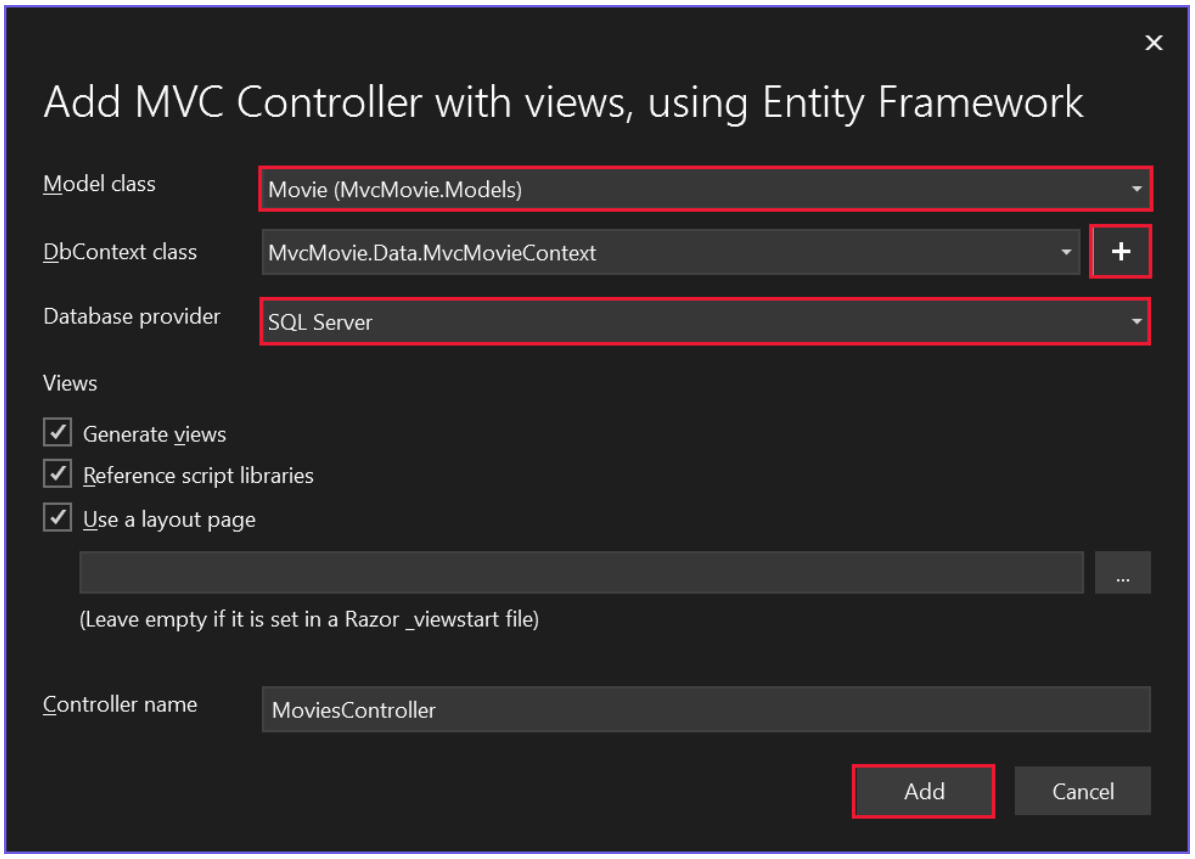
In the **Add New Scaffolded Item** dialog:

- In the left pane, select **Installed** > **Common** > **MVC**.
- Select **MVC Controller with views, using Entity Framework**.
- Select **Add**.



Complete the **Add MVC Controller with views, using Entity Framework** dialog:

- In the **Model class** drop down, select **Movie (MvcMovie.Models)**.
- In the **Data context class** row, select the + (plus) sign.
  - In the **Add Data Context** dialog, the class name *MvcMovie.Data.MvcMovieContext* is generated.
  - Select **Add**.
- In the **Database provider** drop down, select **SQL Server**.
- **Views** and **Controller name**: Keep the default.
- Select **Add**.



If you get an error message, select **Add** a second time to try it again.

Scaffolding adds the following packages:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

Scaffolding creates the following:

- A movies controller: `Controllers/MoviesController.cs`
- Razor view files for **Create**, **Delete**, **Details**, **Edit**, and **Index** pages: `Views/Movies/*.cshtml`
- A database context class: `Data/MvcMovieContext.cs`

Scaffolding updates the following:

- Inserts required package references in the `MvcMovie.csproj` project file.
- Registers the database context in the `Program.cs` file.
- Adds a database connection string to the `appsettings.json` file.

The automatic creation of these files and file updates is known as *scaffolding*.

The scaffolded pages can't be used yet because the database doesn't exist. Running the app and selecting the **Movie App** link results in a *Cannot open database* or *no*

such table: Movie error message.

Build the app to verify that there are no errors.

Initial migration

Use the EF Core Migrations feature to create the database. *Migrations* is a set of tools that create and update a database to match the data model.

Visual Studio

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** .

In the Package Manager Console (PMC), enter the following commands:

```
PowerShell

Add-Migration InitialCreate
Update-Database
```

- `Add-Migration InitialCreate`: Generates a `Migrations/{timestamp}_InitialCreate.cs` migration file. The `InitialCreate` argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the `MvcMovieContext` class.
- `Update-Database`: Updates the database to the latest migration, which the previous command created. This command runs the `Up` method in the `Migrations/{time-stamp}_InitialCreate.cs` file, which creates the database.

The `Update-Database` command generates the following warning:

> No store type was specified for the decimal property 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.

Ignore the preceding warning, it's fixed in a later tutorial.

For more information on the PMC tools for EF Core, see EF Core tools reference - PMC in Visual Studio.

Test the app

Visual Studio

Run the app and select the **Movie App** link.

If you get an exception similar to the following, you may have missed the `Update-Database` command in the migrations step:

```
Console

SqlException: Cannot open database "MvcMovieContext-1" requested by the
login. The login failed.
```

> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

Examine the generated database context class and registration

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. The database context is derived from Microsoft.EntityFrameworkCore.DbContext and specifies the entities to include in the data model.

Scaffolding creates the `Data/MvcMovieContext.cs` database context class:

```
C#
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<MvcMovie.Models.Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property that represents the movies in the database.

Dependency injection

ASP.NET Core is built with dependency injection (DI). Services, such as the database context, are registered with DI in `Program.cs`. These services are provided to components that require them via constructor parameters.

In the `Controllers/MoviesController.cs` file, the constructor uses Dependency Injection to inject the `MvcMovieContext` database context into the controller. The database context is used in each of the CRUD methods in the controller.

Scaffolding generated the following highlighted code in `Program.cs`:

Visual Studio

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context")));
```

The ASP.NET Core configuration system reads the "MvcMovieContext" database connection string.

Examine the generated database connection string

Scaffolding added a connection string to the `appsettings.json` file:

Visual Studio

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Data Source=MvcMovieContext-ea7a4069-f366-4742-
bd1c-3f753a804ce1.db"
  }
}
```

For local development, the ASP.NET Core configuration system reads the `ConnectionString` key from the `appsettings.json` file.

The `InitialCreate` class

Examine the `Migrations/{timestamp}_InitialCreate.cs` migration file:

C#

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace MvcMovie.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Movie",
```

```
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Title = table.Column<string>(type: "nvarchar(max)",
nullable: true),
                    ReleaseDate = table.Column<DateTime>(type: "datetime2",
nullable: false),
                    Genre = table.Column<string>(type: "nvarchar(max)",
nullable: true),
                    Price = table.Column<decimal>(type: "decimal(18,2)",
nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Movie", x => x.Id);
                });
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Movie");
        }
    }
}
```

In the preceding code:

- `InitialCreate.Up` creates the Movie table and configures `Id` as the primary key.
- `InitialCreate.Down` reverts the schema changes made by the `Up` migration.

Dependency injection in the controller

Open the `Controllers/MoviesController.cs` file and examine the constructor:

C#

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the

CRUD methods in the controller.

Test the **Create** page. Enter and submit data.

Test the **Edit**, **Details**, and **Delete** pages.

Strongly typed models and the `@model` directive

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism passed a strongly typed model in the `MoviesController` class and views.

Examine the generated `Details` method in the `Controllers/MoviesController.cs` file:

C#

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example, `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller, the first URL segment.
- The action to `details`, the second URL segment.
- The `id` to 1, the last URL segment.

The `id` can be passed in with a query string, as in the following example:

`https://localhost:5001/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in cases when the `id` value isn't provided.

A lambda expression is passed in to the FirstOrDefaultAsync method to select movie entities that match the route data or query string value.

C#

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

C#

```
return View(movie);
```

Examine the contents of the `Views/Movies/Details.cshtml` file:

CSHTML

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
```

```
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

The `@model` statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following `@model` statement was included:

CSHTML

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows access to the movie that the controller passed to the view. The `Model` object is strongly typed. For example, in the `Details.cshtml` view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the `Index.cshtml` view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

C#

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

The code returns problem details if the `Movie` property of the data context is null.

When the movies controller was created, scaffolding included the following `@model` statement at the top of the `Index.cshtml` file:

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows access to the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the `Index.cshtml` view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
```

```
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Because the `Model` object is strongly typed as an `IEnumerable<Movie>` object, each item in the loop is typed as `Movie`. Among other benefits, the compiler validates the types used in the code.

Additional resources

- Entity Framework Core for Beginners
- Tag Helpers
- Globalization and localization

Previous: Adding a View | Next: Working with SQL

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 5, work with a database in an ASP.NET Core MVC app

Article • 09/10/2024

By Rick Anderson and Jon P Smith.

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `Program.cs` file:

Visual Studio

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context")));
```

The ASP.NET Core Configuration system reads the `ConnectionString` key. For local development, it gets the connection string from the `appsettings.json` file:

JSON

```
"ConnectionStrings": {
  "MvcMovieContext": "Data Source=MvcMovieContext-ea7a4069-f366-4742-
bd1c-3f753a804ce1.db"
}
```

> **Note**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Visual Studio

SQL Server Express LocalDB

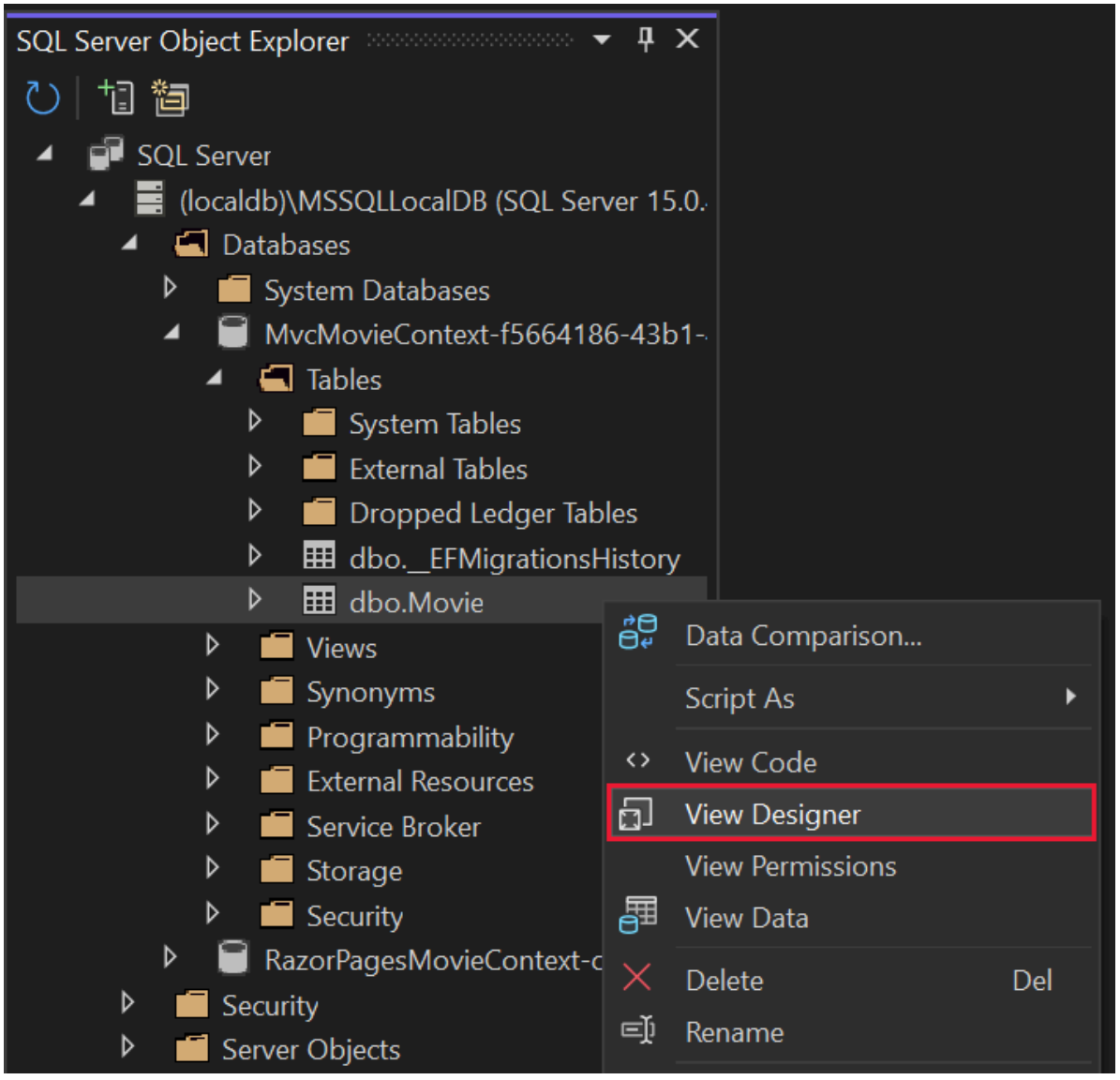
LocalDB:

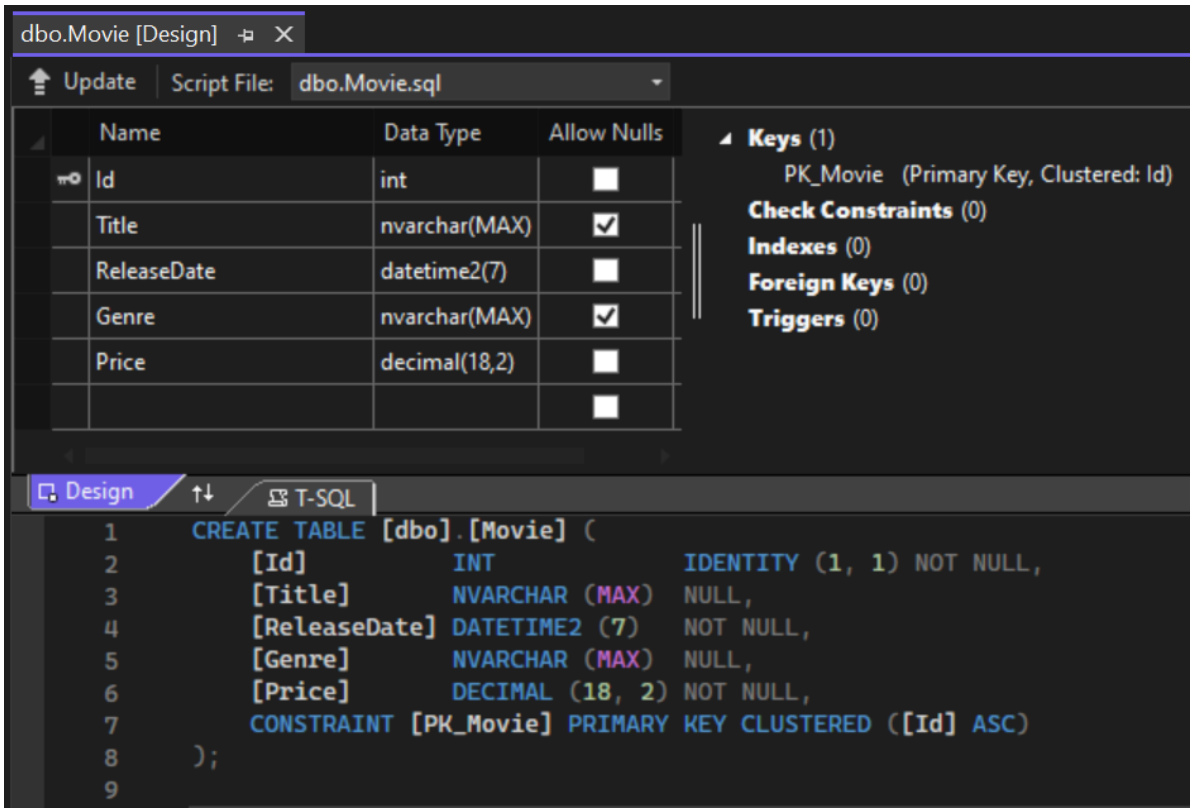
- Is a lightweight version of the SQL Server Express Database Engine, installed by default with Visual Studio.
- Starts on demand by using a connection string.
- Is targeted for program development. It runs in user mode, so there's no complex configuration.
- By default creates *.mdf* files in the *C:/Users/{user}* directory.

Examine the database

From the **View** menu, open **SQL Server Object Explorer** (SSOX).

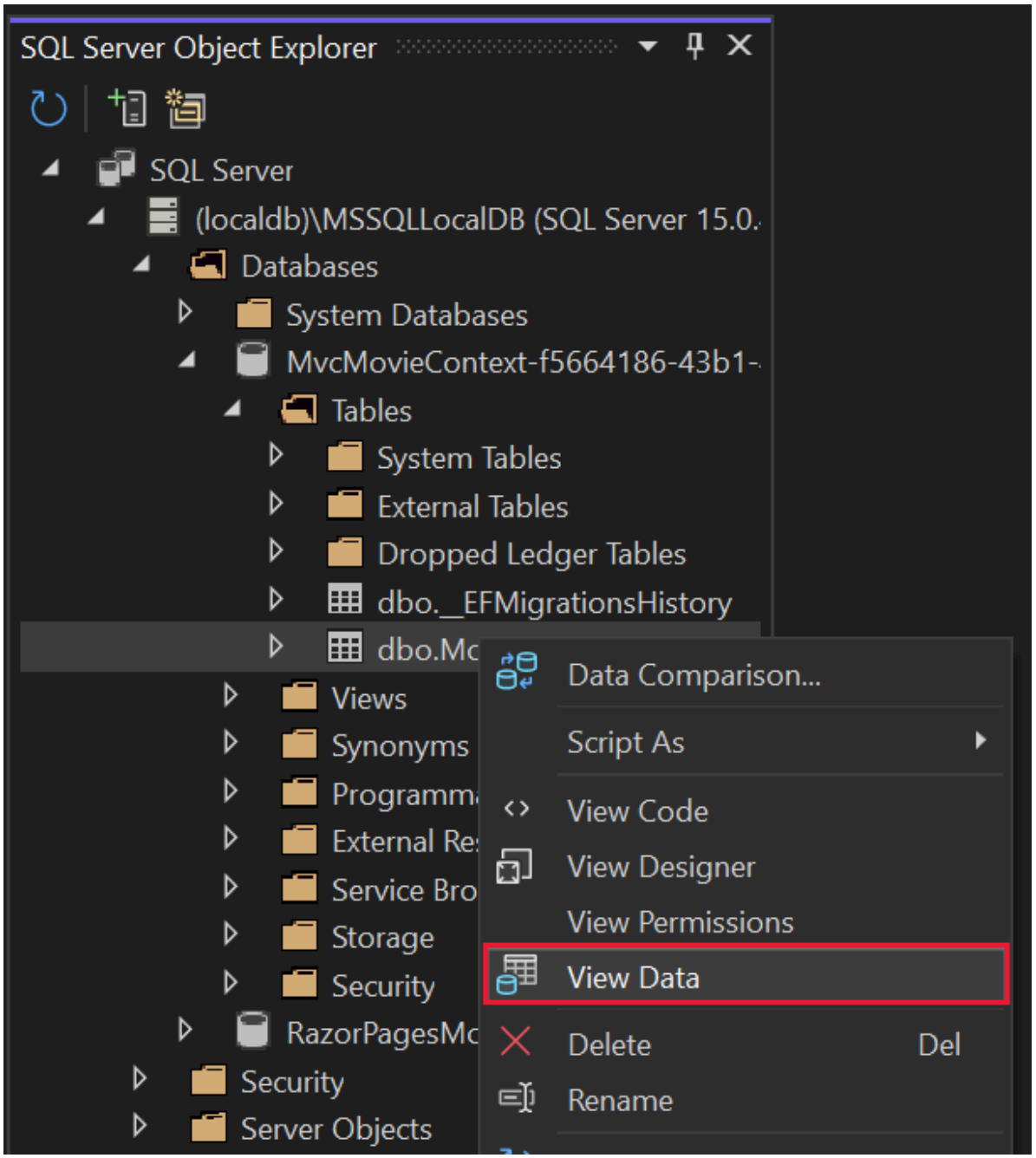
Right-click on the `Movie` table (`dbo.Movie`) > **View Designer**

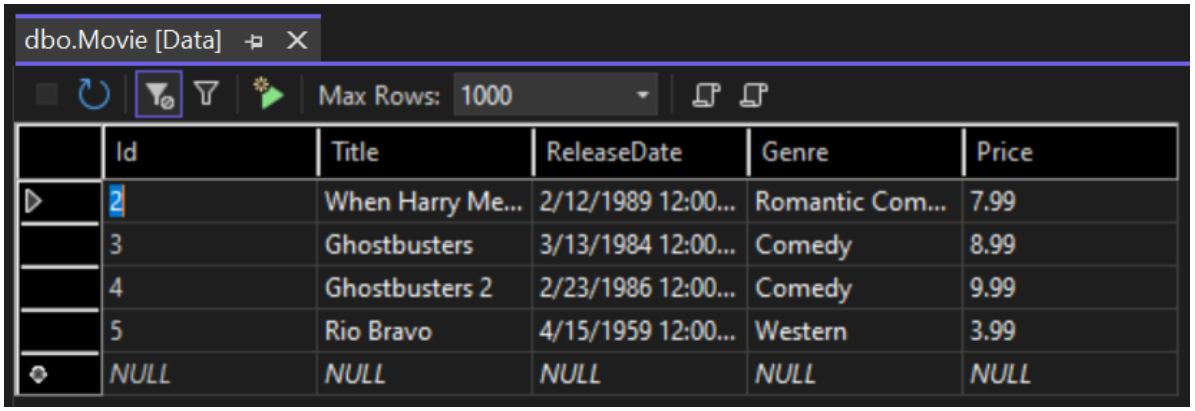




Note the key icon next to `ID`. By default, EF makes a property named `ID` the primary key.

Right-click on the `Movie` table > **View Data**





Id	Title	ReleaseDate	Genre	Price
2	When Harry Me...	2/12/1989 12:00...	Romantic Com...	7.99
3	Ghostbusters	3/13/1984 12:00...	Comedy	8.99
4	Ghostbusters 2	2/23/1986 12:00...	Comedy	9.99
5	Rio Bravo	4/15/1959 12:00...	Western	3.99
NULL	NULL	NULL	NULL	NULL

Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using System;
using System.Linq;

namespace MvcMovie.Models;

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new MvcMovieContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<MvcMovieContext>>()))
        {
            // Look for any movies.
            if (context.Movie.Any())
            {
                return;   // DB has been seeded
            }
            context.Movie.AddRange(
                new Movie
                {
                    Title = "When Harry Met Sally",
                    ReleaseDate = DateTime.Parse("1989-2-12"),
                    Genre = "Romantic Comedy",
                    Price = 7.99M
                },
                new Movie
                {
                    Title = "Ghostbusters ",
                    ReleaseDate = DateTime.Parse("1984-3-13"),
                    Genre = "Comedy",
                    Price = 8.99M
                },
                new Movie
                {
                    Title = "Ghostbusters 2",
                    ReleaseDate = DateTime.Parse("1986-2-23"),
                    Genre = "Comedy",
                    Price = 9.99M
                },
                new Movie
                {
                    Title = "Rio Bravo",
                    ReleaseDate = DateTime.Parse("1959-4-15"),
                    Genre = "Western",
                    Price = 3.99M
```

```
                }
            );
            context.SaveChanges();
        }
    }
}
```

If there are any movies in the database, the seed initializer returns and no movies are added.

C#

```
if (context.Movie.Any())
{
    return;  // DB has been seeded.
}
```

Add the seed initializer

Visual Studio

Replace the contents of `Program.cs` with the following code. The new code is highlighted.

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using MvcMovie.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context")));

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

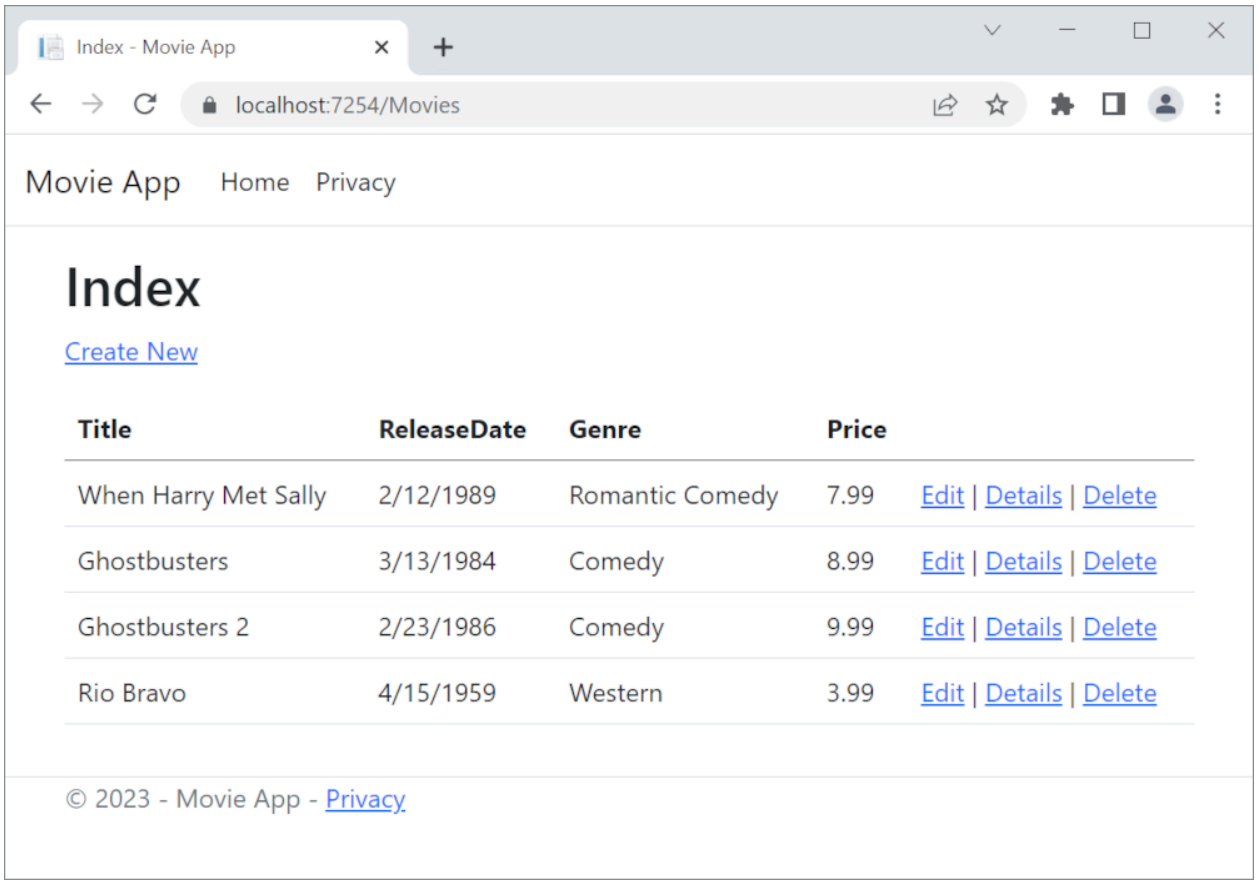
```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}
```

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Delete all the records in the database. You can do this with the delete links in the browser or from SSOX.

Test the app. Force the app to initialize, calling the code in the `Program.cs` file, so the seed method runs. To force initialization, close the command prompt window that Visual Studio opened, and restart by pressing Ctrl+F5.

The app shows the seeded data.



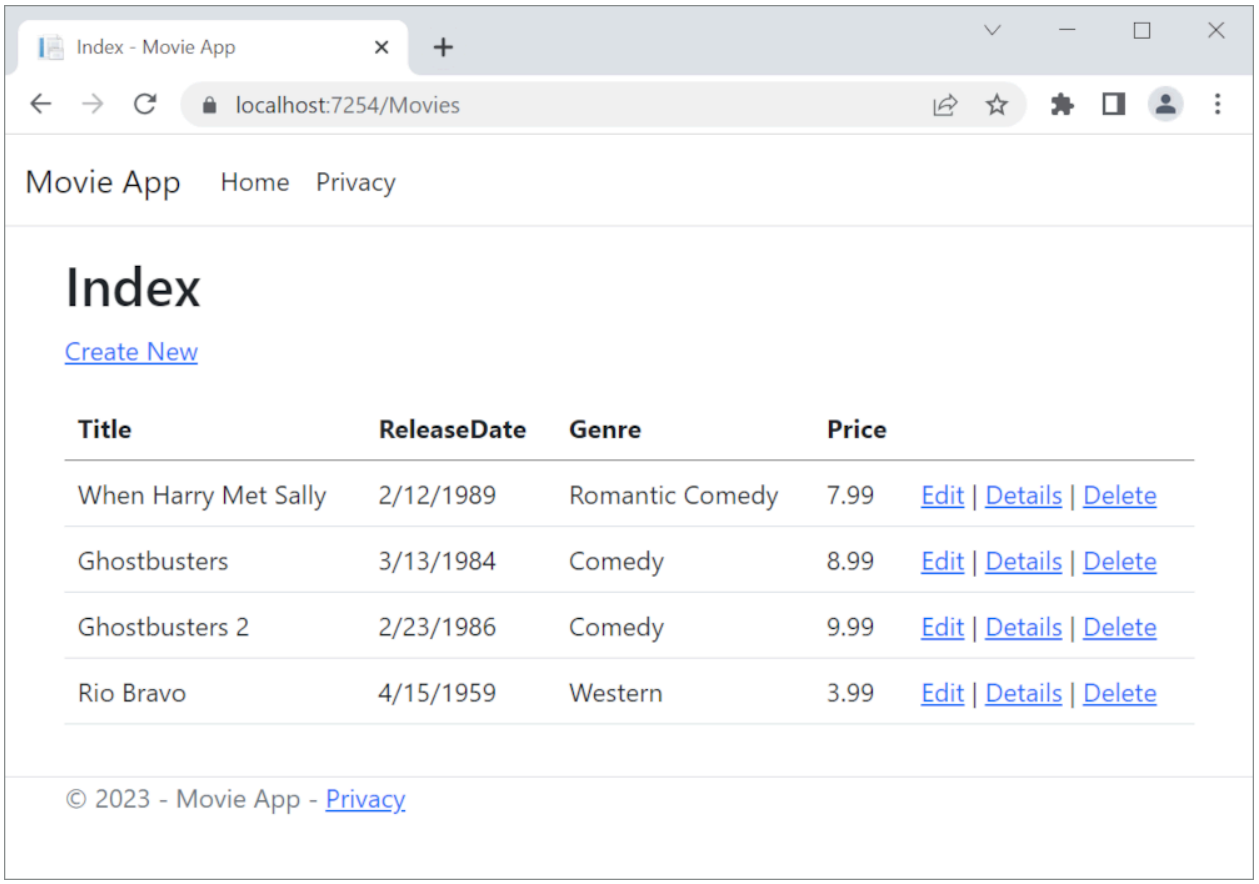
Previous: Adding a model
Next: Adding controller methods and views

Part 6, controller methods and views in ASP.NET Core

Article • 07/26/2024

By Rick Anderson

We have a good start to the movie app, but the presentation isn't ideal, for example, **ReleaseDate** should be two words.



Open the `Models/Movie.cs` file and add the highlighted lines shown below:

C#

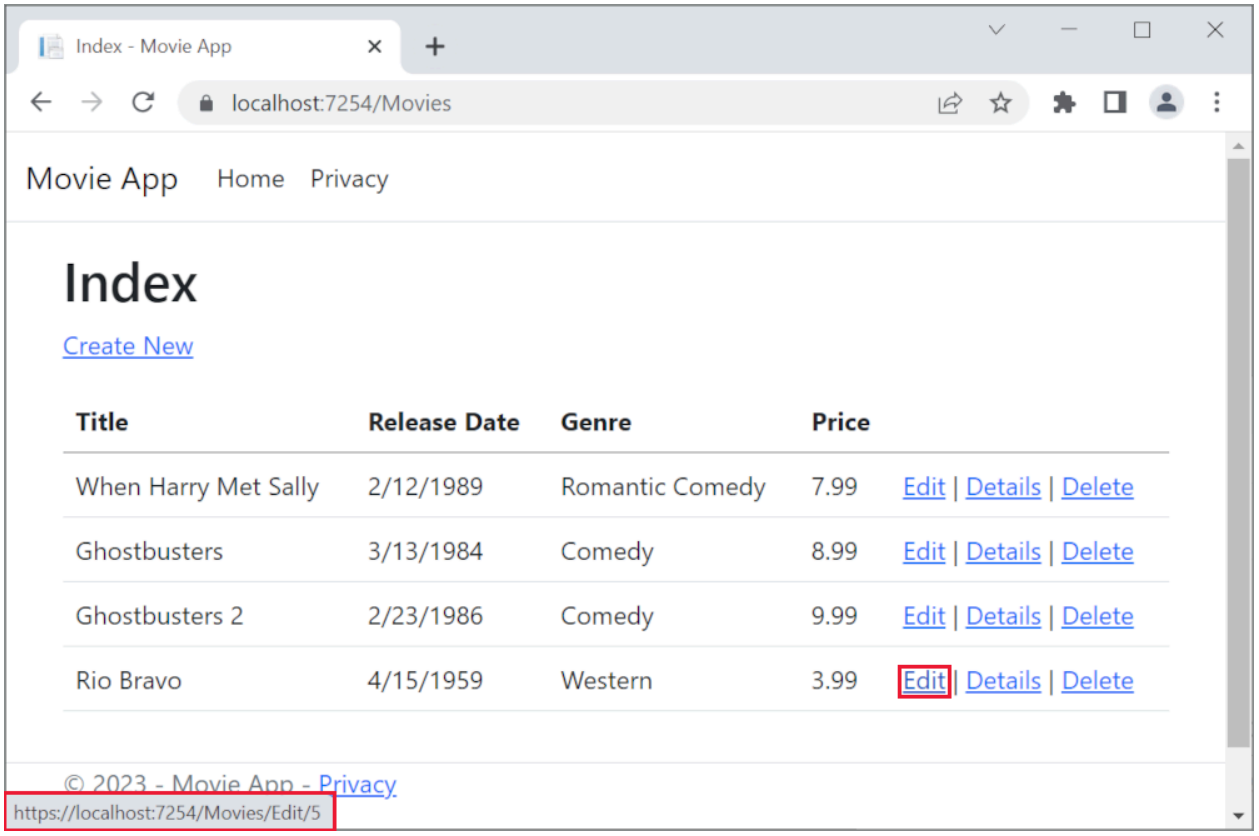
```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
```

```
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
```

`DataAnnotations` are explained in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Core MVC Anchor Tag Helper in the `Views/Movies/Index.cshtml` file.

CSHTML

```
            <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
        </td>
    </tr>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

HTML

```
 <td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for routing set in the `Program.cs` file:

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

ASP.NET Core translates `https://localhost:5001/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. For more information, see Additional resources.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the `Edit.cshtml` Razor file.

C#

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
```

```
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

C#

```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
```

```
    return View(movie);
}
```

The `[Bind]` attribute is one way to protect against over-posting. You should only include properties in the `[Bind]` attribute that you want to change. For more information, see Protect your controller from over-posting. ViewModels provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

C#

```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to prevent forgery of a request and is paired up with an antiforgery token generated in the edit view file (`Views/Movies/Edit.cshtml`). The edit view file generates the antiforgery token with the Form Tag Helper.

CSHTML

```
<form asp-action="Edit">
```

The Form Tag Helper generates a hidden antiforgery token that must match the `[ValidateAntiForgeryToken]` generated antiforgery token in the `Edit` method of the Movies controller. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `FindAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

C#

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

CSHTML

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file. `@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The Label Tag Helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML `<input>` element. The Validation Tag Helper displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

HTML

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field
is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre"
name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Genre" data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true"
data-val-number="The field Price must be a number." data-val-required="The
Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Price" data-valmsg-replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
```

```
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmq
UyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The `<input>` elements are in an `HTML <form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the `Save` button is clicked. The last line before the closing `</form>` element shows the hidden XSRF token generated by the Form Tag Helper.

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

C#

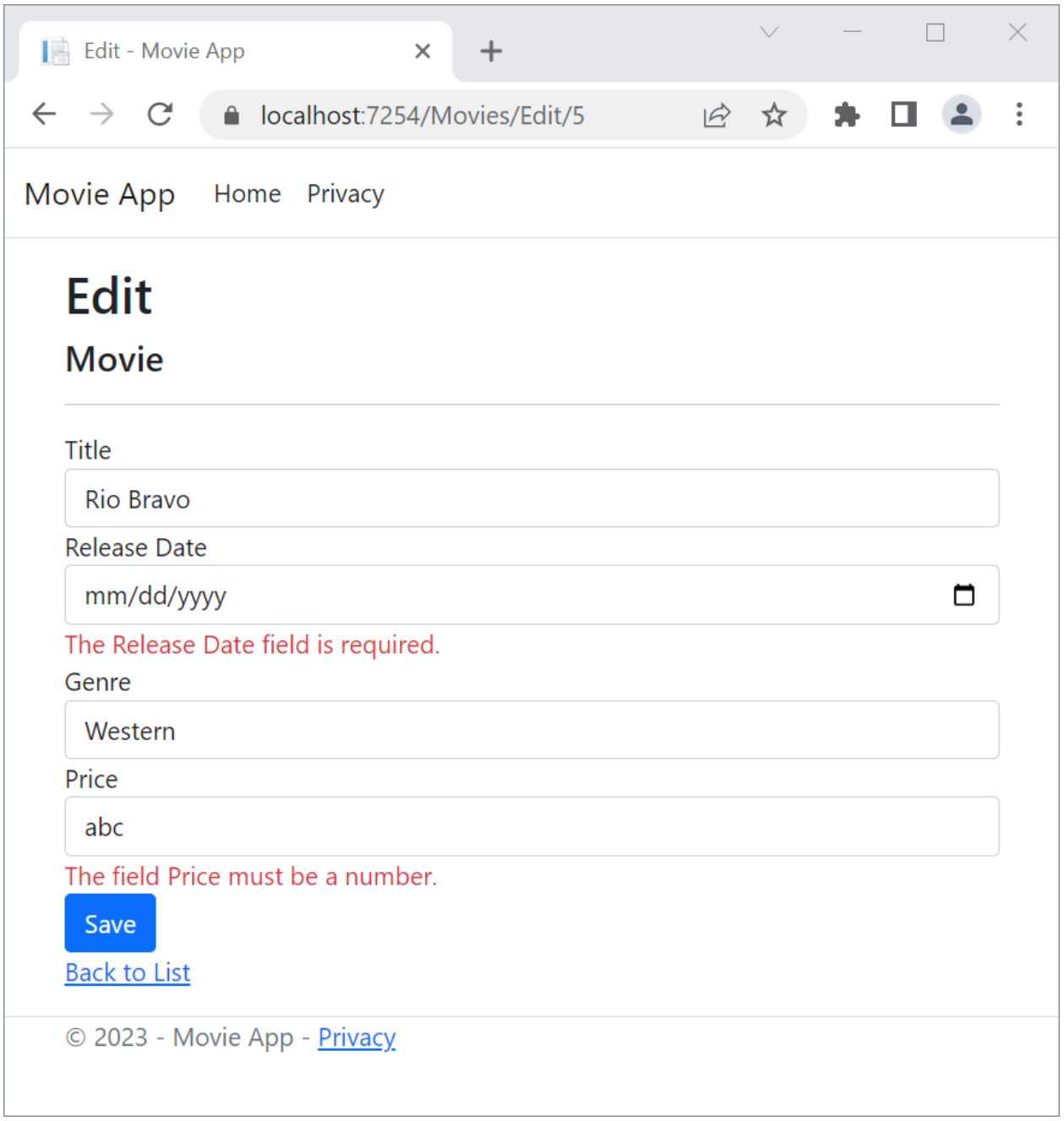
```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
```

```
        }
        return View(movie);
    }
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the antiforgery token generator in the Form Tag Helper

The model binding system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` property verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine Model Validation in more detail. The Validation Tag Helper in the `Views/Movies/Edit.cshtml` view template takes care of displaying appropriate error messages.



All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests shouldn't change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- Globalization and localization

- Introduction to Tag Helpers
- Author Tag Helpers
- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core
- Protect your controller from over-posting
- ViewModels
- Form Tag Helper
- Input Tag Helper
- Label Tag Helper
- Select Tag Helper
- Validation Tag Helper

Previous Next

Part 7, add search to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section, you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method found inside `Controllers/MoviesController.cs` with the following code:

C#

```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

The following line in the `Index` action method creates a LINQ query to select the movies:

C#

```
var movies = from m in _context.Movie
             select m;
```

The query is *only defined* at this point, it has **not** been run against the database.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

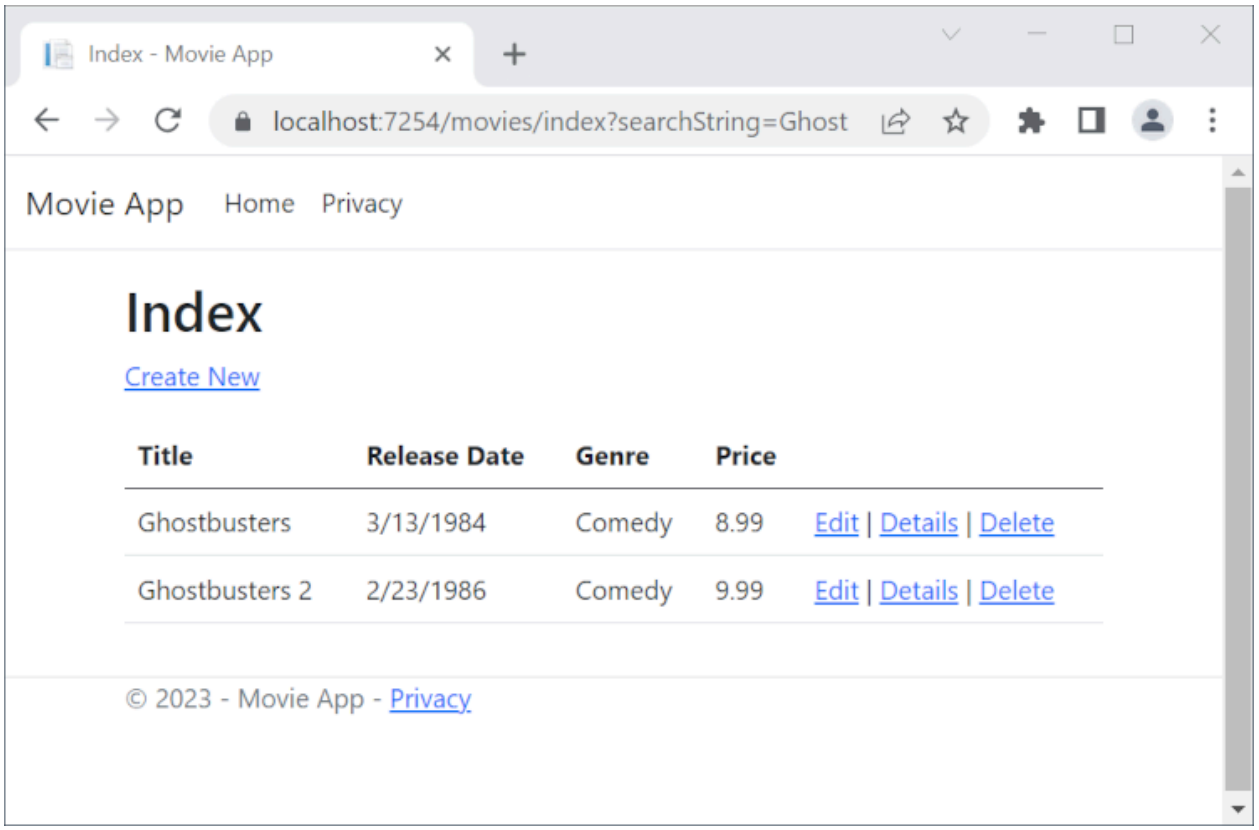
C#

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
}
```

The `s => s.Title!.ToUpper().Contains(searchString.ToUpper())` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see Query Execution.

> **Note**
>
> The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. SQLite with the default collation is a mixture of case sensitive and case ***IN***sensitive, depending on the query. For information on making case insensitive SQLite queries, see the following:
>
> - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
> - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
> - Collations and Case Sensitivity

Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in `Program.cs`.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Change the parameter to `id` and change all occurrences of `searchString` to `id`.

The previous `Index` method:

C#

```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
```

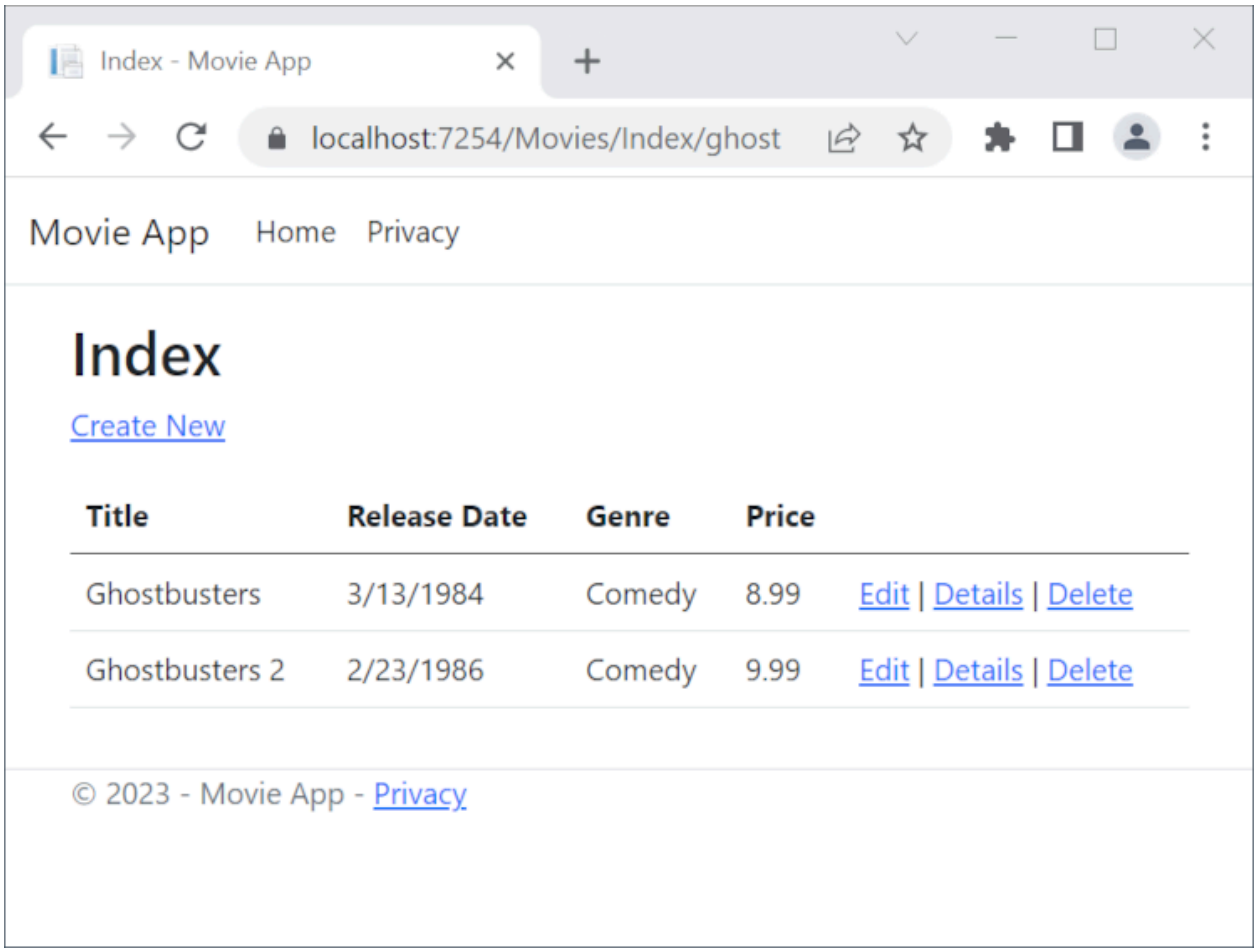
```
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

The updated `Index` method with `id` parameter:

C#

```
public async Task<IActionResult> Index(string id)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(id.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

C#

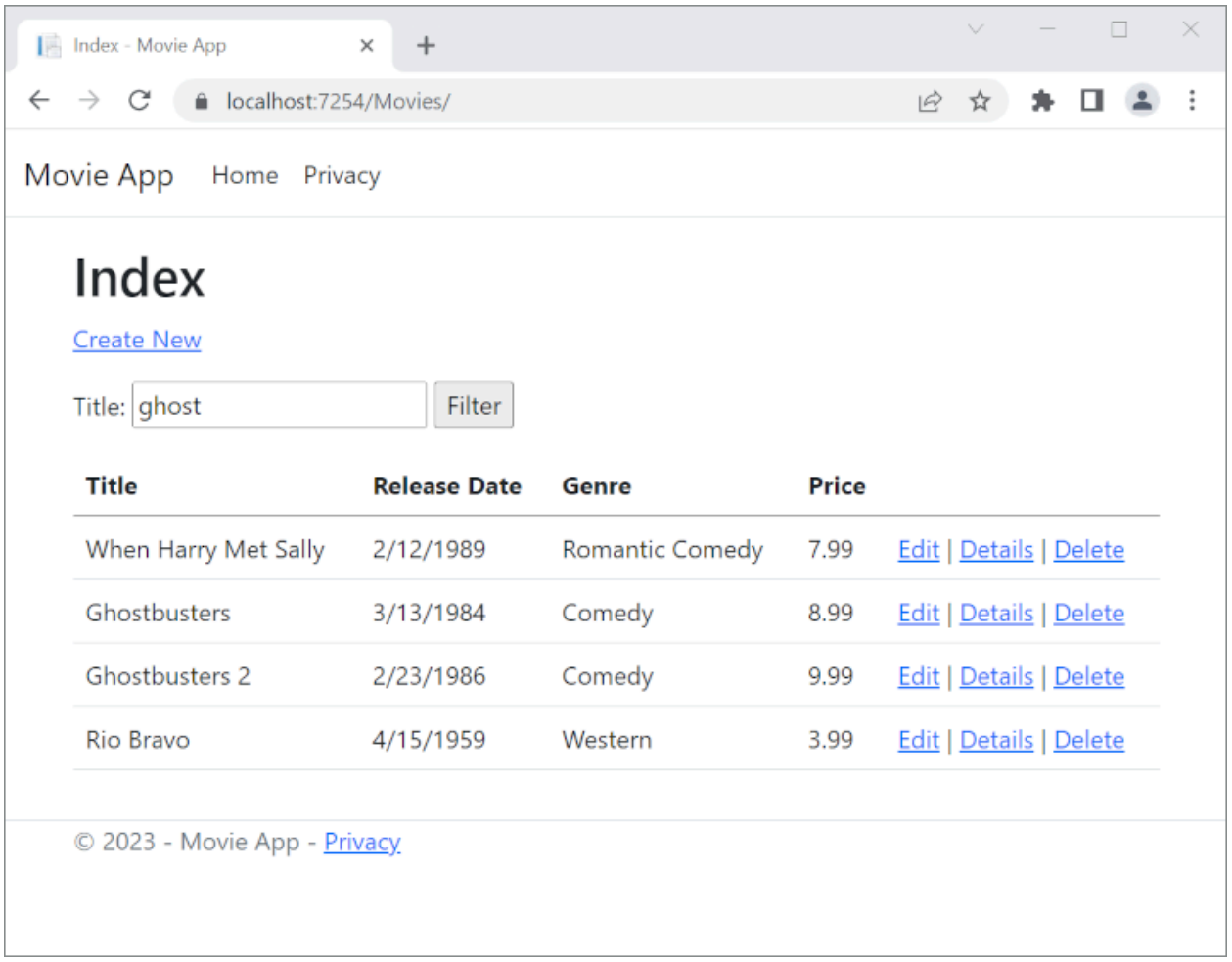
```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

Open the `Views/Movies/Index.cshtml` file, and add the `<form>` markup highlighted below:

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

The HTML `<form>` tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.



There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

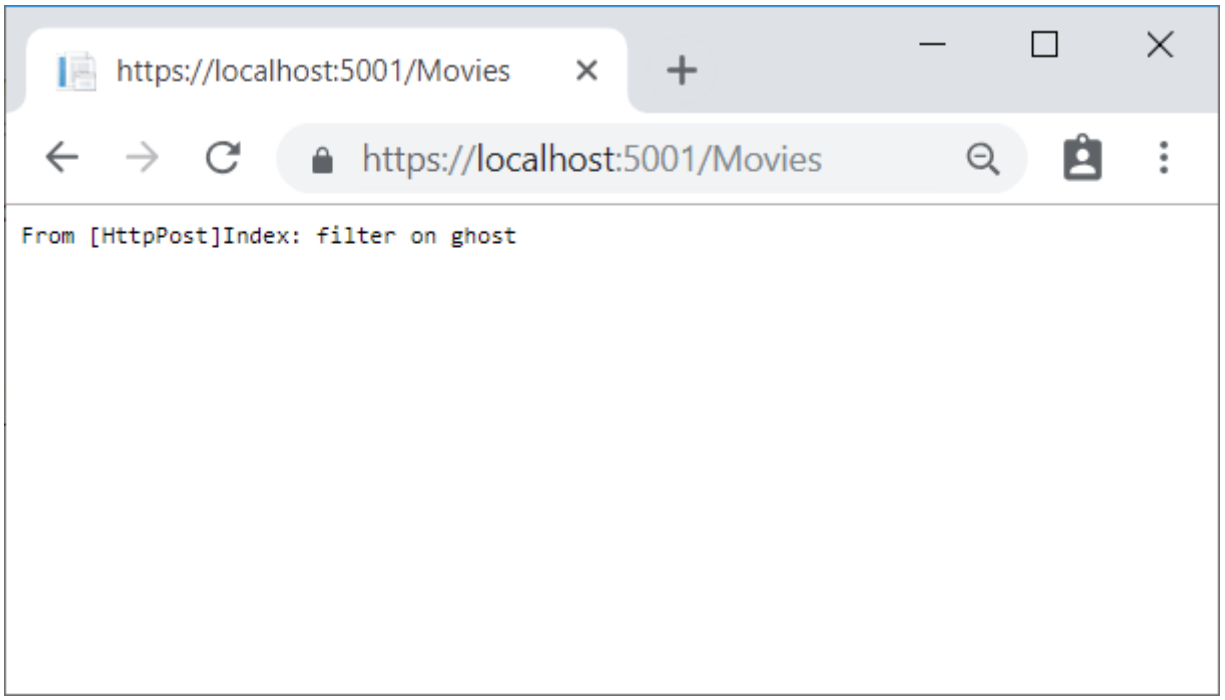
You could add the following `[HttpPost] Index` method.

C#

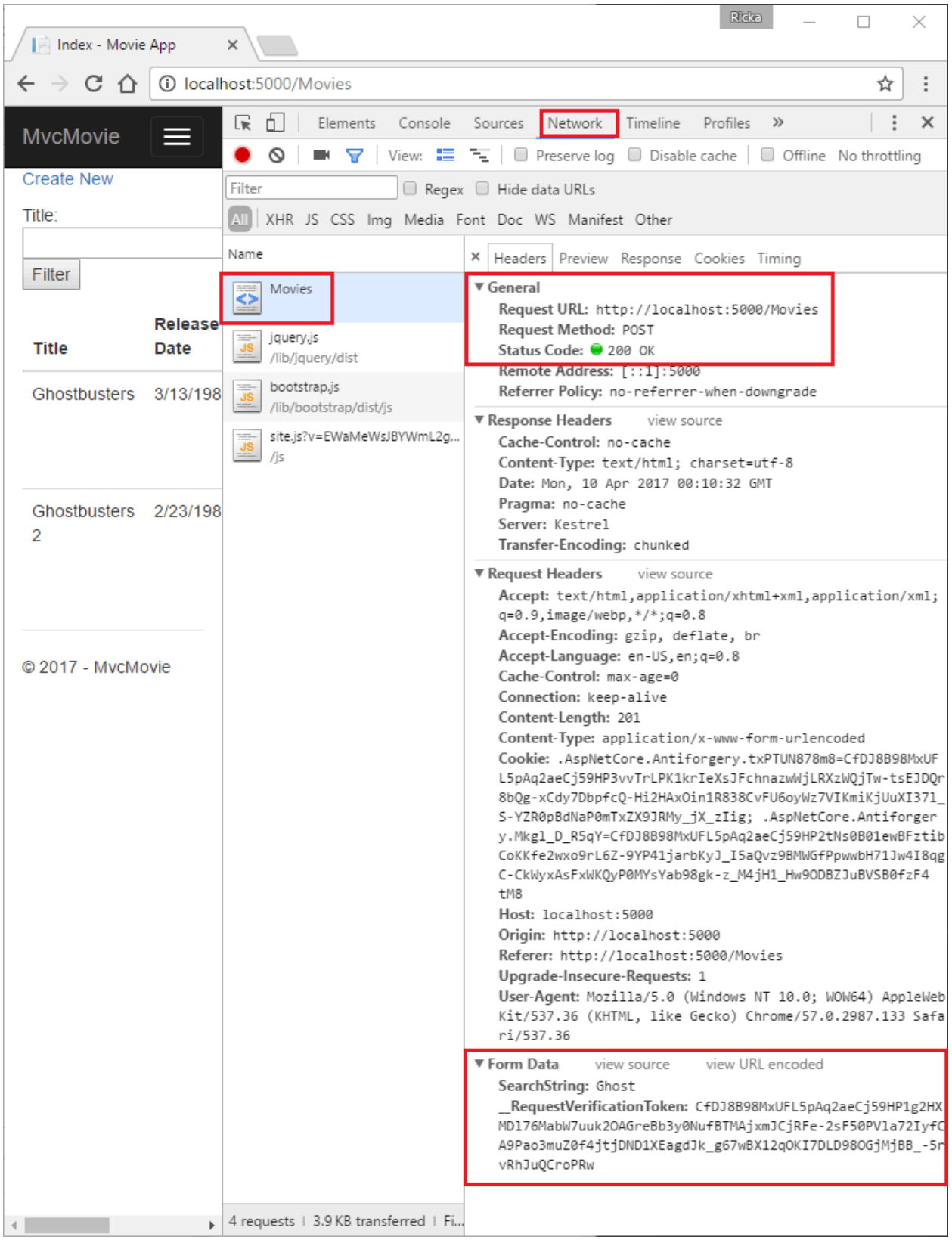
```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:{PORT}/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool. The image below shows the Chrome browser Developer tools:



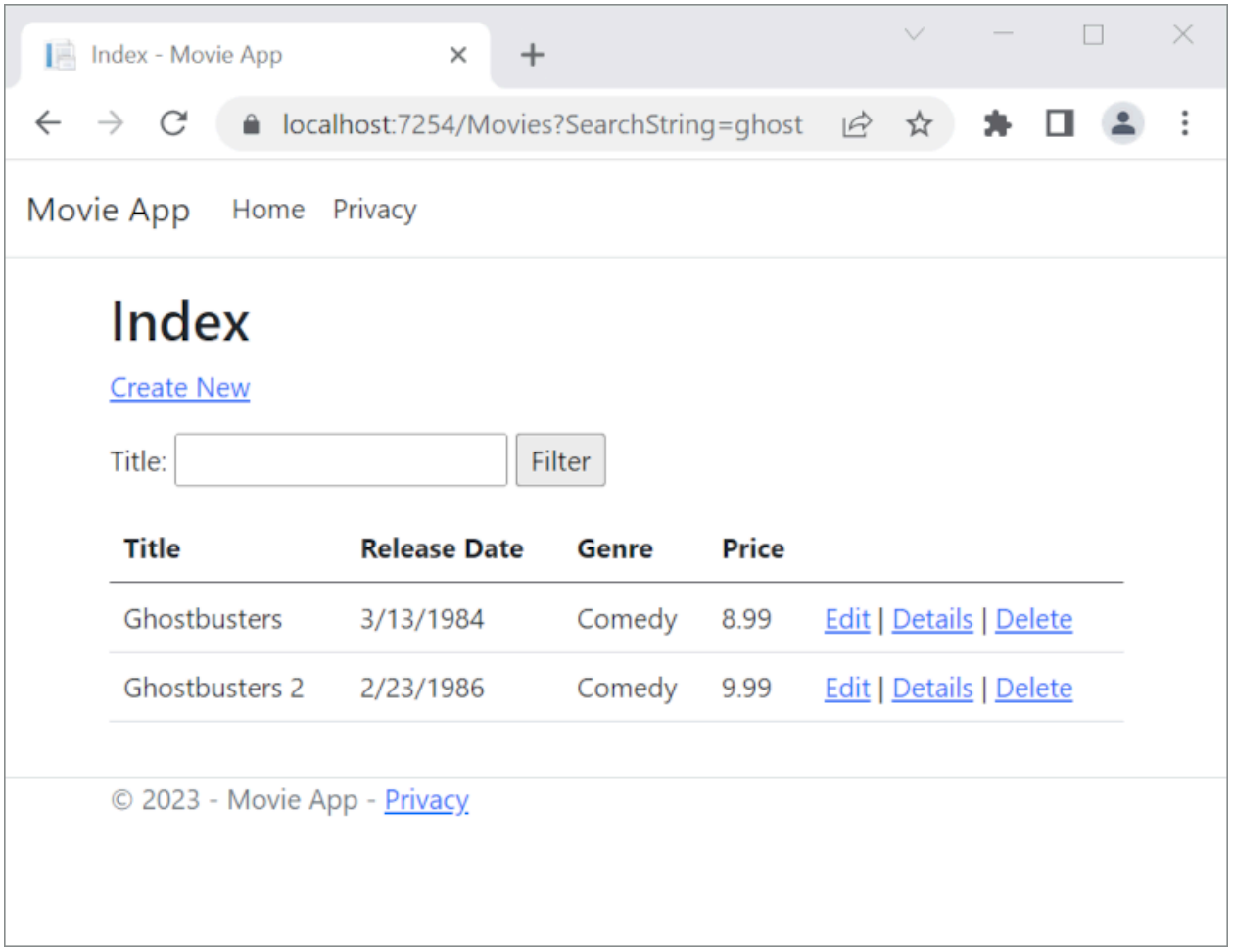
You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF antiforgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be `HTTP GET` found in the `Views/Movies/Index.cshtml` file.

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



The following markup shows the change to the `form` tag:

CSHTML

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Add Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models;

public class MovieGenreViewModel
{
    public List<Movie>? Movies { get; set; }
    public SelectList? Genres { get; set; }
    public string? MovieGenre { get; set; }
    public string? SearchString { get; set; }
}
```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This allows the user to select a genre from the list.
- `MovieGenre`, which contains the selected genre.
- `SearchString`, which contains the text users enter in the search text box.

Replace the `Index` method in `MoviesController.cs` with the following code:

C#

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string
searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    // Use LINQ to get list of genres.
```

```
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;
    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

The following code is a `LINQ` query that retrieves all the genres from the database.

C#

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

When the user searches for the item, the search value is retained in the search box.

Add search by genre to the Index view

Update `Index.cshtml` found in *Views/Movies/* as follows:

CSHTML

```
@model MvcMovie.Models.MovieGenreViewModel

@{
```

```
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies!)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
```

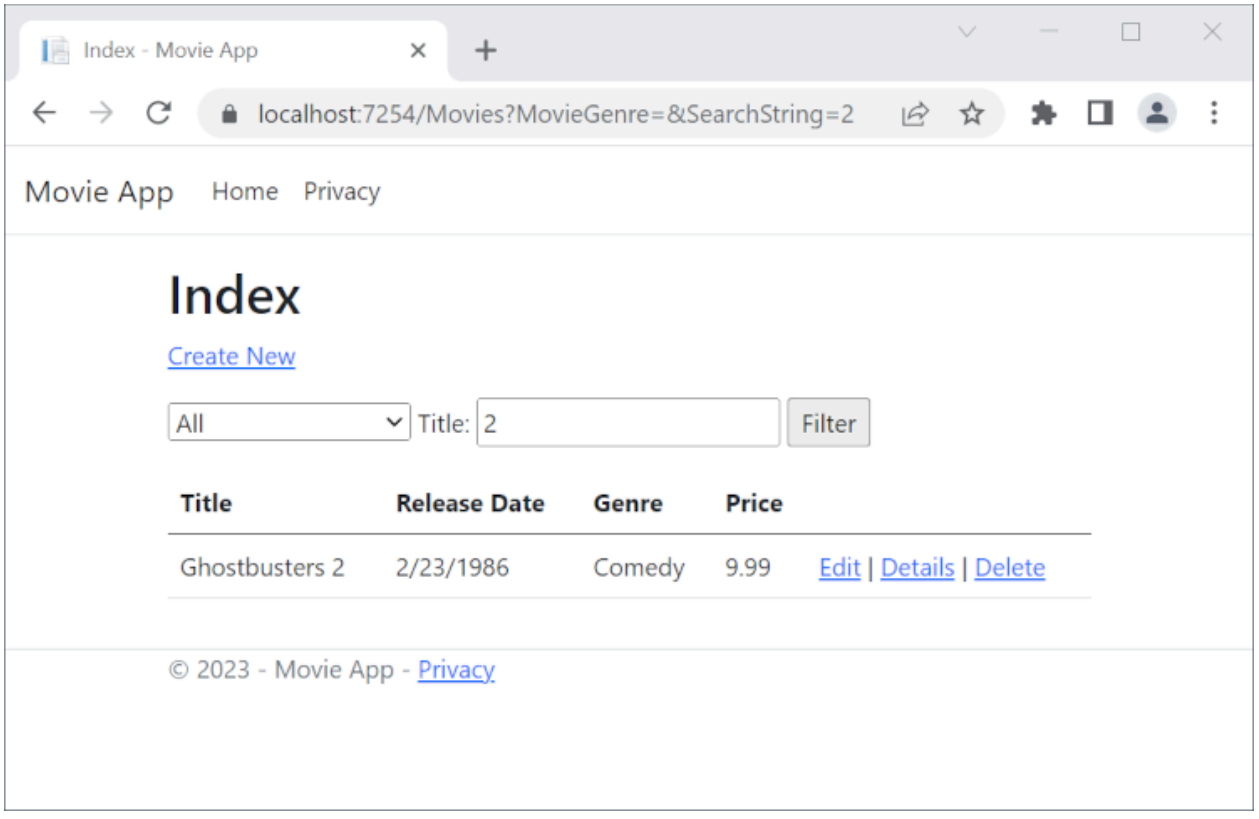
```
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movies![0].Title)
```

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.Movies`, or `model.Movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated. The `!` after `model.Movies` is the null-forgiving operator, which is used to declare that `Movies` isn't null.

Test the app by searching by genre, by movie title, and by both:



Previous Next

Part 8, add a new field to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When EF Code First is used to automatically create a database, Code First:

- Adds a table to the database to track the schema of the database.
- Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Add a Rating Property to the Movie Model

Add a `Rating` property to `Models/Movie.cs`:

C#

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string? Rating {  get; set; }
}
```

Build the app

Visual Studio

Press `Ctrl`+`Shift`+`B`

Because you've added a new field to the `Movie` class, you need to update the property binding list so this new property will be included. In `MoviesController.cs`, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

C#

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

Update the view templates in order to display, create, and edit the new `Rating` property in the browser view.

Edit the `/Views/Movies/Index.cshtml` file and add a `Rating` field:

CSHTML

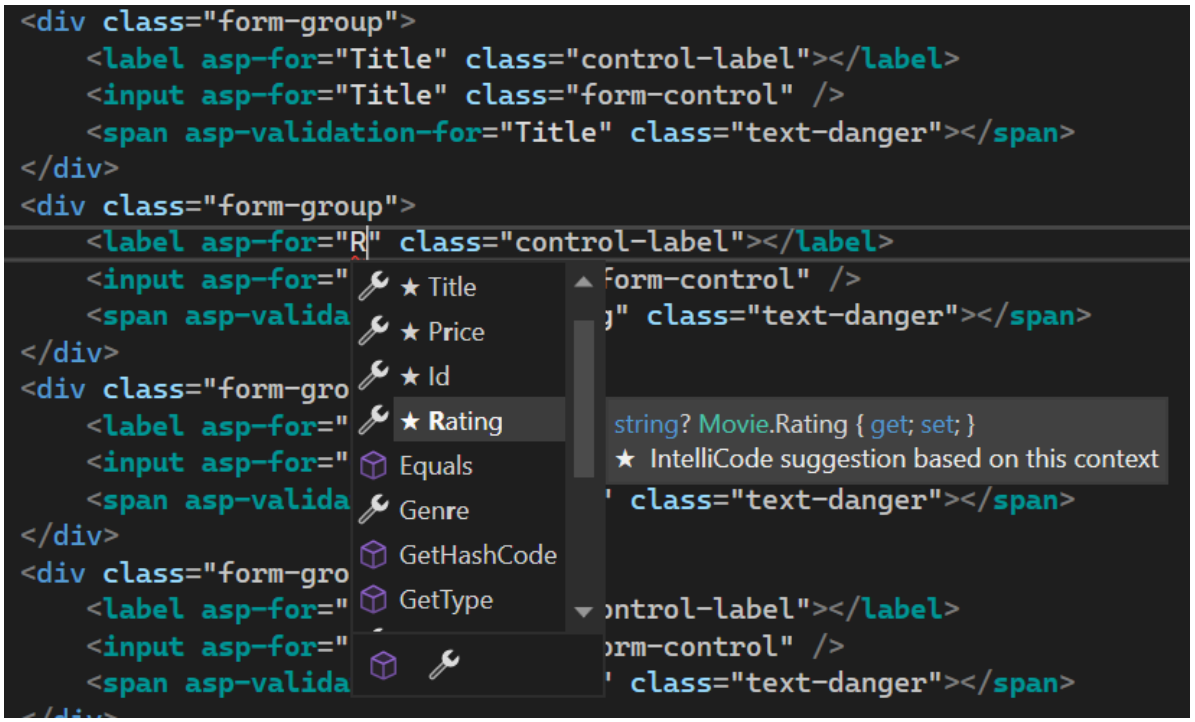
```
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies!)
        {
            <tr>
                <td>
```

```
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the `/Views/Movies/Create.cshtml` with a `Rating` field.

Visual Studio / Visual Studio for Mac

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers.



Update the remaining templates.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

C#

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

The app won't work until the DB is updated to include the new field. If it's run now, the following `SqlException` is thrown:

```
SqlException: Invalid column name 'Rating'.
```

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the

development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.

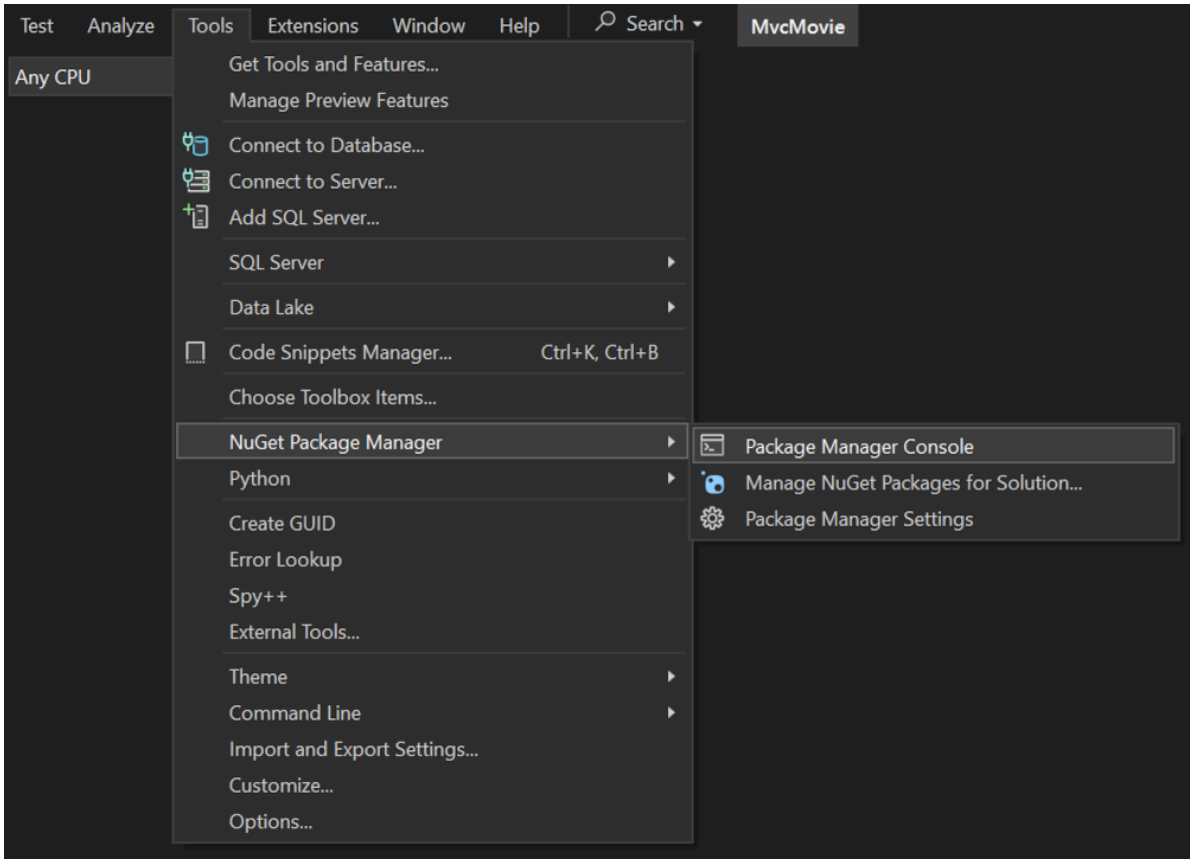
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

3. Use Code First Migrations to update the database schema.

For this tutorial, Code First Migrations is used.

Visual Studio

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
PowerShell
```

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the `Rating` field.

Run the app and verify you can create, edit, and display movies with a `Rating` field.

Previous Next

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 9, add validation to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section:

- Validation logic is added to the `Movie` model.
- You ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Delete the previously edited data

In the next step, validation rules are added that don't allow null values. Run the app, navigate to `/Movies/Index`, delete all listed movies, and stop the app. The app will use the seed data the next time it is run.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in validation attributes `Required`, `StringLength`, `RegularExpression`, `Range` and the `DataType` formatting attribute.

```
C#
```

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string? Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
    [Required]
    [StringLength(30)]
    public string? Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string? Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White spaces are allowed while numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.

    - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".

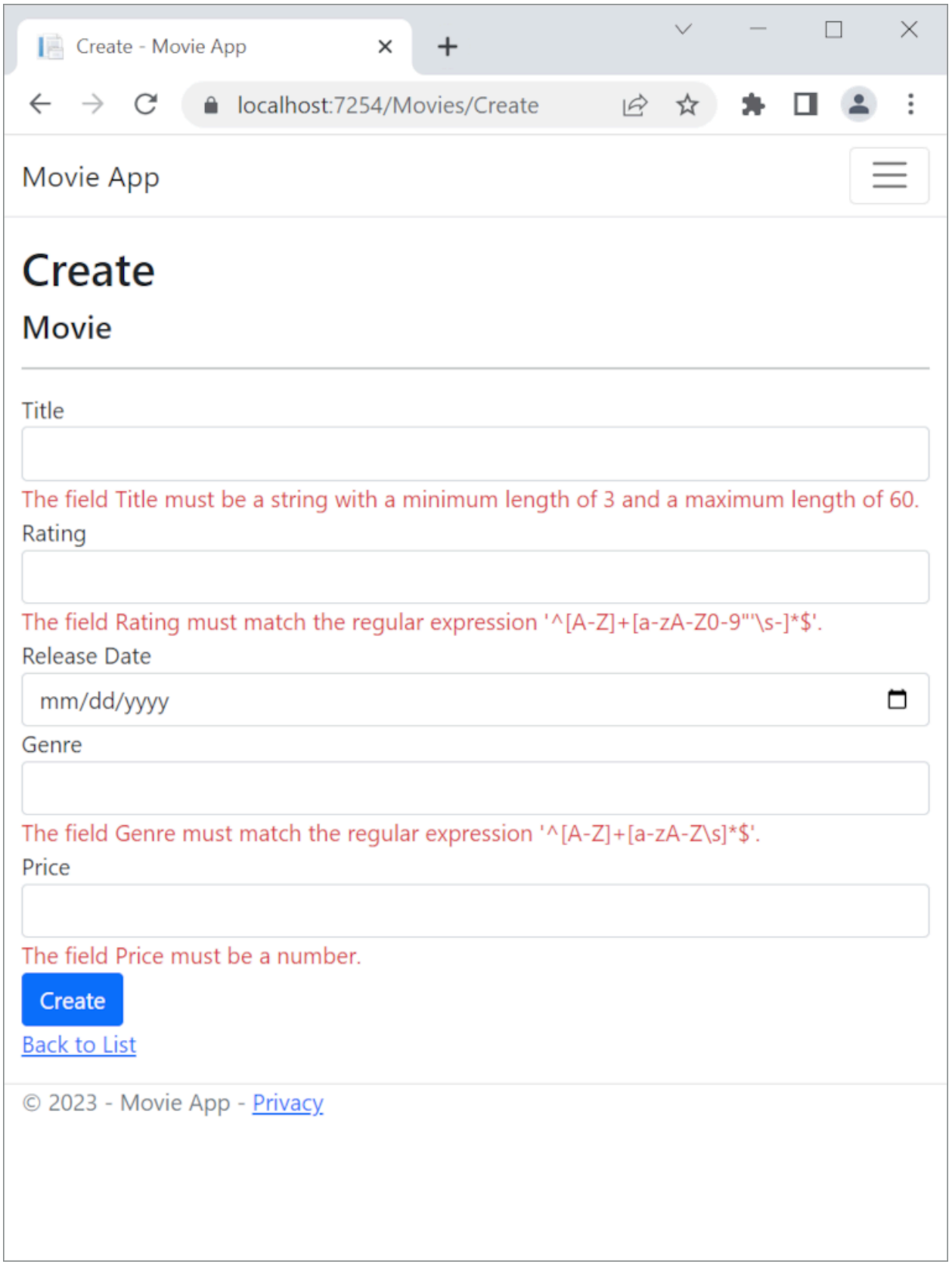
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI

Run the app and navigate to the Movies controller.

Select the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



Note

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the `Create.cshtml` view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data isn't sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the Fiddler tool, or the F12 Developer tools.

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

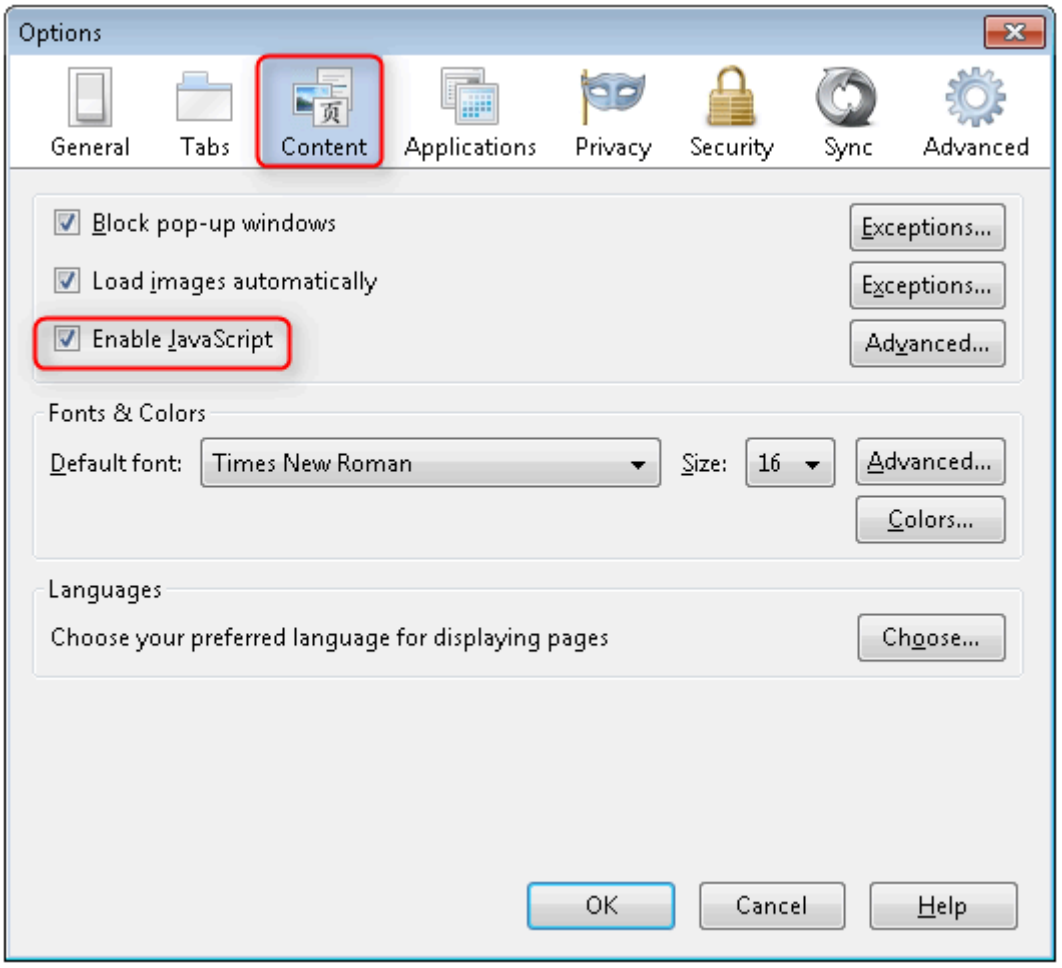
C#

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

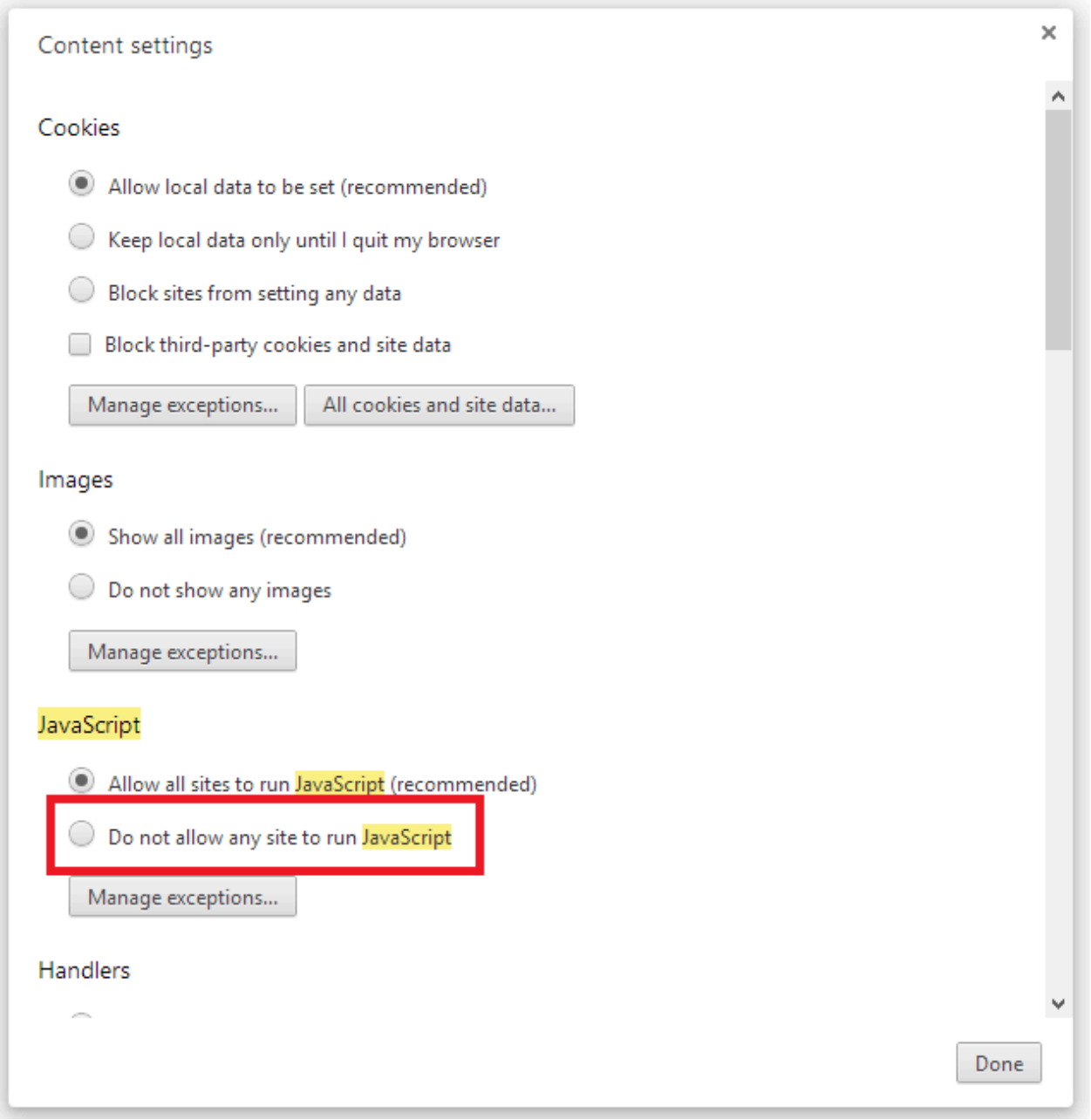
The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation won't submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

The following image shows how to disable JavaScript in the Firefox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.

```
74          // POST: Movies/Create
75          // To protect from overposting attacks, please enable t
76          // more details see http://go.microsoft.com/fwlink/?Lin
77          [HttpPost]
78          [ValidateAntiForgeryToken]
            0 references
79          public async Task<IAc          t> Create([Bind("ID,Title
80          {                              false
81              if (ModelState.IsValid)
82              {
83                  _context.Add(movie);
84                  await _context.SaveChangesAsync();
85                  return RedirectToAction("Index");
86              }
87              return View(movie);
88          }
89
```

A portion of the `Create.cshtml` view template is shown in the following markup:

HTML

```
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>

            @*Markup removed for brevity.*@
```

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The Validation Tag Helper displays validation errors. See Validation for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the `Movie.cs` file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType`

enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

C#

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data and supplies elements/attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

C#

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some

fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

> ⓘ **Note**
>
> jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:
>
> `[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]`

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

C#

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }
    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
```

```
    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$"), Required, StringLength(30)]
    public string Genre { get; set; }
    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

In the next part of the series, we review the app and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- Working with Forms
- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers

Previous Next

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 10, examine the Details and Delete methods of an ASP.NET Core app

Article • 08/04/2024

By Rick Anderson

Open the Movie controller and examine the `Details` method:

C#

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method, and an `id` value. Recall these segments are defined in `Program.cs`.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

EF makes it easy to search for data using the `FirstOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:{PORT}/Movies/Details/1` to something like `http://localhost:`

`{PORT}/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

C#

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{

    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    if (movie != null)
    {
        _context.Movie.Remove(movie);
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

C#

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

C#

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

C#

```
// POST: Movies/Delete/6
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Previous

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core Blazor tutorials

Article • 11/18/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

The following tutorials provide basic working experiences for building Blazor apps.

For an overview of Blazor, see ASP.NET Core Blazor.

- Build your first Blazor app
- Build a Blazor todo list app (Blazor Web App)
- Build a Blazor movie database app (Overview) (Blazor Web App)
- Use ASP.NET Core SignalR with Blazor (Blazor Web App)
- ASP.NET Core Blazor Hybrid tutorials
- Microsoft Learn
  - Blazor Learning Path
  - Blazor Learn Modules

Tutorial: Create a web API with ASP.NET Core

Article • 08/23/2024

By Rick Anderson and Kirk Larkin

This tutorial teaches the basics of building a controller-based web API that uses a database. Another approach to creating APIs in ASP.NET Core is to create *minimal APIs*. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating a minimal API, see Tutorial: Create a minimal API with ASP.NET Core.

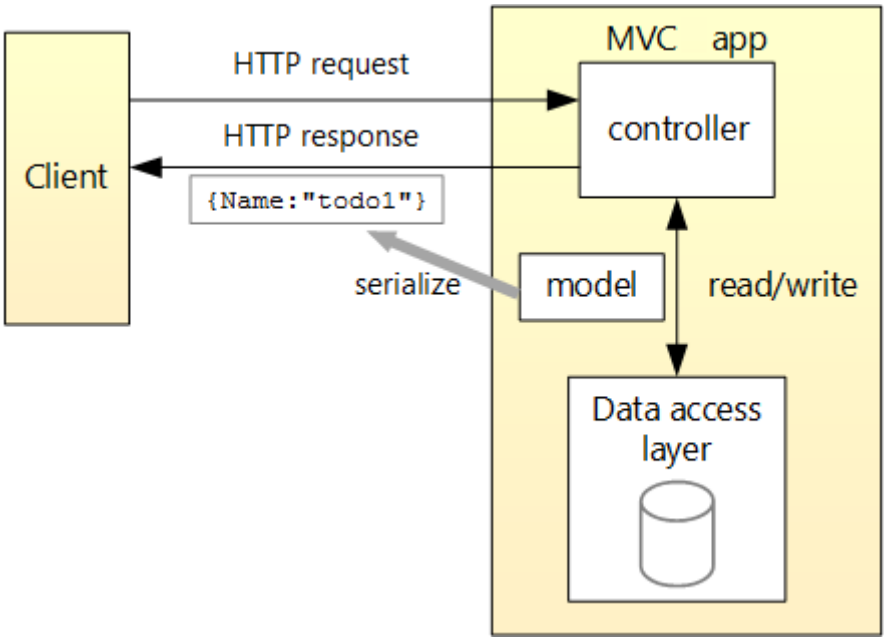
Overview

This tutorial creates the following API:

Expand table

API	Description	Request body	Response body
`GET /api/todoitems`	Get all to-do items	None	Array of to-do items
`GET /api/todoitems/{id}`	Get an item by ID	None	To-do item
`POST /api/todoitems`	Add a new item	To-do item	To-do item
`PUT /api/todoitems/{id}`	Update an existing item	To-do item	None
`DELETE /api/todoitems/{id}`	Delete an item	None	None

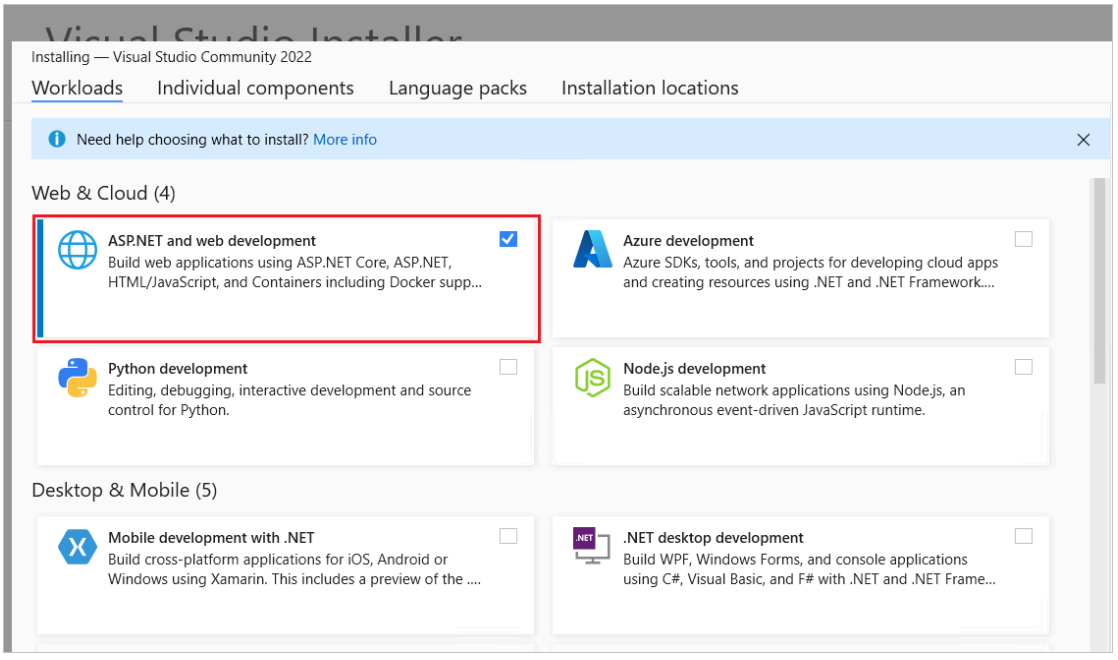
The following diagram shows the design of the app.



Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a web project

Visual Studio

- From the **File** menu, select **New** > **Project**.
- Enter *Web API* in the search box.
- Select the **ASP.NET Core Web API** template and select **Next**.
- In the **Configure your new project dialog**, name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Confirm the **Framework** is **.NET 8.0 (Long Term Support)**.
  - Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
  - Confirm the checkbox for **Enable OpenAPI support** is checked.
  - Select **Create**.

Add a NuGet package

A NuGet package must be added to support the database used in this tutorial.

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab.
- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select `Microsoft.EntityFrameworkCore.InMemory`.
- Select the **Project** checkbox in the right pane and then select **Install**.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

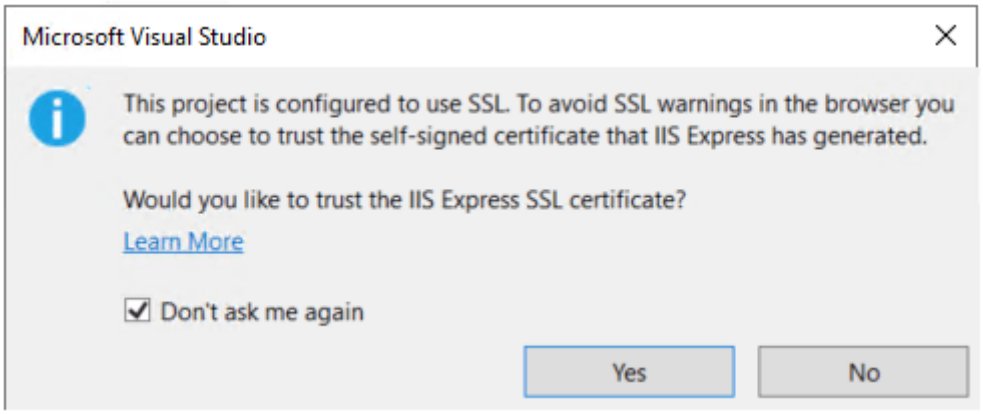
Test the project

The project template creates a `WeatherForecast` API with support for Swagger.

Visual Studio

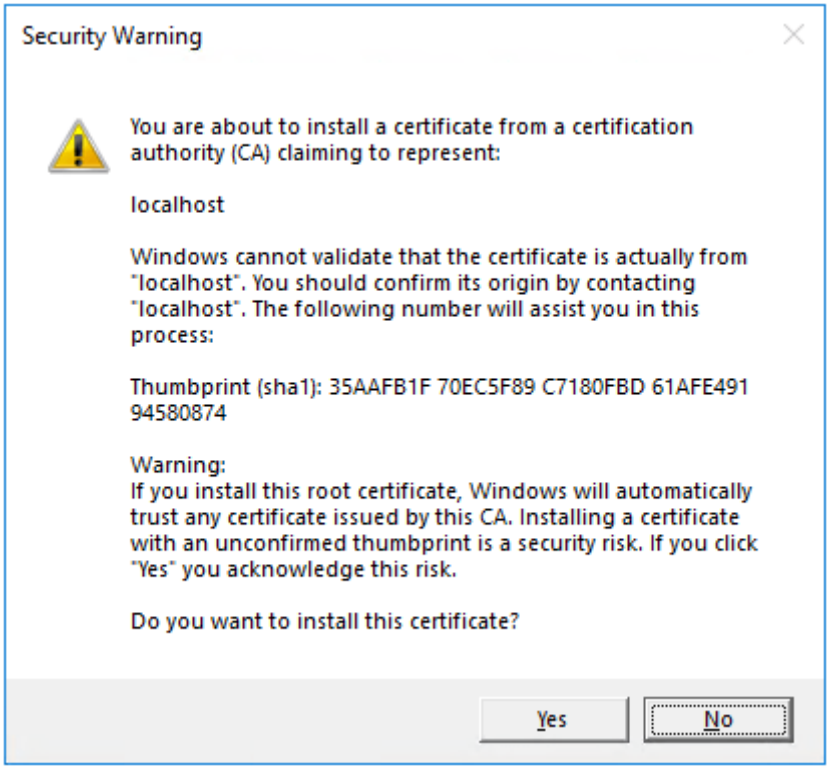
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the default browser and navigates to `https://localhost:<port>/swagger/index.html`, where `<port>` is a randomly chosen port number set at the project creation.

The Swagger page `/swagger/index.html` is displayed. Select **GET** > **Try it out** > **Execute**. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop-down list box with media types and the example value and schema.

If the Swagger page doesn't appear, see this GitHub issue.

Swagger is used to generate useful documentation and help pages for web APIs. This tutorial uses Swagger to test the app. For more information on Swagger, see ASP.NET Core web API documentation with Swagger / OpenAPI.

Copy and paste the **Request URL** in the browser: `https://localhost:<port>/weatherforecast`

JSON similar to the following example is returned:

JSON


```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
```

```
        "summary": "Chilly"
    }
]
```

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is the `TodoItem` class.

Visual Studio

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder `Models`.
- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following:

C#

```
namespace TodoApi.Models;

public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the `Models` folder is used by convention.

Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class.

Visual Studio

- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.

- Enter the following code:

```
C#

using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models;

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update `Program.cs` with the following highlighted code:

```
C#

using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
```

```
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Adds `using` directives.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

Visual Studio

- Right-click the `Controllers` folder.
- Select **Add** > **New Scaffolded Item**.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

  If the scaffolding operation fails, select **Add** to try scaffolding a second time.

The generated code:

- Marks the class with the [ApiController] attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name (method name) isn't included in the endpoint. That is, the action's associated method name isn't used in the matching route.

Update the PostTodoItem create method

Update the return statement in the `PostTodoItem` to use the nameof operator:

C#

```
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //    return CreatedAtAction("GetTodoItem", new { id = todoItem.Id },
todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id },
todoItem);
}
```

The preceding code is an `HTTP POST` method, as indicated by the [HttpPost] attribute. The method gets the value of the `TodoItem` from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. `HTTP 201` is the standard response for an `HTTP POST` method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

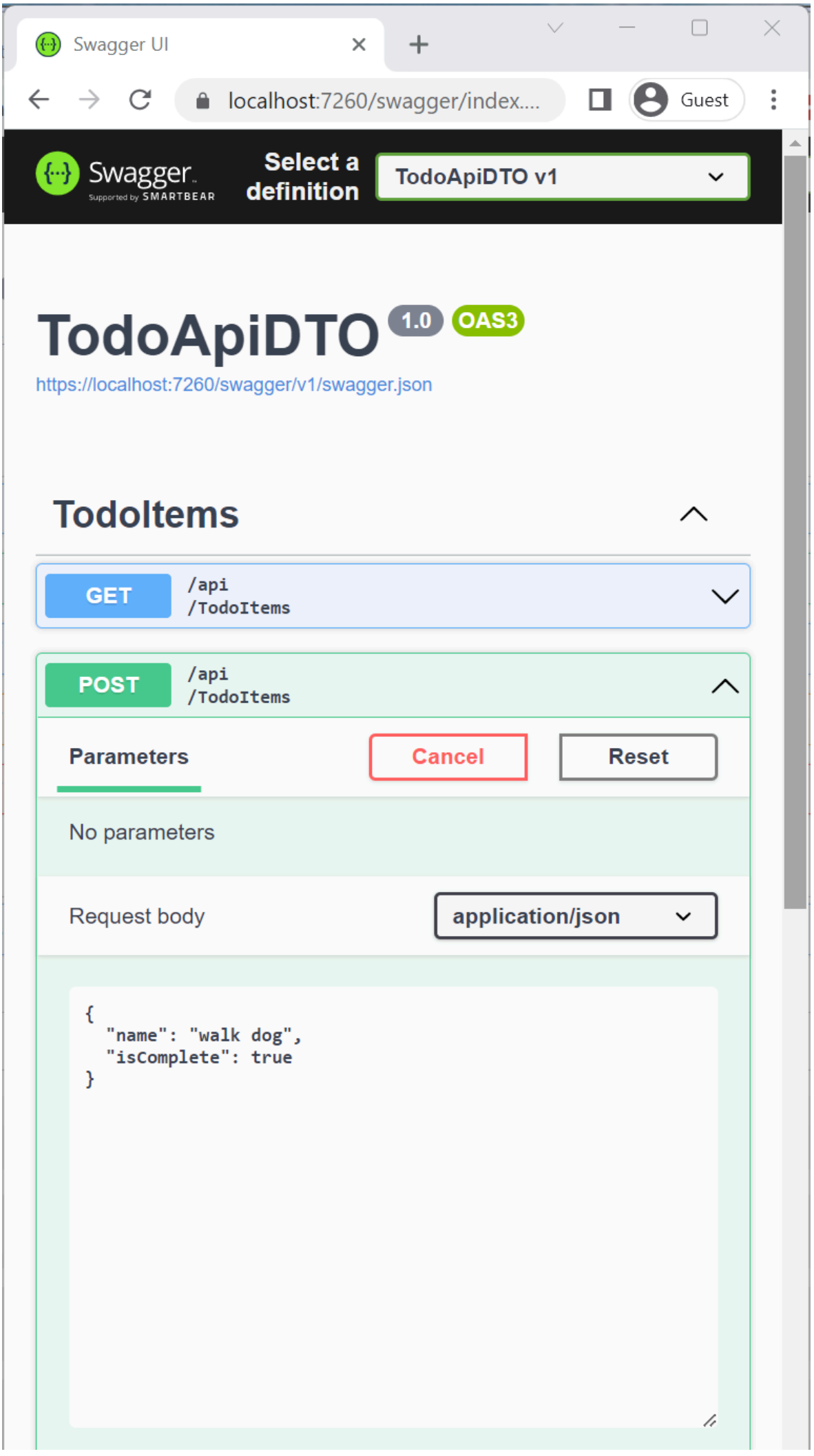
Test PostTodoItem

- Press Ctrl+F5 to run the app.

- In the Swagger browser window, select **POST /api/TodoItems**, and then select **Try it out**.
- In the **Request body** input window, update the JSON. For example,

```JSON
{
  "name": "walk dog",
  "isComplete": true
}
```

- Select **Execute**


Swagger UI
localhost:7260/swagger/index....
Guest
Swagger
Supported by SMARTBEAR
Select a definition
TodoApiDTO v1
TodoApiDTO
1.0
OAS3
https://localhost:7260/swagger/v1/swagger.json
TodoItems
GET
/api
/TodoItems
POST
/api
/TodoItems
Parameters
Cancel
Reset
No parameters
Request body
application/json
{
"name": "walk dog",
"isComplete": true
}



Test the location header URI

In the preceding POST, the Swagger UI shows the location header under **Response headers**. For example, `location: https://localhost:7260/api/TodoItems/1`. The location header shows the URI to the created resource.

To test the location header:

- In the Swagger browser window, select **GET /api/TodoItems/{id}**, and then select **Try it out**.
- Enter `1` in the `id` input box, and then select **Execute**.

GET /api/TodoItems/{id}

Parameters Cancel

Name	Description
id * required integer($int64) (path)	1

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'https://localhost:7260/api/TodoItems/1' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:7260/api/TodoItems/1
```

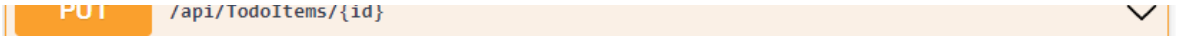
Server response

Code	Details

200	Response body
	`{ "id": 1, "name": "walk dog", "isComplete": true }` Download
	Response headers
	`content-type: application/json; charset=utf-8`
	`date: Fri,21 Oct 2022 00:32:45 GMT`
	`server: Kestrel`

Responses

Code	Description	Links
200	Success Media type: text/plain Controls Accept header. Example Value | Schema `{ "id": 0, "name": "string", "isComplete": true }`	*No links*



Examine the GET methods

Two GET endpoints are implemented:

- `GET /api/todoitems`
- `GET /api/todoitems/{id}`

The previous section showed an example of the `/api/todoitems/{id}` route.

Follow the POST instructions to add another todo item, and then test the `/api/todoitems` route using Swagger.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request doesn't return any data. If no data is returned, POST data to the app.

Routing and URL paths

The [HttpGet] attribute denotes a method that responds to an `HTTP GET` request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
C#

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.

- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an `HTTP 200` response.

The PutTodoItem method

Examine the `PutTodoItem` method:

C#

```
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;
```

```
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses `HTTP PUT`. The response is 204 (No Content). According to the HTTP specification, a `PUT` request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Using the Swagger UI, use the PUT button to update the `TodoItem` that has Id = 1 and set its name to `"feed fish"`. Note the response is HTTP 204 No Content.

The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

C#

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
```

```
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Test the DeleteTodoItem method

Use the Swagger UI to delete the `TodoItem` that has Id = 1. Note the response is HTTP 204 No Content.

Test with other tools

There are many other tools that can be used to test web APIs, for example:

- Visual Studio Endpoints Explorer and .http files
- http-repl
- curl. Swagger uses `curl` and shows the `curl` commands it submits.
- Fiddler

For more information, see:

- Minimal API tutorial: test with .http files and Endpoints Explorer
- Install and test APIs with http-repl

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this, and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this tutorial.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

C#

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
        public string? Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

C#

```
namespace TodoApi.Models;

public class TodoItemDTO
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi.Controllers;

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
```

```
    {
        _context = context;
    }

    // GET: api/TodoItems
    [HttpGet]
    public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
        return await _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

    // GET: api/TodoItems/5
    // <snippet_GetByID>
    [HttpGet("{id}")]
    public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }
    // </snippet_GetByID>

    // PUT: api/TodoItems/5
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Update>
    [HttpPut("{id}")]
    public async Task<IActionResult> PutTodoItem(long id, TodoItemDTO
todoDTO)
    {
        if (id != todoDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoDTO.Name;
        todoItem.IsComplete = todoDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
```

```
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }
    // </snippet_Update>

    // POST: api/TodoItems
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Create>
    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> PostTodoItem(TodoItemDTO
todoDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoDTO.IsComplete,
            Name = todoDTO.Name
        };

        _context.TodoItems.Add(todoItem);
        await _context.SaveChangesAsync();

        return CreatedAtAction(
            nameof(GetTodoItem),
            new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }
    // </snippet_Create>

    // DELETE: api/TodoItems/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        _context.TodoItems.Remove(todoItem);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool TodoItemExists(long id)
    {
        return _context.TodoItems.Any(e => e.Id == id);
    }

    private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
```

```
        new TodoItemDTO
        {
            Id = todoItem.Id,
            Name = todoItem.Name,
            IsComplete = todoItem.IsComplete
        };
}
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

Web API video series

See Video: Beginner's Series to: Web APIs.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see **Migrate from ASP.NET Core 5.0 to 6.0**.

For more information, see the Duende Identity Server documentation (Duende Software website).

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app.

Additional resources

View or download sample code for this tutorial. See how to download.

For more information, see the following resources:

- Create web APIs with ASP.NET Core
- Tutorial: Create a minimal API with ASP.NET Core
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8
- Routing to controller actions in ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Deploy ASP.NET Core apps to Azure App Service
- Host and deploy ASP.NET Core
- Create a web API with ASP.NET Core

Create a web API with ASP.NET Core and MongoDB

Article • 04/17/2024

By Pratik Khandelwal and Scott Addie

This tutorial creates a web API that runs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

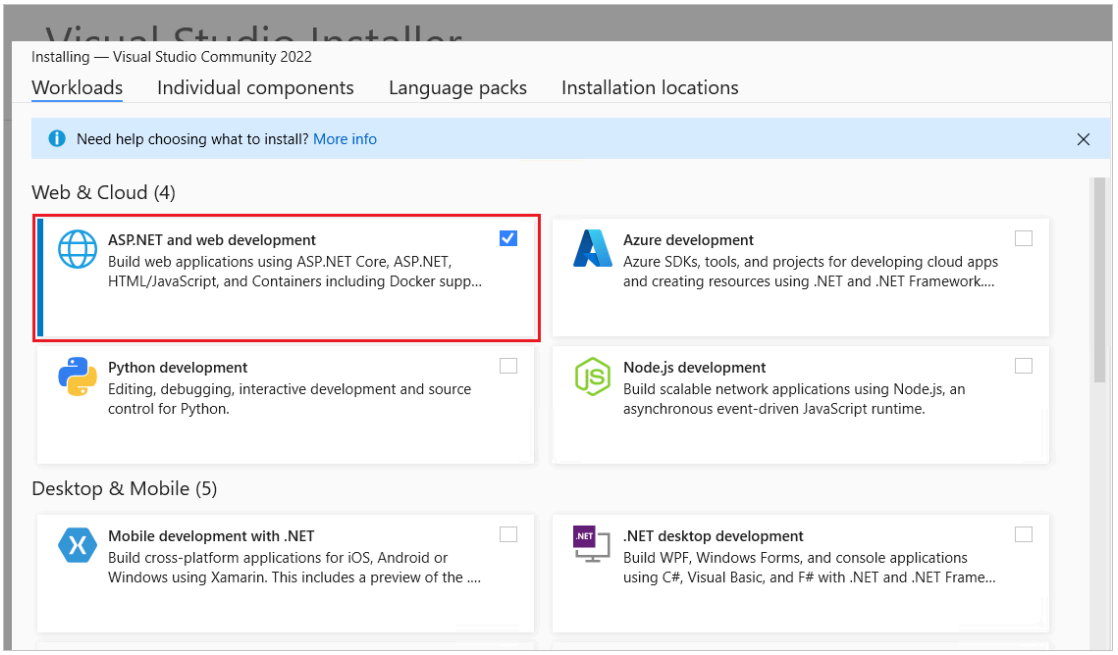
- ✓ Configure MongoDB
- ✓ Create a MongoDB database
- ✓ Define a MongoDB collection and schema
- ✓ Perform MongoDB CRUD operations from a web API
- ✓ Customize JSON serialization

Prerequisites

- MongoDB 6.0.5 or later
- MongoDB Shell

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Configure MongoDB

Enable MongoDB and MongoDB Shell access from anywhere on the development machine (Windows/Linux/macOS):

1. Download and Install MongoDB Shell:

    - macOS/Linux: Choose a directory to extract the MongoDB Shell to. Add the resulting path for `mongosh` to the `PATH` environment variable.
    - Windows: MongoDB Shell (mongosh.exe) is installed at *C:\Users<user>\AppData\Local\Programs\mongosh*. Add the resulting path for `mongosh.exe` to the `PATH` environment variable.

2. Download and Install MongoDB:

    - macOS/Linux: Verify the directory that MongoDB was installed at, usually in */usr/local/mongodb*. Add the resulting path for `mongodb` to the `PATH` environment variable.
    - Windows: MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `PATH` environment variable.

3. Choose a Data Storage Directory: Select a directory on your development machine for storing data. Create the directory if it doesn't exist. The MongoDB Shell doesn't create new directories:

    - macOS/Linux: For example, `/usr/local/var/mongodb`.

- Windows: For example, `C:\\BooksData`.

4. In the OS command shell (not the MongoDB Shell), use the following command to connect to MongoDB on default port 27017. Replace `<data_directory_path>` with the directory chosen in the previous step.

```
Console

mongod --dbpath <data_directory_path>
```

Use the previously installed MongoDB Shell in the following steps to create a database, make collections, and store documents. For more information on MongoDB Shell commands, see mongosh.

1. Open a MongoDB command shell instance by launching `mongosh.exe`.

2. In the command shell, connect to the default test database by running the following command:

```
Console

mongosh
```

3. Run the following command in the command shell:

```
Console

use BookStore
```

   A database named *BookStore* is created if it doesn't already exist. If the database does exist, its connection is opened for transactions.

4. Create a `Books` collection using following command:

```
Console

db.createCollection('Books')
```

   The following result is displayed:

```
Console

{ "ok" : 1 }
```

5. Define a schema for the `Books` collection and insert two documents using the following command:

```
Console

db.Books.insertMany([{ "Name": "Design Patterns", "Price": 54.93,
"Category": "Computers", "Author": "Ralph Johnson" }, { "Name": "Clean
Code", "Price": 43.15, "Category": "Computers","Author": "Robert C.
Martin" }])
```

A result similar to the following is displayed:

```
Console

{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("61a6058e6c43f32854e51f51"),
        ObjectId("61a6058e6c43f32854e51f52")
    ]
}
```

> **Note**
>
> The `ObjectId`s shown in the preceding result won't match those shown in the command shell.

6. View the documents in the database using the following command:

```
Console

db.Books.find().pretty()
```

A result similar to the following is displayed:

```
Console

{
    "_id" : ObjectId("61a6058e6c43f32854e51f51"),
    "Name" : "Design Patterns",
    "Price" : 54.93,
    "Category" : "Computers",
    "Author" : "Ralph Johnson"
}
{
    "_id" : ObjectId("61a6058e6c43f32854e51f52"),
```

```
    "Name" : "Clean Code",
    "Price" : 43.15,
    "Category" : "Computers",
    "Author" : "Robert C. Martin"
}
```

The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

Create the ASP.NET Core web API project

Visual Studio

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web API** project type, and select **Next**.
3. Name the project *BookStoreApi*, and select **Next**.
4. Select the **.NET 8.0 (Long Term support)** framework and select **Create**.
5. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

PowerShell

```
Install-Package MongoDB.Driver
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

C#

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BookStoreApi.Models;

public class Book
{
```

```
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; set; }

    [BsonElement("Name")]
    public string BookName { get; set; } = null!;

    public decimal Price { get; set; }

    public string Category { get; set; } = null!;

    public string Author { get; set; } = null!;
}
```

In the preceding class, the `Id` property is:

- Required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Annotated with [BsonId] to make this property the document's primary key.
- Annotated with [BsonRepresentation(BsonType.ObjectId)] to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the [BsonElement] attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to `appsettings.json`:

JSON

```
{
    "BookStoreDatabase": {
        "ConnectionString": "mongodb://localhost:27017",
        "DatabaseName": "BookStore",
        "BooksCollectionName": "Books"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*"
}
```

2. Add a `BookStoreDatabaseSettings` class to the *Models* directory with the following code:

```C#
namespace BookStoreApi.Models;

public class BookStoreDatabaseSettings
{
    public string ConnectionString { get; set; } = null!;

    public string DatabaseName { get; set; } = null!;

    public string BooksCollectionName { get; set; } = null!;
}
```

   The preceding `BookStoreDatabaseSettings` class is used to store the `appsettings.json` file's `BookStoreDatabase` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));
```

   In the preceding code, the configuration instance to which the `appsettings.json` file's `BookStoreDatabase` section binds is registered in the Dependency Injection (DI) container. For example, the `BookStoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookStoreDatabase:ConnectionString` property in `appsettings.json`.

4. Add the following code to the top of `Program.cs` to resolve the `BookStoreDatabaseSettings` reference:

```C#
using BookStoreApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.

2. Add a `BooksService` class to the *Services* directory with the following code:

C#

```
using BookStoreApi.Models;
using Microsoft.Extensions.Options;
using MongoDB.Driver;

namespace BookStoreApi.Services;

public class BooksService
{
    private readonly IMongoCollection<Book> _booksCollection;

    public BooksService(
        IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
    {
        var mongoClient = new MongoClient(
            bookStoreDatabaseSettings.Value.ConnectionString);

        var mongoDatabase = mongoClient.GetDatabase(
            bookStoreDatabaseSettings.Value.DatabaseName);

        _booksCollection = mongoDatabase.GetCollection<Book>(
            bookStoreDatabaseSettings.Value.BooksCollectionName);
    }

    public async Task<List<Book>> GetAsync() =>
        await _booksCollection.Find(_ => true).ToListAsync();

    public async Task<Book?> GetAsync(string id) =>
        await _booksCollection.Find(x => x.Id ==
id).FirstOrDefaultAsync();

    public async Task CreateAsync(Book newBook) =>
        await _booksCollection.InsertOneAsync(newBook);

    public async Task UpdateAsync(string id, Book updatedBook) =>
        await _booksCollection.ReplaceOneAsync(x => x.Id == id,
updatedBook);

    public async Task RemoveAsync(string id) =>
        await _booksCollection.DeleteOneAsync(x => x.Id == id);
}
```

In the preceding code, a `BookStoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the `appsettings.json` configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));

builder.Services.AddSingleton<BooksService>();
```

In the preceding code, the `BooksService` class is registered with DI to support constructor injection in consuming classes. The singleton service lifetime is most appropriate because `BooksService` takes a direct dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of `Program.cs` to resolve the `BooksService` reference:

```C#
using BookStoreApi.Services;
```

The `BooksService` class uses the following `MongoDB.Driver` members to run CRUD operations against the database:

- MongoClient: Reads the server instance for running database operations. The constructor of this class is provided in the MongoDB connection string:

```C#
public BooksService(
    IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
{
    var mongoClient = new MongoClient(
        bookStoreDatabaseSettings.Value.ConnectionString);

    var mongoDatabase = mongoClient.GetDatabase(
        bookStoreDatabaseSettings.Value.DatabaseName);

    _booksCollection = mongoDatabase.GetCollection<Book>(
        bookStoreDatabaseSettings.Value.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for running operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Run CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOneAsync: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOneAsync: Inserts the provided object as a new document in the collection.
- ReplaceOneAsync: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

C#

```
using BookStoreApi.Models;
using BookStoreApi.Services;
using Microsoft.AspNetCore.Mvc;

namespace BookStoreApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BooksController : ControllerBase
{
    private readonly BooksService _booksService;

    public BooksController(BooksService booksService) =>
        _booksService = booksService;

    [HttpGet]
    public async Task<List<Book>> Get() =>
        await _booksService.GetAsync();

    [HttpGet("{id:length(24)}")]
```

```
    public async Task<ActionResult<Book>> Get(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        return book;
    }

    [HttpPost]
    public async Task<IActionResult> Post(Book newBook)
    {
        await _booksService.CreateAsync(newBook);

        return CreatedAtAction(nameof(Get), new { id = newBook.Id },
newBook);
    }

    [HttpPut("{id:length(24)}")]
    public async Task<IActionResult> Update(string id, Book updatedBook)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        updatedBook.Id = book.Id;

        await _booksService.UpdateAsync(id, updatedBook);

        return NoContent();
    }

    [HttpDelete("{id:length(24)}")]
    public async Task<IActionResult> Delete(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        await _booksService.RemoveAsync(id);

        return NoContent();
    }
}
```

The preceding web API controller:

- Uses the `BooksService` class to run CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtAction in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtAction` also adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.

2. Navigate to `https://localhost:<port>/api/books`, where `<port>` is the automatically assigned port number for the app, to test the controller's parameterless `Get` action method, select **Try it out** > **Execute**. A JSON response similar to the following is displayed:

   JSON

   ```
   [
     {
       "id": "61a6058e6c43f32854e51f51",
       "bookName": "Design Patterns",
       "price": 54.93,
       "category": "Computers",
       "author": "Ralph Johnson"
     },
     {
       "id": "61a6058e6c43f32854e51f52",
       "bookName": "Clean Code",
       "price": 43.15,
       "category": "Computers",
       "author": "Robert C. Martin"
     }
   ]
   ```

3. Navigate to `https://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. A JSON response similar to the following is displayed:

   JSON

   ```
   {
     "id": "61a6058e6c43f32854e51f52",
   ```

```
  "bookName": "Clean Code",
  "price": 43.15,
  "category": "Computers",
  "author": "Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. In `Program.cs`, chain the following highlighted code on to the `AddControllers` method call:

   C#

   ```
   var builder = WebApplication.CreateBuilder(args);

   // Add services to the container.
   builder.Services.Configure<BookStoreDatabaseSettings>(
       builder.Configuration.GetSection("BookStoreDatabase"));

   builder.Services.AddSingleton<BooksService>();

   builder.Services.AddControllers()
       .AddJsonOptions(
           options => options.JsonSerializerOptions.PropertyNamingPolicy =
   null);
   ```

   With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author` instead of `author`.

2. In `Models/Book.cs`, annotate the `BookName` property with the [JsonPropertyName] attribute:

   C#

   ```
   [BsonElement("Name")]
   [JsonPropertyName("Name")]
   ```

```
public string BookName { get; set; } = null!;
```

The `[JsonPropertyName]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

3. Add the following code to the top of `Models/Book.cs` to resolve the `[JsonProperty]` attribute reference:

```
C#

using System.Text.Json.Serialization;
```

4. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

For more information, see the Duende Identity Server documentation (Duende Software website).

Additional resources

- View or download sample code (how to download)
- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Create a web API with ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Call an ASP.NET Core web API with JavaScript

Article • 04/24/2024

By Rick Anderson

This tutorial shows how to call an ASP.NET Core web API with JavaScript, using the Fetch API.

Prerequisites

- Complete Tutorial: Create a web API
- Familiarity with CSS, HTML, and JavaScript

Call the web API with JavaScript

In this section, you'll add an HTML page containing forms for creating and managing to-do items. Event handlers are attached to elements on the page. The event handlers result in HTTP requests to the web API's action methods. The Fetch API's `fetch` function initiates each HTTP request.

The `fetch` function returns a Promise object, which contains an HTTP response represented as a `Response` object. A common pattern is to extract the JSON response body by invoking the `json` function on the `Response` object. JavaScript updates the page with the details from the web API's response.

The simplest `fetch` call accepts a single parameter representing the route. A second parameter, known as the `init` object, is optional. `init` is used to configure the HTTP request.

1. Configure the app to serve static files and enable default file mapping. The following highlighted code is needed in `Program.cs`:

C#

```
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
```

```
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));

var app = builder.Build();

if (builder.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.UseDefaultFiles();
app.UseStaticFiles();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

1. Create a *wwwroot* folder in the project root.
2. Create a *css* folder inside of the *wwwroot* folder.
3. Create a *js* folder inside of the *wwwroot* folder.
4. Add an HTML file named `index.html` to the *wwwroot* folder. Replace the contents of `index.html` with the following markup:

HTML

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>To-do CRUD</title>
    <link rel="stylesheet" href="css/site.css" />
</head>
<body>
    <h1>To-do CRUD</h1>
    <h3>Add</h3>
    <form action="javascript:void(0);" method="POST"
onsubmit="addItem()">
        <input type="text" id="add-name" placeholder="New to-do">
        <input type="submit" value="Add">
    </form>

    <div id="editForm">
        <h3>Edit</h3>
        <form action="javascript:void(0);" onsubmit="updateItem()">
            <input type="hidden" id="edit-id">
```

```
            <input type="checkbox" id="edit-isComplete">
            <input type="text" id="edit-name">
            <input type="submit" value="Save">
            <a onclick="closeInput()" aria-label="Close">&#10006;</a>
        </form>
    </div>

    <p id="counter"></p>

    <table>
        <tr>
            <th>Is Complete?</th>
            <th>Name</th>
            <th></th>
            <th></th>
        </tr>
        <tbody id="todos"></tbody>
    </table>

    <script src="js/site.js" asp-append-version="true"></script>
    <script type="text/javascript">
        getItems();
    </script>
</body>
</html>
```

5. Add a CSS file named `site.css` to the *wwwroot/css* folder. Replace the contents of `site.css` with the following styles:

```css
input[type='submit'], button, [aria-label] {
    cursor: pointer;
}

#editForm {
    display: none;
}

table {
    font-family: Arial, sans-serif;
    border: 1px solid;
    border-collapse: collapse;
}

th {
    background-color: #f8f8f8;
    padding: 5px;
}

td {
    border: 1px solid;
```

```
    padding: 5px;
}
```

6. Add a JavaScript file named `site.js` to the *wwwroot/js* folder. Replace the contents of `site.js` with the following code:

JavaScript

```
const uri = 'api/todoitems';
let todos = [];

function getItems() {
  fetch(uri)
    .then(response => response.json())
    .then(data => _displayItems(data))
    .catch(error => console.error('Unable to get items.', error));
}

function addItem() {
  const addNameTextbox = document.getElementById('add-name');

  const item = {
    isComplete: false,
    name: addNameTextbox.value.trim()
  };

  fetch(uri, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
    .then(response => response.json())
    .then(() => {
      getItems();
      addNameTextbox.value = '';
    })
    .catch(error => console.error('Unable to add item.', error));
}

function deleteItem(id) {
  fetch(`${uri}/${id}`, {
    method: 'DELETE'
  })
  .then(() => getItems())
  .catch(error => console.error('Unable to delete item.', error));
}

function displayEditForm(id) {
  const item = todos.find(item => item.id === id);
```

```
  document.getElementById('edit-name').value = item.name;
  document.getElementById('edit-id').value = item.id;
  document.getElementById('edit-isComplete').checked = item.isComplete;
  document.getElementById('editForm').style.display = 'block';
}

function updateItem() {
  const itemId = document.getElementById('edit-id').value;
  const item = {
    id: parseInt(itemId, 10),
    isComplete: document.getElementById('edit-isComplete').checked,
    name: document.getElementById('edit-name').value.trim()
  };

  fetch(`${uri}/${itemId}`, {
    method: 'PUT',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
  .then(() => getItems())
  .catch(error => console.error('Unable to update item.', error));

  closeInput();

  return false;
}

function closeInput() {
  document.getElementById('editForm').style.display = 'none';
}

function _displayCount(itemCount) {
  const name = (itemCount === 1) ? 'to-do' : 'to-dos';

  document.getElementById('counter').innerText = `${itemCount}
${name}`;
}

function _displayItems(data) {
  const tBody = document.getElementById('todos');
  tBody.innerHTML = '';

  _displayCount(data.length);

  const button = document.createElement('button');

  data.forEach(item => {
    let isCompleteCheckbox = document.createElement('input');
    isCompleteCheckbox.type = 'checkbox';
    isCompleteCheckbox.disabled = true;
    isCompleteCheckbox.checked = item.isComplete;
```

```
    let editButton = button.cloneNode(false);
    editButton.innerText = 'Edit';
    editButton.setAttribute('onclick', `displayEditForm(${item.id})`);

    let deleteButton = button.cloneNode(false);
    deleteButton.innerText = 'Delete';
    deleteButton.setAttribute('onclick', `deleteItem(${item.id})`);

    let tr = tBody.insertRow();

    let td1 = tr.insertCell(0);
    td1.appendChild(isCompleteCheckbox);

    let td2 = tr.insertCell(1);
    let textNode = document.createTextNode(item.name);
    td2.appendChild(textNode);

    let td3 = tr.insertCell(2);
    td3.appendChild(editButton);

    let td4 = tr.insertCell(3);
    td4.appendChild(deleteButton);
  });

  todos = data;
}
```

A change to the ASP.NET Core project's launch settings may be required to test the HTML page locally:

1. Open *Properties\launchSettings.json*.
2. Remove the `launchUrl` property to force the app to open at `index.html`—the project's default file.

This sample calls all of the CRUD methods of the web API. Following are explanations of the web API requests.

Get a list of to-do items

In the following code, an HTTP GET request is sent to the *api/todoitems* route:

JavaScript

```
fetch(uri)
  .then(response => response.json())
  .then(data => _displayItems(data))
  .catch(error => console.error('Unable to get items.', error));
```

When the web API returns a successful status code, the `_displayItems` function is invoked. Each to-do item in the array parameter accepted by `_displayItems` is added to a table with **Edit** and **Delete** buttons. If the web API request fails, an error is logged to the browser's console.

Add a to-do item

In the following code:

- An `item` variable is declared to construct an object literal representation of the to-do item.
- A Fetch request is configured with the following options:
  - `method`—specifies the POST HTTP action verb.
  - `body`—specifies the JSON representation of the request body. The JSON is produced by passing the object literal stored in `item` to the JSON.stringify function.
  - `headers`—specifies the `Accept` and `Content-Type` HTTP request headers. Both headers are set to `application/json` to specify the media type being received and sent, respectively.
- An HTTP POST request is sent to the *api/todoitems* route.

JavaScript

```
function addItem() {
  const addNameTextbox = document.getElementById('add-name');

  const item = {
    isComplete: false,
    name: addNameTextbox.value.trim()
  };

  fetch(uri, {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(item)
  })
    .then(response => response.json())
    .then(() => {
      getItems();
      addNameTextbox.value = '';
    })
    .catch(error => console.error('Unable to add item.', error));
}
```

When the web API returns a successful status code, the `getItems` function is invoked to update the HTML table. If the web API request fails, an error is logged to the browser's console.

Update a to-do item

Updating a to-do item is similar to adding one; however, there are two significant differences:

- The route is suffixed with the unique identifier of the item to update. For example, *api/todoitems/1*.
- The HTTP action verb is PUT, as indicated by the `method` option.

JavaScript

```
fetch(`${uri}/${itemId}`, {
  method: 'PUT',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(item)
})
.then(() => getItems())
.catch(error => console.error('Unable to update item.', error));
```

Delete a to-do item

To delete a to-do item, set the request's `method` option to `DELETE` and specify the item's unique identifier in the URL.

JavaScript

```
fetch(`${uri}/${id}`, {
  method: 'DELETE'
})
.then(() => getItems())
.catch(error => console.error('Unable to delete item.', error));
```

Advance to the next tutorial to learn how to generate web API help pages:

Get started with Swashbuckle and ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Create backend services for native mobile apps with ASP.NET Core

Article • 04/22/2024

By James Montemagno

Mobile apps can communicate with ASP.NET Core backend services. For instructions on connecting local web services from iOS simulators and Android emulators, see Connect to Local Web Services from iOS Simulators and Android Emulators.

View or download sample backend services code

The Sample Native Mobile App

This tutorial demonstrates how to create backend services using ASP.NET Core to support native mobile apps. It uses the Xamarin.Forms TodoRest app as its native client, which includes separate native clients for Android, iOS, and Windows. You can follow the linked tutorial to create the native app (and install the necessary free Xamarin tools), and download the Xamarin sample solution. The Xamarin sample includes an ASP.NET Core Web API services project, which this article's ASP.NET Core app replaces (with no changes required by the client).



Features

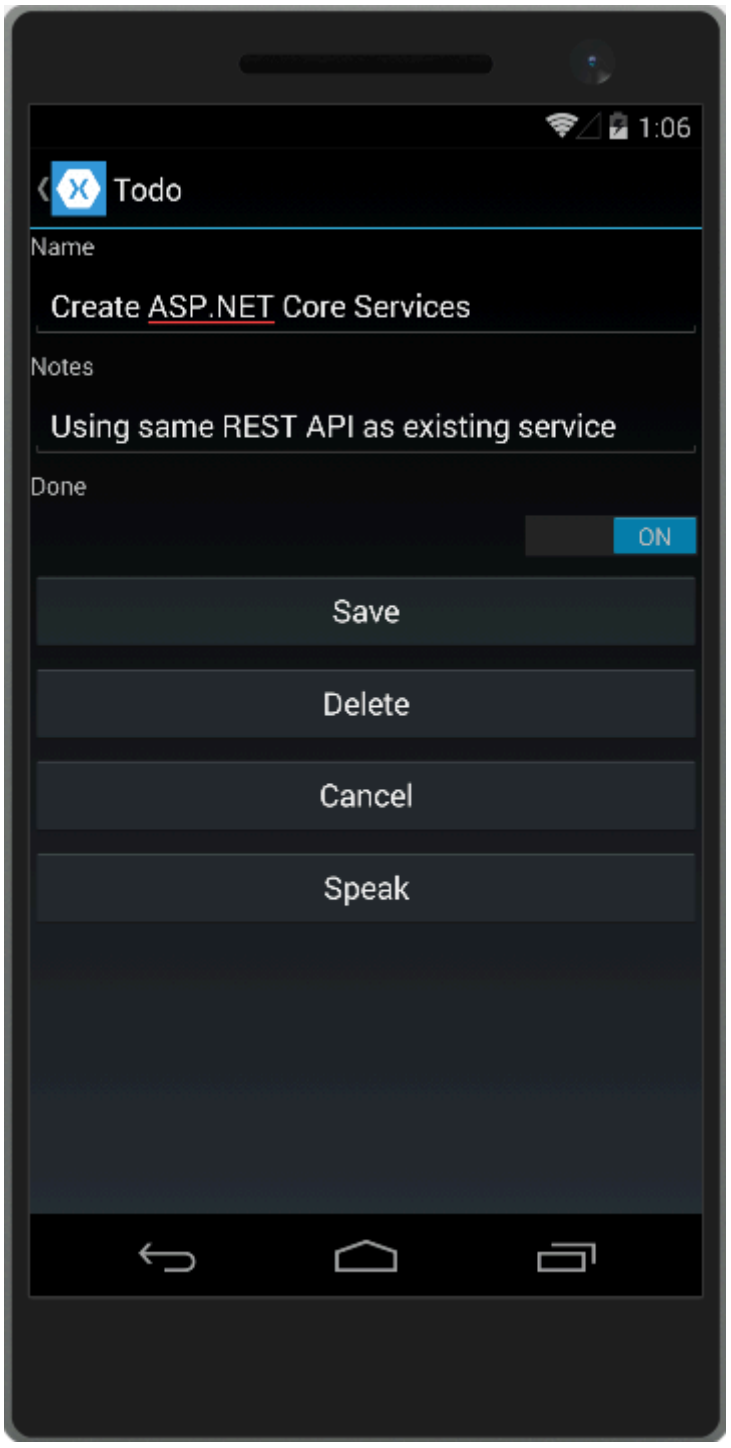
The TodoREST app supports listing, adding, deleting, and updating To-Do items. Each item has an ID, a Name, Notes, and a property indicating whether it's been Done yet.

In the previous example, The main view of the items lists each item's name and indicates if it's done with a checkmark.

Tapping the `+` icon opens an add item dialog:



Tapping an item on the main list screen opens up an edit dialog where the item's Name, Notes, and Done settings can be modified, or the item can be deleted:



To test it out yourself against the ASP.NET Core app created in the next section running on your computer, update the app's RestUrl constant.

Android emulators don't run on the local machine and use a loopback IP (10.0.2.2) to communicate with the local machine. Use Xamarin.Essentials DeviceInfo to detect what operating the system is running to use the correct URL.

Navigate to the TodoREST project and open the Constants.cs file. The `Constants.cs` file contains the following configuration.

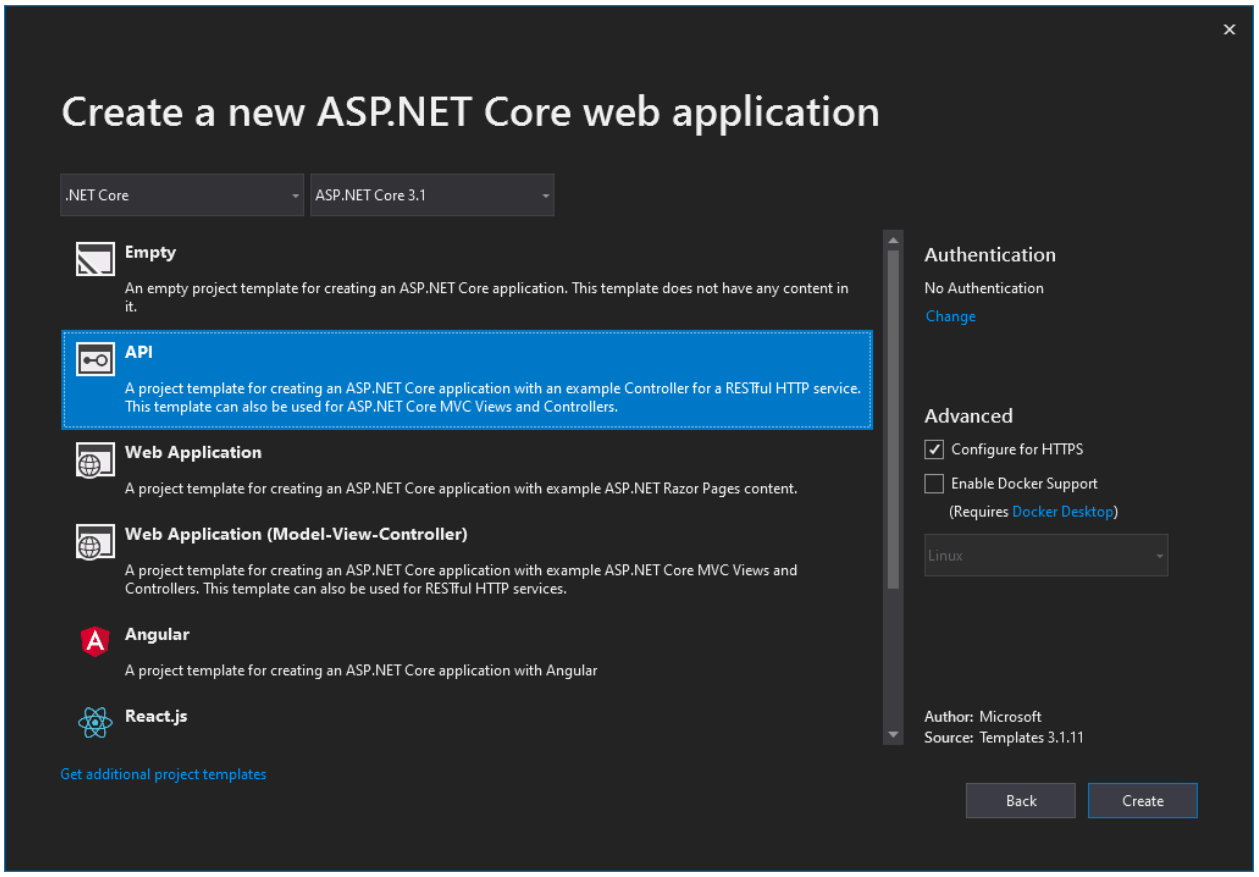
```
C#
```

```
using Xamarin.Essentials;
using Xamarin.Forms;

namespace TodoREST
{
    public static class Constants
    {
        // URL of REST service
        //public static string RestUrl =
"https://YOURPROJECT.azurewebsites.net:8081/api/todoitems/{0}";

        // URL of REST service (Android does not use localhost)
        // Use http cleartext for local deployment. Change to https for
production
        public static string RestUrl = DeviceInfo.Platform ==
DevicePlatform.Android ? "http://10.0.2.2:5000/api/todoitems/{0}" :
"http://localhost:5000/api/todoitems/{0}";
    }
}
```

You can optionally deploy the web service to a cloud service such as Azure and update the `RestUrl`.

Creating the ASP.NET Core Project

Create a new ASP.NET Core Web Application in Visual Studio. Choose the Web API template. Name the project *TodoAPI*.



The app should respond to all requests made to port 5000 including clear-text HTTP traffic for our mobile client. Update `Startup.cs` so UseHttpsRedirection doesn't run in development:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // For mobile apps, allow http traffic.
        app.UseHttpsRedirection();
    }

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

> **Note**
>
> Run the app directly, rather than behind IIS Express. IIS Express ignores non-local requests by default. Run **dotnet run** from a command prompt, or choose the app name profile from the Debug Target dropdown in the Visual Studio toolbar.

Add a model class to represent To-Do items. Mark required fields with the `[Required]` attribute:

C#

```
using System.ComponentModel.DataAnnotations;

namespace TodoAPI.Models
{
    public class TodoItem
    {
        [Required]
        public string ID { get; set; }

        [Required]
        public string Name { get; set; }

        [Required]
        public string Notes { get; set; }

        public bool Done { get; set; }
    }
}
```

The API methods require some way to work with data. Use the same `ITodoRepository` interface the original Xamarin sample uses:

C#

```
using System.Collections.Generic;
using TodoAPI.Models;

namespace TodoAPI.Interfaces
{
    public interface ITodoRepository
    {
        bool DoesItemExist(string id);
        IEnumerable<TodoItem> All { get; }
        TodoItem Find(string id);
        void Insert(TodoItem item);
        void Update(TodoItem item);
        void Delete(string id);
```

```
    }
}
```

For this sample, the implementation just uses a private collection of items:

C#

```
using System.Collections.Generic;
using System.Linq;
using TodoAPI.Interfaces;
using TodoAPI.Models;

namespace TodoAPI.Services
{
    public class TodoRepository : ITodoRepository
    {
        private List<TodoItem> _todoList;

        public TodoRepository()
        {
            InitializeData();
        }

        public IEnumerable<TodoItem> All
        {
            get { return _todoList; }
        }

        public bool DoesItemExist(string id)
        {
            return _todoList.Any(item => item.ID == id);
        }

        public TodoItem Find(string id)
        {
            return _todoList.FirstOrDefault(item => item.ID == id);
        }

        public void Insert(TodoItem item)
        {
            _todoList.Add(item);
        }

        public void Update(TodoItem item)
        {
            var todoItem = this.Find(item.ID);
            var index = _todoList.IndexOf(todoItem);
            _todoList.RemoveAt(index);
            _todoList.Insert(index, item);
        }

        public void Delete(string id)
        {
```

```
            _todoList.Remove(this.Find(id));
        }

        private void InitializeData()
        {
            _todoList = new List<TodoItem>();

            var todoItem1 = new TodoItem
            {
                ID = "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
                Name = "Learn app development",
                Notes = "Take Microsoft Learn Courses",
                Done = true
            };

            var todoItem2 = new TodoItem
            {
                ID = "b94afb54-a1cb-4313-8af3-b7511551b33b",
                Name = "Develop apps",
                Notes = "Use Visual Studio and Visual Studio for Mac",
                Done = false
            };

            var todoItem3 = new TodoItem
            {
                ID = "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
                Name = "Publish apps",
                Notes = "All app stores",
                Done = false,
            };

            _todoList.Add(todoItem1);
            _todoList.Add(todoItem2);
            _todoList.Add(todoItem3);
        }
    }
}
```

Configure the implementation in `Startup.cs`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<ITodoRepository, TodoRepository>();
    services.AddControllers();
}
```

Creating the Controller

Add a new controller to the project, TodoItemsController. It should inherit from ControllerBase. Add a `Route` attribute to indicate that the controller handles requests made to paths starting with `api/todoitems`. The `[controller]` token in the route is replaced by the name of the controller (omitting the `Controller` suffix), and is especially helpful for global routes. Learn more about routing.

The controller requires an `ITodoRepository` to function; request an instance of this type through the controller's constructor. At runtime, this instance is provided using the framework's support for dependency injection.

C#

```
[ApiController]
[Route("api/[controller]")]
public class TodoItemsController : ControllerBase
{
    private readonly ITodoRepository _todoRepository;

    public TodoItemsController(ITodoRepository todoRepository)
    {
        _todoRepository = todoRepository;
    }
```

This API supports four different HTTP verbs to perform CRUD (Create, Read, Update, Delete) operations on the data source. The simplest of these is the Read operation, which corresponds to an HTTP `GET` request.

Test the API using curl

You can test the API method using a variety of tools. For this tutorial the following open source command-line tools are used:

- curl: Transfers data using various protocols including HTTP and HTTPS. curl is used in this tutorial to call the API using HTTP methods `GET`, `POST`, `PUT`, and `DELETE`.
- jq: A JSON processor used in this tutorial to format JSON data so that it's easy to read from the API response.

Install curl and jq

Windows

curl is installed with Windows 10, version 1802 or higher. For more information on installing curl, see the Official curl website.

Install jq with the following command in PowerShell or the Command Prompt:

PowerShell

```
winget install jqlang.jq
```

The `jq` command is available once the PowerShell or Command Prompt is closed and restarted.

For more details on jq installation, see jq.

Reading Items

Requesting a list of items is done with a GET request to the `List` method. The `[HttpGet]` attribute on the `List` method indicates that this action should only handle GET requests. The route for this action is the route specified on the controller. You don't necessarily need to use the action name as part of the route. You just need to ensure each action has a unique and unambiguous route. Routing attributes can be applied at both the controller and method levels to build up specific routes.

C#

```
[HttpGet]
public IActionResult List()
{
    return Ok(_todoRepository.All);
}
```

Windows

> **Note**
>
> Windows PowerShell 5.1 recognizes `curl` as an alias to `Invoke-WebRequest`. To use `curl.exe` instead, type the `&` operator followed by the full path to `curl.exe`. Find the full path to `curl.exe` by typing `where curl` in the Command Prompt. For example, if the full path to `curl.exe` is `C:\Windows\System32\curl.exe`. Then instead of typing the command `curl --`

`help`, use `& 'C:\Windows\System32\curl.exe' --help`. PowerShell 7 uses `curl` as the command for `curl.exe` and so a full path is not required.

In PowerShell, call the following curl command:

PowerShell

```
curl -v -X GET 'http://localhost:5000/api/todoitems/' | jq
```

The previous curl command includes the following components:

- `-v`: Activates verbose mode, providing detailed information about the HTTP response and is useful for API testing and troubleshooting.
- `-X GET`: Specifies the use of the HTTP `GET` method for the request. While curl can often infer the intended HTTP method, this option makes it explicit.
- `'http://localhost:5000/api/todoitems/'`: This is the request's target URL. In this instance, it's a REST API endpoint.
- `| jq`: This segment isn't related to curl directly. The pipe `|` is a shell operator that takes the output from the command on its left and "pipes" it to the command on its right. `jq` is a command-line JSON processor. While not required, `jq` makes the returned JSON data easier to read.

The `List` method returns a 200 OK response code and all of the Todo items, serialized as JSON:

Output


```
[
  {
    "id": "6bb8a868-dba1-4f1a-93b7-24ebce87e243",
    "name": "Learn app development",
    "notes": "Take Microsoft Learn Courses",
    "done": true
  },
  {
    "id": "b94afb54-a1cb-4313-8af3-b7511551b33b",
    "name": "Develop apps",
    "notes": "Use Visual Studio and Visual Studio for Mac",
    "done": false
  },
  {
    "id": "ecfa6f80-3671-4911-aabe-63cc442c1ecf",
    "name": "Publish apps",
    "notes": "All app stores",
    "done": false
```

```
  }
]
```

Creating Items

By convention, creating new data items is mapped to the HTTP `POST` verb. The `Create` method has an `[HttpPost]` attribute applied to it and accepts a `TodoItem` instance. Since the `item` argument is passed in the body of the POST, this parameter specifies the `[FromBody]` attribute.

Inside the method, the item is checked for validity and prior existence in the data store, and if no issues occur, it's added using the repository. Checking `ModelState.IsValid` performs model validation, and should be done in every API method that accepts user input.

C#

```
[HttpPost]
public IActionResult Create([FromBody]TodoItem item)
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return
BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        bool itemExists = _todoRepository.DoesItemExist(item.ID);
        if (itemExists)
        {
            return StatusCode(StatusCodes.Status409Conflict,
ErrorCode.TodoItemIDInUse.ToString());
        }
        _todoRepository.Insert(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotCreateItem.ToString());
    }
    return Ok(item);
}
```

The sample uses an `enum` containing error codes that are passed to the mobile client:

C#

```
public enum ErrorCode
{
    TodoItemNameAndNotesRequired,
    TodoItemIDInUse,
    RecordNotFound,
    CouldNotCreateItem,
    CouldNotUpdateItem,
    CouldNotDeleteItem
}
```

In the terminal, test adding new items by calling the following curl command using the `POST` verb and providing the new object in JSON format in the Body of the request.

Windows

PowerShell

```
curl -v -X POST 'http://localhost:5000/api/todoitems/' `
--header 'Content-Type: application/json' `
--data '{
  "id": "6bb8b868-dba1-4f1a-93b7-24ebce87e243",
  "name": "A Test Item",
  "notes": "asdf",
  "done": false
}' | jq
```

The previous curl command includes the following options:

- `--header 'Content-Type: application/json'`: Sets the `Content-Type` header to `application/json`, indicating that the request body contains JSON data.
- `--data '{...}'`: Sends the specified data in the request body.

The method returns the newly created item in the response.

Updating Items

Modifying records is done using HTTP `PUT` requests. Other than this change, the `Edit` method is almost identical to `Create`. If the record isn't found, the `Edit` action returns a `NotFound` (404) response.

C#

```
[HttpPut]
public IActionResult Edit([FromBody] TodoItem item)
```

```
{
    try
    {
        if (item == null || !ModelState.IsValid)
        {
            return
BadRequest(ErrorCode.TodoItemNameAndNotesRequired.ToString());
        }
        var existingItem = _todoRepository.Find(item.ID);
        if (existingItem == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Update(item);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotUpdateItem.ToString());
    }
    return NoContent();
}
```

To test with curl, change the verb to `PUT`. Specify the updated object data in the Body of the request.

Windows

PowerShell

```
curl -v -X PUT 'http://localhost:5000/api/todoitems/' `
--header 'Content-Type: application/json' `
--data '{
  "id": "6bb8b868-dba1-4f1a-93b7-24ebce87e243",
  "name": "A Test Item",
  "notes": "asdf",
  "done": true
}' | jq
```

This method returns a `NoContent` (204) response when successful, for consistency with the pre-existing API.

Deleting Items

Deleting records is accomplished by making `DELETE` requests to the service, and passing the ID of the item to be deleted. As with updates, requests for items that don't exist receive `NotFound` responses. Otherwise, a successful request returns a `NoContent` (204) response.

C#

```
[HttpDelete("{id}")]
public IActionResult Delete(string id)
{
    try
    {
        var item = _todoRepository.Find(id);
        if (item == null)
        {
            return NotFound(ErrorCode.RecordNotFound.ToString());
        }
        _todoRepository.Delete(id);
    }
    catch (Exception)
    {
        return BadRequest(ErrorCode.CouldNotDeleteItem.ToString());
    }
    return NoContent();
}
```

Test with curl by changing the HTTP verb to `DELETE` and appending the ID of the data object to delete at the end of the URL. Nothing is required in the Body of the request.

Windows

PowerShell

```
curl -v -X DELETE 'http://localhost:5000/api/todoitems/6bb8b868-dba1-4f1a-93b7-24ebce87e243'
```

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients aren't supposed to view.
- Omit some properties to reduce payload size.

- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, see Prevent over-posting

Common Web API Conventions

As you develop the backend services for your app, you'll want to come up with a consistent set of conventions or policies for handling cross-cutting concerns. For example, in the service shown previously, requests for specific records that weren't found received a `NotFound` response, rather than a `BadRequest` response. Similarly, commands made to this service that passed in model bound types always checked `ModelState.IsValid` and returned a `BadRequest` for invalid model types.

Once you've identified a common policy for your APIs, you can usually encapsulate it in a filter. Learn more about how to encapsulate common API policies in ASP.NET Core MVC applications.

Additional resources

- Xamarin.Forms: Web Service Authentication
- Xamarin.Forms: Consume a RESTful Web Service
- Consume REST web services in Xamarin Apps
- Create a web API with ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Publish an ASP.NET Core web API to Azure API Management with Visual Studio

Article • 11/04/2022

By Matt Soucoup

In this tutorial you'll learn how to create an ASP.NET Core web API project using Visual Studio, ensure it has OpenAPI support, and then publish the web API to both Azure App Service and Azure API Management.

Set up

To complete the tutorial you'll need an Azure account.

- Open a free Azure account if you don't have one.

Create an ASP.NET Core web API

Visual Studio allows you to easily create a new ASP.NET Core web API project from a template. Follow these directions to create a new ASP.NET Core web API project:

- From the File menu, select **New** > **Project**.
- Enter **Web API** in the search box.
- Select the **ASP.NET Core Web API** template and select **Next**.
- In the **Configure your new project dialog**, name the project **WeatherAPI** and select **Next**.
- In the **Additional information** dialog:
- Confirm the Framework is **.NET 6.0 (Long-term support)**.
- Confirm the checkbox for **Use controllers (uncheck to use minimal APIs)** is checked.
- Confirm the checkbox for **Enable OpenAPI support** is checked.
- Select **Create**.

Explore the code

Swagger definitions allow Azure API Management to read the app's API definitions. By checking the **Enable OpenAPI support** checkbox during app creation, Visual Studio

automatically adds the code to create the Swagger definitions. Open up the `Program.cs` file which shows the following code:

```
C#


...

builder.Services.AddSwaggerGen();

...

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(); // Protected by if (env.IsDevelopment())
}

...
```

Ensure the Swagger definitions are always generated

Azure API Management needs the Swagger definitions to always be present, regardless of the application's environment. To ensure they are always generated, move `app.UseSwagger();` outside of the `if (app.Environment.IsDevelopment())` block.

The updated code:

```
C#


...

app.UseSwagger();

if (app.Environment.IsDevelopment())
{
    app.UseSwaggerUI();
}

...
```

Change the API routing

Change the URL structure needed to access the `Get` action of the `WeatherForecastController`. Complete the following steps:

1. Open the `WeatherForecastController.cs` file.
2. Replace the `[Route("[controller]")]` class-level attribute with `[Route("/")]`. The updated class definition :

```
C#

[ApiController]
[Route("/")]
public class WeatherForecastController : ControllerBase
```

Publish the web API to Azure App Service

Complete the following steps to publish the ASP.NET Core web API to Azure API Management:

1. Publish the API app to Azure App Service.
2. Publish the ASP.NET Core web API app to the Azure API Management service instance.

Publish the API app to Azure App Service

Complete the following steps to publish the ASP.NET Core web API to Azure API Management:

1. In **Solution Explorer**, right-click the project and select **Publish**.
2. In the **Publish** dialog, select **Azure** and select the **Next** button.
3. Select **Azure App Service (Windows)** and select the **Next** button.
4. Select **Create a new Azure App Service**.

   The **Create App Service** dialog appears. The **App Name**, **Resource Group**, and **App Service Plan** entry fields are populated. You can keep these names or change them.
5. Select the **Create** button.
6. Once the app service is created, select the **Next** button.

7. Select **Create a new API Management Service**.

   The **Create API Management Service** dialog appears. You can leave the **API Name**, **Subscription Name**, and **Resource Group** entry fields as they are. Select the **new** button next to the **API Management Service** entry and enter the required fields from that dialog box.

   Select the **OK** button to create the API Management service.

8. Select the **Create** button to proceed with the API Management service creation. This step may take several minutes to complete.

9. When that completes, select the **Finish** button.

10. The dialog closes and a summary screen appears with information about the publish. Select the **Publish** button.

    The web API publishes to both Azure App Service and Azure API Management. A new browser window will appear and show the API running in Azure App Service. You can close that window.

11. Open up the Azure portal in a web browser and navigate to the API Management instance you created.

12. Select the **APIs** option from the left-hand menu.

13. Select the API you created in the preceding steps. It's now populated and you can explore around.

Configure the published API name

Notice the name of the API is named *WeatherAPI*; however, we would like to call it *Weather Forecasts*. Complete the following steps to update the name:

1. Add the following to `Program.cs` immediately after `servies.AddSwaggerGen();`

```
C#
```

```
builder.Services.ConfigureSwaggerGen(setup =>
{
    setup.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo
    {
        Title = "Weather Forecasts",
        Version = "v1"
    });
});
```

2. Republish the ASP.NET Core web API and open the Azure API Management instance in the Azure portal.

3. Refresh the page in your browser. You'll see the name of the API is now correct.

Verify the web API is working

You can test the deployed ASP.NET Core web API in Azure API Management from the Azure portal with the following steps:

1. Open the **Test** tab.
2. Select **/** or the **Get** operation.
3. Select **Send**.

Clean up

When you've finished testing the app, go to the Azure portal and delete the app.

1. Select **Resource groups**, then select the resource group you created.

2. In the **Resource groups** page, select **Delete**.

3. Enter the name of the resource group and select **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure.

Additional resources

- Azure API Management
- Azure App Service

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Create a minimal API with ASP.NET Core

Article • 08/21/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Rick Anderson and Tom Dykstra

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They're ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core.

This tutorial teaches the basics of building a minimal API with ASP.NET Core. Another approach to creating APIs in ASP.NET Core is to use controllers. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating an API project based on controllers that contains more features, see Create a web API.

Overview

This tutorial creates the following API:

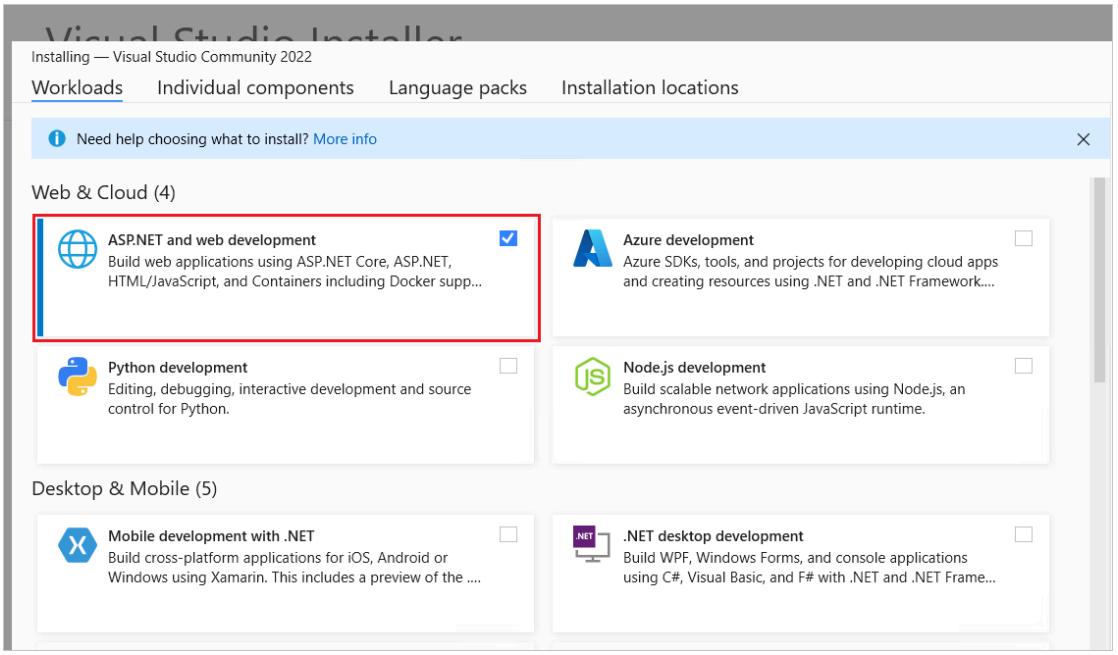
Expand table

API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item
`POST /todoitems`	Add a new item	To-do item	To-do item
`PUT /todoitems/{id}`	Update an existing item	To-do item	None
`DELETE /todoitems/{id}`	Delete an item	None	None

Prerequisites

Visual Studio

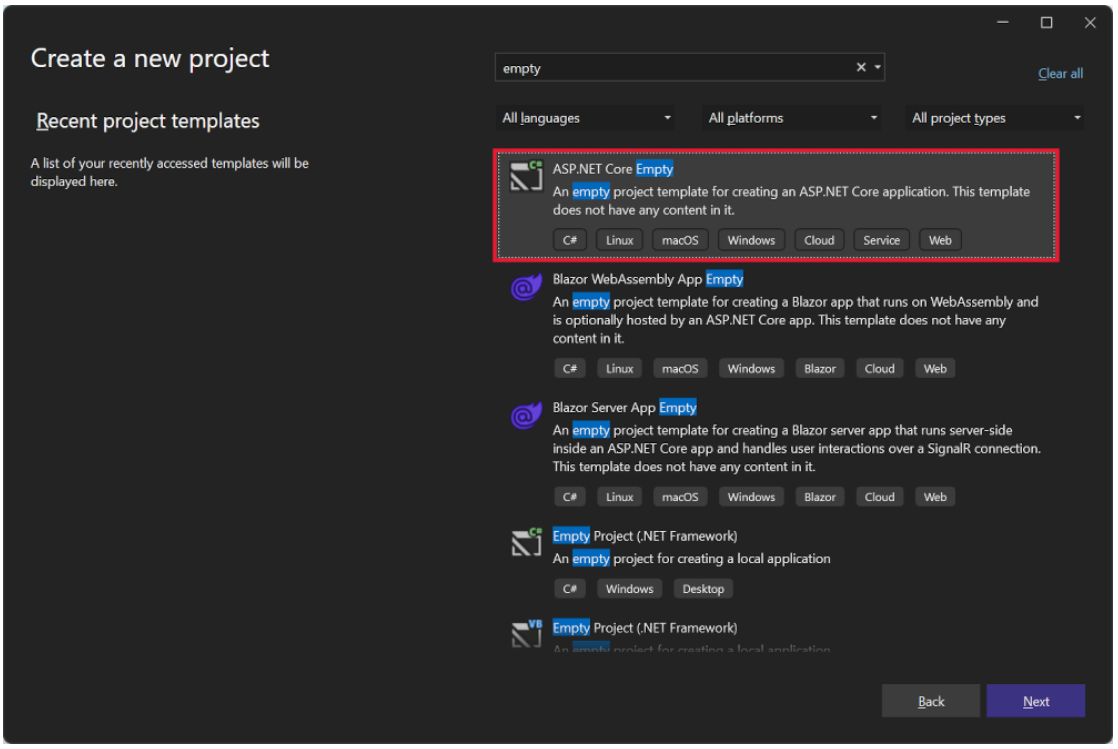
- Visual Studio 2022 with the **ASP.NET and web development** workload.



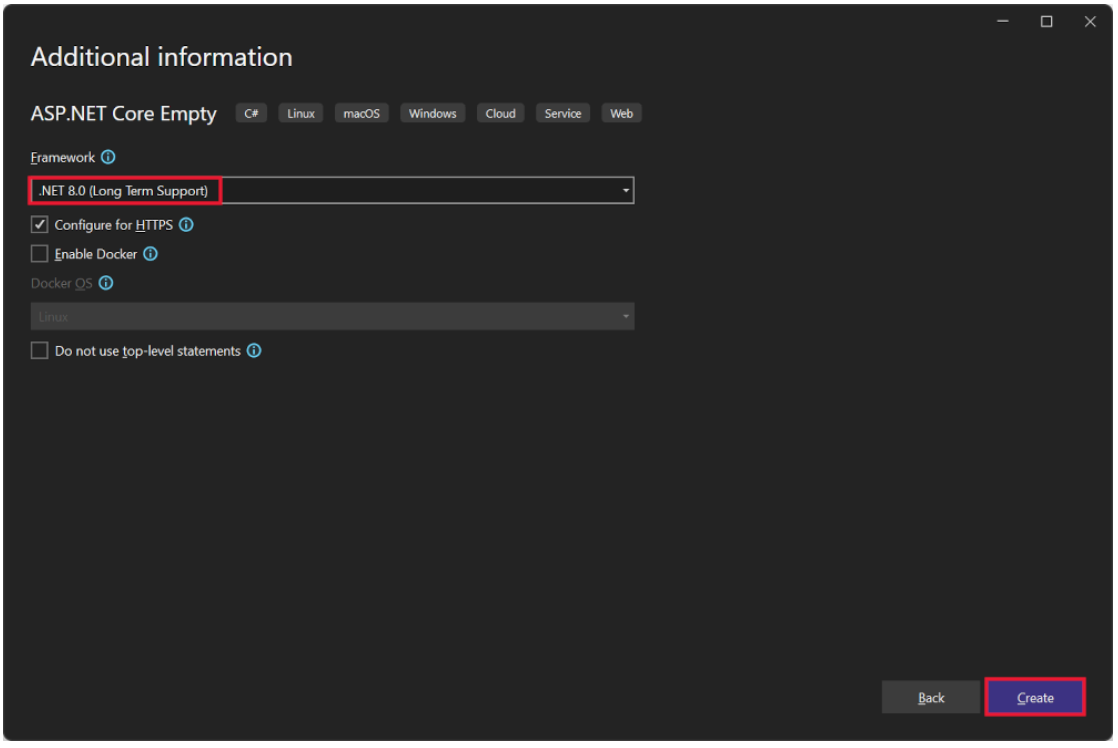
Create an API project

Visual Studio

- Start Visual Studio 2022 and select **Create a new project**.
- In the **Create a new project** dialog:
  - Enter `Empty` in the **Search for templates** search box.
  - Select the **ASP.NET Core Empty** template and select **Next**.



- Name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 8.0 (Long Term Support)**
  - Uncheck **Do not use top-level statements**
  - Select **Create**



Examine the code

The `Program.cs` file contains the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding code:

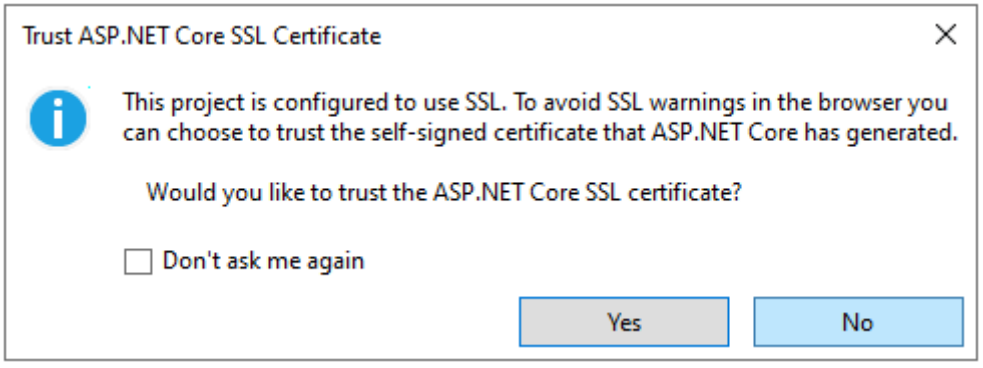
- Creates a WebApplicationBuilder and a WebApplication with preconfigured defaults.
- Creates an HTTP GET endpoint `/` that returns `Hello World!`:

Run the app

Visual Studio

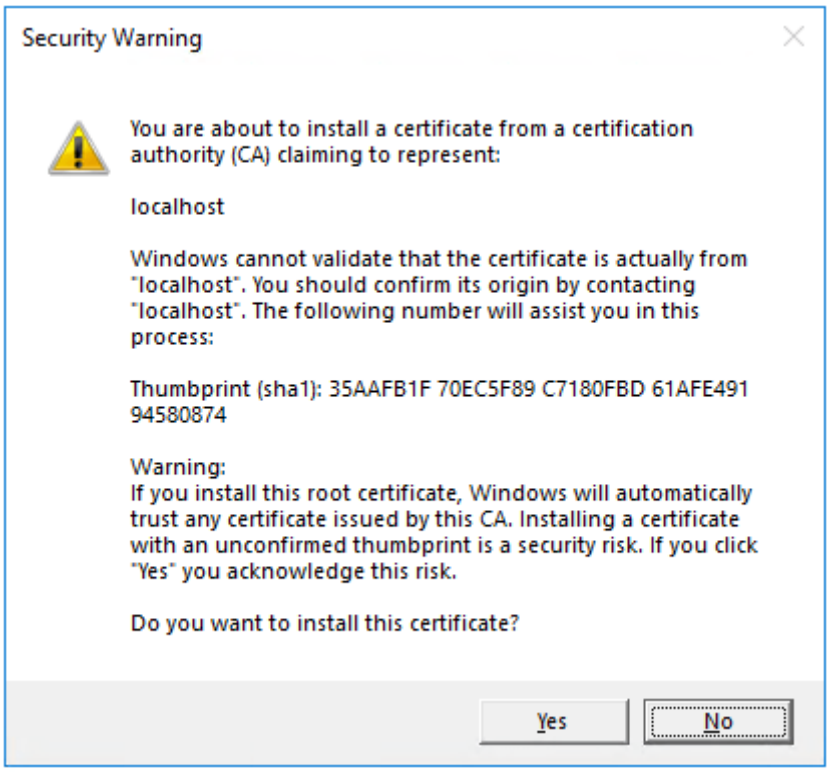
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the Kestrel web server and opens a browser window.

`Hello World!` is displayed in the browser. The `Program.cs` file contains a minimal but complete app.

Close the browser window.

Add NuGet packages

NuGet packages must be added to support the database and diagnostics used in this tutorial.

Visual Studio

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab.
- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select `Microsoft.EntityFrameworkCore.InMemory`.
- Select the **Project** checkbox in the right pane and then select **Install**.

- Follow the preceding instructions to add the `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` package.

The model and database context classes

- In the project folder, create a file named `Todo.cs` with the following code:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The preceding code creates the model for this app. A *model* is a class that represents data that the app manages.

- Create a file named `TodoDb.cs` with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

class TodoDb : DbContext
{
    public TodoDb(DbContextOptions<TodoDb> options)
        : base(options) { }

    public DbSet<Todo> Todos => Set<Todo>();
}
```

The preceding code defines the *database context*, which is the main class that coordinates Entity Framework functionality for a data model. This class derives from the Microsoft.EntityFrameworkCore.DbContext class.

Add the API code

- Replace the contents of the `Program.cs` file with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

```
app.Run();
```

The following highlighted code adds the database context to the dependency injection (DI) container and enables displaying database-related exceptions:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();
```

The DI container provides access to the database context and other services.

Visual Studio

This tutorial uses Endpoints Explorer and .http files to test the API.

Test posting data

The following code in `Program.cs` creates an HTTP POST endpoint `/todoitems` that adds data to the in-memory database:

C#

```
app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});
```

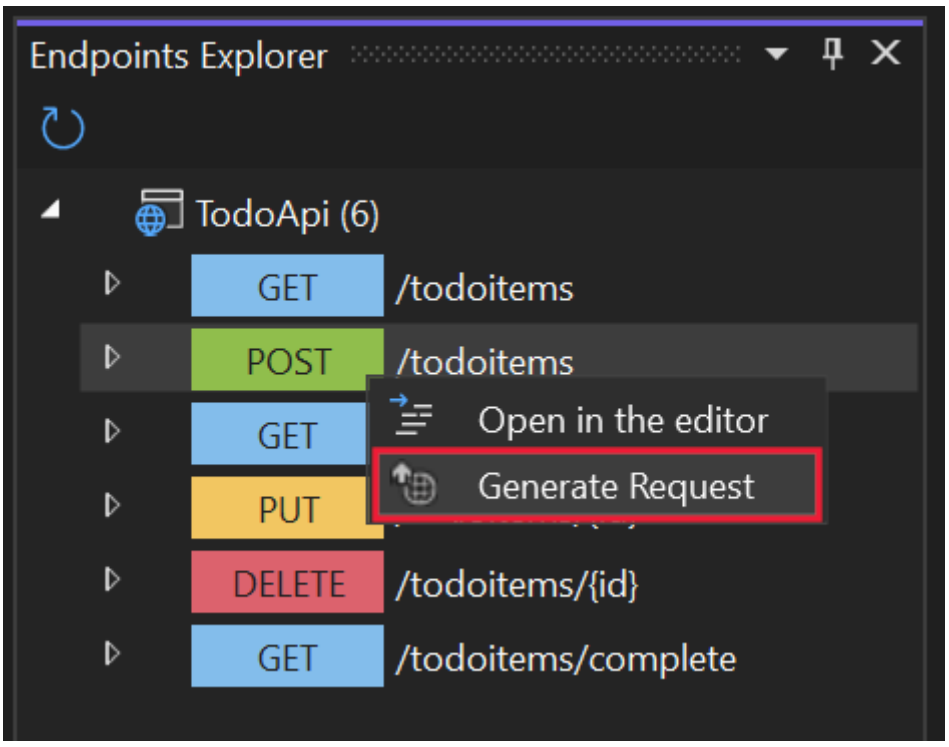
Run the app. The browser displays a 404 error because there's no longer a `/` endpoint.

The POST endpoint will be used to add data to the app.

Visual Studio

- Select **View** > **Other Windows** > **Endpoints Explorer**.

- Right-click the **POST** endpoint and select **Generate request**.



A new file is created in the project folder named `TodoApi.http`, with contents similar to the following example:

```
@TodoApi_HostAddress = https://localhost:7031

Post {{TodoApi_HostAddress}}/todoitems

###
```

  - The first line creates a variable that is used for all of the endpoints.
  - The next line defines a POST request.
  - The triple hashtag (`###`) line is a request delimiter: what comes after it is for a different request.

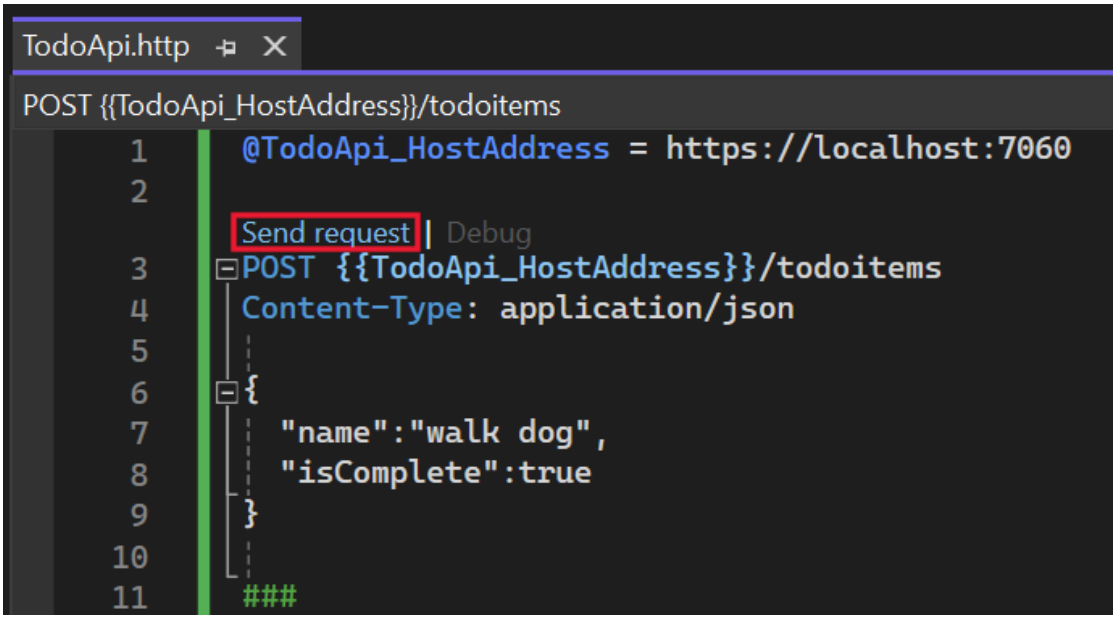
- The POST request needs headers and a body. To define those parts of the request, add the following lines immediately after the POST request line:

```
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}
```

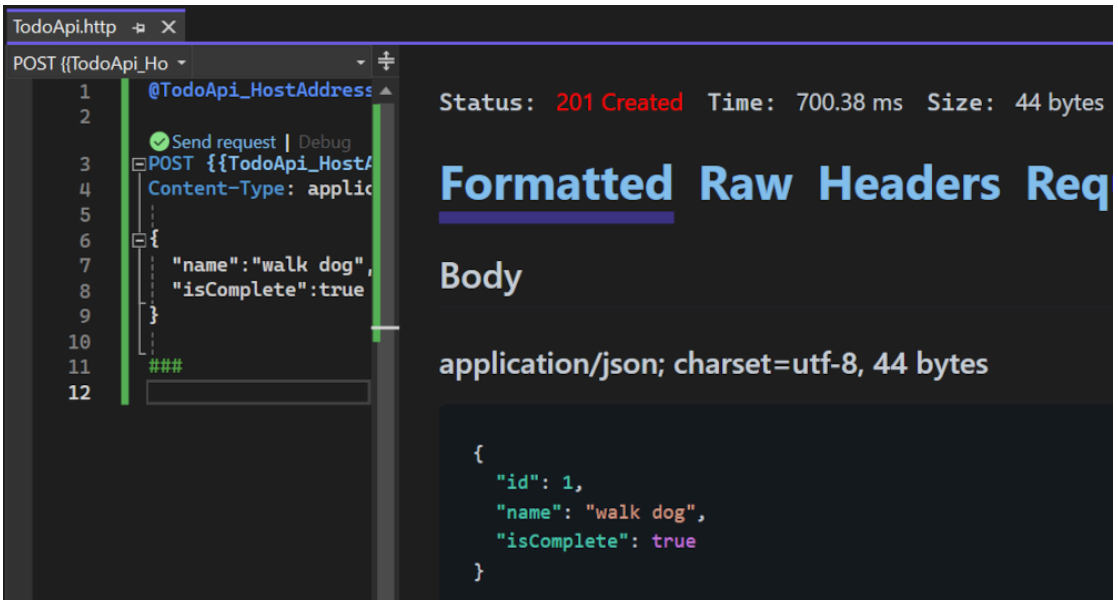
The preceding code adds a Content-Type header and a JSON request body. The TodoApi.http file should now look like the following example, but with your port number:

```
@TodoApi_HostAddress = https://localhost:7057

Post {{TodoApi_HostAddress}}/todoitems
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}

###
```

- Run the app.
- Select the **Send request** link that is above the `POST` request line.



The POST request is sent to the app and the response is displayed in the **Response** pane.



Examine the GET endpoints

The sample app implements several GET endpoints by calling `MapGet`:

Expand table

API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get all completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item

C#

```
app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

Test the GET endpoints

Visual Studio

Test the app by calling the `GET` endpoints from a browser or by using **Endpoints Explorer**. The following steps are for **Endpoints Explorer**.

- In **Endpoints Explorer**, right-click the first **GET** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Get {{TodoApi_HostAddress}}/todoitems

  ###
  ```

- Select the **Send request** link that is above the new `GET` request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  JSON

  ```
  [
    {
      "id": 1,
      "name": "walk dog",
      "isComplete": true
    }
  ]
  ```

- In **Endpoints Explorer**, right-click the `/todoitems/{id}` **GET** endpoint and select **Generate request**. The following content is added to the `TodoApi.http` file:

  ```
  GET {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- Replace `{id}` with `1`.

- Select the **Send request** link that is above the new GET request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  ```
  JSON

  {
    "id": 1,
    "name": "walk dog",
    "isComplete": true
  }
  ```

This app uses an in-memory database. If the app is restarted, the GET request doesn't return any data. If no data is returned, POST data to the app and try the GET request again.

Return values

ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

The return types can represent a wide range of HTTP status codes. For example, `GET /todoitems/{id}` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

Examine the PUT endpoint

The sample app implements a single PUT endpoint using `MapPut`:

```
C#

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
```

```
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});
```

This method is similar to the `MapPost` method, except it uses HTTP PUT. A successful response returns 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PUT endpoint

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has `Id = 1` and set its name to `"feed fish"`.

Visual Studio

- In **Endpoints Explorer**, right-click the **PUT** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Put {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- In the PUT request line, replace `{id}` with `1`.
- Add the following lines immediately after the PUT request line:

  ```
  ```

```
Content-Type: application/json

{
  "name": "feed fish",
  "isComplete": false
}
```

The preceding code adds a Content-Type header and a JSON request body.

- Select the **Send request** link that is above the new PUT request line.

  The PUT request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Examine and test the DELETE endpoint

The sample app implements a single DELETE endpoint using `MapDelete`:

C#

```
app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

Visual Studio

- In **Endpoints Explorer**, right-click the **DELETE** endpoint and select **Generate request**.

  A DELETE request is added to `TodoApi.http`.

- Replace `{id}` in the DELETE request line with `1`. The DELETE request should look like the following example:

  ```
  
  ```

```
DELETE {{TodoApi_HostAddress}}/todoitems/1

###
```

- Select the **Send request** link for the DELETE request.

  The DELETE request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Use the MapGroup API

The sample app code repeats the `todoitems` URL prefix each time it sets up an endpoint. APIs often have groups of endpoints with a common URL prefix, and the MapGroup method is available to help organize such groups. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata.

Replace the contents of `Program.cs` with the following code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", async (TodoDb db) =>
    await db.Todos.ToListAsync());

todoItems.MapGet("/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

todoItems.MapGet("/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

```
todoItems.MapPost("/", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

todoItems.MapPut("/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

todoItems.MapDelete("/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});

app.Run();
```

The preceding code has the following changes:

- Adds `var todoItems = app.MapGroup("/todoitems");` to set up the group using the URL prefix `/todoitems`.
- Changes all the `app.Map<HttpVerb>` methods to `todoItems.Map<HttpVerb>`.
- Removes the URL prefix `/todoitems` from the `Map<HttpVerb>` method calls.

Test the endpoints to verify that they work the same.

Use the TypedResults API

Returning TypedResults rather than Results has several advantages, including testability and automatically returning the response type metadata for OpenAPI to describe the

endpoint. For more information, see TypedResults vs Results.

The `Map<HttpVerb>` methods can call route handler methods instead of using lambdas. To see an example, update *Program.cs* with the following code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);

app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();
```

```
    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

The `Map<HttpVerb>` code now calls methods instead of lambdas:

C#

```
var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

These methods return objects that implement IResult and are defined by TypedResults:

C#

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

```
static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Unit tests can call these methods and test that they return the correct type. For example, if the method is `GetAllTodos`:

```
C#
```

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

Unit test code can verify that an object of type Ok<Todo[]> is returned from the handler method. For example:

C#

```
public async Task GetAllTodos_ReturnsOkOfTodosResult()
{
    // Arrange
    var db = CreateDbContext();

    // Act
    var result = await TodosApi.GetAllTodos(db);

    // Assert: Check for the correct returned type
    Assert.IsType<Ok<Todo[]>>(result);
}
```

Prevent over-posting

Currently the sample app exposes the entire `Todo` object. Production apps In production applications, a subset of the model is often used to restrict the data that can be input and returned. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO can be used to:

- Prevent over-posting.
- Hide properties that clients aren't supposed to view.
- Omit some properties to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `Todo` class to include a secret field:

C#

```
public class Todo
{
    public int Id { get; set; }
```

```
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
    public string? Secret { get; set; }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a file named `TodoItemDTO.cs` with the following code:

C#

```
public class TodoItemDTO
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    public TodoItemDTO() { }
    public TodoItemDTO(Todo todoItem) =>
    (Id, Name, IsComplete) = (todoItem.Id, todoItem.Name,
todoItem.IsComplete);
}
```

Replace the contents of the `Program.cs` file with the following code to use this DTO model:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

RouteGroupBuilder todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

```
app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Select(x => new
TodoItemDTO(x)).ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db) {
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).Select(x => new TodoItemDTO(x)).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(new TodoItemDTO(todo))
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(TodoItemDTO todoItemDTO, TodoDb
db)
{
    var todoItem = new Todo
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

    db.Todos.Add(todoItem);
    await db.SaveChangesAsync();

    todoItemDTO = new TodoItemDTO(todoItem);

    return TypedResults.Created($"/todoitems/{todoItem.Id}",
todoItemDTO);
}

static async Task<IResult> UpdateTodo(int id, TodoItemDTO todoItemDTO,
TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = todoItemDTO.Name;
    todo.IsComplete = todoItemDTO.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}
```

```
static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Verify you can post and get all fields except the secret field.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

- Configure JSON serialization options.
- Handle errors and exceptions: The developer exception page is enabled by default in the development environment for minimal API apps. For information about how to handle errors and exceptions, see Handle errors in ASP.NET Core APIs.
- For an example of testing a minimal API app, see this GitHub sample.
- OpenAPI support in minimal APIs.
- Quickstart: Publish to Azure.
- Organizing ASP.NET Core Minimal APIs.

Learn more

See Minimal APIs quick reference

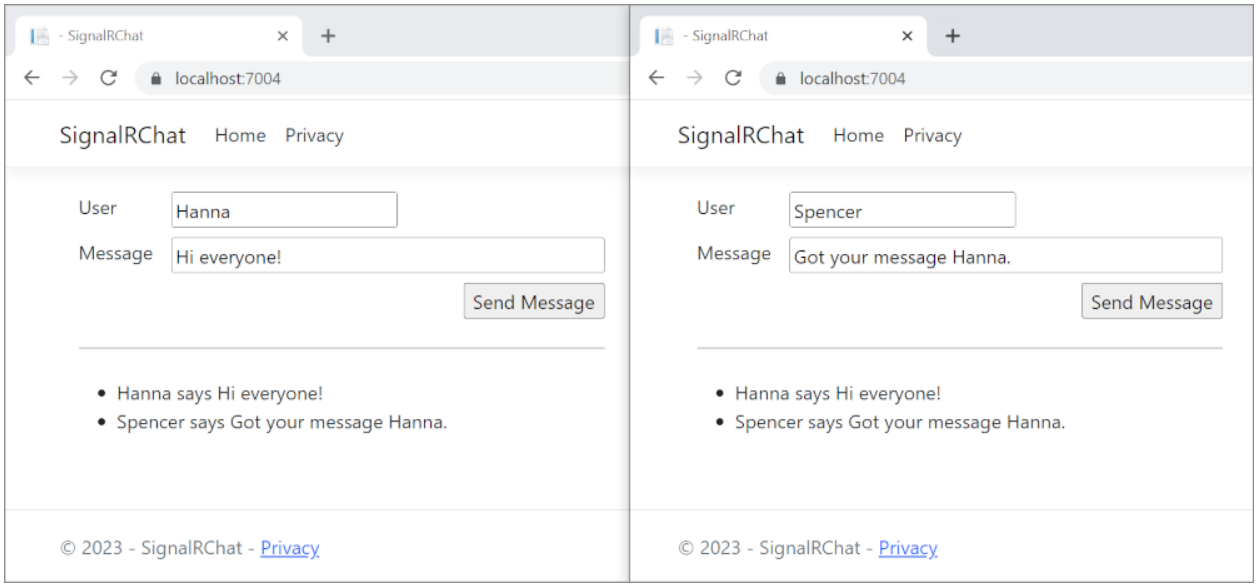
Tutorial: Get started with ASP.NET Core SignalR

Article • 11/16/2023

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- ✓ Create a web project.
- ✓ Add the SignalR client library.
- ✓ Create a SignalR hub.
- ✓ Configure the project to use SignalR.
- ✓ Add code that sends messages from any client to all connected clients.

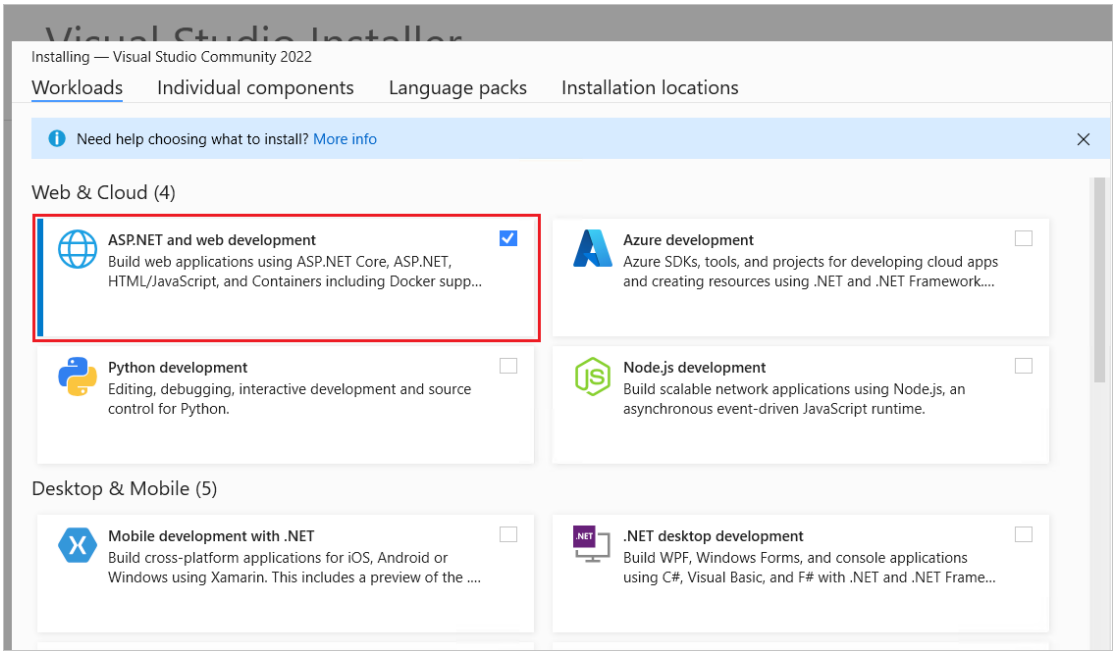
At the end, you'll have a working chat app:



Prerequisites

Visual Studio

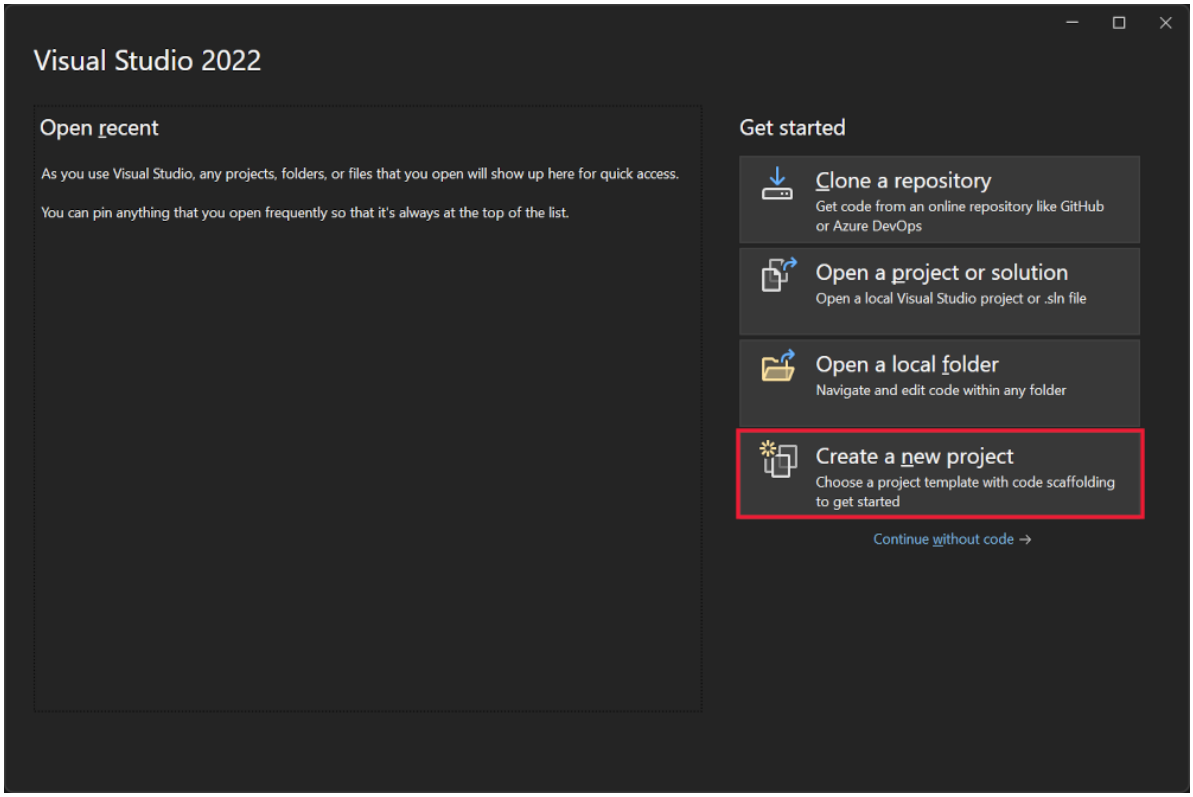
- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



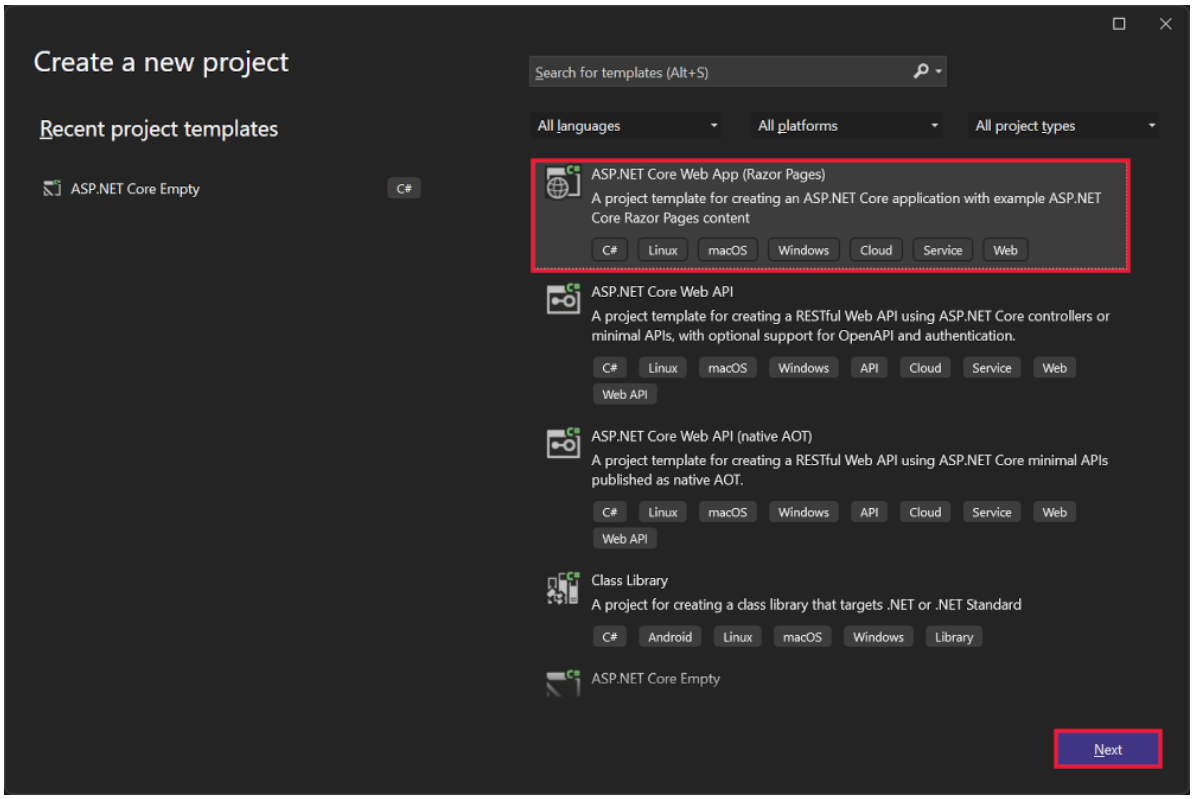
Create a web app project

Visual Studio

Start Visual Studio 2022 and select **Create a new project**.



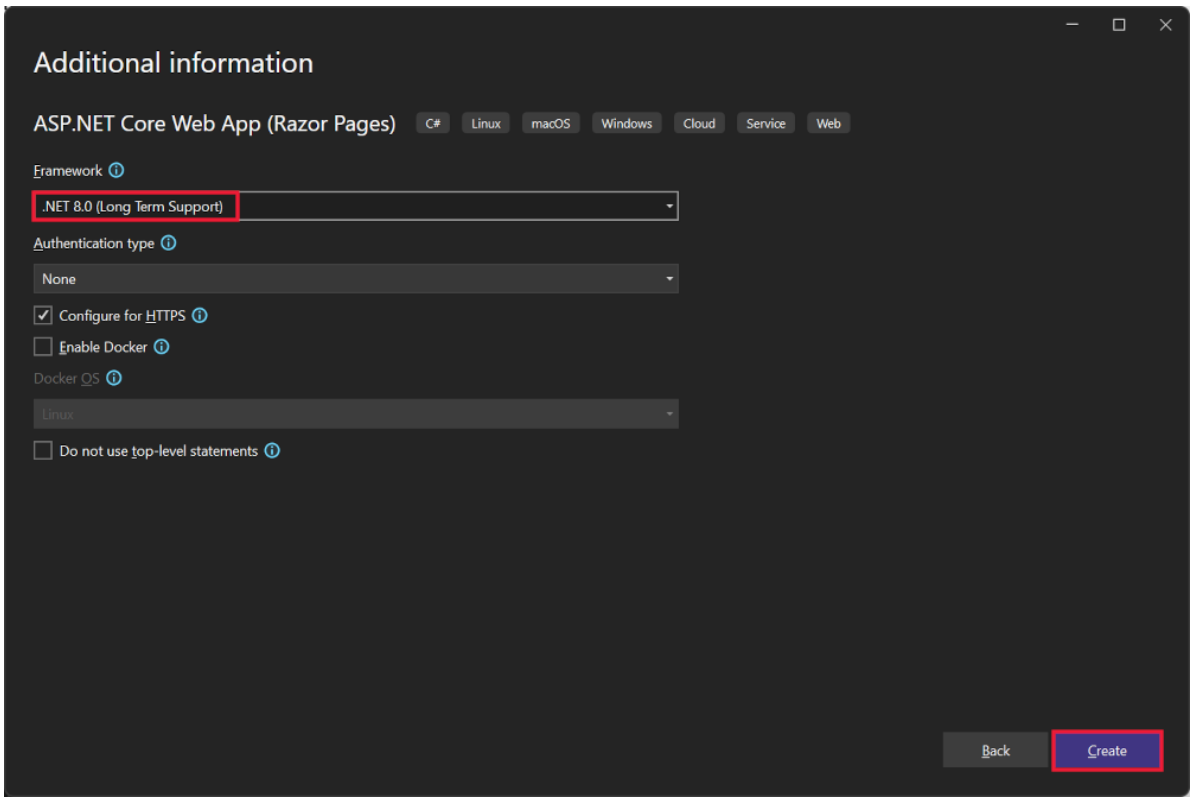
In the **Create a new project** dialog, select **ASP.NET Core Web App (Razor Pages)**, and then select **Next**.



In the **Configure your new project** dialog, enter `SignalRChat` for **Project name**. It's important to name the project `SignalRChat`, including matching the capitalization, so the namespaces match the code in the tutorial.

Select **Next**.

In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.



Add the SignalR client library

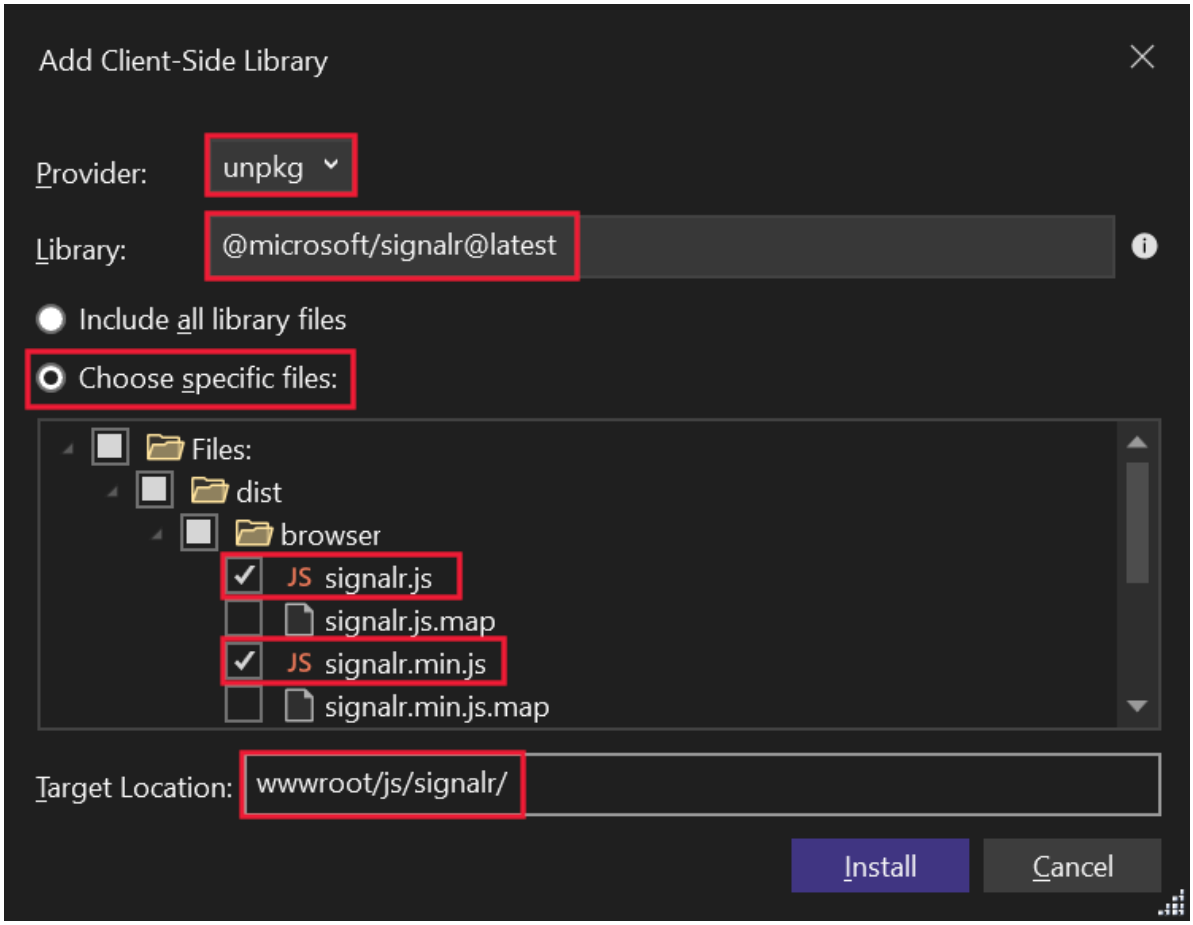
The SignalR server library is included in the ASP.NET Core shared framework. The JavaScript client library isn't automatically included in the project. For this tutorial, use Library Manager (LibMan) to get the client library from unpkg. `unpkg` is a fast, global content delivery network for everything on npm.

Visual Studio

In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.

In the **Add Client-Side Library** dialog:

- Select **unpkg** for **Provider**
- Enter `@microsoft/signalr@latest` for **Library**.
- Select **Choose specific files**, expand the *dist/browser* folder, and select `signalr.js` and `signalr.min.js`.
- Set **Target Location** to `wwwroot/js/signalr/`.
- Select **Install**.



LibMan creates a `wwwroot/js/signalr` folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

In the SignalRChat project folder, create a `Hubs` folder.

In the `Hubs` folder, create the `ChatHub` class with the following code:

C#

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

The `ChatHub` class inherits from the SignalR Hub class. The `Hub` class manages connections, groups, and messaging.

The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR. Add the following highlighted code to the `Program.cs` file.

C#

```
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();
```

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");

app.Run();
```

The preceding highlighted code adds SignalR to the ASP.NET Core dependency injection and routing systems.

Add SignalR client code

Replace the content in `Pages/Index.cshtml` with the following code:

CSHTML

```
@page
<div class="container">
    <div class="row p-1">
        <div class="col-1">User</div>
        <div class="col-5"><input type="text" id="userInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-1">Message</div>
        <div class="col-5"><input type="text" class="w-100"
id="messageInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-6 text-end">
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <hr />
        </div>
```

```
    </div>
    <div class="row p-1">
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

The preceding markup:

- Creates text boxes and a submit button.
- Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
- Includes script references to SignalR and the `chat.js` app code is created in the next step.

In the `wwwroot/js` folder, create a `chat.js` file with the following code:

JavaScript

```
"use strict";

var connection = new
signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable the send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var li = document.createElement("li");
    document.getElementById("messagesList").appendChild(li);
    // We can assign user-supplied strings to an element's textContent
because it
    // is not interpreted as markup. If you're assigning in any other way,
you
    // should be aware of possible script injection concerns.
    li.textContent = `${user} says ${message}`;
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function
(event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
```

```
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding JavaScript:

- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

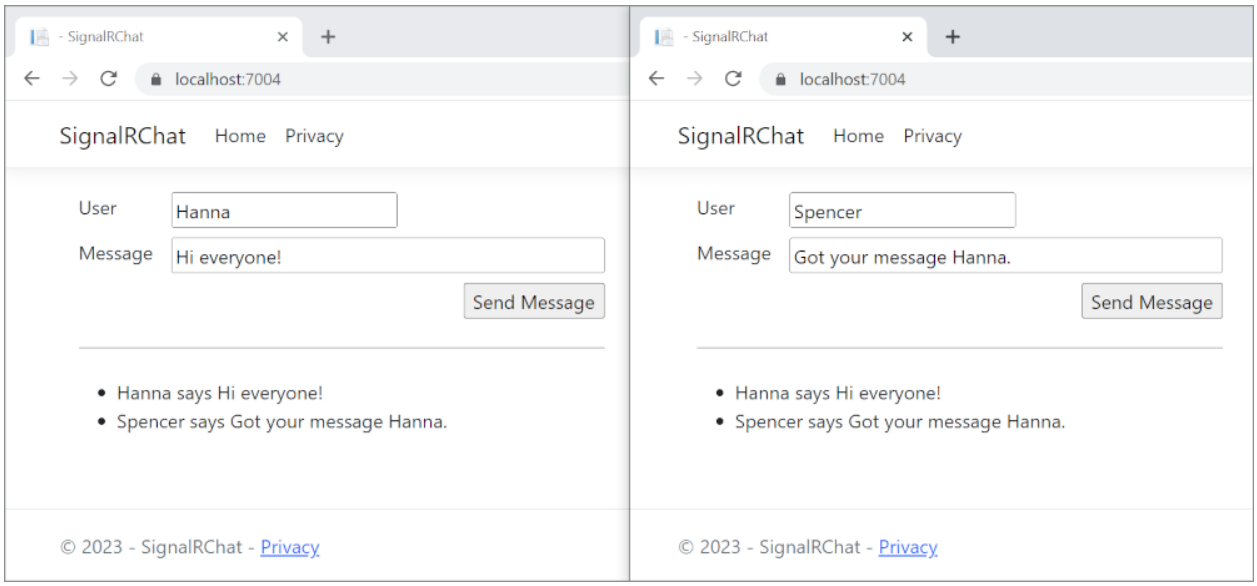
Visual Studio

Select `Ctrl`+`F5` to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

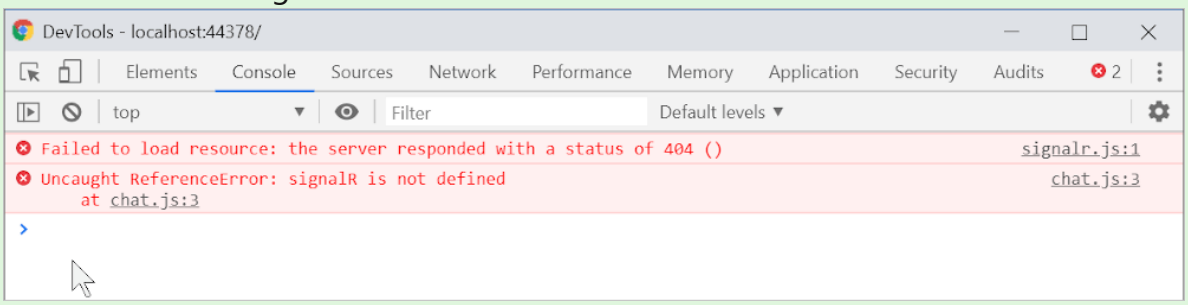
Choose either browser, enter a name and message, and select the **Send Message** button.

The name and message are displayed on both pages instantly.



Tip

If the app doesn't work, open the browser developer tools (F12) and go to the console. Look for possible errors related to HTML and JavaScript code. For example, if `signalr.js` was put in a different folder than directed, the reference to that file won't work resulting in a 404 error in the console.



If an `ERR_SPDY_INADEQUATE_TRANSPORT_SECURITY` error has occurred in Chrome, run the following commands to update the development certificate:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app. For more information on Azure SignalR Service, see What is Azure SignalR Service?.

Next steps

- Use hubs
- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- View or download sample code (how to download)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Tutorial: Get started with ASP.NET Core SignalR using TypeScript and Webpack

Article • 11/16/2023

By Sébastien Sougnez

This tutorial demonstrates using Webpack in an ASP.NET Core SignalR web app to bundle and build a client written in TypeScript. Webpack enables developers to bundle and build the client-side resources of a web app.

In this tutorial, you learn how to:

- ✓ Create an ASP.NET Core SignalR app
- ✓ Configure the SignalR server
- ✓ Configure a build pipeline using Webpack
- ✓ Configure the SignalR TypeScript client
- ✓ Enable communication between the client and the server

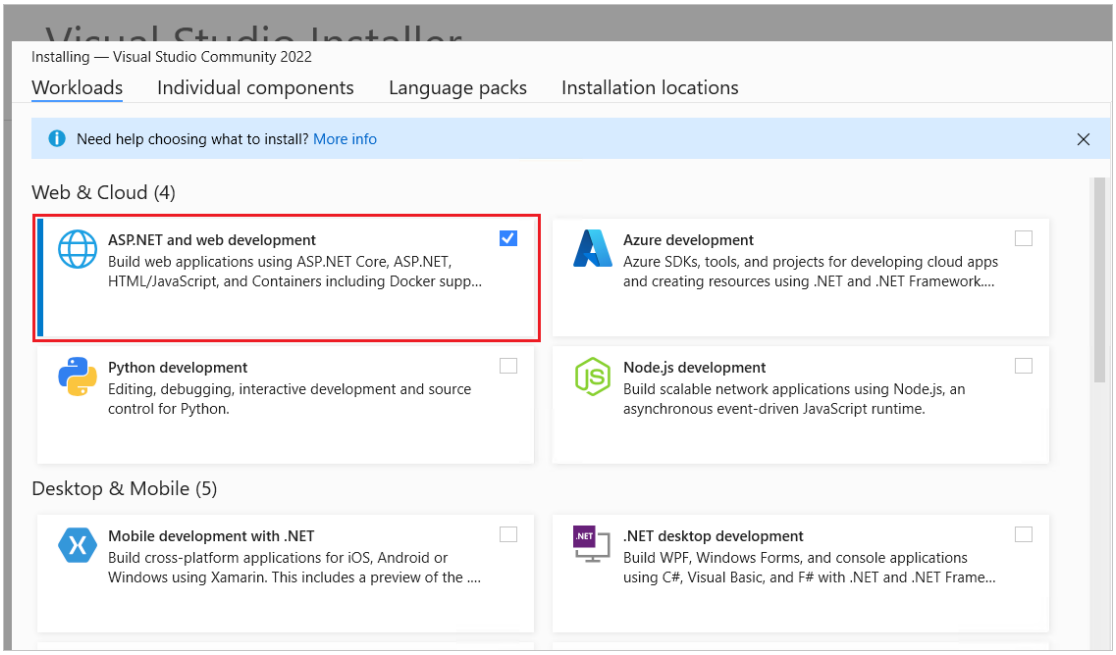
View or download sample code (how to download)

Prerequisites

- Node.js with npm

Visual Studio

- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



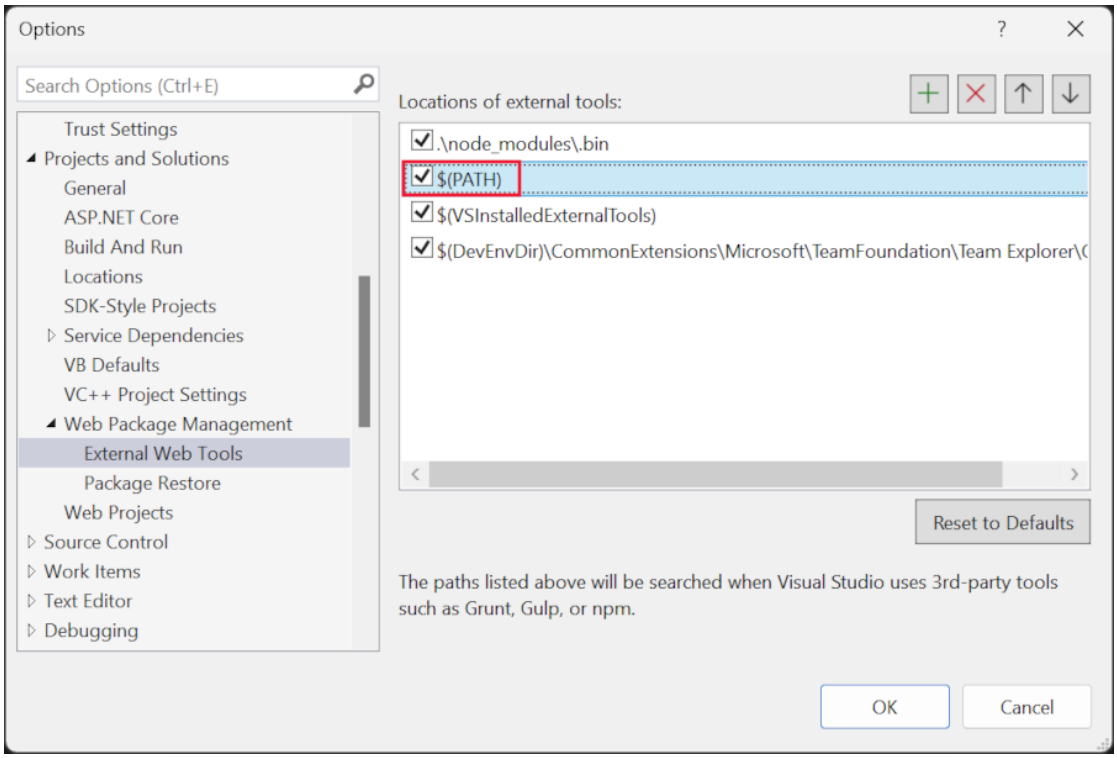
Create the ASP.NET Core web app

Visual Studio

By default, Visual Studio uses the version of npm found in its installation directory. To configure Visual Studio to look for npm in the `PATH` environment variable:

Launch Visual Studio. At the start window, select **Continue without code**.

1. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.
2. Select the `$(PATH)` entry from the list. Select the up arrow to move the entry to the second position in the list, and select **OK**:



.

To create a new ASP.NET Core web app:

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Empty** template. Select **Next**.
2. Name the project `SignalRWebpack`, and select **Create**.
3. Select **.NET 8.0 (Long Term Support)** from the **Framework** drop-down. Select **Create**.

Add the Microsoft.TypeScript.MSBuild NuGet package to the project:

1. In **Solution Explorer**, right-click the project node and select **Manage NuGet Packages**. In the **Browse** tab, search for `Microsoft.TypeScript.MSBuild` and then select **Install** on the right to install the package.

Visual Studio adds the NuGet package under the **Dependencies** node in **Solution Explorer**, enabling TypeScript compilation in the project.

Configure the server

In this section, you configure the ASP.NET Core web app to send and receive SignalR messages.

1. In `Program.cs`, call AddSignalR:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();
```

2. Again, in `Program.cs`, call UseDefaultFiles and UseStaticFiles:

```
C#

var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();
```

   The preceding code allows the server to locate and serve the `index.html` file. The file is served whether the user enters its full URL or the root URL of the web app.

3. Create a new directory named `Hubs` in the project root, `SignalRWebpack/`, for the SignalR hub class.

4. Create a new file, `Hubs/ChatHub.cs`, with the following code:

```
C#

using Microsoft.AspNetCore.SignalR;

namespace SignalRWebpack.Hubs;

public class ChatHub : Hub
{
    public async Task NewMessage(long username, string message) =>
        await Clients.All.SendAsync("messageReceived", username,
message);
}
```

   The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic `on` method to receive all the messages. A method named after the message name is enough.

   In this example:

   - The TypeScript client sends a message identified as `newMessage`.
   - The C# `NewMessage` method expects the data sent by the client.
   - A call is made to SendAsync on Clients.All.
   - The received messages are sent to all clients connected to the hub.

5. Add the following `using` statement at the top of `Program.cs` to resolve the `ChatHub` reference:

```C#
using SignalRWebpack.Hubs;
```

6. In `Program.cs`, map the `/hub` route to the `ChatHub` hub. Replace the code that displays `Hello World!` with the following code:

```C#
app.MapHub<ChatHub>("/hub");
```

Configure the client

In this section, you create a Node.js project to convert TypeScript to JavaScript and bundle client-side resources, including HTML and CSS, using Webpack.

1. Run the following command in the project root to create a `package.json` file:

```Console
npm init -y
```

2. Add the highlighted property to the `package.json` file and save the file changes:

```JSON
{
  "name": "SignalRWebpack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Setting the `private` property to `true` prevents package installation warnings in the next step.

3. Install the required npm packages. Run the following command from the project root:

```
Console

npm i -D -E clean-webpack-plugin css-loader html-webpack-plugin mini-
css-extract-plugin ts-loader typescript webpack webpack-cli
```

The `-E` option disables npm's default behavior of writing semantic versioning range operators to `package.json`. For example, `"webpack": "5.76.1"` is used instead of `"webpack": "^5.76.1"`. This option prevents unintended upgrades to newer package versions.

For more information, see the npm-install documentation.

4. Replace the `scripts` property of `package.json` file with the following code:

```
JSON

"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

The following scripts are defined:

- `build`: Bundles the client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The `mode` option disables production optimizations, such as tree shaking and minification. use `build` in development only.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET CLI's publish command to publish the app.

5. Create a file named `webpack.config.js` in the project root, with the following code:

```
JavaScript

const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
```

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/",
    },
    resolve: {
        extensions: [".js", ".ts"],
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader",
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"],
            },
        ],
    },
    plugins: [
        new CleanWebpackPlugin(),
        new HtmlWebpackPlugin({
            template: "./src/index.html",
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css",
        }),
    ],
};
```

The preceding file configures the Webpack compilation process:

- The `output` property overrides the default value of `dist`. The bundle is instead emitted in the `wwwroot` directory.
- The `resolve.extensions` array includes `.js` to import the SignalR client JavaScript.

6. Create a new directory named `src` in the project root, `SignalRWebpack/`, for the client code.

7. Copy the `src` directory and its contents from the sample project into the project root. The `src` directory contains the following files:

    - `index.html`, which defines the homepage's boilerplate markup:

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR with TypeScript and
Webpack</title>
  </head>
  <body>
    <div id="divMessages" class="messages"></div>
    <div class="input-zone">
      <label id="lblMessage" for="tbMessage">Message:</label>
      <input id="tbMessage" class="input-zone-input" type="text"
/>
      <button id="btnSend">Send</button>
    </div>
  </body>
</html>
```

- `css/main.css`, which provides CSS styles for the homepage:

css

```
*,
*::before,
*::after {
  box-sizing: border-box;
}

html,
body {
  margin: 0;
  padding: 0;
}

.input-zone {
  align-items: center;
  display: flex;
  flex-direction: row;
  margin: 10px;
}

.input-zone-input {
  flex: 1;
  margin-right: 10px;
}

.message-author {
  font-weight: bold;
}

.messages {
```

```
  border: 1px solid #000;
  margin: 10px;
  max-height: 300px;
  min-height: 300px;
  overflow-y: auto;
  padding: 5px;
}
```

- `tsconfig.json`, which configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript:

JSON

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

- `index.ts`:

TypeScript

```
import * as signalR from "@microsoft/signalr";
import "./css/main.css";

const divMessages: HTMLDivElement =
document.querySelector("#divMessages");
const tbMessage: HTMLInputElement =
document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement =
document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message:
string) => {
  const m = document.createElement("div");

  m.innerHTML = `<div class="message-author">${username}</div>
<div>${message}</div>`;

  divMessages.appendChild(m);
  divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch((err) => document.write(err));
```

```
tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
  if (e.key === "Enter") {
    send();
  }
});

btnSend.addEventListener("click", send);

function send() {
  connection.send("newMessage", username, tbMessage.value)
    .then(() => (tbMessage.value = ""));
}
```

The preceding code retrieves references to DOM elements and attaches two event handlers:

- `keyup`: Fires when the user types in the `tbMessage` textbox and calls the `send` function when the user presses the **Enter** key.
- `click`: Fires when the user selects the **Send** button and calls `send` function is called.

The `HubConnectionBuilder` class creates a new builder for configuring the server connection. The `withUrl` function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. Any number of message names can be listened to. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new `div` element is created with the author's name and the message content in its `innerHTML` attribute. It's added to the main `div` element displaying the messages.

Sending a message through the WebSockets connection requires calling the `send` method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as `newMessage` is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

8. Run the following command at the project root:

```
Console
```

```
npm i @microsoft/signalr @types/node
```

The preceding command installs:

- The SignalR TypeScript client, which allows the client to send messages to the server.
- The TypeScript type definitions for Node.js, which enables compile-time checking of Node.js types.

Test the app

Confirm that the app works with the following steps:

Visual Studio

1. Run Webpack in `release` mode. Using the **Package Manager Console** window, run the following command in the project root.

   Console

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the `wwwroot` folder.

   Webpack completed the following tasks:

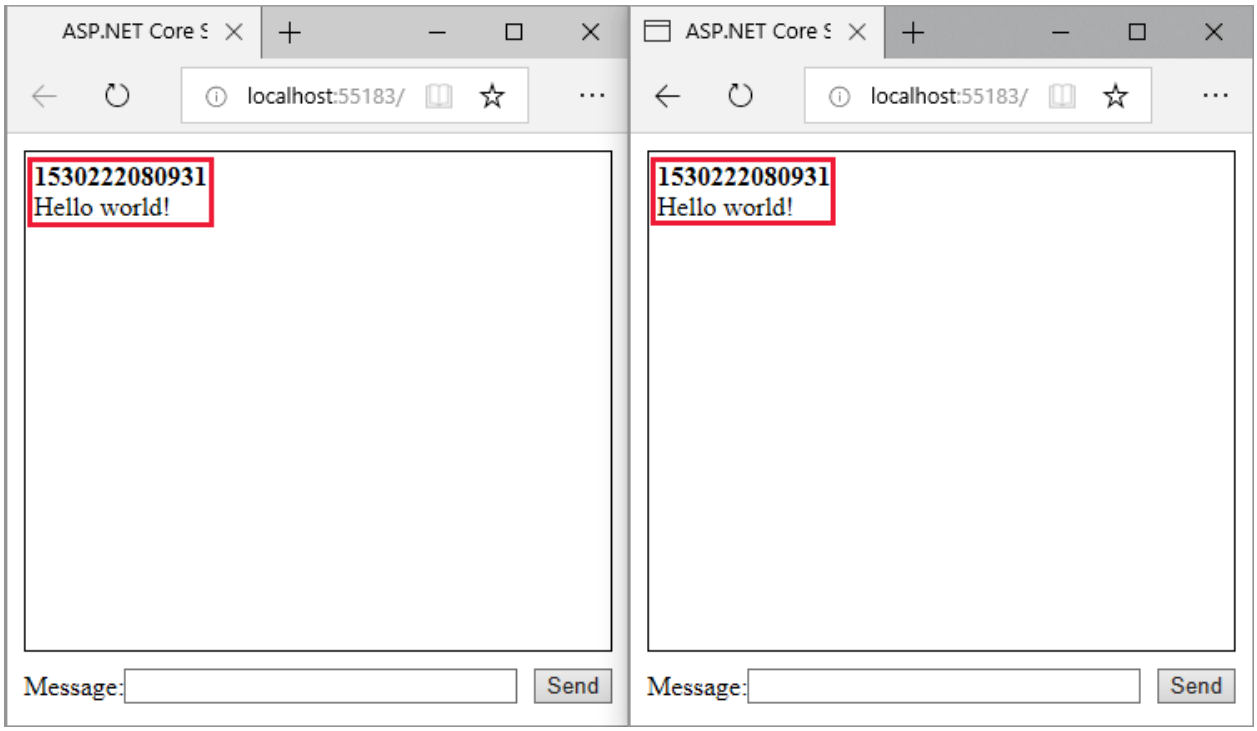
   - Purged the contents of the `wwwroot` directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from `src` to the `wwwroot` directory.
   - Injected the following elements into the `wwwroot/index.html` file:
     - A `<link>` tag, referencing the `wwwroot/main.<hash>.css` file. This tag is placed immediately before the closing `</head>` tag.
     - A `<script>` tag, referencing the minified `wwwroot/main.<hash>.js` file. This tag is placed immediately after the closing `</title>` tag.

2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The `wwwroot/index.html` file is served at `https://localhost:<port>`.

   If there are compile errors, try closing and reopening the solution.

3. Open another browser instance (any browser) and paste the URL in the address bar.

4. Choose either browser, type something in the **Message** text box, and select the **Send** button. The unique user name and message are displayed on both pages instantly.



Next steps

- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- MessagePack Hub Protocol in SignalR for ASP.NET Core

Additional resources

- ASP.NET Core SignalR JavaScript client
- Use hubs in ASP.NET Core SignalR

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Use ASP.NET Core SignalR with Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial provides a basic working experience for building a real-time app using SignalR with Blazor. This article is useful for developers who are already familiar with SignalR and are seeking to understand how to use SignalR in a Blazor app. For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets and the API documentation:

- Overview of ASP.NET Core SignalR
- ASP.NET Core Blazor
- .NET API browser

Learn how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

Prerequisites

Visual Studio

Visual Studio (latest release) with the **ASP.NET and web development** workload

Sample app

Downloading the tutorial's sample chat app isn't required for this tutorial. The sample app is the final, working app produced by following the steps of this tutorial. When you

open the samples repository, open the version folder that you plan to target and find the sample named `BlazorSignalRApp`.

View or download sample code (how to download)

Create a Blazor Web App

Follow the guidance for your choice of tooling:

Visual Studio

> **Note**
>
> Visual Studio 2022 or later and .NET Core SDK 8.0.0 or later are required.

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorSignalRApp` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.
- Confirm that the **Location** for the app is suitable. Leave the **Place solution and project in the same directory** checkbox selected. Select the **Next** button.
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Confirm that the latest framework is selected. If Visual Studio's **Framework** dropdown list doesn't include the latest available .NET framework, update Visual Studio and restart the tutorial.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **WebAssembly**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected
  - Select **Create**.

The guidance in this article uses a WebAssembly component for the SignalR client because it doesn't make sense to use SignalR to connect to a hub from an Interactive Server component in the same app, as that can lead to server port exhaustion.

Add the SignalR client library

Visual Studio

In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.

In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.

With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.

In the search results, select the latest release of the Microsoft.AspNetCore.SignalR.Client package. Select **Install**.

If the **Preview Changes** dialog appears, select **OK**.

If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the server `BlazorSignalRApp` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

C#

```
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Hubs;

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

Add services and an endpoint for the SignalR hub

Open the `Program` file of the server `BlazorSignalRApp` project.

Add the namespaces for Microsoft.AspNetCore.ResponseCompression and the `ChatHub` class to the top of the file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using BlazorSignalRApp.Hubs;
```

Add SignalR and Response Compression Middleware services:

C#

```
builder.Services.AddSignalR();

builder.Services.AddResponseCompression(opts =>
{
   opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
       ["application/octet-stream"]);
});
```

Use Response Compression Middleware at the top of the processing pipeline's configuration. Place the following line of code immediately after the line that builds the app (`var app = builder.Build();`):

C#

```
app.UseResponseCompression();
```

Add an endpoint for the hub immediately before the line that runs the app (`app.Run();`):

C#

```
app.MapHub<ChatHub>("/chathub");
```

Add Razor component code for chat

Add the following `Pages/Chat.razor` file to the `BlazorSignalRApp.Client` project:

razor

```razor
@page "/chat"
@rendermode InteractiveWebAssembly
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager Navigation
@implements IAsyncDisposable

<PageTitle>Chat</PageTitle>

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection? hubConnection;
    private List<string> messages = [];
    private string? userInput;
    private string? messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message)
=>
        {
            var encodedMsg = $"{user}: {message}";
            messages.Add(encodedMsg);
            InvokeAsync(StateHasChanged);
        });

        await hubConnection.StartAsync();
```

```
    }

    private async Task Send()
    {
        if (hubConnection is not null)
        {
            await hubConnection.SendAsync("SendMessage", userInput,
messageInput);
        }
    }

    public bool IsConnected =>
        hubConnection?.State == HubConnectionState.Connected;

    public async ValueTask DisposeAsync()
    {
        if (hubConnection is not null)
        {
            await hubConnection.DisposeAsync();
        }
    }
}
```

Add an entry to the `NavMenu` component to reach the chat page. In `Components/Layout/NavMenu.razor` immediately after the `<div>` block for the `Weather` component, add the following `<div>` block:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="chat">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Chat
    </NavLink>
</div>
```

> **Note**
>
> Disable Response Compression Middleware in the `Development` environment when using Hot Reload. For more information, see ASP.NET Core Blazor SignalR guidance.

Run the app

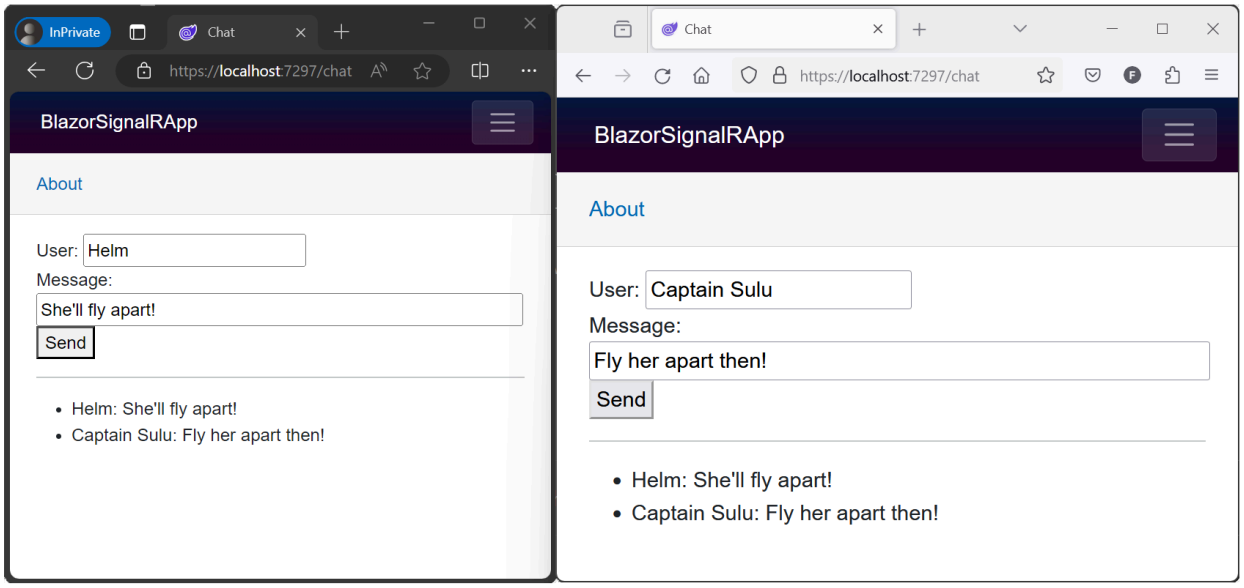
Follow the guidance for your tooling:

Visual Studio

With the server `BlazorSignalRApp` project selected in **Solution Explorer**, press `F5` to run the app with debugging or `Ctrl`+`F5` (Windows)/`⌘`+`F5` (macOS) to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* ©1991 Paramount

Next steps

In this tutorial, you learned how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets:

Overview of ASP.NET Core SignalR | ASP.NET Core Blazor

Additional resources

- Bearer token authentication with Identity Server, WebSockets, and Server-Sent Events
- Secure a SignalR hub in Blazor WebAssembly apps
- SignalR cross-origin negotiation for authentication
- SignalR configuration
- Debug ASP.NET Core Blazor apps
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Tutorial: Create a gRPC client and server in ASP.NET Core

Article • 11/16/2023

This tutorial shows how to create a .NET Core gRPC client and an ASP.NET Core gRPC Server. At the end, you'll have a gRPC client that communicates with the gRPC Greeter service.

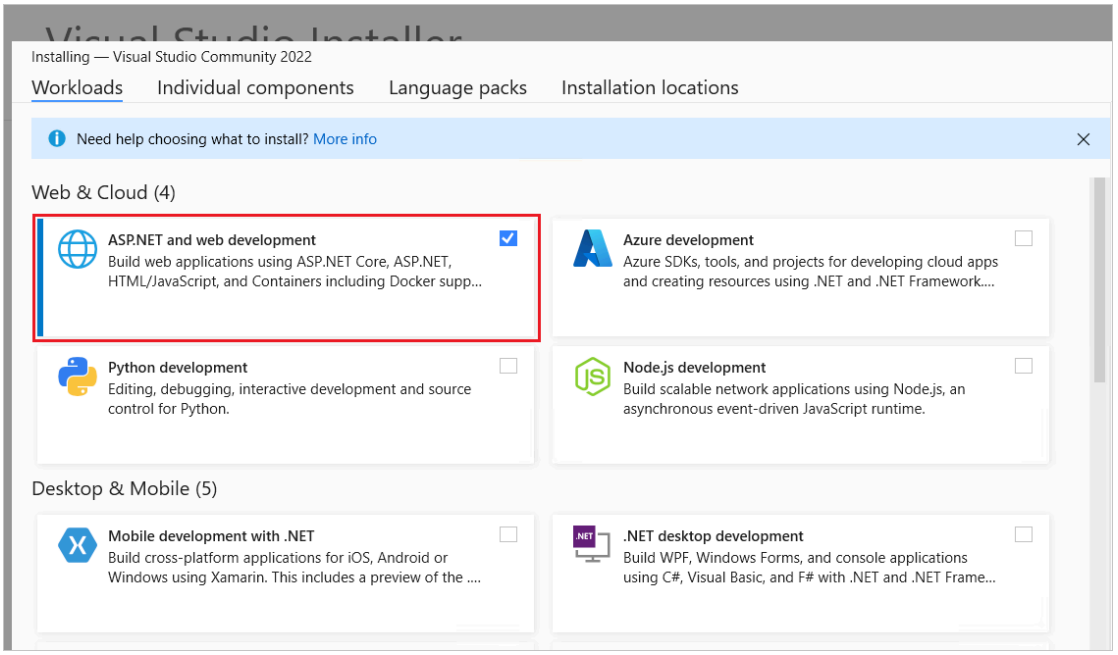
In this tutorial, you:

- ✓ Create a gRPC Server.
- ✓ Create a gRPC client.
- ✓ Test the gRPC client with the gRPC Greeter service.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a gRPC service

Visual Studio

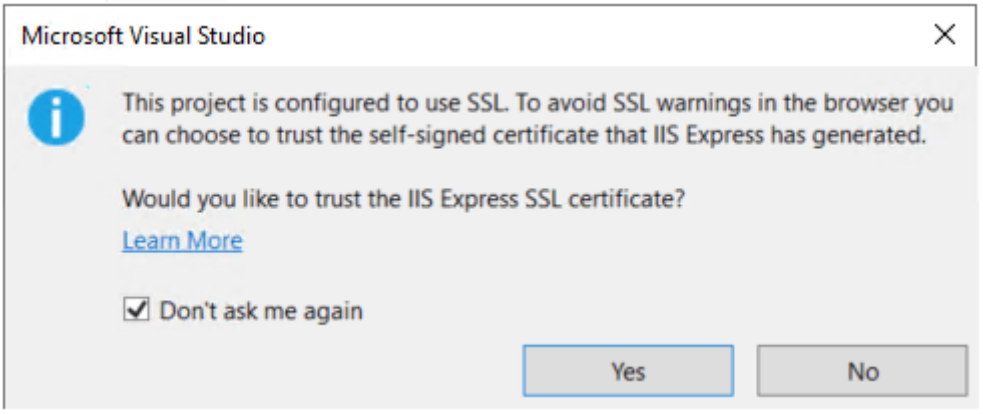
- Start Visual Studio 2022 and select **New Project**.
- In the **Create a new project** dialog, search for `gRPC`. Select **ASP.NET Core gRPC Service** and select **Next**.
- In the **Configure your new project** dialog, enter `GrpcGreeter` for **Project name**. It's important to name the project *GrpcGreeter* so the namespaces match when you copy and paste code.
- Select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Run the service

Visual Studio

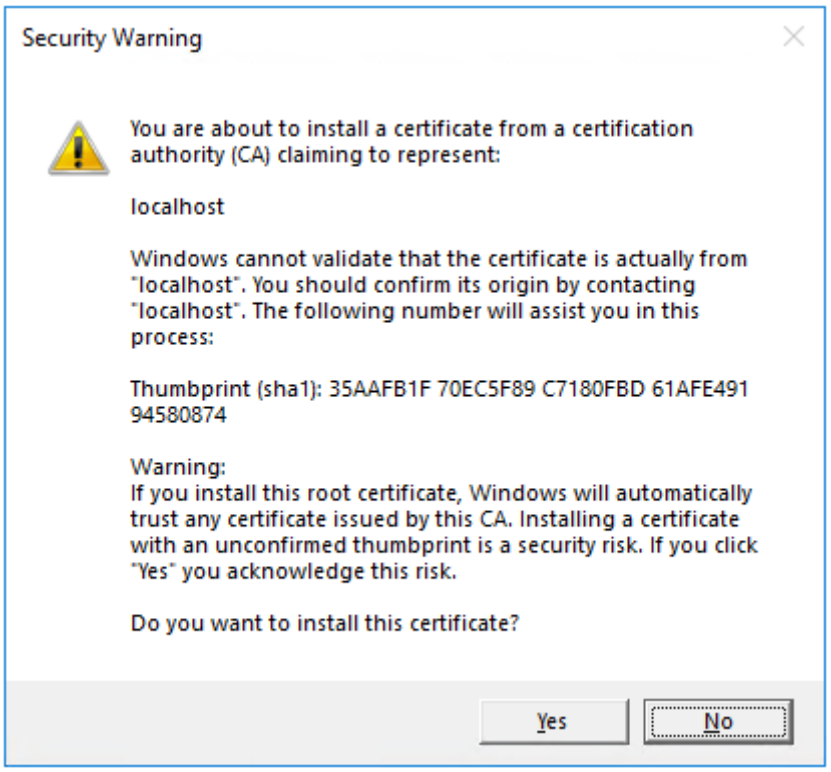
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:

  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio:

- Starts Kestrel server.
- Launches a browser.
- Navigates to `http://localhost:port`, such as `http://localhost:7042`.
  - *port*: A randomly assigned port number for the app.
  - `localhost`: The standard hostname for the local computer. Localhost only serves web requests from the local computer.

The logs show the service listening on `https://localhost:<port>`, where `<port>` is the localhost port number randomly assigned when the project is created and set in `Properties/launchSettings.json`.

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

> **Note**
>
> The gRPC template is configured to use Transport Layer Security (TLS). gRPC clients need to use HTTPS to call the server. The gRPC service localhost port number is randomly assigned when the project is created and set in the *Properties\launchSettings.json* file of the gRPC service project.

Examine the project files

GrpcGreeter project files:

- `Protos/greet.proto`: defines the `Greeter` gRPC and is used to generate the gRPC server assets. For more information, see Introduction to gRPC.
- `Services` folder: Contains the implementation of the `Greeter` service.
- `appSettings.json`: Contains configuration data such as the protocol used by Kestrel. For more information, see Configuration in ASP.NET Core.
- `Program.cs`, which contains:
  - The entry point for the gRPC service. For more information, see .NET Generic Host in ASP.NET Core.
  - Code that configures app behavior. For more information, see App startup.

Create the gRPC client in a .NET console app

Visual Studio

- Open a second instance of Visual Studio and select **New Project**.
- In the **Create a new project** dialog, select **Console App**, and select **Next**.
- In the **Project name** text box, enter **GrpcGreeterClient** and select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Add required NuGet packages

The gRPC client project requires the following NuGet packages:

- Grpc.Net.Client, which contains the .NET Core client.
- Google.Protobuf, which contains protobuf message APIs for C#.

- Grpc.Tools, which contain C# tooling support for protobuf files. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`.

Visual Studio

Install the packages using either the Package Manager Console (PMC) or Manage NuGet Packages.

PMC option to install packages

- From Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**
- From the **Package Manager Console** window, run `cd GrpcGreeterClient` to change directories to the folder containing the `GrpcGreeterClient.csproj` files.
- Run the following commands:

PowerShell

```
Install-Package Grpc.Net.Client
Install-Package Google.Protobuf
Install-Package Grpc.Tools
```

Manage NuGet Packages option to install packages

- Right-click the project in **Solution Explorer** > **Manage NuGet Packages**.
- Select the **Browse** tab.
- Enter **Grpc.Net.Client** in the search box.
- Select the **Grpc.Net.Client** package from the **Browse** tab and select **Install**.
- Repeat for `Google.Protobuf` and `Grpc.Tools`.

Add greet.proto

- Create a *Protos* folder in the gRPC client project.
- Copy the *Protos\greet.proto* file from the gRPC Greeter service to the *Protos* folder in the gRPC client project.
- Update the namespace inside the `greet.proto` file to the project's namespace:

JSON

```
option csharp_namespace = "GrpcGreeterClient";
```

- Edit the `GrpcGreeterClient.csproj` project file:

Visual Studio

Right-click the project and select **Edit Project File**.

- Add an item group with a `<Protobuf>` element that refers to the *greet.proto* file:

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Create the Greeter client

- Build the client project to create the types in the `GrpcGreeterClient` namespace.

> **Note**
>
> The `GrpcGreeterClient` types are generated automatically by the build process. The tooling package Grpc.Tools generates the following files based on the *greet.proto* file:
>
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\Greet.cs`: The protocol buffer code which populates, serializes and retrieves the request and response message types.
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\GreetGrpc.cs`: Contains the generated client classes.
>
> For more information on the C# assets automatically generated by Grpc.Tools, see gRPC services with C#: Generated C# assets.

- Update the gRPC client `Program.cs` file with the following code.

C#

```
using System.Threading.Tasks;
using Grpc.Net.Client;
using GrpcGreeterClient;

// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

- In the preceding highlighted code, replace the localhost port number `7042` with the `HTTPS` port number specified in `Properties/launchSettings.json` within the `GrpcGreeter` service project.

`Program.cs` contains the entry point and logic for the gRPC client.

The Greeter client is created by:

- Instantiating a `GrpcChannel` containing the information for creating the connection to the gRPC service.
- Using the `GrpcChannel` to construct the Greeter client:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

The Greeter client calls the asynchronous `SayHello` method. The result of the `SayHello` call is displayed:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
```

```
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

Test the gRPC client with the gRPC Greeter service

Update the `appsettings.Development.json` file by adding the following highlighted lines:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
      ,"Microsoft.AspNetCore.Hosting": "Information",
      "Microsoft.AspNetCore.Routing.EndpointMiddleware": "Information"
    }
  }
}
```

Visual Studio

- In the Greeter service, press `Ctrl+F5` to start the server without the debugger.
- In the `GrpcGreeterClient` project, press `Ctrl+F5` to start the client without the debugger.

The client sends a greeting to the service with a message containing its name, *GreeterClient*. The service sends the message "Hello GreeterClient" as a response. The "Hello GreeterClient" response is displayed in the command prompt:

Console

```
Greeting: Hello GreeterClient
Press any key to exit...
```

The gRPC service records the details of the successful call in the logs written to the command prompt:

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path:
C:\GH\aspnet\docs\4\Docs\aspnetcore\tutorials\grpc\grpc-
start\sample\GrpcGreeter
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:
<port>/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 78.32260000000001ms 200 application/grpc
```

> **Note**
>
> The code in this article requires the ASP.NET Core HTTPS development certificate to secure the gRPC service. If the .NET gRPC client fails with the message `The remote certificate is invalid according to the validation procedure.` or `The SSL connection could not be established.`, the development certificate isn't trusted. To fix this issue, see Call a gRPC service with an untrusted/invalid certificate.

Next steps

- View or download the completed sample code for this tutorial (how to download).
- Overview for gRPC on .NET
- gRPC services with C#
- Migrate gRPC from C-core to gRPC for .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8

Article • 09/27/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Tom Dykstra, Jeremy Likness, and Jon P Smith

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

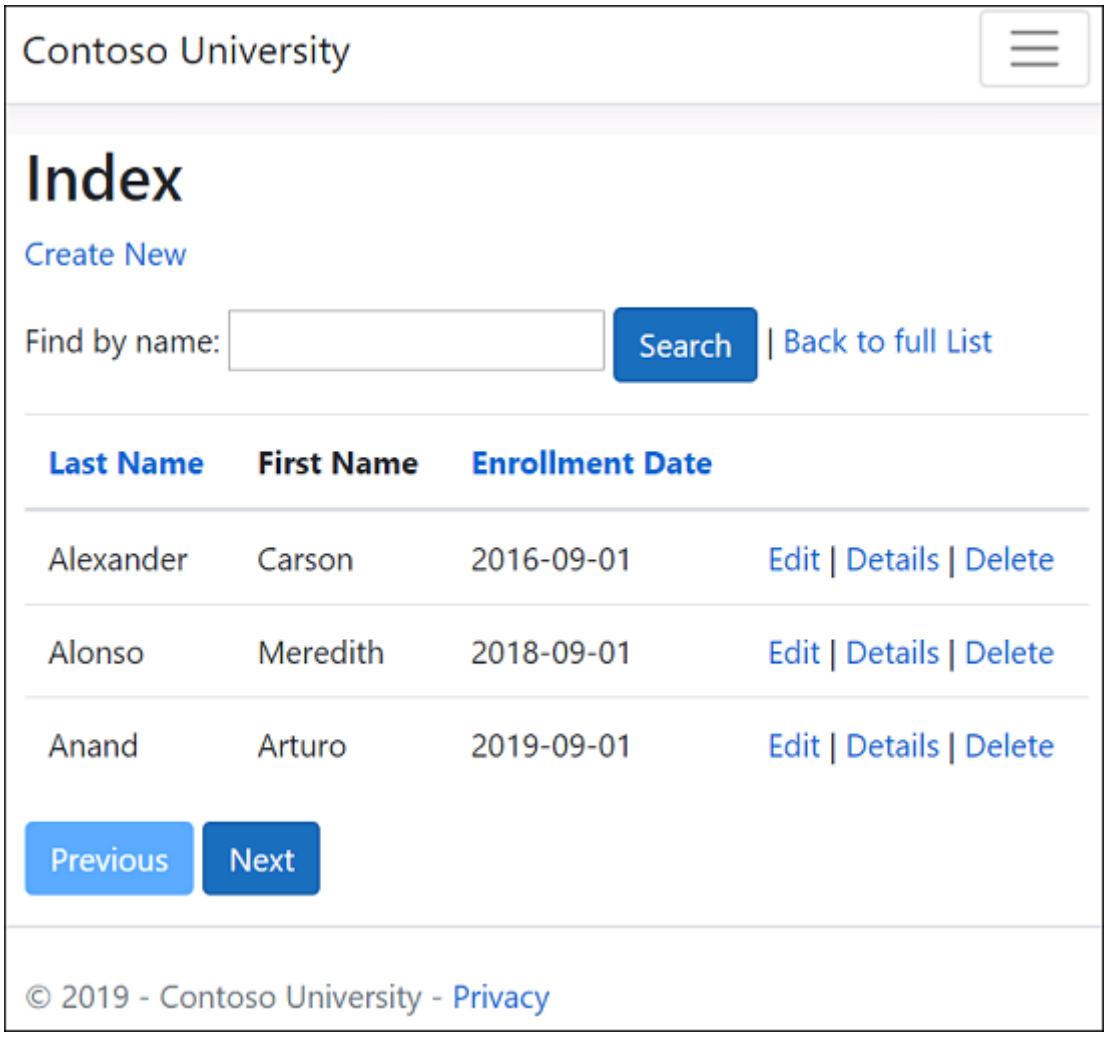
Troubleshooting

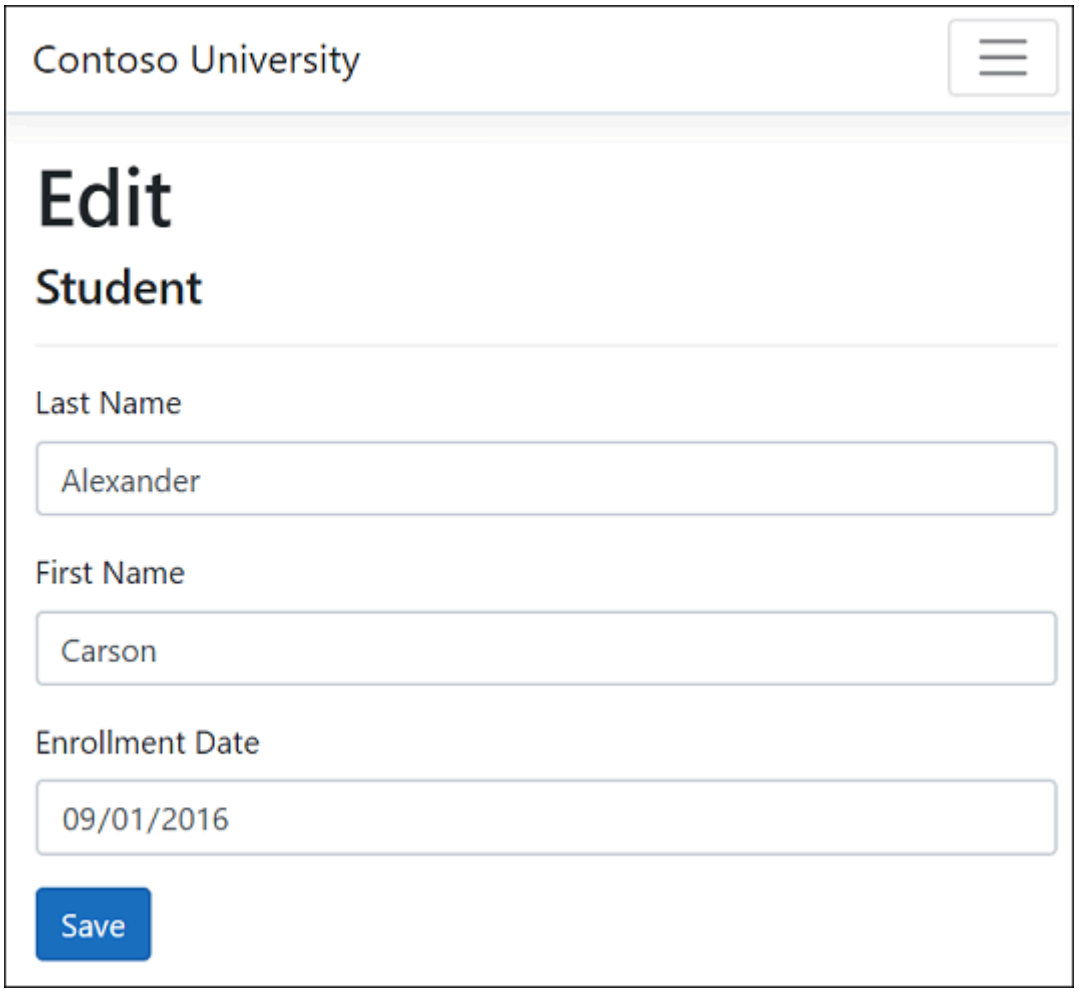
If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using

the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core with ASP.NET Core, not how to customize the UI.

Optional: Build the sample download

This step is optional. Building the completed app is recommended when you have problems you can't solve. If you run into a problem you can't resolve, compare your code to the completed project. Download instructions.

Visual Studio

Select `ContosoUniversity.csproj` to open the project.

- Build the project.
- In Package Manager Console (PMC) run the following command:

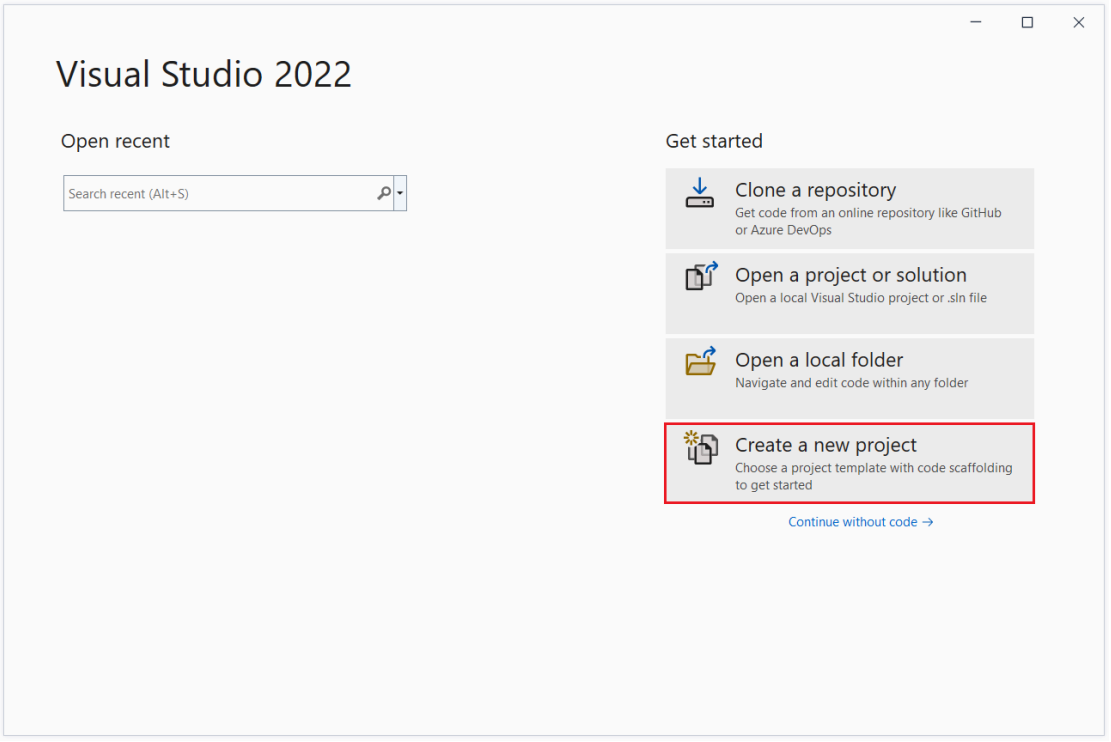
```
PowerShell
Update-Database
```

Run the project to seed the database.

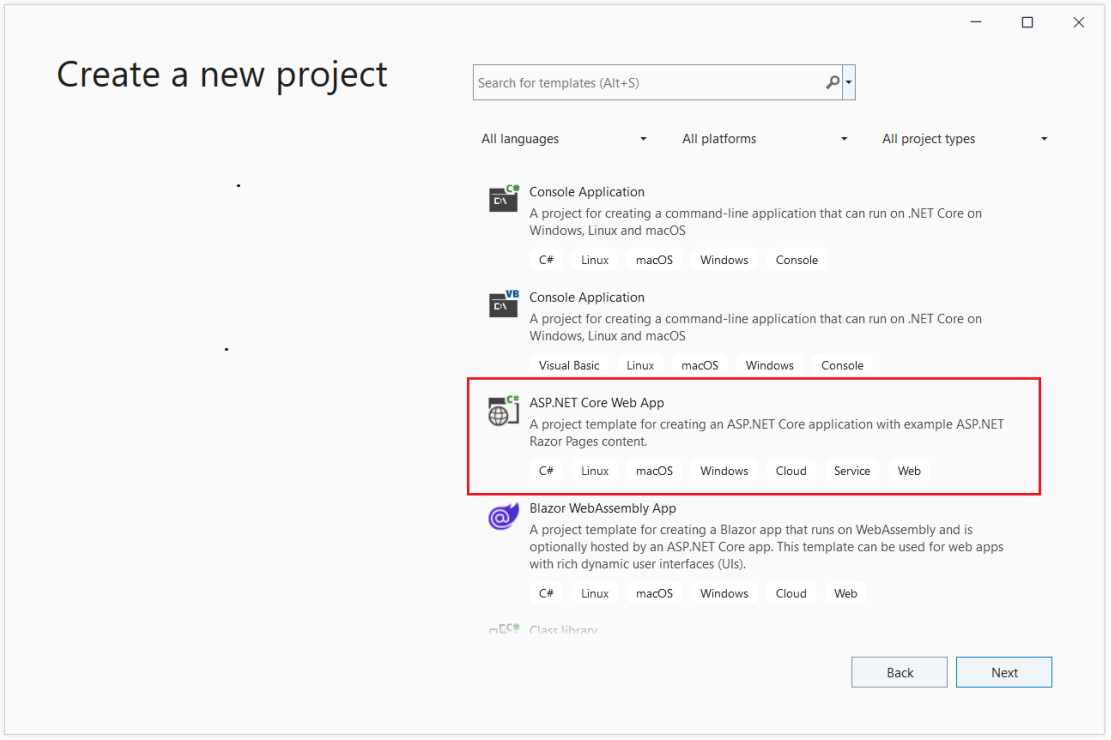
Create the web app project

Visual Studio

1. Start Visual Studio 2022 and select **Create a new project**.



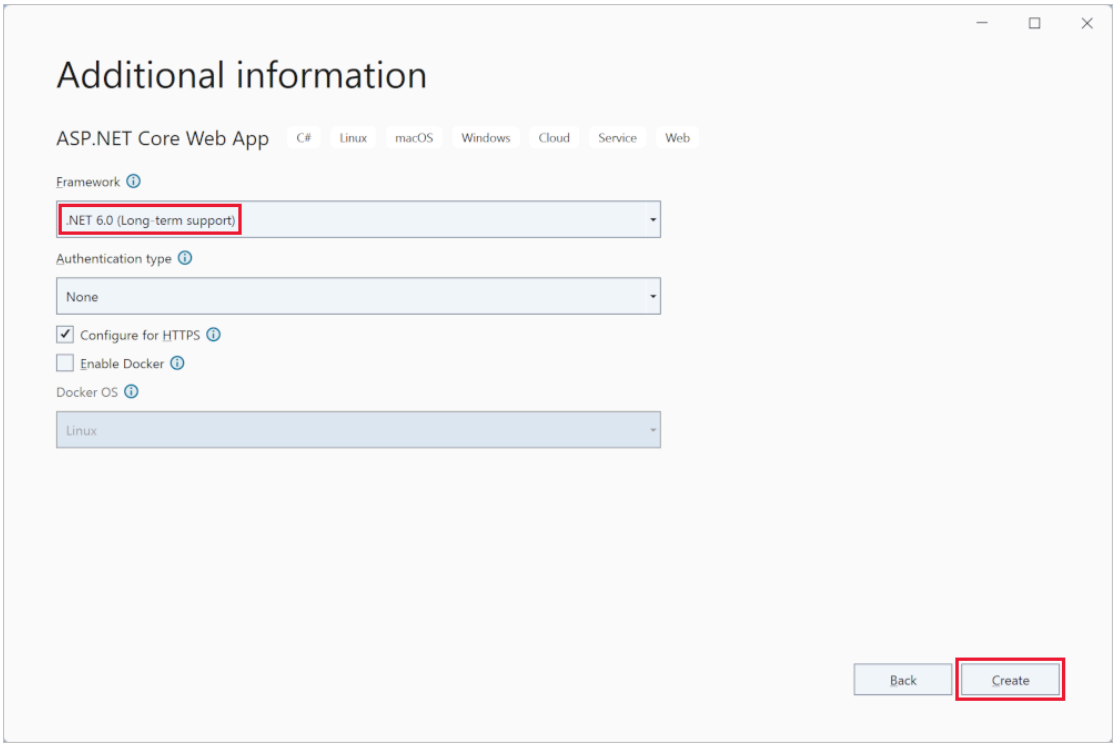
2. In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.



3. In the **Configure your new project** dialog, enter `ContosoUniversity` for **Project name**. It's important to name the project *ContosoUniversity*, including matching the capitalization, so the namespaces will match when you copy and paste example code.

4. Select **Next**.

5. In the **Additional information** dialog, select **.NET 6.0 (Long-term support)** and then select **Create**.



Set up the site style

Copy and paste the following code into the `Pages/Shared/_Layout.cshtml` file:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/ContosoUniversity.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
```

```
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - Contoso University - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

The layout file sets the site header, footer, and menu. The preceding code makes the following changes:

- Each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- The **Home** and **Privacy** menu entries are deleted.

- Entries are added for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

In `Pages/Index.cshtml`, replace the contents of the file with the following code:

CSHTML

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in
a series of tutorials.
                </p>
                <p>
@*                      <a
href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro"
class="stretched-link">See the tutorial</a>
*@                  </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
@*                      <a
href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef
-rp/intro/samples" class="stretched-link">See project source code</a>
*@                  </p>
            </div>
        </div>
```

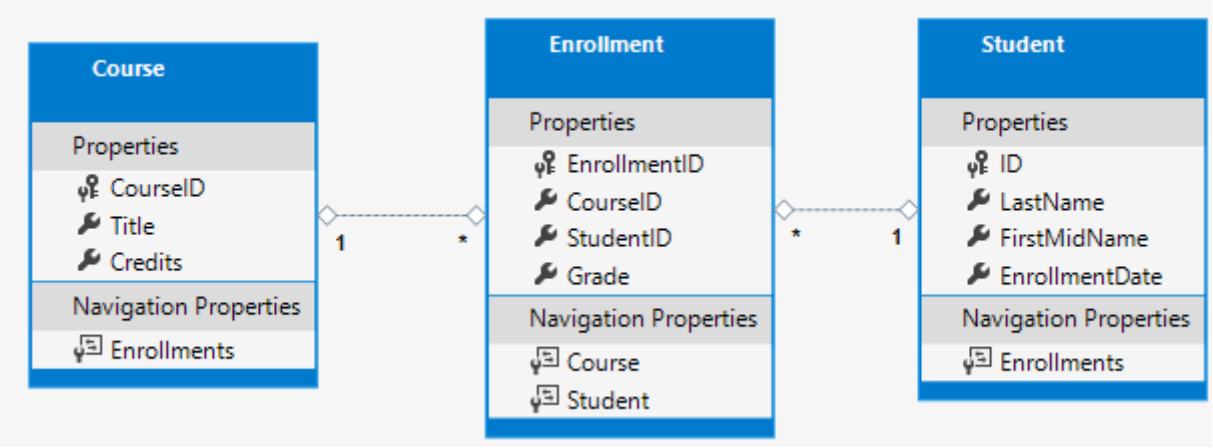
```
    </div>
</div>
```

The preceding code replaces the text about ASP.NET Core with text about this app.

Run the app to verify that the home page appears.

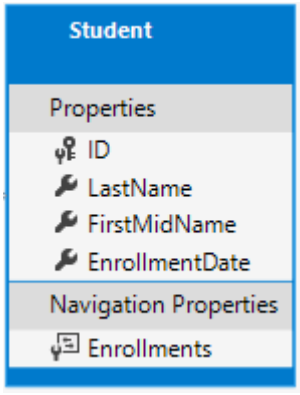
The data model

The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity



- Create a *Models* folder in the project folder.
- Create `Models/Student.cs` with the following code:

```
C#

namespace ContosoUniversity.Models
{
    public class Student
```

```
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

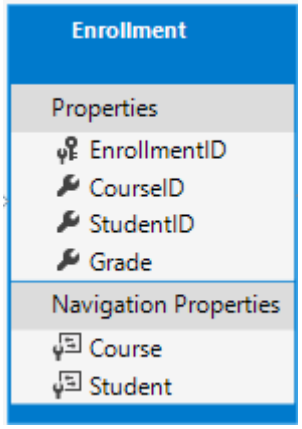
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its `StudentID` column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have `StudentID` = 1. `StudentID` is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. Other collection types can be used, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



Create `Models/Enrollment.cs` with the following code:

```
C#

using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, many developers choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

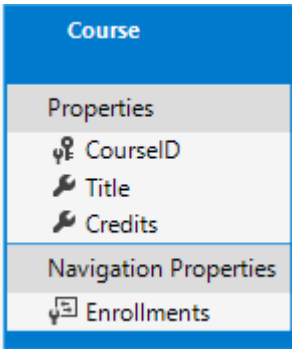
The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties

can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create `Models/Course.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the app. The compiler generates several warnings about how `null` values are handled. See this GitHub issue, Nullable reference types, and Tutorial: Express your design intent more clearly with nullable and non-nullable reference types for more information.

To eliminate the warnings from nullable reference types, remove the following line from the `ContosoUniversity.csproj` file:

XML

```
<Nullable>enable</Nullable>
```

The scaffolding engine currently does not support nullable reference types, therefore the models used in scaffold can't either.

Remove the `?` nullable reference type annotation from `public string? RequestId { get; set; }` in `Pages/Error.cshtml.cs` so the project builds without compiler warnings.

Scaffold Student pages

In this section, the ASP.NET Core scaffolding tool is used to generate:

- An EF Core `DbContext` class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the Microsoft.EntityFrameworkCore.DbContext class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

Visual Studio

- Create a *Pages/Students* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add New Scaffold Item** dialog:
  - In the left tab, select **Installed** > **Common** > **Razor Pages**
  - Select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the + (plus) sign.
    - Change the data context name to end in `SchoolContext` rather than `ContosoUniversityContext`. The updated context name: `ContosoUniversity.Data.SchoolContext`
    - Select **Add** to finish adding the data context class.
    - Select **Add** to finish the **Add Razor Pages** dialog.

The following packages are automatically installed:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

If the preceding step fails, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - `Create.cshtml` and `Create.cshtml.cs`
  - `Delete.cshtml` and `Delete.cshtml.cs`
  - `Details.cshtml` and `Details.cshtml.cs`
  - `Edit.cshtml` and `Edit.cshtml.cs`
  - `Index.cshtml` and `Index.cshtml.cs`
- Creates `Data/SchoolContext.cs`.
- Adds the context to dependency injection in `Program.cs`.
- Adds a database connection string to `appsettings.json`.

Database connection string

The scaffolding tool generates a connection string in the `appsettings.json` file.

Visual Studio

The connection string specifies SQL Server LocalDB:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext-
0e9;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf*

files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update `Data/SchoolContext.cs` with the following code:

C#

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext (DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

The preceding code changes from the singular `DbSet<Student> Student` to the plural `DbSet<Student> Students`. To make the Razor Pages code match the new `DBSet` name, make a global change from: `_context.Student.` to: `_context.Students.`

There are 8 occurrences.

Because an entity set contains multiple entities, many developers prefer the `DBSet` property names should be plural.

The highlighted code:

- Creates a DbSet<TEntity> property for each entity set. In EF Core terminology:
  - An entity set typically corresponds to a database table.
  - An entity corresponds to a row in the table.
- Calls OnModelCreating. `OnModelCreating`:
  - Is called when `SchoolContext` has been initialized but before the model has been secured and used to initialize the context.
  - Is required because later in the tutorial the `Student` entity will have references to the other entities.

We hope to fix this issue in a future release.

Program.cs

ASP.NET Core is built with dependency injection. Services such as the `SchoolContext` are registered with dependency injection during app startup. Components that require these services, such as Razor Pages, are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

Visual Studio

The following highlighted lines were added by the scaffolder:

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration

system reads the connection string from the `appsettings.json` or the `appsettings.Development.json` file.

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter and UseMigrationsEndPoint as shown in the following code:

Visual Studio

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}
```

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package.

In the Package Manager Console, enter the following to add the NuGet package:

PowerShell

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

The `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` NuGet package provides ASP.NET Core middleware for Entity Framework Core error pages. This middleware helps to detect and diagnose errors with Entity Framework Core migrations.

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment for EF migrations errors.

Create the database

Update `Program.cs` to create the database if it doesn't exist:

Visual Studio

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    // DbInitializer.Initialize(context);
}
```

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works early in development when the schema is rapidly evolving, as long as data doesn't need to be preserved. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, the database is deleted that was created by `EnsureCreated` and migrations is used. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create `Data/DbInitializer.cs` with the following code:

```
C#
```

```
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.P
arse("2019-09-01")},
                new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Pa
rse("2017-09-01")},
                new
Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse
("2018-09-01")},
                new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Pa
rse("2017-09-01")},
                new
Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017
-09-01")},
                new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Pars
e("2016-09-01")},
                new
Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse
("2018-09-01")},
                new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Pars
e("2019-09-01")}
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
```

```
            };

            context.Courses.AddRange(courses);
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In `Program.cs`, remove `//` from the `DbInitializer.Initialize` line:

C#

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    DbInitializer.Initialize(context);
}
```

Visual Studio

- Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

PowerShell

```
Drop-Database -Confirm
```

- Respond with `Y` to delete the database.

- Restart the app.
- Select the Students page to see the seeded data.

View the database

Visual Studio

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB** > **Databases** > **SchoolContext-{GUID}**. The database name is generated from the context name provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous EF methods in ASP.NET Core web apps

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't doing work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
C#

public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

> ⚠ **Warning**
>
> The async implementation of Microsoft.Data.SqlClient has some known issues (#593, #601, and others). If you're seeing unexpected performance problems,

try using sync command execution instead, especially when dealing with large text or binary values.

Performance considerations

In general, a web page shouldn't be loading an arbitrary number of rows. A query should use paging or a limiting approach. For example, the preceding query could use `Take` to limit the rows returned:

C#

```
public async Task OnGetAsync()
{
    Student = await _context.Students.Take(10).ToListAsync();
}
```

Enumerating a large table in a view could return a partially constructed HTTP 200 response if a database exception occurs part way through the enumeration.

Paging is covered later in the tutorial.

For more information, see Performance considerations (EF).

Next steps

Use SQLite for development, SQL Server for production

Next tutorial

Part 2, Razor Pages with EF Core in ASP.NET Core - CRUD

Article • 04/11/2023

By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, enrollments are added to the `Details` page.

Read enrollments

To display a student's enrollment data on the page, the enrollment data must be read. The scaffolded code in `Pages/Students/Details.cshtml.cs` reads only the `Student` data, without the `Enrollment` data:

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
```

```
    }

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the Read related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in `Pages/Students/Details.cshtml` with the following code to display a list of enrollments. The changes are highlighted.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
```

```
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the `Course` entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for the selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is

unnecessary if the query can only return one entity, as when it searches on a unique key.

- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in `Pages/Students/Create.cshtml.cs` with the following code.

C#

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` is converted to `DateTime`.

Run the app, and create a student entity to test the Create page.

Overposting

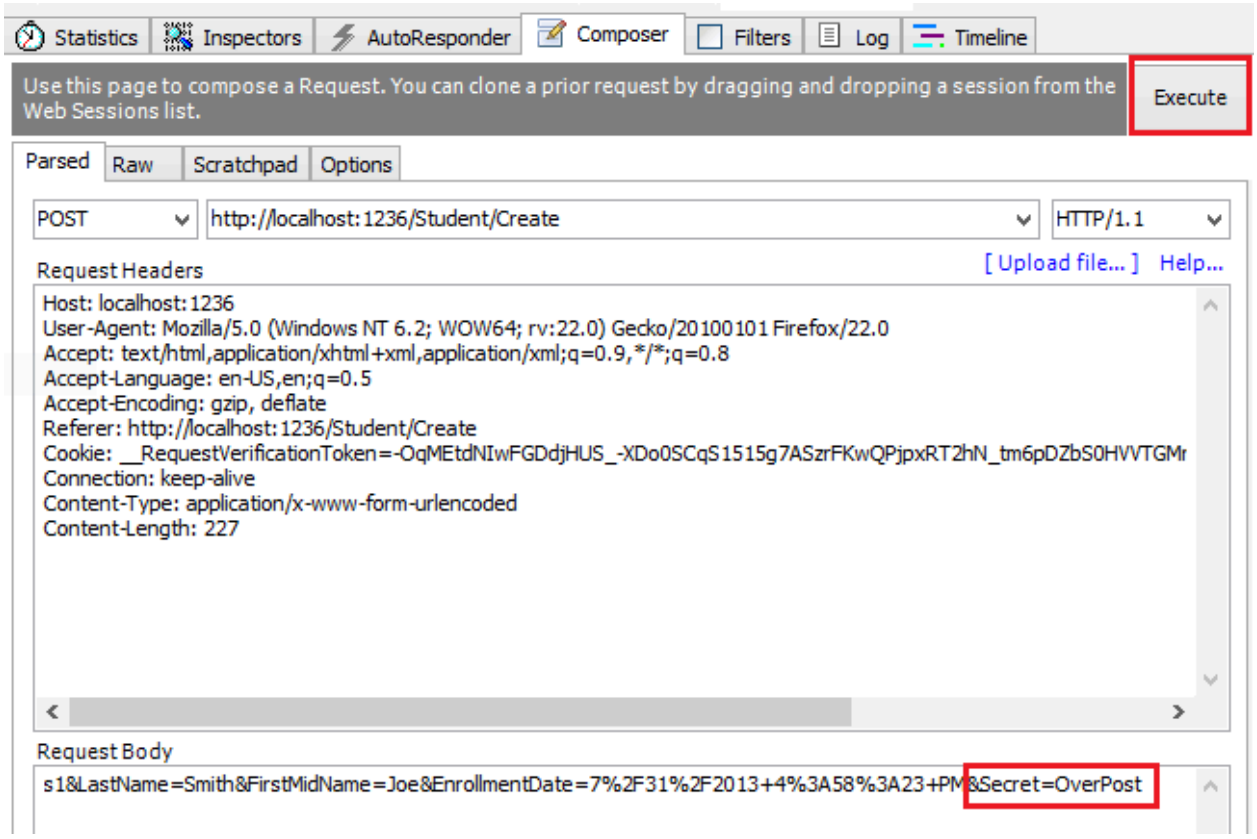
Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

C#

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field, with the value "OverPost", to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI page, for example, the Create page.

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `StudentVM` view model:

C#

```
public class StudentVM
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
```

```
    public DateTime EnrollmentDate { get; set; }
}
```

The following code uses the `StudentVM` view model to create a new student:

C#

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type:

- Doesn't need to be related to the model type.
- Needs to have properties that match.

Using `StudentVM` requires the Create page use `StudentVM` rather than `Student`:

CSHTML

```
@page
@model CreateVMModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Student</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
```

```
                <label asp-for="StudentVM.LastName" class="control-label">
</label>
                <input asp-for="StudentVM.LastName" class="form-control" />
                <span asp-validation-for="StudentVM.LastName" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.FirstMidName" class="control-
label"></label>
                <input asp-for="StudentVM.FirstMidName" class="form-control"
/>
                <span asp-validation-for="StudentVM.FirstMidName"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.EnrollmentDate" class="control-
label"></label>
                <input asp-for="StudentVM.EnrollmentDate" class="form-
control" />
                <span asp-validation-for="StudentVM.EnrollmentDate"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Edit page

In `Pages/Students/Edit.cshtml.cs`, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
```

```
    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

    if (studentToUpdate == null)
    {
        return NotFound();
    }

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When you don't have to include related data, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL `INSERT` command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an `INSERT` statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an `UPDATE` statement.
- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a `DELETE` statement.
- `Detached`: The entity isn't being tracked by the database context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL `UPDATE` statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, a custom error message is implemented when the call to `SaveChanges` fails.

Replace the code in `Pages/Students/Delete.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
```

```
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<DeleteModel> _logger;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context,
                           ILogger<DeleteModel> logger)
        {
            _context = context;
            _logger = logger;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool?
saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Student == null)
            {
                return NotFound();
            }

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = String.Format("Delete {ID} failed. Try
again", id);
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var student = await _context.Students.FindAsync(id);

            if (student == null)
            {
                return NotFound();
            }
```

```
            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException ex)
            {
                _logger.LogError(ex, ErrorMessage);

                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code:

- Adds Logging.
- Adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object.

The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is `false` when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` because the delete operation failed, the `saveChangesError` parameter is `true`.

The `OnPostAsync` method retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL `DELETE` command is generated. If `Remove` fails:

- The database exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to `Pages/Students/Delete.cshtml`:

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}
```

```
<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Part 3, Razor Pages with EF Core in ASP.NET Core - Sort, Filter, Paging

Article • 04/11/2023

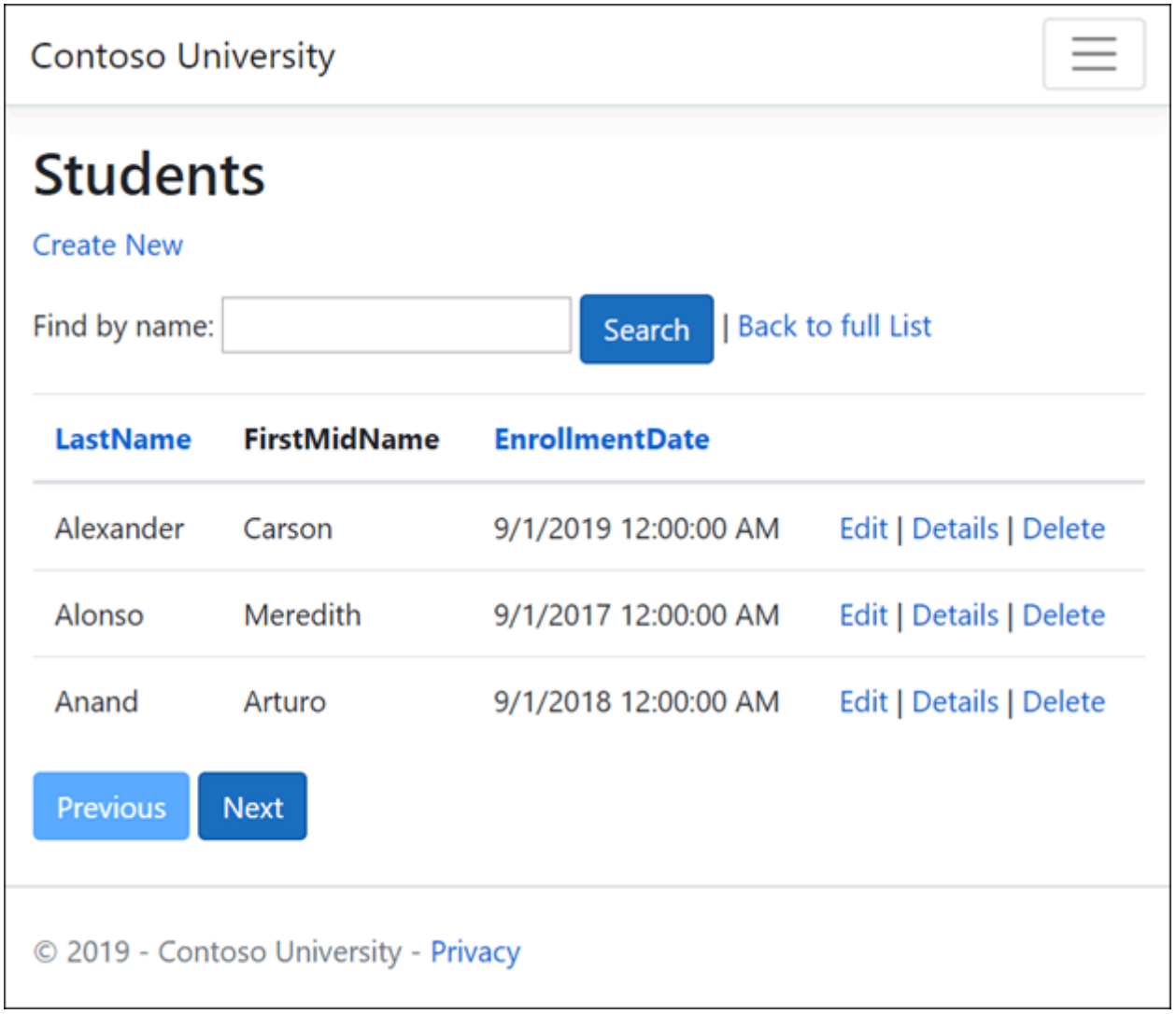
By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial adds sorting, filtering, and paging functionality to the Students pages.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Click a column heading repeatedly to switch between ascending and descending sort order.



Add sorting

Replace the code in `Pages/Students/Index.cshtml.cs` with the following code to add sorting.

C#

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;
    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder)
    {
        // using System;
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;

        switch (sortOrder)
        {
            case "name_desc":
                studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                break;
            case "Date":
                studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                break;
            case "date_desc":
                studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                break;
            default:
                studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                break;
        }

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Requires adding `using System;`.
- Adds properties to contain the sorting parameters.
- Changes the name of the `Student` property to `Students`.
- Replaces the code in the `OnGetAsync` method.

The `OnGetAsync` method receives a `sortOrder` parameter from the query string in the URL. The URL and query string is generated by the Anchor Tag Helper.

The `sortOrder` parameter is either `Name` or `Date`. The `sortOrder` parameter is optionally followed by `_desc` to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the `default` in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```
C#
```

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The code uses the C# conditional operator ?:. The `?:` operator is a ternary operator, it takes three operands. The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to `name_desc`. If `sortOrder` is ***not*** null or empty, `NameSort` is set to an empty string.

These two statements enable the page to set the column heading hyperlinks as follows:

Expand table

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

C#

```
IQueryable<Student> studentsIQ = from s in _context.Students
                                 select s;

switch (sortOrder)
{
    case "name_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
        break;
    case "Date":
        studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
        break;
    case "date_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
        break;
    default:
        studentsIQ = studentsIQ.OrderBy(s => s.LastName);
        break;
}

Students = await studentsIQ.AsNoTracking().ToListAsync();
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following statement:

C#

```
Students = await studentsIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns. For information about an alternative way to code this functionality, see Use dynamic LINQ to simplify code in the MVC version of this tutorial series.

Add column heading hyperlinks to the Student Index page

Replace the code in `Students/Index.cshtml`, with the following code. The changes are highlighted.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
```

```
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.
- Changes the page heading from Index to Students.
- Changes `Model.Student` to `Model.Students`.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click the column headings.

Add filtering

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Update the OnGetAsync method

Replace the code in `Students/Index.cshtml.cs` with the following code to add filtering:

C#

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }
```

```
    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder, string searchString)
    {
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        CurrentFilter = searchString;

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;
        if (!String.IsNullOrEmpty(searchString))
        {
            studentsIQ = studentsIQ.Where(s =>
s.LastName.Contains(searchString)
                                   ||
s.FirstMidName.Contains(searchString));
        }

        switch (sortOrder)
        {
            case "name_desc":
                studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                break;
            case "Date":
                studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                break;
            case "date_desc":
                studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                break;
            default:
                studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                break;
        }

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method, and saves the parameter value in the `CurrentFilter` property. The search string value is received from a text box that's added in the next section.
- Adds to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is

executed only if there's a value to search for.

IQueryable vs. IEnumerable

The code calls the Where method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. SQLite defaults to case-sensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
C#

Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())`
```

The preceding code would ensure that the filter is case-insensitive even if the `Where` method is called on an `IEnumerable` or runs on SQLite.

When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used.

Calling `Contains` on an `IQueryable` is usually preferable for performance reasons. With `IQueryable`, the filtering is done by the database server. If an `IEnumerable` is created first, all the rows have to be returned from the database server.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

For more information, see How to use case-insensitive query with Sqlite provider.

Update the Razor page

Replace the code in `Pages/Students/Index.cshtml` to add a **Search** button.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString"
value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
```

```
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string. If you're using SQLite, the filter is case-insensitive only if you implemented the optional `ToUpper` code shown earlier.
- Select **Search**.

Notice that the URL contains the search string. For example:

```
browser-address-bar
```

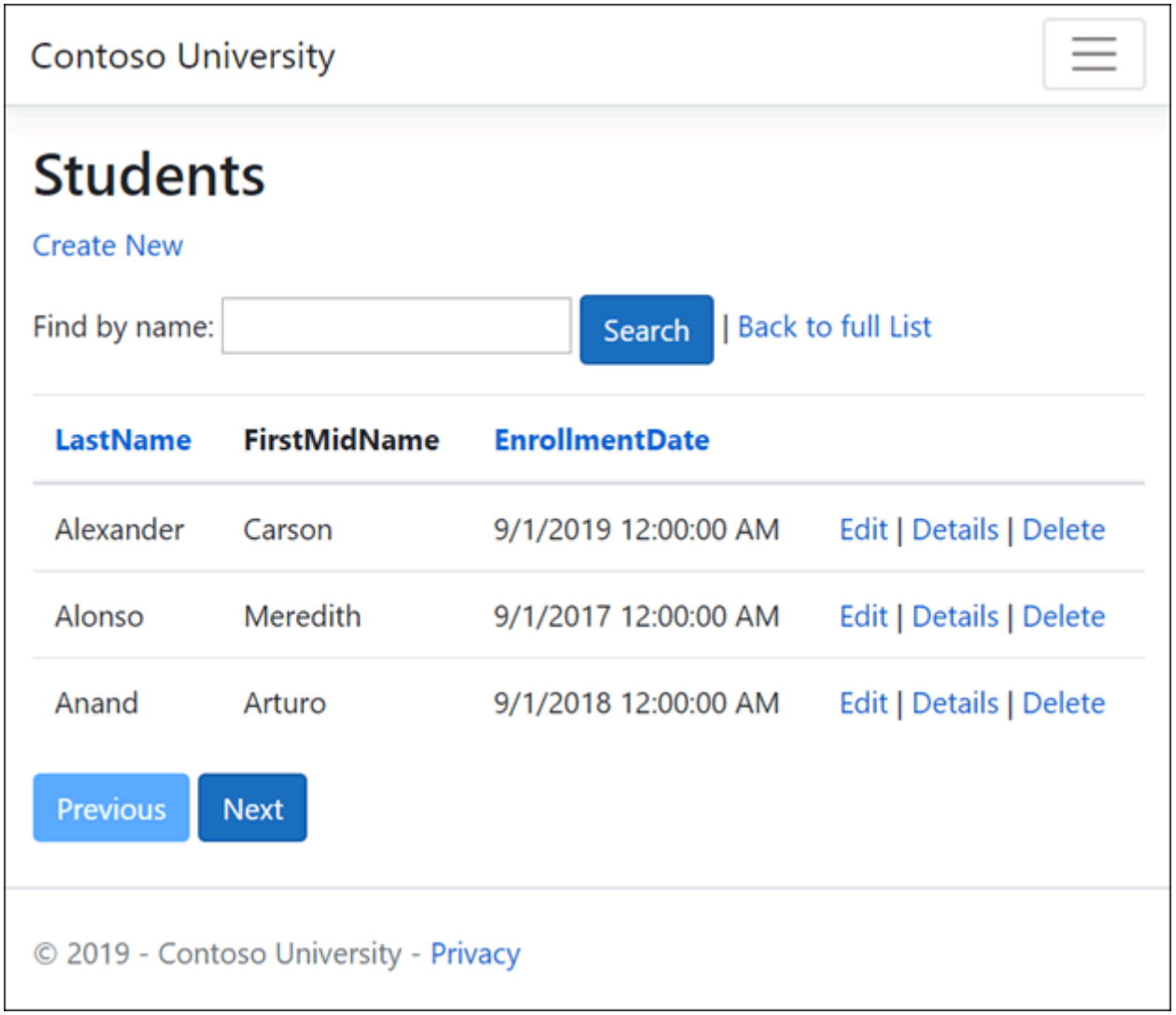
```
https://localhost:5001/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



Create the PaginatedList class

In the project folder, create `PaginatedList.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
```

```
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int
pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage => PageIndex > 1;

        public bool HasNextPage => PageIndex < TotalPages;

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object; constructors can't run asynchronous code.

Add page size to configuration

Add `PageSize` to the `appsettings.json` Configuration file:

JSON

```
{
  "PageSize": 3,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
```

```
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=(localdb)\\mssqllocaldb;Database=CU-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

Add paging to IndexModel

Replace the code in `Students/Index.cshtml.cs` to add paging.

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;
        private readonly IConfiguration Configuration;

        public IndexModel(SchoolContext context, IConfiguration
configuration)
        {
            _context = context;
            Configuration = configuration;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }
```

```
        public PaginatedList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder,
            string currentFilter, string searchString, int? pageIndex)
        {
            CurrentSort = sortOrder;
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";
            if (searchString != null)
            {
                pageIndex = 1;
            }
            else
            {
                searchString = currentFilter;
            }

            CurrentFilter = searchString;

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s =>
s.LastName.Contains(searchString)
                                       ||
s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s =>
s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            var pageSize = Configuration.GetValue("PageSize", 4);
            Students = await PaginatedList<Student>.CreateAsync(
                studentsIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
        }
    }
}
```

The preceding code:

- Changes the type of the `Students` property from `IList<Student>` to `PaginatedList<Student>`.
- Adds the page index, the current `sortOrder`, and the `currentFilter` to the `OnGetAsync` method signature.
- Saves the sort order in the `CurrentSort` property.
- Resets page index to 1 when there's a new search string.
- Uses the `PaginatedList` class to get Student entities.
- Sets `pageSize` to 3 from Configuration, 4 if configuration fails.

All the parameters that `OnGetAsync` receives are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

The `CurrentSort` property provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

The `CurrentFilter` property provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

The two question marks after `pageIndex` in the `PaginatedList.CreateAsync` call represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `pageIndex ?? 1` returns the value of `pageIndex` if it has a value, otherwise, it returns 1.

Add paging links

Replace the code in `Students/Index.cshtml` with the following code. The changes are highlighted:

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString"
value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
```

```
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

```
@{
    var prevDisabled = !Model.Students.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Students.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method:

CSHTML

```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Students[0].LastName)
</a>
```

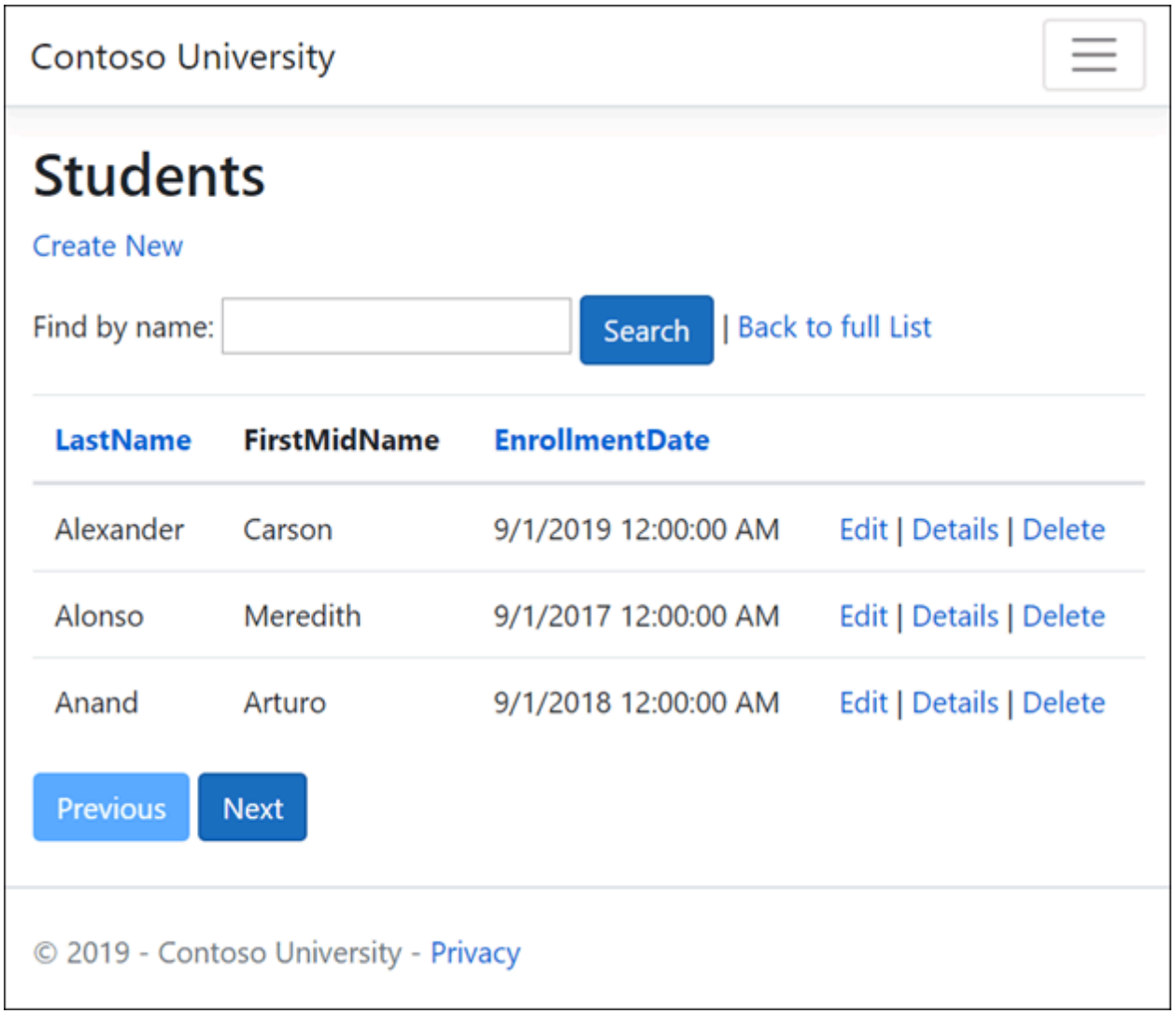
The paging buttons are displayed by tag helpers:

CSHTML

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



Grouping

This section creates an `About` page that displays how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the `About` page.
- Update the `About` page to use the view model.

Create the view model

Create a *Models/SchoolViewModels* folder.

Create `SchoolViewModels/EnrollmentDateGroup.cs` with the following code:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
```

```
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Create the Razor Page

Create a `Pages/About.cshtml` file with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Students)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Create the page model

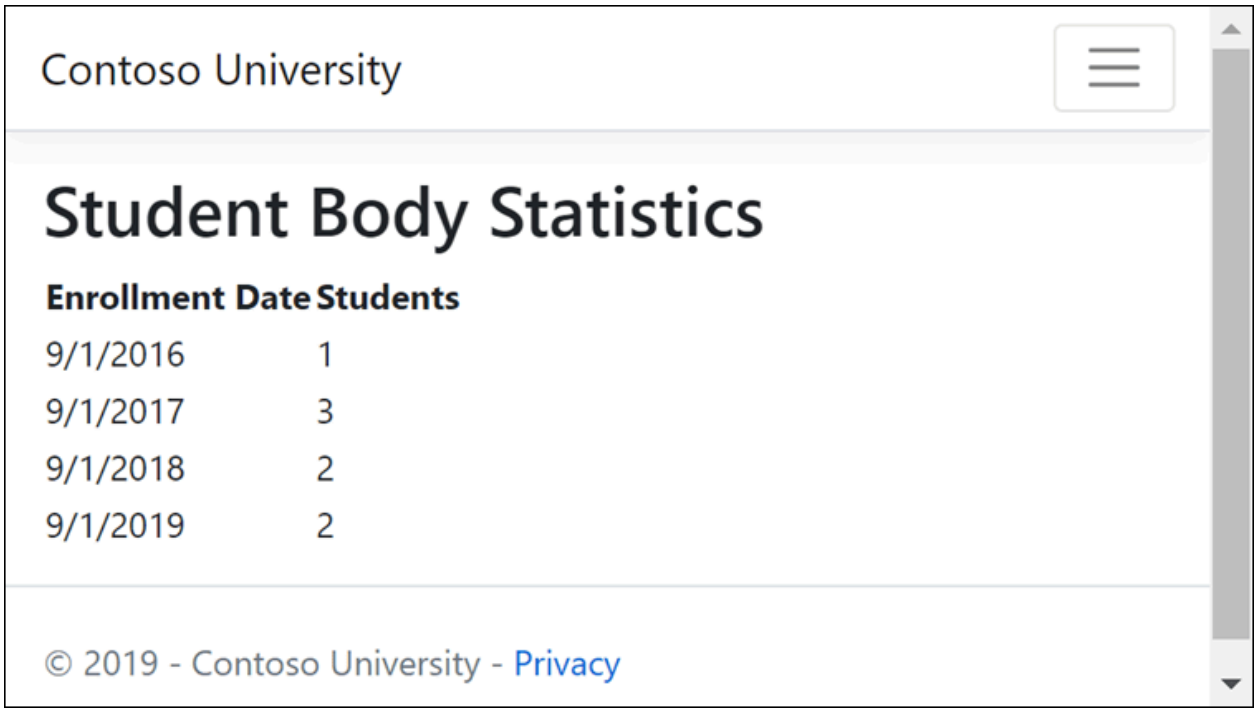
Update the `Pages/About.cshtml.cs` file with the following code:

C#

```
using ContosoUniversity.Models.SchoolViewModels;
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Students { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Students = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.



Next steps

In the next tutorial, the app uses migrations to update the data model.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 4, Razor Pages with EF Core migrations in ASP.NET Core

Article • 07/28/2023

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial introduces the EF Core migrations feature for managing data model changes.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial series started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes, the database needs to be dropped. The next time the app runs, the call to `EnsureCreated` re-creates the database to match the new data model. The `DbInitializer` class then runs to seed the new database.

This approach to keeping the DB in sync with the data model works well until the app needs to be deployed to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the DB schema instead of creating a new database.

Rather than dropping and recreating the database when the data model changes, migrations updates the schema and retains existing data.

> **Note**
>
> **SQLite limitations**
>
> This tutorial uses the Entity Framework Core migrations feature where possible. Migrations updates the database schema to match changes in the data model. However, migrations only does the kinds of changes that the database engine supports, and SQLite's schema change capabilities are limited. For example, adding

a column is supported, but removing a column is not supported. If a migration is created to remove a column, the `ef migrations add` command succeeds but the `ef database update` command fails.

The workaround for the SQLite limitations is to manually write migrations code to perform a table rebuild when something in the table changes. The code goes in the `Up` and `Down` methods for a migration and involves:

- Creating a new table.
- Copying data from the old table to the new table.
- Dropping the old table.
- Renaming the new table.

Writing database-specific code of this type is outside the scope of this tutorial. Instead, this tutorial drops and re-creates the database whenever an attempt to apply a migration would fail. For more information, see the following resources:

- SQLite EF Core Database Provider Limitations
- Customize migration code
- Data seeding
- SQLite ALTER TABLE statement

Drop the database

Visual Studio

Use **SQL Server Object Explorer** (SSOX) to delete the database, or run the following command in the **Package Manager Console** (PMC):

PowerShell

```
Drop-Database
```

Create an initial migration

Visual Studio

Run the following commands in the PMC:

PowerShell

```
Add-Migration InitialCreate
Update-Database
```

Remove EnsureCreated

This tutorial series started by using EnsureCreated. `EnsureCreated` doesn't create a migrations history table and so can't be used with migrations. It's designed for testing or rapid prototyping where the database is dropped and re-created frequently.

From this point forward, the tutorials will use migrations.

In `Program.cs`, delete the following line:

C#

```
context.Database.EnsureCreated();
```

Run the app and verify that the database is seeded.

Up and Down methods

The EF Core `migrations add` command generated code to create the database. This migrations code is in the `Migrations\<timestamp>_InitialCreate.cs` file. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

C#

```
using System;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ContosoUniversity.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Course",
                columns: table => new
                {
```

```
                    CourseID = table.Column<int>(nullable: false),
                    Title = table.Column<string>(nullable: true),
                    Credits = table.Column<int>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Course", x => x.CourseID);
                });

            migrationBuilder.CreateTable(
                name: "Student",
                columns: table => new
                {
                    ID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    LastName = table.Column<string>(nullable: true),
                    FirstMidName = table.Column<string>(nullable: true),
                    EnrollmentDate = table.Column<DateTime>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Student", x => x.ID);
                });

            migrationBuilder.CreateTable(
                name: "Enrollment",
                columns: table => new
                {
                    EnrollmentID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    CourseID = table.Column<int>(nullable: false),
                    StudentID = table.Column<int>(nullable: false),
                    Grade = table.Column<int>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Enrollment", x => x.EnrollmentID);
                    table.ForeignKey(
                        name: "FK_Enrollment_Course_CourseID",
                        column: x => x.CourseID,
                        principalTable: "Course",
                        principalColumn: "CourseID",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_Enrollment_Student_StudentID",
                        column: x => x.StudentID,
                        principalTable: "Student",
                        principalColumn: "ID",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_CourseID",
```

```
                table: "Enrollment",
                column: "CourseID");

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_StudentID",
                table: "Enrollment",
                column: "StudentID");
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Enrollment");

            migrationBuilder.DropTable(
                name: "Course");

            migrationBuilder.DropTable(
                name: "Student");
        }
    }
}
```

The preceding code is for the initial migration. The code:

- Was generated by the `migrations add InitialCreate` command.
- Is executed by the `database update` command.
- Creates a database for the data model specified by the database context class.

The migration name parameter (`InitialCreate` in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

The migrations history table

- Use SSOX or SQLite tool to inspect the database.
- Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the database.
- View the data in the `__EFMigrationsHistory` table. It shows one row for the first migration.

The data model snapshot

Migrations creates a *snapshot* of the current data model in `Migrations/SchoolContextModelSnapshot.cs`. When add a migration is added, EF determines what changed by comparing the current data model to the snapshot file.

Because the snapshot file tracks the state of the data model, a migration cannot be deleted by deleting the `<timestamp>_<migrationname>.cs` file. To back out the most recent migration, use the migrations remove command. `migrations remove` deletes the migration and ensures the snapshot is correctly reset. For more information, see dotnet ef migrations remove.

See Resetting all migrations to remove all migrations.

Applying migrations in production

We recommend that production apps **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app that is deployed to a server farm. If the app is scaled out to multiple server instances, it's hard to ensure database schema updates don't happen from multiple servers or conflict with read/write access.

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

Troubleshooting

If the app uses SQL Server LocalDB and displays the following exception:

```text
SqlException: Cannot open database "ContosoUniversity" requested by the
login.
The login failed.
Login failed for user 'user name'.
```

The solution may be to run `dotnet ef database update` at a command prompt.

Additional resources

- EF Core CLI.
- dotnet ef migrations CLI commands

- Package Manager Console (Visual Studio)

Next steps

The next tutorial builds out the data model, adding entity properties and new entities.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 5, Razor Pages with EF Core in ASP.NET Core - Data Model

Article • 04/11/2023

By Tom Dykstra, Jeremy Likness, and Jon P Smith

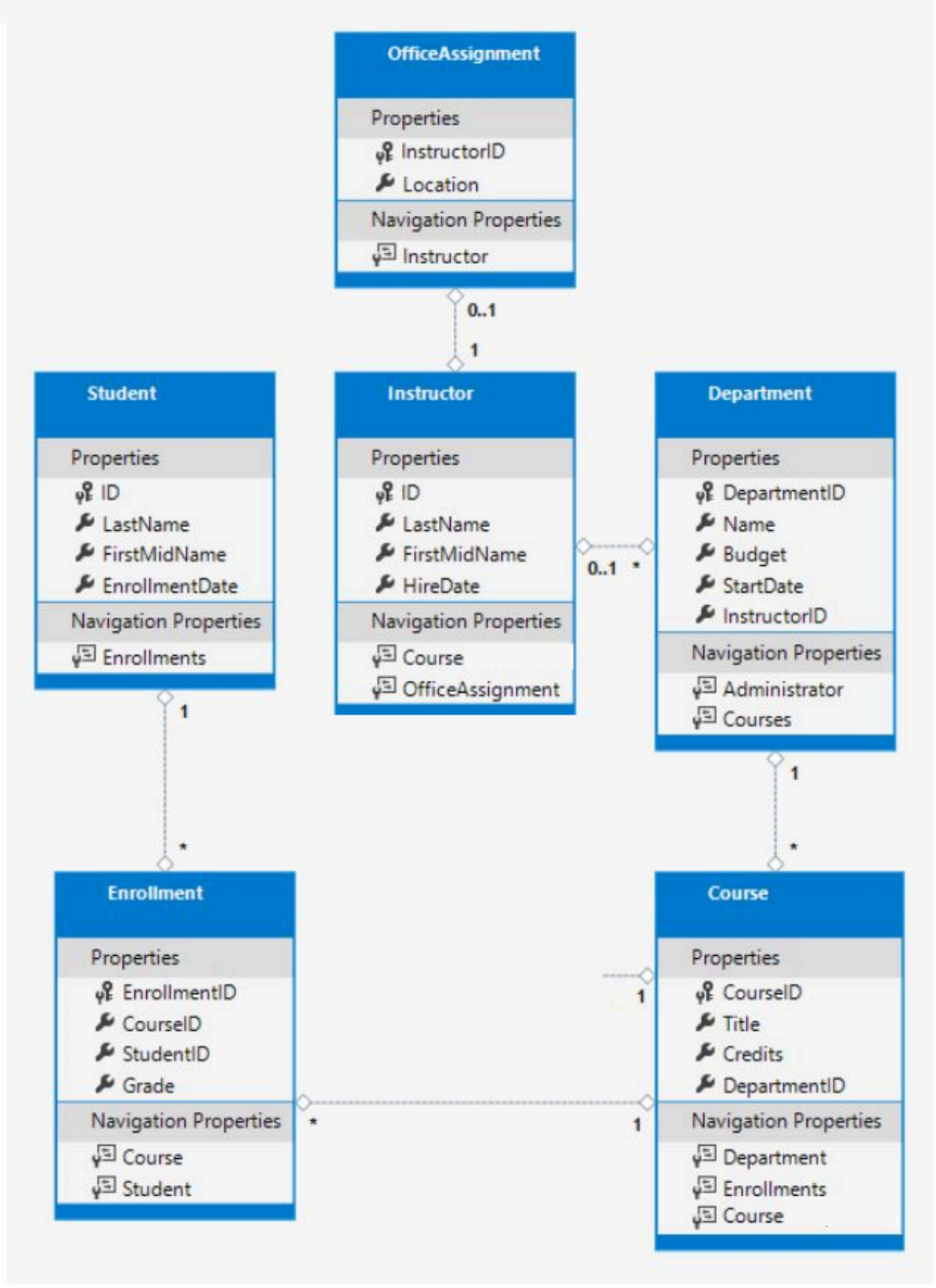
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

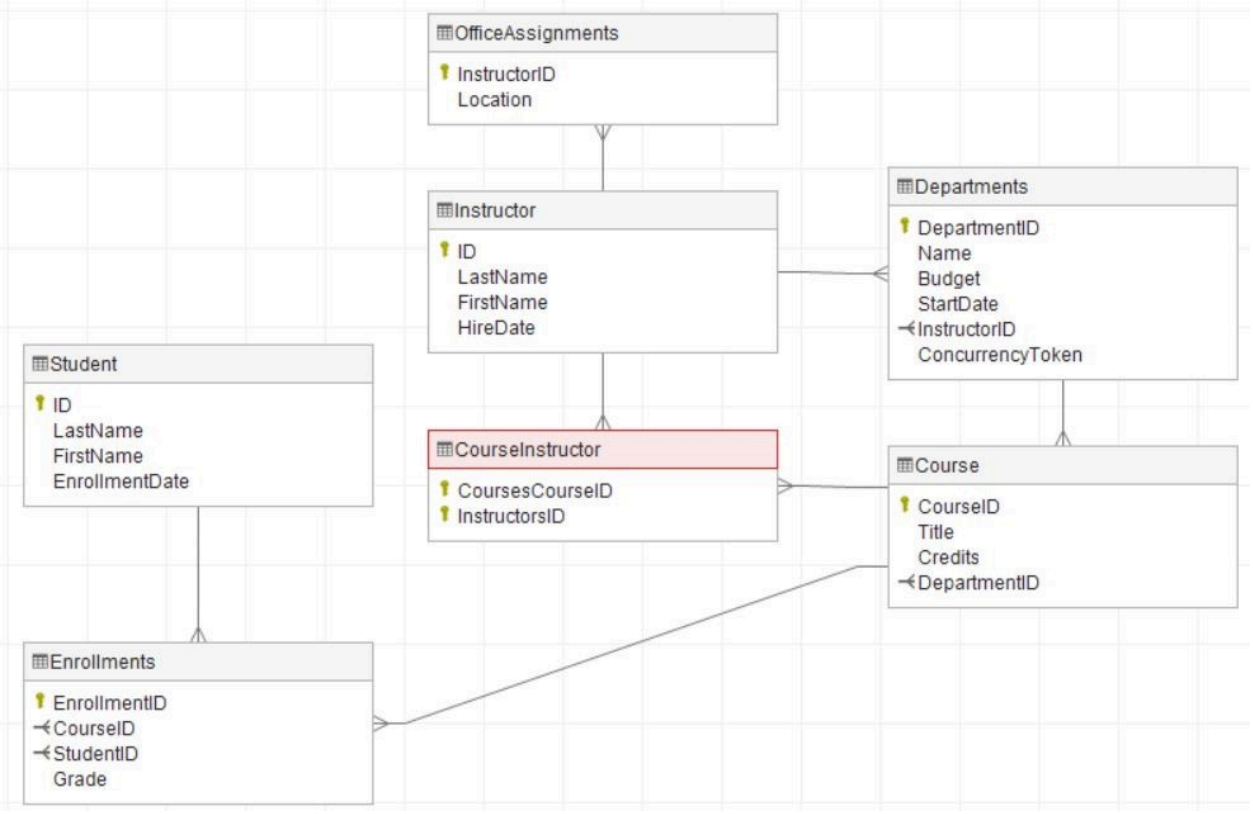
The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The completed data model is shown in the following illustration:



The following database diagram was made with Dataedo:



To create a database diagram with Dataedo:

- Deploy the app to Azure
- Download and install Dataedo on your computer.
- Follow the instructions Generate documentation for Azure SQL Database in 5 minutes

In the preceding Dataedo diagram, the `CourseInstructor` is a join table created by Entity Framework. For more information, see Many-to-many

The Student entity

Replace the code in `Models/Student.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
```

```
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than
50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code adds a `FullName` property and adds the following attributes to existing properties:

- [DataType]
- [DisplayFormat]
- [StringLength]
- [Column]
- [Required]
- [Display]

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` can't be set, so it has only a get accessor. No `FullName` column is created in the database.

The DataType attribute

```
C#
```

```
[DataType(DataType.Date)]
```

For student enrollment dates, all of the pages currently display the time of day along with the date, although only the date is relevant. By using data annotation attributes, you can make one code change that will fix the display format in every page that shows the data.

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes. The `DataType` attributes don't provide validation.

The DisplayFormat attribute

C#

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
```

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format. The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, and client-side input validation.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

The StringLength attribute

C#

```
[StringLength(50, ErrorMessage = "First name cannot be longer than 50
characters.")]
```

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The code shown limits names to no more than 50 characters. An example that sets the minimum string length is shown later.

The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

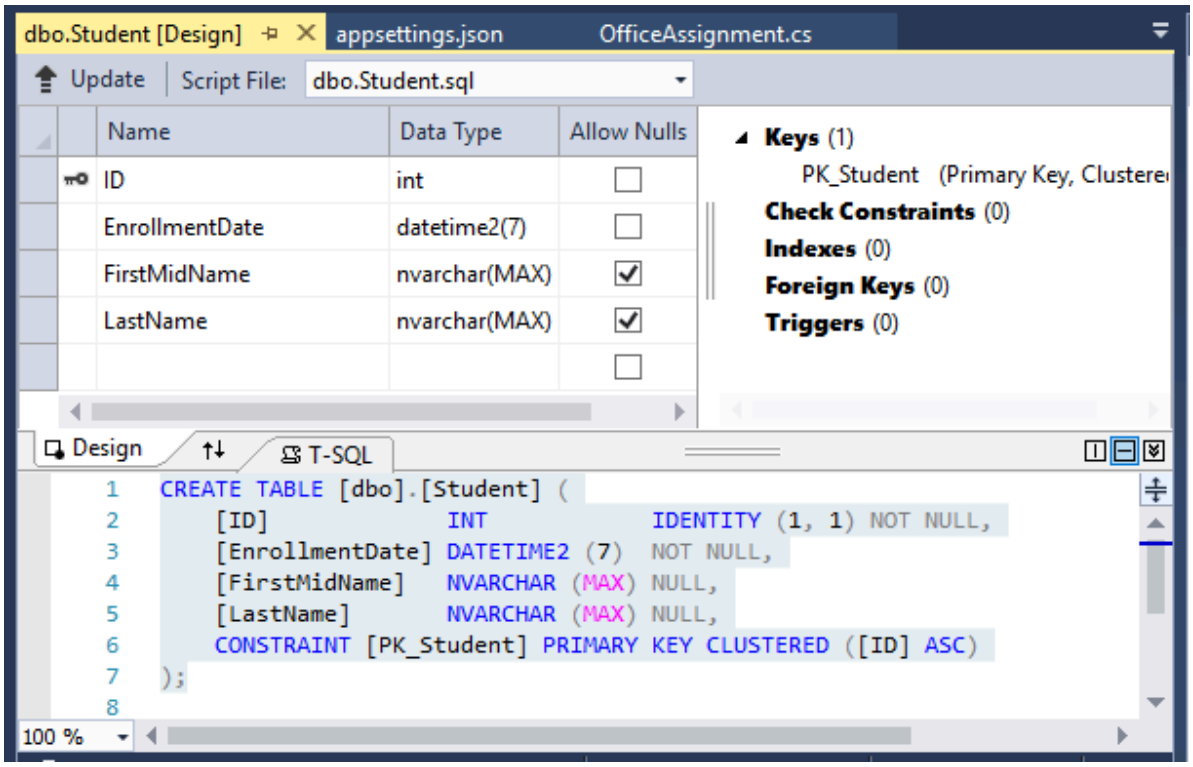
The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute can be used to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

C#

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

Visual Studio

In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)`. When a migration is created and applied later in this tutorial, the name fields become `nvarchar(50)` as a result of the string length attributes.

The Column attribute

C#

```
[Column("FirstName")]
public string FirstMidName { get; set; }
```

Attributes can control how classes and properties are mapped to the database. In the `Student` model, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the database.

When the database is created, property names on the model are used for column names (except when the `Column` attribute is used). The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

With the `[Column]` attribute, `Student.FirstMidName` in the data model maps to the `FirstName` column of the `Student` table. The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. That discrepancy will be resolved by adding a migration later in this tutorial.

The Required attribute

```
C#

[Required]
```

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (for example, `DateTime`, `int`, and `double`). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
C#

[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for full control over the string.

The Display attribute

```
C#

[Display(Name = "Last Name")]
```

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

Create a migration

Run the app and go to the Students page. An exception is thrown. The `[Column]` attribute causes EF to expect to find a column named `FirstName`, but the column name in the database is still `FirstMidName`.

Visual Studio

The error message is similar to the following example:

```
SqlException: Invalid column name 'FirstName'.
There are pending model changes
Pending model changes are detected in the following:

SchoolContext
```

- In the PMC, enter the following commands to create a new migration and update the database:

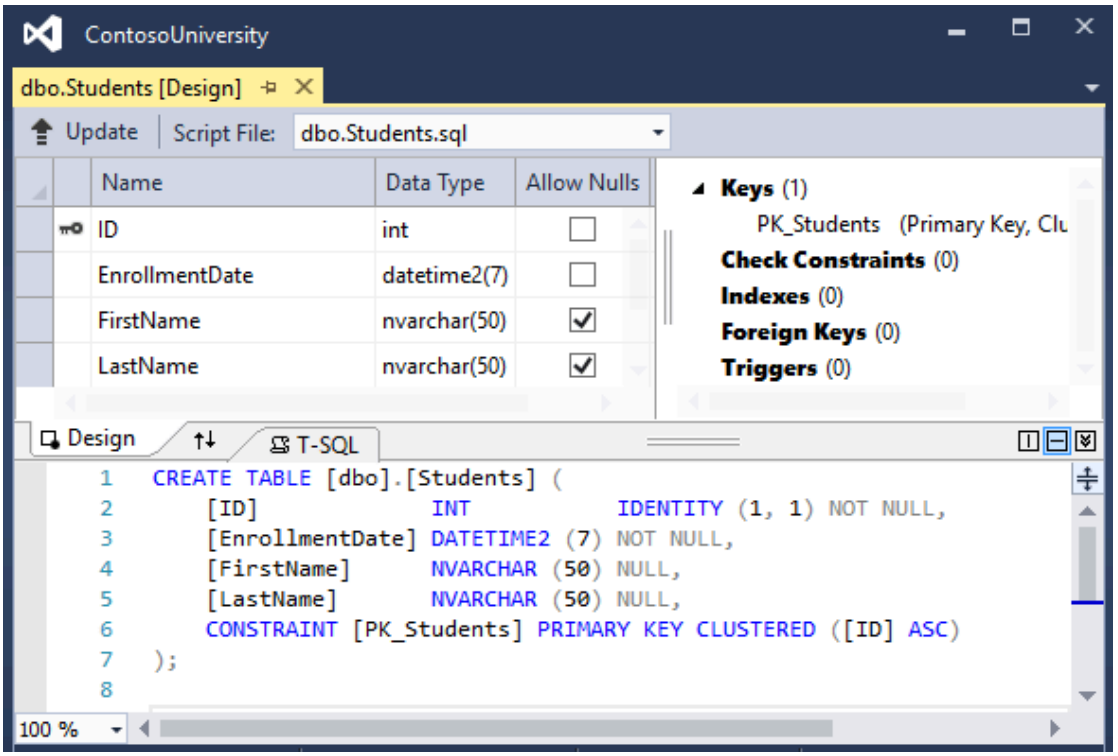
  ```PowerShell
  Add-Migration ColumnFirstName
  Update-Database
  ```

  The first of these commands generates the following warning message:

  ```text
  An operation was scaffolded that may result in the loss of data.
  Please review the migration for accuracy.
  ```

  The warning is generated because the name fields are now limited to 50 characters. If a name in the database had more than 50 characters, the 51 to last character would be lost.

- Open the Student table in SSOX:



Before the migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

- Run the app and go to the Students page.
- Notice that times are not input or displayed along with dates.
- Select **Create New**, and try to enter a name longer than 50 characters.

> ⓘ Note
>
> In the following sections, building the app at some stages generates compiler errors. The instructions specify when to build the app.

The Instructor Entity

Create `Models/Instructor.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
```

```
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<Course> Courses { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

C#

```
[DataType(DataType.Date),Display(Name = "Hire
Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

Navigation properties

The `Courses` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `Courses` is defined as a collection.

C#

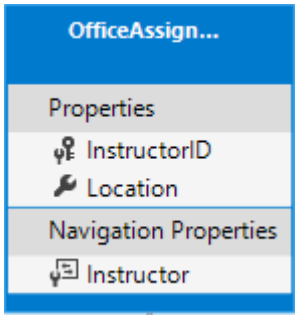
```
public ICollection<Course> Courses { get; set; }
```

An instructor can have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

C#

```
public OfficeAssignment OfficeAssignment { get; set; }
```

The OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The [Key] attribute is used to identify a property as the primary key (PK) when the property name is something other than `classnameID` or `ID`.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity. A one-to-zero-or-one relationship occurs when a PK in one table is both a PK and a FK in another table.

EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because `InstructorID` doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

C#

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship. For more information, see EF Keys.

The Instructor navigation property

The `Instructor.OfficeAssignment` navigation property can be null because there might not be an `OfficeAssignment` row for a given instructor. An instructor might not have an office assignment.

The `OfficeAssignment.Instructor` navigation property will always have an instructor entity because the foreign key `InstructorID` type is `int`, a non-nullable value type. An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The Course Entity

Update `Models/Course.cs` with the following code:

C#

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
```

```
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<Instructor> Instructors { get; set; }
    }
}
```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a foreign key property for a data model when the model has a navigation property for a related entity. EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. However, explicitly including the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is ***not*** included. When a course entity is fetched to edit:

- The `Department` property is `null` if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

C#

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, EF Core assumes that PK values are generated by the database. Database-generated is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the database can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

C#

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

C#

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `Instructors` navigation property is a collection:

C#

```
public ICollection<Instructor> Instructors { get; set; }
```

The Department entity

Create `Models/Department.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
                       ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the database:

C#

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required. EF Core chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

C#

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

The `?` in the preceding code specifies the property is nullable.

A department may have many courses, so there's a Courses navigation property:

C#

```
public ICollection<Course> Courses { get; set; }
```

By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. This default behavior can result in circular cascade delete rules. Circular cascade delete rules cause an exception when a migration is added.

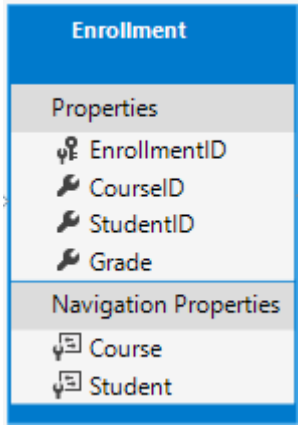
For example, if the `Department.InstructorID` property was defined as non-nullable, EF Core would configure a cascade delete rule. In that case, the department would be deleted when the instructor assigned as its administrator is deleted. In this scenario, a restrict rule would make more sense. The following fluent API would set a restrict rule and disable cascade delete.

C#

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

The Enrollment foreign key and navigation properties

An enrollment record is for one course taken by one student.



Update `Models/Enrollment.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

C#

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

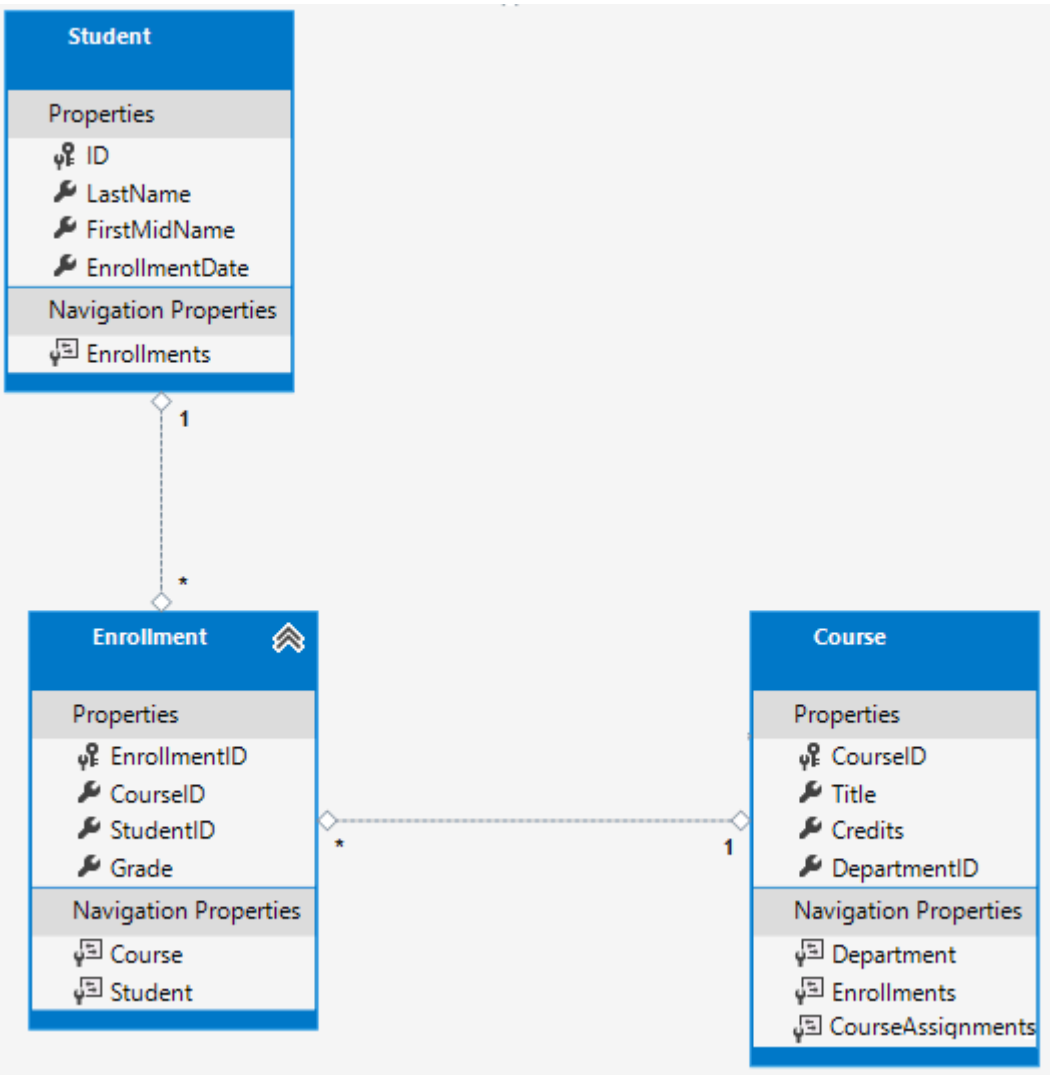
C#

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities. The `Enrollment` entity functions as a many-to-many join table ***with payload*** in the database. ***With payload*** means that the `Enrollment` table contains additional data besides FKs for the joined tables. In the `Enrollment` entity, the additional data besides FKs are the PK and `Grade`.

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs, `CourseID` and `StudentID`. A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a PJT.

Update the database context

Update `Data/SchoolContext.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable(nameof(Course))
                .HasMany(c => c.Instructors)
                .WithMany(i => i.Courses);
            modelBuilder.Entity<Student>().ToTable(nameof(Student));
            modelBuilder.Entity<Instructor>().ToTable(nameof(Instructor));
        }
    }
}
```

The preceding code adds the new entities and configures the many-to-many relationship between the `Instructor` and `Course` entities.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

```
C#
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

In this tutorial, the fluent API is used only for database mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes *clean*. Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API, for example, specifying a composite PK. There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

Some of the attributes used in this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Seed the database

Update the code in `Data/DbInitializer.cs`:

```
C#
```

```
using ContosoUniversity.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var alexander = new Student
            {
                FirstMidName = "Carson",
                LastName = "Alexander",
                EnrollmentDate = DateTime.Parse("2016-09-01")
            };

            var alonso = new Student
            {
                FirstMidName = "Meredith",
                LastName = "Alonso",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var anand = new Student
            {
                FirstMidName = "Arturo",
                LastName = "Anand",
                EnrollmentDate = DateTime.Parse("2019-09-01")
            };

            var barzdukas = new Student
            {
                FirstMidName = "Gytis",
                LastName = "Barzdukas",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var li = new Student
            {
                FirstMidName = "Yan",
                LastName = "Li",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var justice = new Student
            {
```

```
            FirstMidName = "Peggy",
            LastName = "Justice",
            EnrollmentDate = DateTime.Parse("2017-09-01")
        };

        var norman = new Student
        {
            FirstMidName = "Laura",
            LastName = "Norman",
            EnrollmentDate = DateTime.Parse("2019-09-01")
        };

        var olivetto = new Student
        {
            FirstMidName = "Nino",
            LastName = "Olivetto",
            EnrollmentDate = DateTime.Parse("2011-09-01")
        };

        var students = new Student[]
        {
            alexander,
            alonso,
            anand,
            barzdukas,
            li,
            justice,
            norman,
            olivetto
        };

        context.AddRange(students);

        var abercrombie = new Instructor
        {
            FirstMidName = "Kim",
            LastName = "Abercrombie",
            HireDate = DateTime.Parse("1995-03-11")
        };

        var fakhouri = new Instructor
        {
            FirstMidName = "Fadi",
            LastName = "Fakhouri",
            HireDate = DateTime.Parse("2002-07-06")
        };

        var harui = new Instructor
        {
            FirstMidName = "Roger",
            LastName = "Harui",
            HireDate = DateTime.Parse("1998-07-01")
        };

        var kapoor = new Instructor
```

```
            {
                FirstMidName = "Candace",
                LastName = "Kapoor",
                HireDate = DateTime.Parse("2001-01-15")
            };

            var zheng = new Instructor
            {
                FirstMidName = "Roger",
                LastName = "Zheng",
                HireDate = DateTime.Parse("2004-02-12")
            };

            var instructors = new Instructor[]
            {
                abercrombie,
                fakhouri,
                harui,
                kapoor,
                zheng
            };

            context.AddRange(instructors);

            var officeAssignments = new OfficeAssignment[]
            {
                new OfficeAssignment {
                    Instructor = fakhouri,
                    Location = "Smith 17" },
                new OfficeAssignment {
                    Instructor = harui,
                    Location = "Gowan 27" },
                new OfficeAssignment {
                    Instructor = kapoor,
                    Location = "Thompson 304" }
            };

            context.AddRange(officeAssignments);

            var english = new Department
            {
                Name = "English",
                Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = abercrombie
            };

            var mathematics = new Department
            {
                Name = "Mathematics",
                Budget = 100000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = fakhouri
            };
```

```
            var engineering = new Department
            {
                Name = "Engineering",
                Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = harui
            };

            var economics = new Department
            {
                Name = "Economics",
                Budget = 100000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = kapoor
            };

            var departments = new Department[]
            {
                english,
                mathematics,
                engineering,
                economics
            };

            context.AddRange(departments);

            var chemistry = new Course
            {
                CourseID = 1050,
                Title = "Chemistry",
                Credits = 3,
                Department = engineering,
                Instructors = new List<Instructor> { kapoor, harui }
            };

            var microeconomics = new Course
            {
                CourseID = 4022,
                Title = "Microeconomics",
                Credits = 3,
                Department = economics,
                Instructors = new List<Instructor> { zheng }
            };

            var macroeconmics = new Course
            {
                CourseID = 4041,
                Title = "Macroeconomics",
                Credits = 3,
                Department = economics,
                Instructors = new List<Instructor> { zheng }
            };

            var calculus = new Course
            {
```

```
            CourseID = 1045,
            Title = "Calculus",
            Credits = 4,
            Department = mathematics,
            Instructors = new List<Instructor> { fakhouri }
        };

        var trigonometry = new Course
        {
            CourseID = 3141,
            Title = "Trigonometry",
            Credits = 4,
            Department = mathematics,
            Instructors = new List<Instructor> { harui }
        };

        var composition = new Course
        {
            CourseID = 2021,
            Title = "Composition",
            Credits = 3,
            Department = english,
            Instructors = new List<Instructor> { abercrombie }
        };

        var literature = new Course
        {
            CourseID = 2042,
            Title = "Literature",
            Credits = 4,
            Department = english,
            Instructors = new List<Instructor> { abercrombie }
        };

        var courses = new Course[]
        {
            chemistry,
            microeconomics,
            macroeconmics,
            calculus,
            trigonometry,
            composition,
            literature
        };

        context.AddRange(courses);

        var enrollments = new Enrollment[]
        {
            new Enrollment {
                Student = alexander,
                Course = chemistry,
                Grade = Grade.A
            },
            new Enrollment {
```

```
                    Student = alexander,
                    Course = microeconomics,
                    Grade = Grade.C
                },
                new Enrollment {
                    Student = alexander,
                    Course = macroeconmics,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = alonso,
                    Course = calculus,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = alonso,
                    Course = trigonometry,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = alonso,
                    Course = composition,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = anand,
                    Course = chemistry
                },
                new Enrollment {
                    Student = anand,
                    Course = microeconomics,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = barzdukas,
                    Course = chemistry,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = li,
                    Course = composition,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = justice,
                    Course = literature,
                    Grade = Grade.B
                }
            };

            context.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing.

Apply the migration or drop and re-create

With the existing database, there are two approaches to changing the database:

- Drop and re-create the database. Choose this section when using SQLite.
- Apply the migration to the existing database. The instructions in this section work for SQL Server only, ***not for SQLite***.

Either choice works for SQL Server. While the apply-migration method is more complex and time-consuming, it's the preferred approach for real-world, production environments.

Drop and re-create the database

To force EF Core to create a new database, drop and update the database:

Visual Studio

- Delete the *Migrations* folder.
- In the **Package Manager Console** (PMC), run the following commands:

PowerShell

```
Drop-Database
Add-Migration InitialCreate
Update-Database
```

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

Visual Studio

Open the database in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.

Next steps

The next two tutorials show how to read and update related data.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 6, Razor Pages with EF Core in ASP.NET Core - Read Related Data

Article • 04/11/2023

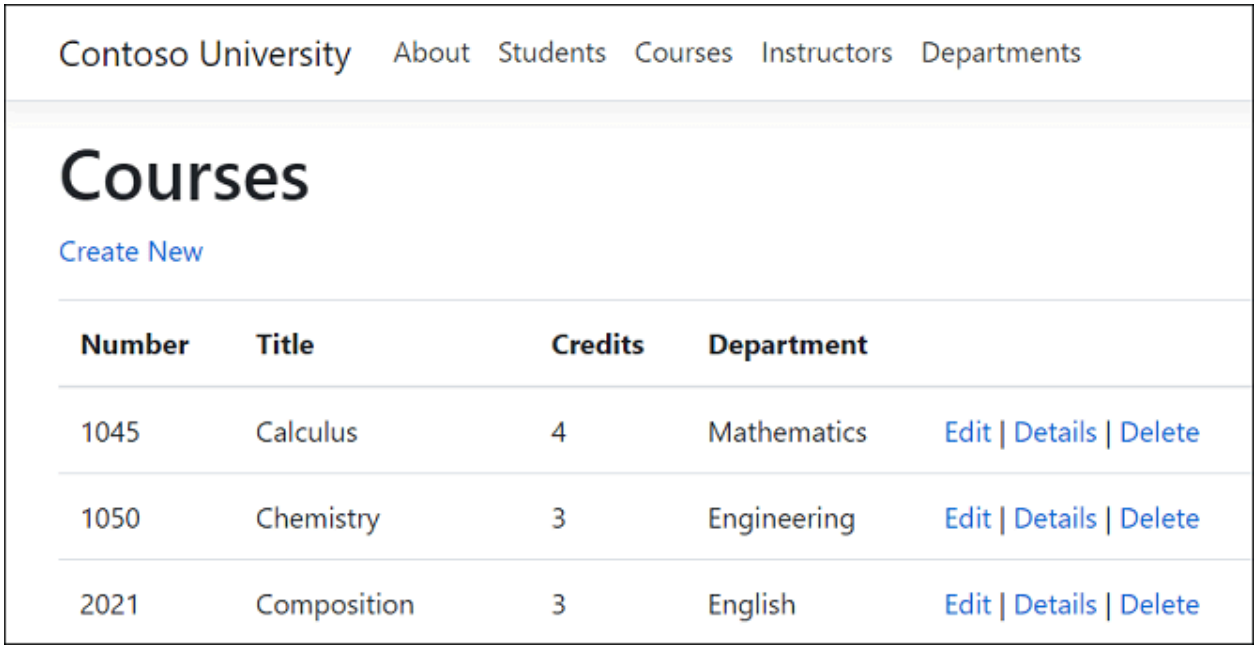
By Tom Dykstra, Jon P Smith, and Rick Anderson

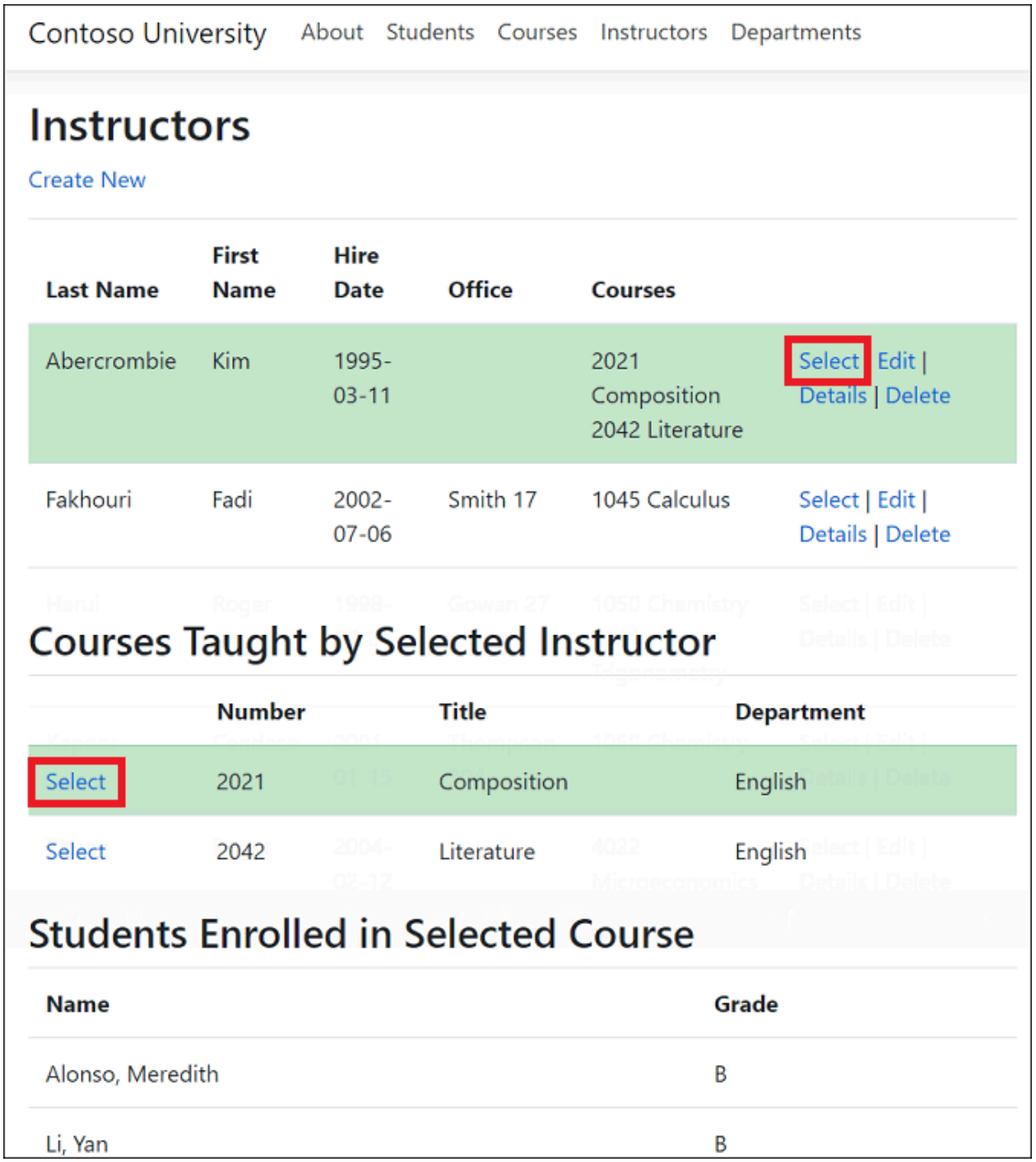
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to read and display related data. Related data is data that EF Core loads into navigation properties.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy loading

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When an entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than a large single query. Eager loading is specified with the Include and ThenInclude methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

Eager loading sends multiple queries when a collection navigation is included:

- One query for the main query
- One query for each collection "edge" in the load tree.

- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "Fixes up" means that EF Core automatically populates the navigation properties. Separate queries with `Load` is more like explicit loading than eager loading.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the database. With explicit loading, the code specifies the navigation properties to be loaded. Use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

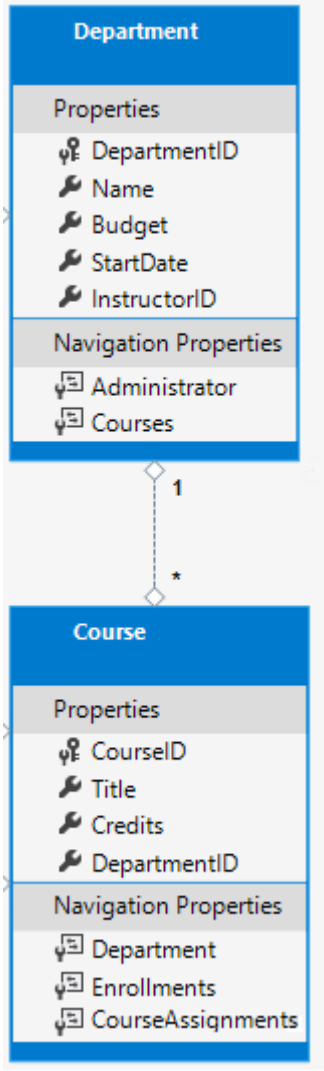
Query: Course rows related to Department d

- Lazy loading. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the database each time a navigation property is accessed for the first time. Lazy loading can hurt performance, for

example when developers use N+1 queries. N+1 queries load a parent and enumerate through children.

Create Course pages

The `Course` entity includes a navigation property that contains the related `Department` entity.



To display the name of the assigned department for a course:

- Load the related `Department` entity into the `Course.Department` navigation property.
- Get the name from the `Department` entity's `Name` property.

Scaffold Course pages

Visual Studio

- Follow the instructions in Scaffold Student pages with the following exceptions:
  - Create a *Pages/Courses* folder.
  - Use `Course` for the model class.
  - Use the existing context class instead of creating a new one.

- Open `Pages/Courses/Index.cshtml.cs` and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

- Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Display the department name

Update Pages/Courses/Index.cshtml.cs with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IList<Course> Courses { get; set; }

        public async Task OnGetAsync()
        {
            Courses = await _context.Courses
                .Include(c => c.Department)
                .AsNoTracking()
                .ToListAsync();
        }
```

```
    }
}
```

The preceding code changes the `Course` property to `Courses` and adds `AsNoTracking`.

No-tracking queries are useful when the results are used in a read-only scenario. They're generally quicker to execute because there's no need to set up the change tracking information. If the entities retrieved from the database don't need to be updated, then a no-tracking query is likely to perform better than a tracking query.

In some cases a tracking query is more efficient than a no-tracking query. For more information, see Tracking vs. No-Tracking Queries. In the preceding code, `AsNoTracking` is called because the entities aren't updated in the current context.

Update `Pages/Courses/Index.cshtml` with the following code.

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel

@{
    ViewData["Title"] = "Courses";
}

<h1>Courses</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Courses)
{
```

```
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.CourseID)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Credits)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Department.Name)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a>
|
                <a asp-page="./Details" asp-route-
id="@item.CourseID">Details</a> |
                <a asp-page="./Delete" asp-route-
id="@item.CourseID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

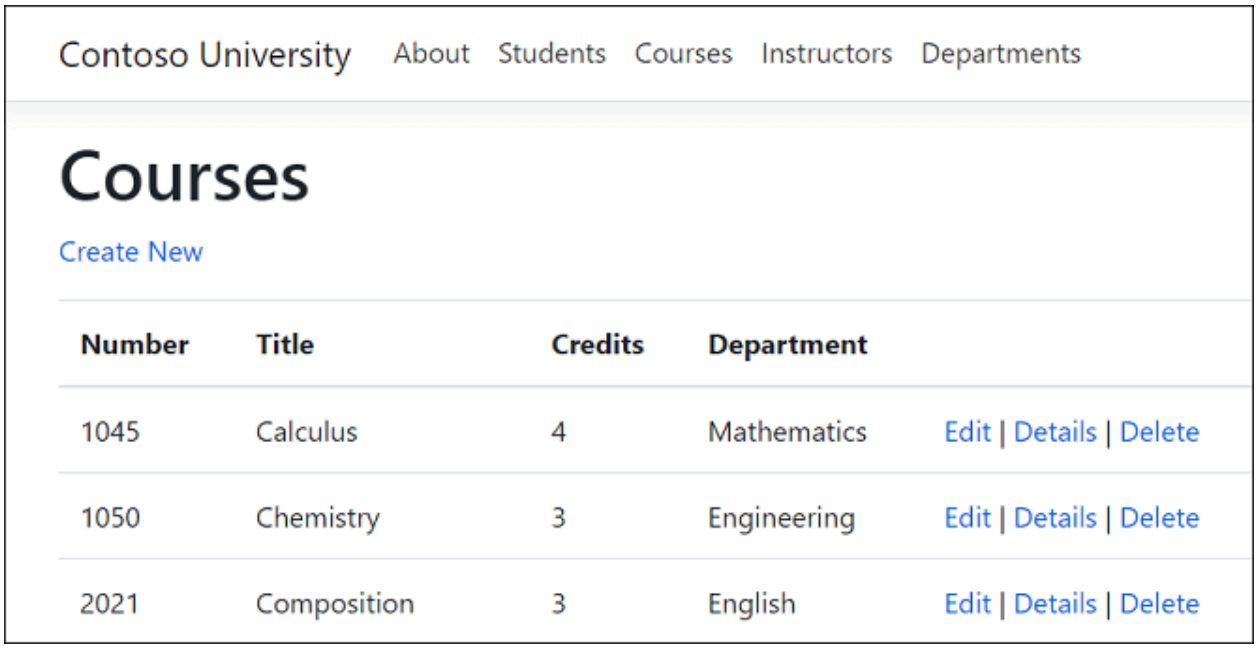
The following changes have been made to the scaffolded code:

- Changed the `Course` property name to `Courses`.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

  HTML

  ```
  @Html.DisplayFor(modelItem => item.Department.Name)
  ```

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method. The `Select` method is an alternative that loads only the related data needed. For single items, like the `Department.Name` it uses a `SQL INNER JOIN`. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

C#

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
    .Select(p => new CourseViewModel
    {
        CourseID = p.CourseID,
        Title = p.Title,
        Credits = p.Credits,
        DepartmentName = p.Department.Name
    }).ToListAsync();
}
```

The preceding code doesn't return any entity types, therefore no tracking is done. For more information about the EF tracking, see Tracking vs. No-Tracking Queries.

The `CourseViewModel`:

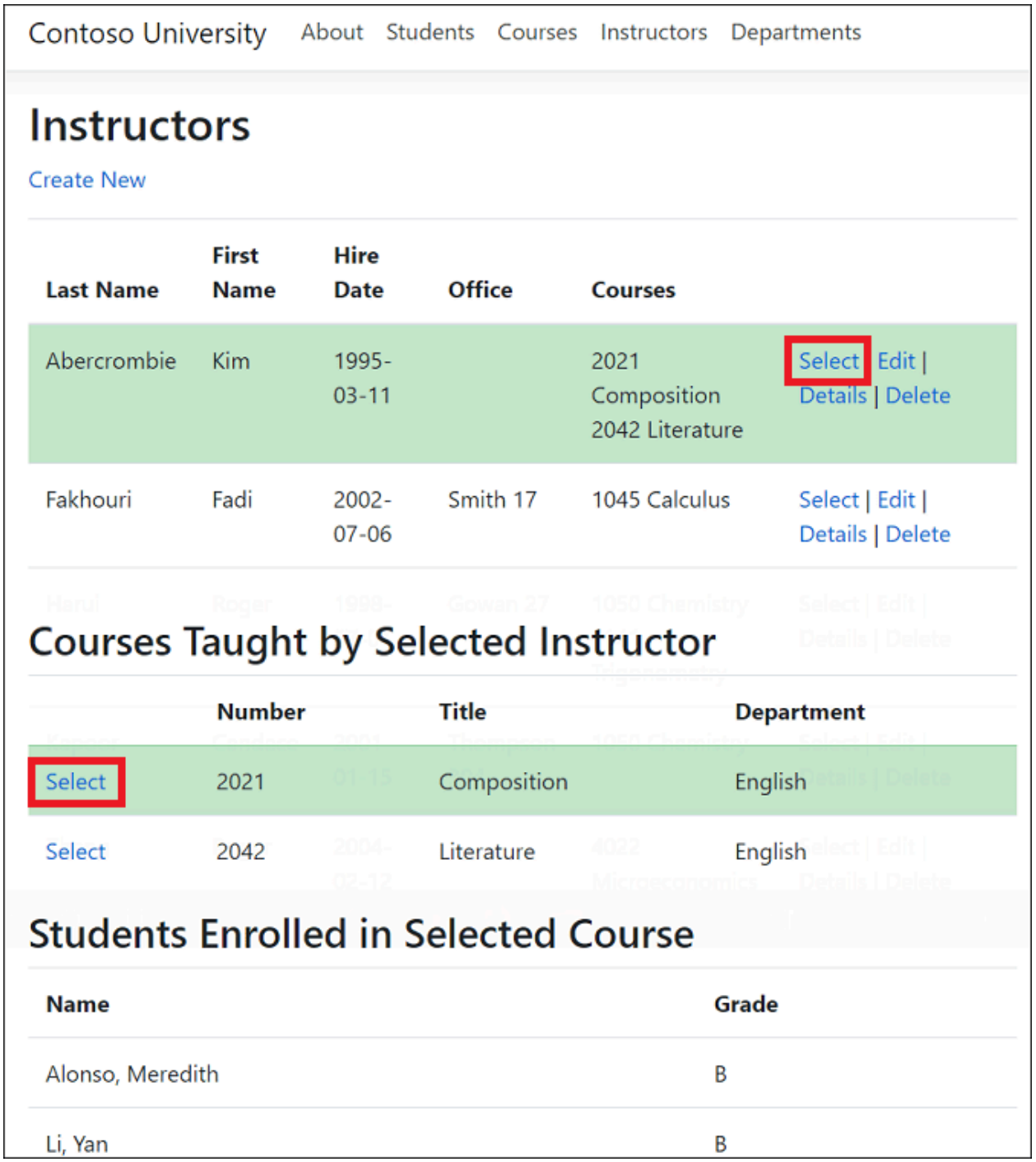
C#

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelectModel for the complete Razor Pages.

Create Instructor pages

This section scaffolds Instructor pages and adds related Courses and Enrollments to the Instructors Index page.



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected

instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.

- When the user selects a course, related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model

The instructors page shows data from three different tables. A view model is needed that includes three properties representing the three tables.

Create `Models/SchoolViewModels/InstructorIndexData.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Scaffold Instructor pages

Visual Studio

- Follow the instructions in Scaffold the student pages with the following exceptions:
  - Create a *Pages/Instructors* folder.
  - Use `Instructor` for the model class.
  - Use the existing context class instead of creating a new one.

Run the app and navigate to the Instructors page.

Update `Pages/Instructors/Index.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.Courses)
                    .ThenInclude(c => c.Department)
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
                Instructor instructor = InstructorData.Instructors
                    .Where(i => i.ID == id.Value).Single();
                InstructorData.Courses = instructor.Courses;
            }

            if (courseID != null)
            {
                CourseID = courseID.Value;
                IEnumerable<Enrollment> Enrollments = await
_context.Enrollments
                    .Where(x => x.CourseID == CourseID)
                    .Include(i=>i.Student)
                    .ToListAsync();
                InstructorData.Enrollments = Enrollments;
```

```
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the `Pages/Instructors/Index.cshtml.cs` file:

C#

```
InstructorData = new InstructorIndexData();
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.Courses)
        .ThenInclude(c => c.Department)
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The code specifies eager loading for the following navigation properties:

- `Instructor.OfficeAssignment`
- `Instructor.Courses`
  - `Course.Department`

The following code executes when an instructor is selected, that is, `id != null`.

C#

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = InstructorData.Instructors
        .Where(i => i.ID == id.Value).Single();
    InstructorData.Courses = instructor.Courses;
}
```

The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from the selected instructor's `Courses` navigation property.

The `Where` method returns a collection. In this case, the filter select a single entity, so the `Single` method is called to convert the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `Course` navigation property.

The Single method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is SingleOrDefault, which returns a default value if the collection is empty. For this query, `null` in the default returned.

The following code populates the view model's `Enrollments` property when a course is selected:

C#

```
if (courseID != null)
{
    CourseID = courseID.Value;
    IEnumerable<Enrollment> Enrollments = await _context.Enrollments
        .Where(x => x.CourseID == CourseID)
        .Include(i=>i.Student)
        .ToListAsync();
    InstructorData.Enrollments = Enrollments;
}
```

Update the instructors Index page

Update `Pages/Instructors/Index.cshtml` with the following code.

CSHTML

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
```

```
    <tbody>
        @foreach (var item in Model.InstructorData.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.Courses)
                        {
                            @course.CourseID @:  @course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a>
|
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.InstructorData.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
```

```
        </tr>

        @foreach (var item in Model.InstructorData.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    <a asp-page="./Index" asp-route-
courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}

@if (Model.InstructorData.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.InstructorData.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding code makes the following changes:

- Updates the `page` directive to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `https://localhost:5001/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the URL is: `https://localhost:5001/Instructors/2`

- Adds an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

  HTML

  ```
  @if (item.OfficeAssignment != null)
  {
      @item.OfficeAssignment.Location
  }
  ```

- Adds a **Courses** column that displays courses taught by each instructor. See Explicit line transition for more about this razor syntax.

- Adds code that dynamically adds `class="table-success"` to the `tr` element of the selected instructor and course. This sets a background color for the selected row using a Bootstrap class.

  HTML

  ```
  string selectedRow = "";
  if (item.CourseID == Model.CourseID)
  {
      selectedRow = "table-success";
  }
  <tr class="@selectedRow">
  ```

- Adds a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

  HTML

  ```
  <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
  ```

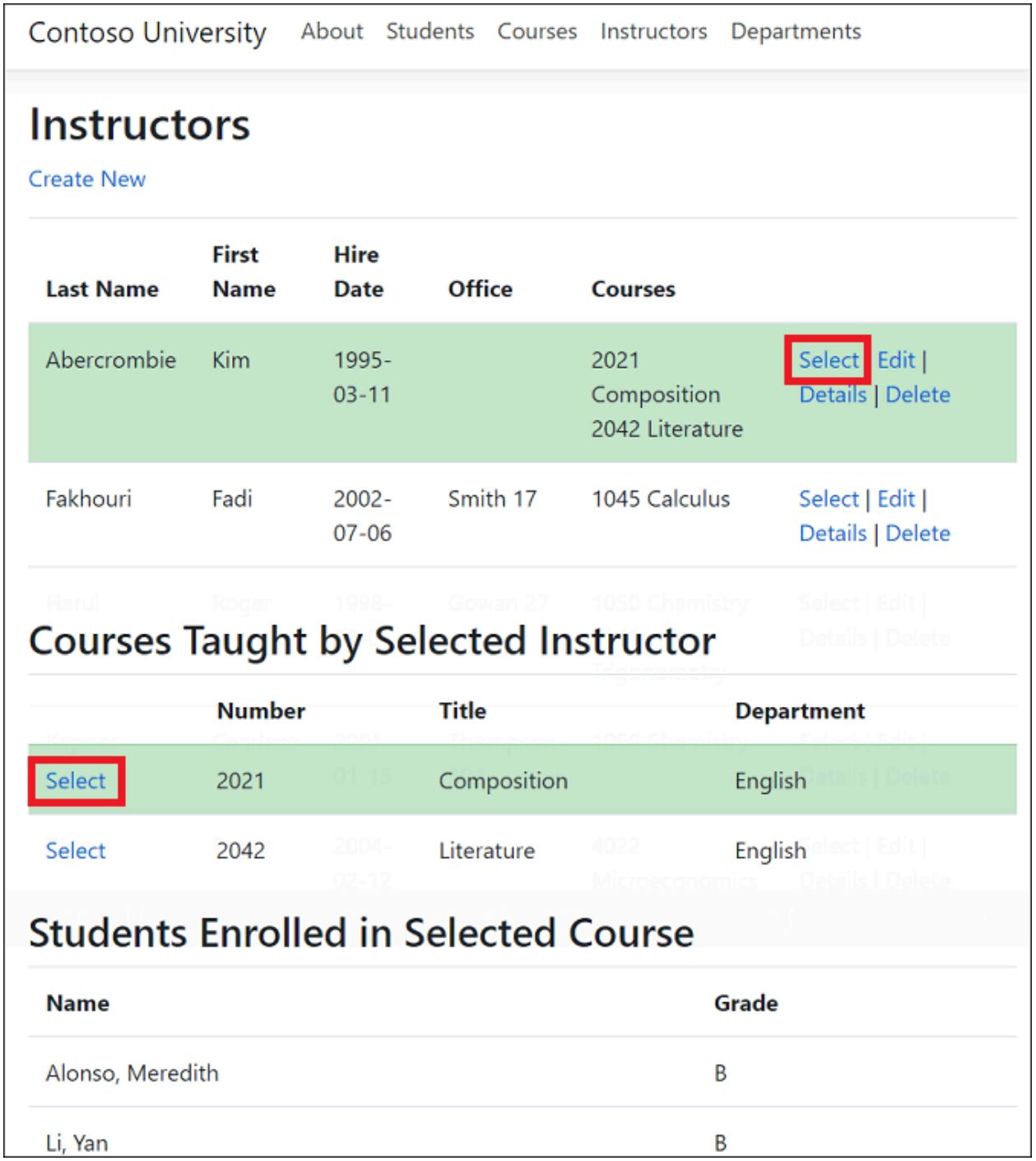
- Adds a table of courses for the selected Instructor.

- Adds a table of student enrollments for the selected course.

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If `OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link for an instructor. The row style changes and courses assigned to that instructor are displayed.

Select a course to see the list of enrolled students and their grades.



Next steps

The next tutorial shows how to update related data.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 7, Razor Pages with EF Core in ASP.NET Core - Update Related Data

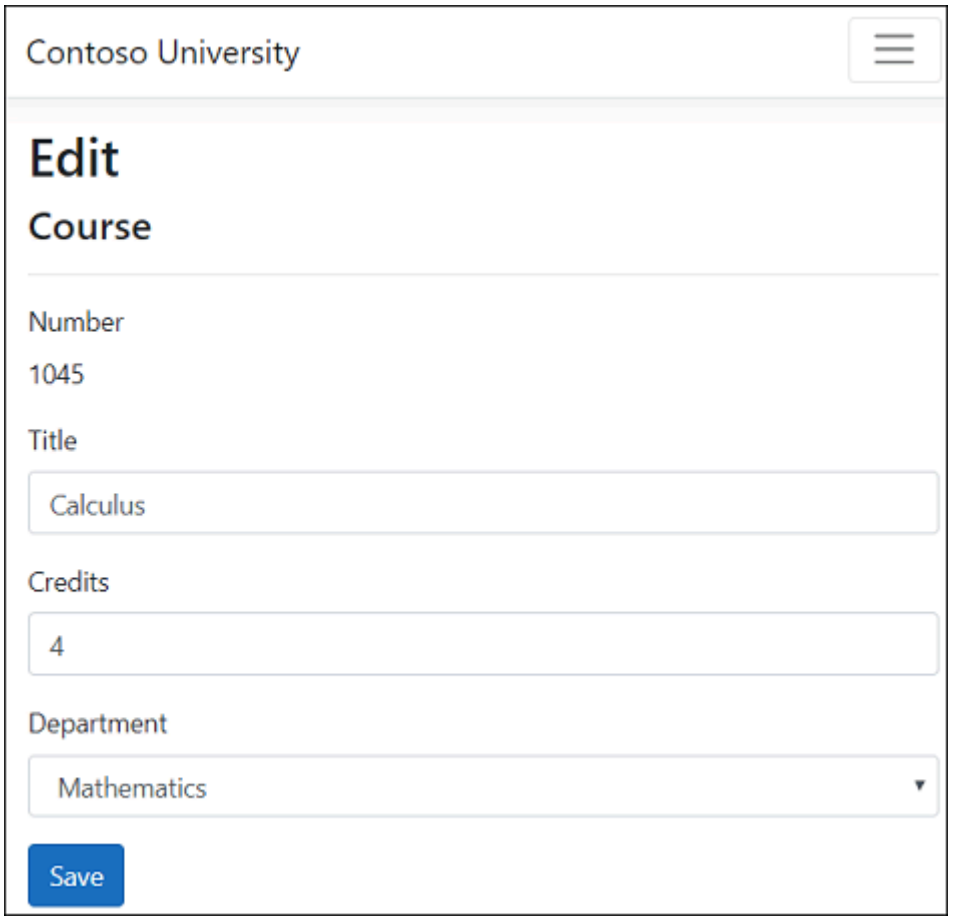
Article • 04/11/2023

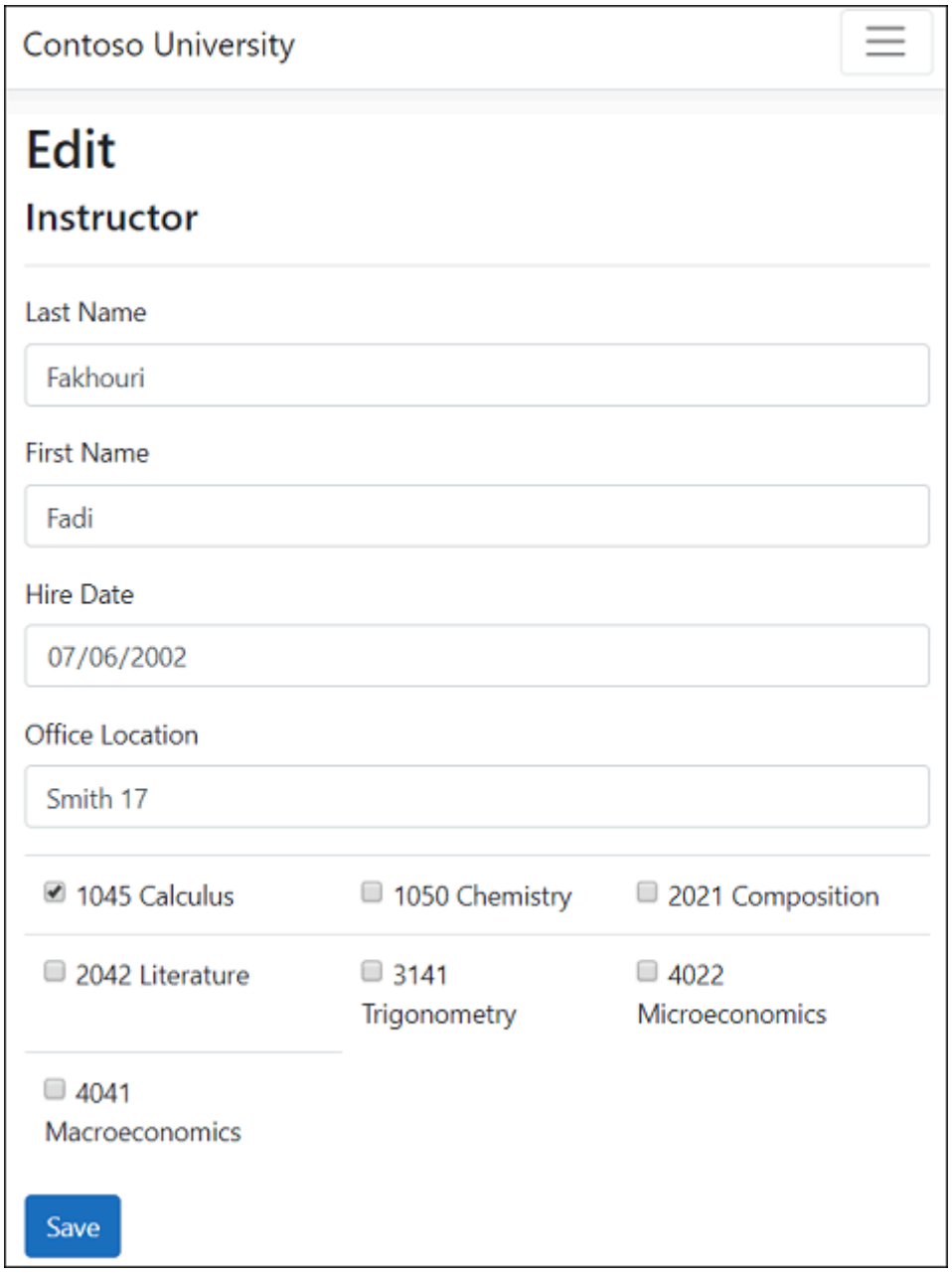
By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to update related data. The following illustrations show some of the completed pages.


Contoso University
Edit
Course
Number
1045
Title
Calculus
Credits
4
Department
Mathematics
Save



Update the Course Create and Edit pages

The scaffolded code for the Course Create and Edit pages has a Department drop-down list that shows `DepartmentID`, an `int`. The drop-down should show the Department name, so both of these pages need a list of department names. To provide that list, use a base class for the Create and Edit pages.

Create a base class for Course Create and Edit

Create a `Pages/Courses/DepartmentNamePageModel.cs` file with the following code:

```
C#

using ContosoUniversity.Data;
using ContosoUniversity.Models;
```

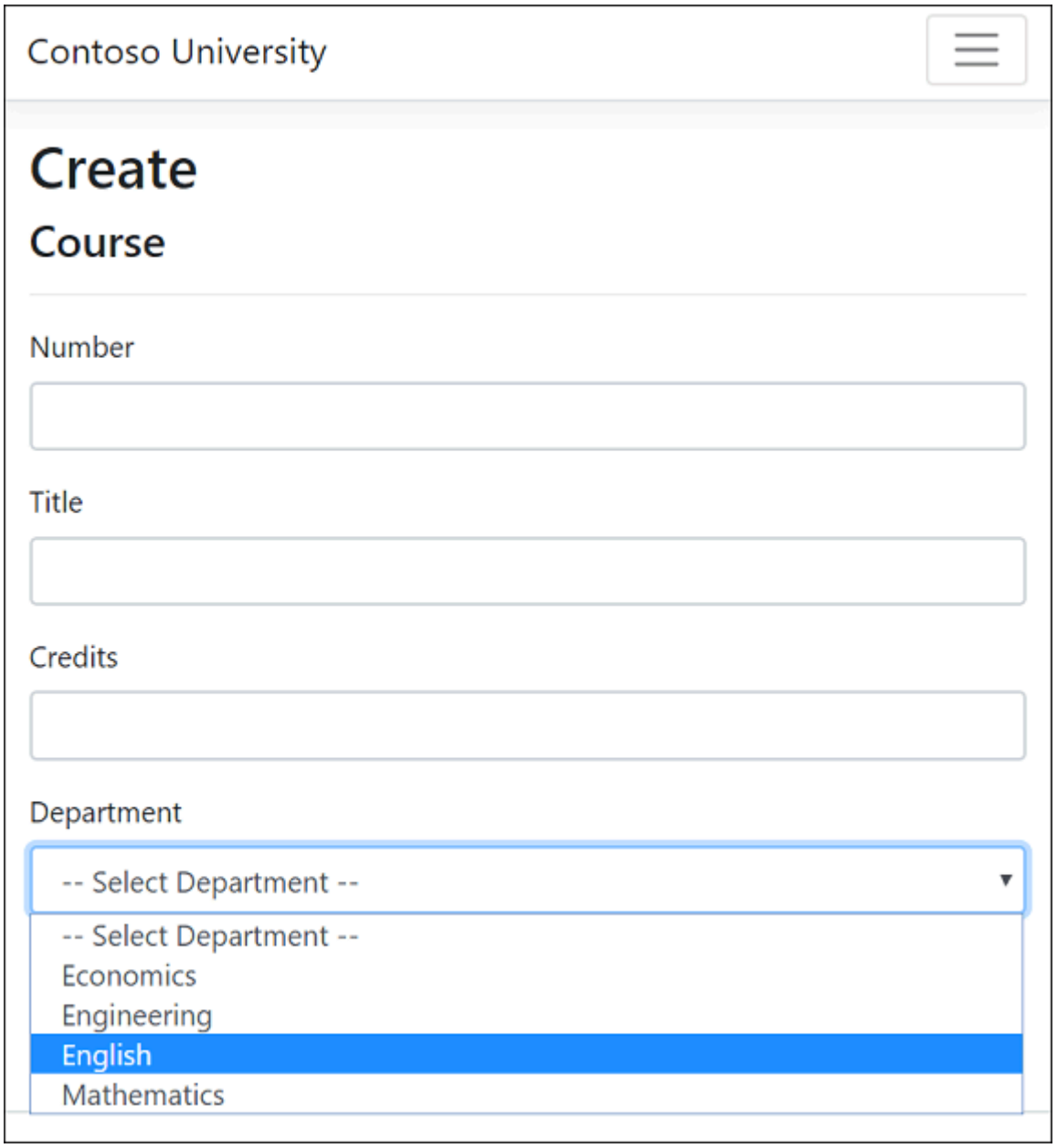
```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

            DepartmentNameSL = new
SelectList(departmentsQuery.AsNoTracking(),
                nameof(Department.DepartmentID),
                nameof(Department.Name),
                selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Update the Course Create page model

A Course is assigned to a Department. The base class for the Create and Edit pages provides a `SelectList` for selecting the department. The drop-down list that uses the `SelectList` sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update `Pages/Courses/Create.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
```

```
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s =>
s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context,
emptyCourse.DepartmentID);
            return Page();
        }
    }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses TryUpdateModelAsync to prevent overposting.
- Removes `ViewData["DepartmentID"]`. The `DepartmentNameSL` `SelectList` is a strongly typed model and will be used by the Razor page. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Course Create Razor page

Update `Pages/Courses/Create.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label">
</label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label">
</label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label">
</label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-
danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
/>
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
```

```
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" in the drop-down when no department has been selected yet, rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

CSHTML

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Course Edit page model

Update `Pages/Courses/Edit.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
```

```
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m =>
m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (courseToUpdate == null)
            {
                return NotFound();
            }

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context,
courseToUpdate.DepartmentID);
            return Page();
        }
```

```
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which selects that department in the drop-down list.

Update the Course Edit Razor page

Update `Pages/Courses/Edit.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label">
</label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label">
</label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-
danger"></span>
            </div>
            <div class="form-group">
```

```
                <label asp-for="Course.Department" class="control-label">
</label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption for the Department drop-down from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL`, which is in the base class.

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for the course number to be included in the posted data when the user selects **Save**.

Update the Course page models

AsNoTracking can improve performance when tracking isn't required.

Update `Pages/Courses/Delete.cshtml.cs` and `Pages/Courses/Details.cshtml.cs` by adding `AsNoTracking` to the `OnGetAsync` methods:

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
```

```
    {
        return NotFound();
    }

    Course = await _context.Courses
        .AsNoTracking()
        .Include(c => c.Department)
        .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

Update the Course Razor pages

Update `Pages/Courses/Delete.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
```

```
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
```

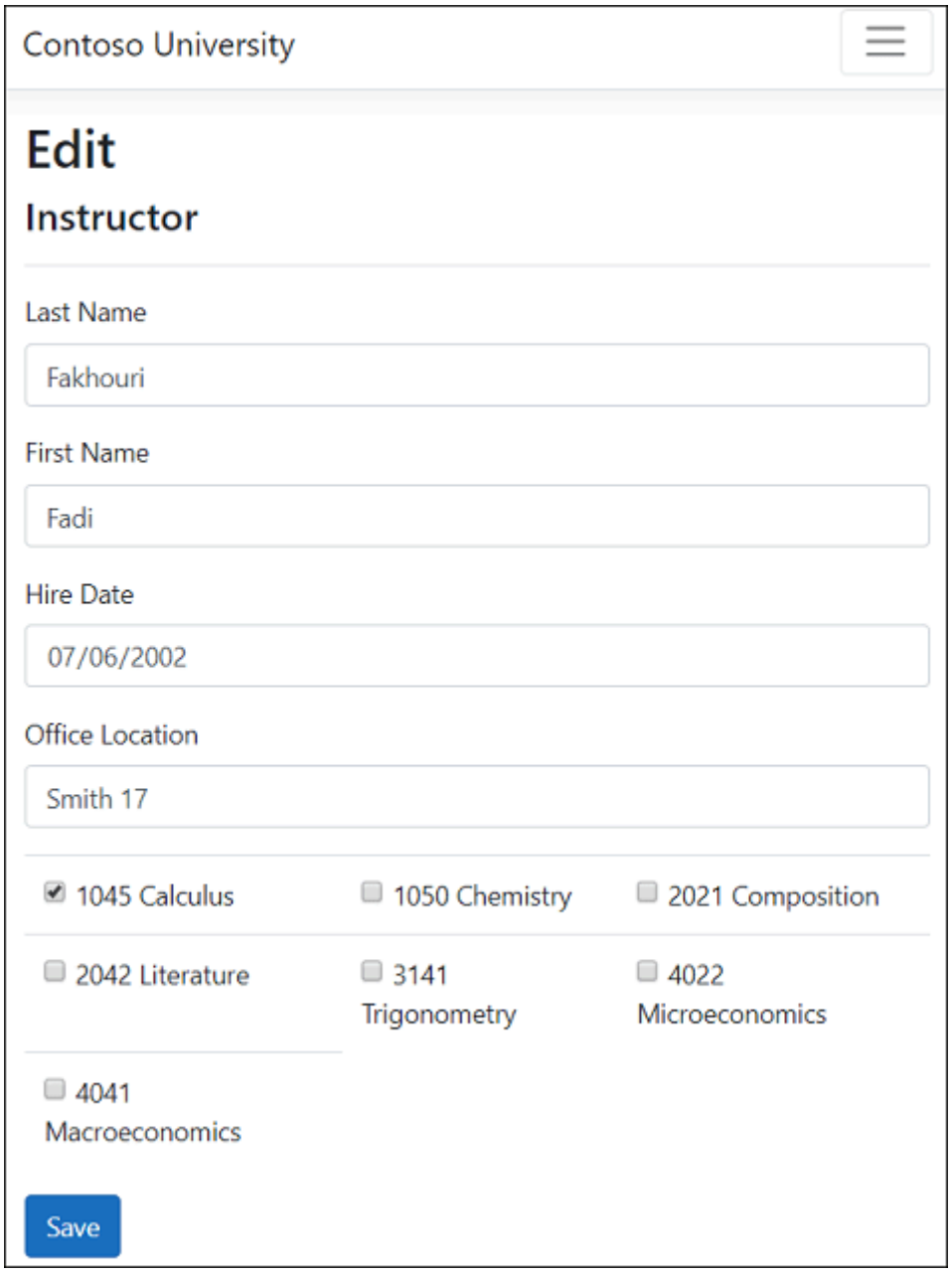
```
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Course.CourseID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

Test the Course pages

Test the create, edit, details, and delete pages.

Update the instructor Create and Edit pages

Instructors may teach any number of courses. The following image shows the instructor Edit page with an array of course checkboxes.



The checkboxes enable changes to courses an instructor is assigned to. A checkbox is displayed for every course in the database. Courses that the instructor is assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, a different UI might work better. But the method of managing a many-to-many relationship shown here wouldn't change. To create or delete relationships, you manipulate a join entity.

Create a class for assigned courses data

Create `Models/SchoolViewModels/AssignedCourseData.cs` with the following code:

C#

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
```

```
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the checkboxes for courses assigned to an instructor.

Create an Instructor page model base class

Create the `Pages/Instructors/InstructorCoursesPageModel.cs` base class:

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {
        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.Courses.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used for efficient lookups.

Handle office location

Another relationship the edit page has to handle is the one-to-zero-or-one relationship that the Instructor entity has with the `OfficeAssignment` entity. The instructor edit code must handle the following scenarios:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.
- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the Instructor Edit page model

Update `Pages/Instructors/Edit.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class EditModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
```

```
            {
                return NotFound();
            }

            Instructor = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.Courses)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            PopulateAssignedCourseData(_context, Instructor);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id, string[]
selectedCourses)
        {
            if (id == null)
            {
                return NotFound();
            }

            var instructorToUpdate = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.Courses)
                .FirstOrDefaultAsync(s => s.ID == id);

            if (instructorToUpdate == null)
            {
                return NotFound();
            }

            if (await TryUpdateModelAsync<Instructor>(
                instructorToUpdate,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                if (String.IsNullOrWhiteSpace(
                    instructorToUpdate.OfficeAssignment?.Location))
                {
                    instructorToUpdate.OfficeAssignment = null;
                }
                UpdateInstructorCourses(selectedCourses,
instructorToUpdate);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            UpdateInstructorCourses(selectedCourses, instructorToUpdate);
            PopulateAssignedCourseData(_context, instructorToUpdate);
            return Page();
```

```
        }

        public void UpdateInstructorCourses(string[] selectedCourses,
                                            Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.Courses = new List<Course>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.Courses.Select(c => c.CourseID));
            foreach (var course in _context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.Courses.Add(course);
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        var courseToRemove =
instructorToUpdate.Courses.Single(
                                                              c => c.CourseID ==
course.CourseID);
                        instructorToUpdate.Courses.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` and `Courses` navigation properties.
- Updates the retrieved `Instructor` entity with values from the model binder. TryUpdateModelAsync prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.
- Calls `PopulateAssignedCourseData` in `OnGetAsync` to provide information for the checkboxes using the `AssignedCourseData` view model class.

- Calls `UpdateInstructorCourses` in `OnPostAsync` to apply information from the checkboxes to the Instructor entity being edited.
- Calls `PopulateAssignedCourseData` and `UpdateInstructorCourses` in `OnPostAsync` if TryUpdateModelAsync fails. These method calls restore the assigned course data entered on the page when it is redisplayed with an error message.

Since the Razor page doesn't have a collection of Course entities, the model binder can't automatically update the `Courses` navigation property. Instead of using the model binder to update the `Courses` navigation property, that's done in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `Courses` property from model binding. This doesn't require any change to the code that calls TryUpdateModelAsync because you're using the overload with declared properties and `Courses` isn't in the include list.

If no checkboxes were selected, the code in `UpdateInstructorCourses` initializes the `instructorToUpdate.Courses` with an empty collection and returns:

C#

```
if (selectedCourses == null)
{
    instructorToUpdate.Courses = new List<Course>();
    return;
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the page. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the checkbox for a course is selected but the course is ***not*** in the `Instructor.Courses` navigation property, the course is added to the collection in the navigation property.

C#

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Add(course);
    }
}
```

If the checkbox for a course is ***not*** selected, but the course is in the `Instructor.Courses` navigation property, the course is removed from the navigation property.

C#

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        var courseToRemove = instructorToUpdate.Courses.Single(
                                        c => c.CourseID == course.CourseID);
        instructorToUpdate.Courses.Remove(courseToRemove);
    }
}
```

Update the Instructor Edit Razor page

Update `Pages/Instructors/Edit.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label">
</label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-
label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-
control" />
                <span asp-validation-for="Instructor.FirstMidName"
class="text-danger"></span>
            </div>
            <div class="form-group">
```

```
                <label asp-for="Instructor.HireDate" class="control-label">
</label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location"
class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location"
class="form-control" />
                <span asp-validation-
for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in
Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                               @course.CourseID @:
@course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
```

```
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code creates an HTML table that has three columns. Each column has a checkbox and a caption containing the course number and title. The checkboxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each checkbox is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the checkboxes that are selected.

When the checkboxes are initially rendered, courses assigned to the instructor are selected.

Note: The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Run the app and test the updated Instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Update the Instructor Create page

Update the Instructor Create page model and with code similar to the Edit page:

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<InstructorCoursesPageModel> _logger;

        public CreateModel(SchoolContext context,
                           ILogger<InstructorCoursesPageModel> logger)
        {
            _context = context;
            _logger = logger;
        }
```

```
        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.Courses = new List<Course>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[]
selectedCourses)
        {
            var newInstructor = new Instructor();

            if (selectedCourses.Length > 0)
            {
                newInstructor.Courses = new List<Course>();
                // Load collection with one DB call.
                _context.Courses.Load();
            }

            // Add selected Courses courses to the new instructor.
            foreach (var course in selectedCourses)
            {
                var foundCourse = await
_context.Courses.FindAsync(int.Parse(course));
                if (foundCourse != null)
                {
                    newInstructor.Courses.Add(foundCourse);
                }
                else
                {
                    _logger.LogWarning("Course {course} not found", course);
                }
            }

            try
            {
                if (await TryUpdateModelAsync<Instructor>(
                                newInstructor,
                                "Instructor",
                                i => i.FirstMidName, i => i.LastName,
                                i => i.HireDate, i => i.OfficeAssignment))
                {
                    _context.Instructors.Add(newInstructor);
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
```

```
                return RedirectToPage("./Index");
            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }

            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

The preceding code:

- Adds logging for warning and error messages.
- Calls Load, which fetches all the Courses in one database call. For small collections this is an optimization when using FindAsync. `FindAsync` returns the tracked entity without a request to the database.

```
C#

public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
{
    var newInstructor = new Instructor();

    if (selectedCourses.Length > 0)
    {
        newInstructor.Courses = new List<Course>();
        // Load collection with one DB call.
        _context.Courses.Load();
    }

    // Add selected Courses courses to the new instructor.
    foreach (var course in selectedCourses)
    {
        var foundCourse = await
_context.Courses.FindAsync(int.Parse(course));
        if (foundCourse != null)
        {
            newInstructor.Courses.Add(foundCourse);
        }
        else
        {
            _logger.LogWarning("Course {course} not found", course);
        }
    }

    try
    {
        if (await TryUpdateModelAsync<Instructor>(
```

```
                        newInstructor,
                        "Instructor",
                        i => i.FirstMidName, i => i.LastName,
                        i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            return RedirectToPage("./Index");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex.Message);
        }

        PopulateAssignedCourseData(_context, newInstructor);
        return Page();
    }
```

- `_context.Instructors.Add(newInstructor)` creates a new `Instructor` using many-to-many relationships without explicitly mapping the join table. Many-to-many was added in EF 5.0.

Test the instructor Create page.

Update the Instructor Create Razor page with code similar to the Edit page:

CSHTML

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label">
</label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-
danger"></span>
```

```
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-
label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-
control" />
                <span asp-validation-for="Instructor.FirstMidName"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label">
</label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location"
class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location"
class="form-control" />
                <span asp-validation-
for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in
Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />

                                               @course.CourseID @:
@course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
```

```
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Instructor Delete page

Update `Pages/Instructors/Delete.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.FirstOrDefaultAsync(m =>
m.ID == id);
```

```
            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor instructor = await _context.Instructors
                .Include(i => i.Courses)
                .SingleAsync(i => i.ID == id);

            if (instructor == null)
            {
                return RedirectToPage("./Index");
            }

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `Courses` navigation property. `Courses` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Run the app and test the Delete page.

Next steps

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 8, Razor Pages with EF Core in ASP.NET Core - Concurrency

Article • 04/11/2023

Tom Dykstra, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to handle conflicts when multiple users update an entity concurrently.

Concurrency conflicts

A concurrency conflict occurs when:

- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the database.

If concurrency detection isn't enabled, whoever updates the database last overwrites the other user's changes. If this risk is acceptable, the cost of programming for concurrency might outweigh the benefit.

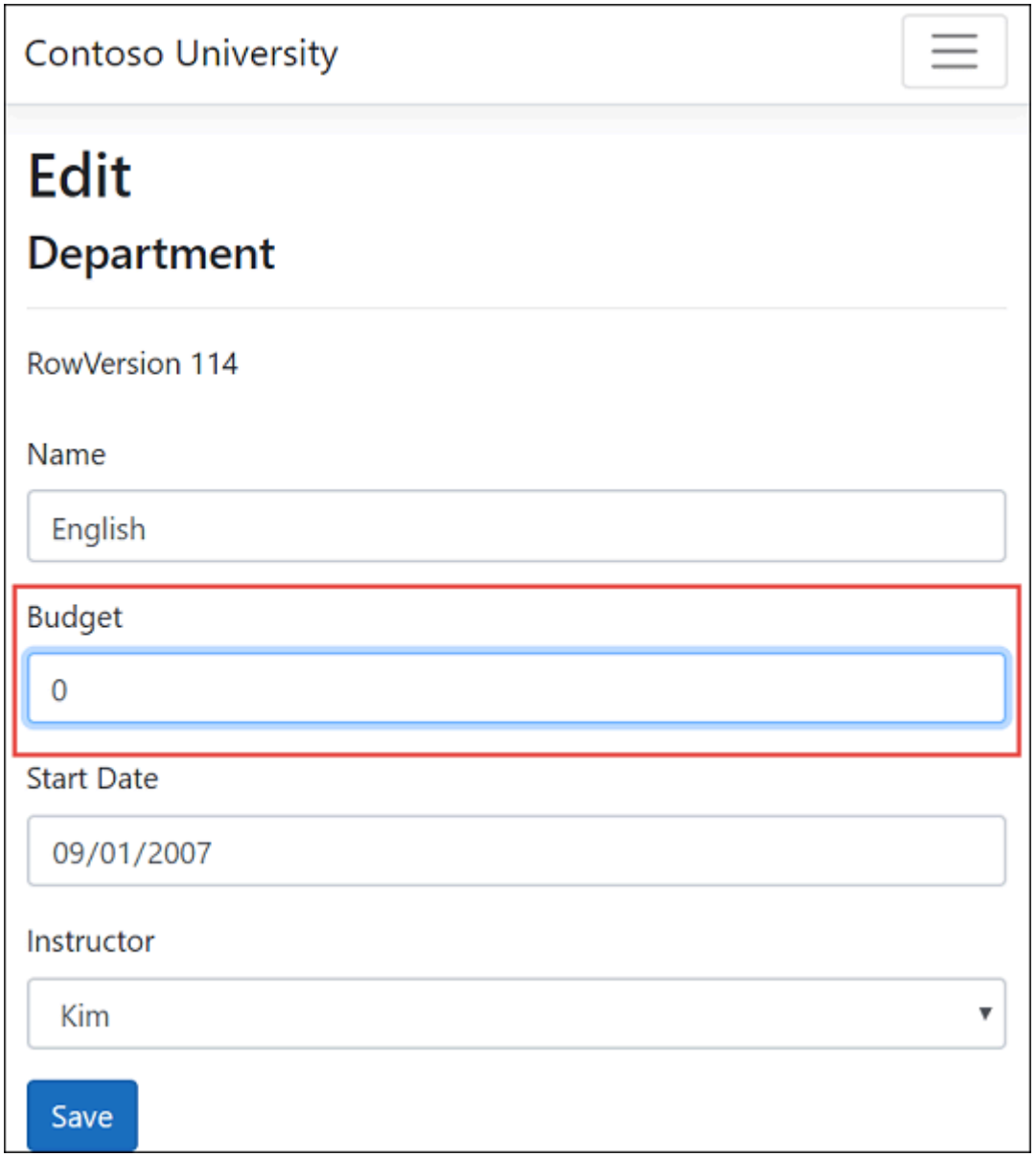
Pessimistic concurrency

One way to prevent concurrency conflicts is to use database locks. This is called pessimistic concurrency. Before the app reads a database row that it intends to update, it requests a lock. Once a row is locked for update access, no other users are allowed to lock the row until the first lock is released.

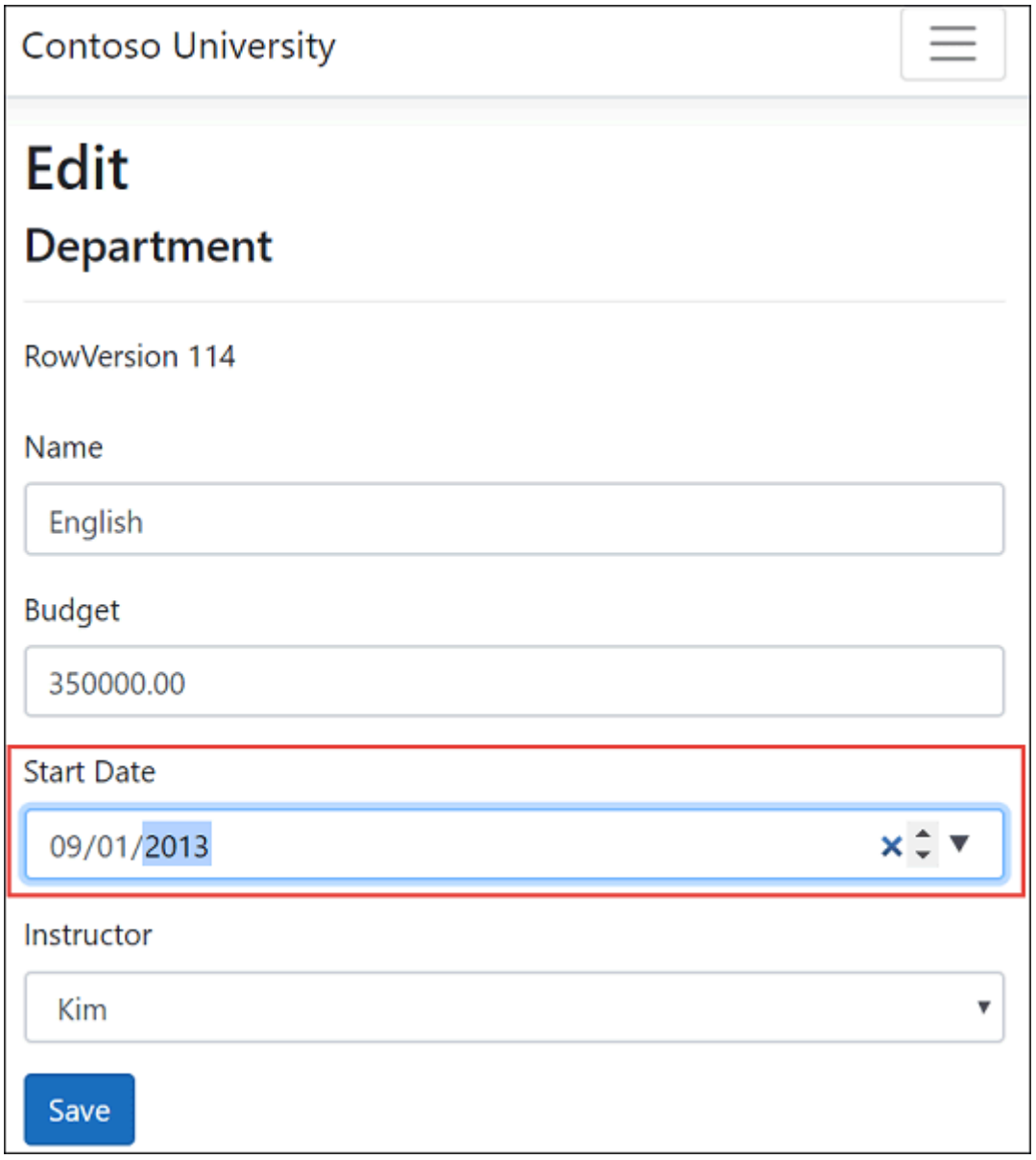
Managing locks has disadvantages. It can be complex to program and can cause performance problems as the number of users increases. Entity Framework Core provides no built-in support for pessimistic concurrency.

Optimistic concurrency

Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change take effect, since the browser displays the Index page with zero as the Budget amount.

John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts:

- Keep track of which property a user has modified and update only the corresponding columns in the database.

  In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach has some disadvantages:
  - Can't avoid data loss if competing changes are made to the same property.
  - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.

    - Can increase app complexity compared to concurrency detection on an entity.

- Let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the fetched $350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. All values from the client take precedence over what's in the data store. The scaffolded code does no concurrency handling, Client Wins happens automatically.

- Prevent John's change from being updated in the database. Typically, the app would:
    - Display an error message.
    - Show the current state of the data.
    - Allow the user to reapply the changes.

  This is called a *Store Wins* scenario. The data-store values take precedence over the values submitted by the client. The Store Wins scenario is used in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Conflict detection in EF Core

Properties configured as concurrency tokens are used to implement optimistic concurrency control. When an update or delete operation is triggered by SaveChanges or SaveChangesAsync, the value of the concurrency token in the database is compared against the original value read by EF Core:

- If the values match, the operation can complete.
- If the values do not match, EF Core assumes that another user has performed a conflicting operation, aborts the current transaction, and throws a DbUpdateConcurrencyException.

Another user or process performing an operation that conflicts with the current operation is known as *concurrency conflict*.

On relational databases EF Core checks for the value of the concurrency token in the `WHERE` clause of `UPDATE` and `DELETE` statements to detect a concurrency conflict.

The data model must be configured to enable conflict detection by including a tracking column that can be used to determine when a row has been changed. EF provides two approaches for concurrency tokens:

- Applying [ConcurrencyCheck] or IsConcurrencyToken to a property on the model. This approach is not recommended. For more information, see Concurrency Tokens in EF Core.
- Applying TimestampAttribute or IsRowVersion to a concurrency token in the model. This is the approach used in this tutorial.

The SQL Server approach and SQLite implementation details are slightly different. A difference file is shown later in the tutorial listing the differences. The Visual Studio tab shows the SQL Server approach. The Visual Studio Code tab shows the approach for non-SQL Server databases, such as SQLite.

Visual Studio

- In the model, include a tracking column that is used to determine when a row has been changed.
- Apply the TimestampAttribute to the concurrency property.

Update the `Models/Department.cs` file with the following highlighted code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
                       ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
```

```
        public byte[] ConcurrencyToken { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The TimestampAttribute is what identifies the column as a concurrency tracking column. The fluent API is an alternative way to specify the tracking property:

C#

```
modelBuilder.Entity<Department>()
  .Property<byte[]>("ConcurrencyToken")
  .IsRowVersion();
```

The [Timestamp] attribute on an entity property generates the following code in the ModelBuilder method:

C#

```
 b.Property<byte[]>("ConcurrencyToken")
     .IsConcurrencyToken()
     .ValueGeneratedOnAddOrUpdate()
     .HasColumnType("rowversion");
```

The preceding code:

- Sets the property type `ConcurrencyToken` to byte array. `byte[]` is the required type for SQL Server.
- Calls IsConcurrencyToken. `IsConcurrencyToken` configures the property as a concurrency token. On updates, the concurrency token value in the database is compared to the original value to ensure it has not changed since the instance was retrieved from the database. If it has changed, a DbUpdateConcurrencyException is thrown and changes are not applied.
- Calls ValueGeneratedOnAddOrUpdate, which configures the `ConcurrencyToken` property to have a value automatically generated when adding or updating an entity.
- `HasColumnType("rowversion")` sets the column type in the SQL Server database to rowversion.

The following code shows a portion of the T-SQL generated by EF Core when the `Department` name is updated:

```
```

SQL

```
SET NOCOUNT ON;
UPDATE [Departments] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [ConcurrencyToken] = @p2;
SELECT [ConcurrencyToken]
FROM [Departments]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

The preceding highlighted code shows the `WHERE` clause containing `ConcurrencyToken`. If the database `ConcurrencyToken` doesn't equal the `ConcurrencyToken` parameter `@p2`, no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

SQL

```
SET NOCOUNT ON;
UPDATE [Departments] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [ConcurrencyToken] = @p2;
SELECT [ConcurrencyToken]
FROM [Departments]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT returns the number of rows affected by the last statement. If no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

Add a migration

Adding the `ConcurrencyToken` property changes the data model, which requires a migration.

Build the project.

Visual Studio

Run the following commands in the PMC:

PowerShell

```
Add-Migration RowVersion
Update-Database
```

The preceding commands:

- Creates the `Migrations/{time stamp}_RowVersion.cs` migration file.
- Updates the `Migrations/SchoolContextModelSnapshot.cs` file. The update adds the following code to the `BuildModel` method:

C#

```
b.Property<byte[]>("ConcurrencyToken")
    .IsConcurrencyToken()
    .ValueGeneratedOnAddOrUpdate()
    .HasColumnType("rowversion");
```

Scaffold Department pages

Visual Studio

Follow the instructions in Scaffold Student pages with the following exceptions:

- Create a *Pages/Departments* folder.
- Use `Department` for the model class.
- Use the existing context class instead of creating a new one.

Add a utility class

In the project folder, create the `Utility` class with the following code:

Visual Studio

C#

```
namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(byte[] token)
        {
            return token[7].ToString();
        }
    }
}
```

The `Utility` class provides the `GetLastChars` method used to display the last few characters of the concurrency token. The following code shows the code that works with both SQLite ad SQL Server:

C#

```
#if SQLiteVersion
using System;

namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(Guid token)
        {
            return token.ToString().Substring(
                                    token.ToString().Length - 3);
        }
    }
}
#else
namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(byte[] token)
        {
            return token[7].ToString();
        }
    }
}
#endif
```

The `#if SQLiteVersion` preprocessor directive isolates the differences in the SQLite and SQL Server versions and helps:

- The author maintain one code base for both versions.
- SQLite developers deploy the app to Azure and use SQL Azure.

Build the project.

Update the Index page

The scaffolding tool created a `ConcurrencyToken` column for the Index page, but that field wouldn't be displayed in a production app. In this tutorial, the last portion of the `ConcurrencyToken` is displayed to help show how concurrency handling works. The last portion isn't guaranteed to be unique by itself.

Update *Pages\Departments\Index.cshtml* page:

- Replace Index with Departments.
- Change the code containing `ConcurrencyToken` to show just the last few characters.
- Replace `FirstMidName` with `FullName`.

The following code shows the updated page:

CSHTML

```
@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Department[0].Administrator)
            </th>
            <th>
                Token
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Department)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
```

```
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem =>
item.Administrator.FullName)
                </td>
                <td>
                    @Utility.GetLastChars(item.ConcurrencyToken)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-
id="@item.DepartmentID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.DepartmentID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the Edit page model

Update `Pages/Departments/Edit.cshtml.cs` with the following code:

Visual Studio

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
```

```
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            // Fetch current department from DB.
            // ConcurrencyToken may have changed.
            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (departmentToUpdate == null)
            {
                return HandleDeletedDepartment();
            }

            // Set ConcurrencyToken to value read in OnGetAsync
            _context.Entry(departmentToUpdate).Property(
                 d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;

            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s =>
s.InstructorID))
            {
```

```
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues =
(Department)exceptionEntry.Entity;
                    var databaseEntry =
exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable
to save. " +
                            "The department was deleted by another
user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await SetDbErrorMessage(dbValues, clientValues,
_context);

                    // Save the current ConcurrencyToken so next
postback
                    // matches unless an new concurrency issue happens.
                    Department.ConcurrencyToken =
(byte[])dbValues.ConcurrencyToken;
                    // Clear the model error for the next postback.
                    ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

        private IActionResult HandleDeletedDepartment()
        {
            // ModelState contains the posted data because of the
deletion error
            // and overides the Department instance values when
displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another
user.");
            InstructorNameSL = new SelectList(_context.Instructors,
"ID", "FullName", Department.InstructorID);
            return Page();
        }
```

```
        private async Task SetDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
            }
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in
the database "
              + "have been displayed. If you still want to edit this
record, click "
              + "the Save button again.");
        }
    }
}
```

The concurrency updates

OriginalValue is updated with the `ConcurrencyToken` value from the entity when it was fetched in the `OnGetAsync` method. EF Core generates a `SQL UPDATE` command with a `WHERE` clause containing the original `ConcurrencyToken` value. If no rows are affected by the `UPDATE` command, a `DbUpdateConcurrencyException` exception is thrown. No rows are affected by the `UPDATE` command when no rows have the original `ConcurrencyToken` value.

Visual Studio

C#

```
public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // Fetch current department from DB.
    // ConcurrencyToken may have changed.
    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        return HandleDeletedDepartment();
    }

    // Set ConcurrencyToken to value read in OnGetAsync
    _context.Entry(departmentToUpdate).Property(
         d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;
```

In the preceding highlighted code:

- The value in `Department.ConcurrencyToken` is the value when the entity was fetched in the `Get` request for the `Edit` page. The value is provided to the `OnPost` method by a hidden field in the Razor page that displays the entity to be edited. The hidden field value is copied to `Department.ConcurrencyToken` by the model binder.
- `OriginalValue` is what EF Core uses in the `WHERE` clause. Before the highlighted line of code executes:
  - `OriginalValue` has the value that was in the database when `FirstOrDefaultAsync` was called in this method.
  - This value might be different from what was displayed on the Edit page.
- The highlighted code makes sure that EF Core uses the original `ConcurrencyToken` value from the displayed `Department` entity in the SQL `UPDATE` statement's `WHERE` clause.

The following code shows the `Department` model. `Department` is initialized in the:

- `OnGetAsync` method by the EF query.

- `OnPostAsync` method by the hidden field in the Razor page using model binding:

Visual Studio

C#

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Department Department { get; set; }
    // Replace ViewData["InstructorID"]
    public SelectList InstructorNameSL { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Department = await _context.Departments
            .Include(d => d.Administrator)  // eager loading
            .AsNoTracking()                 // tracking not required
            .FirstOrDefaultAsync(m => m.DepartmentID == id);

        if (Department == null)
        {
            return NotFound();
        }

        // Use strongly typed data rather than ViewData.
        InstructorNameSL = new SelectList(_context.Instructors,
            "ID", "FirstMidName");

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        // Fetch current department from DB.
        // ConcurrencyToken may have changed.
        var departmentToUpdate = await _context.Departments
            .Include(i => i.Administrator)
            .FirstOrDefaultAsync(m => m.DepartmentID == id);

        if (departmentToUpdate == null)
```

```
            {
                return HandleDeletedDepartment();
            }

            // Set ConcurrencyToken to value read in OnGetAsync
            _context.Entry(departmentToUpdate).Property(
                 d => d.ConcurrencyToken).OriginalValue =
    Department.ConcurrencyToken;
```

The preceding code shows the `ConcurrencyToken` value of the `Department` entity from the `HTTP POST` request is set to the `ConcurrencyToken` value from the `HTTP GET` request.

When a concurrency error happens, the following highlighted code gets the client values (the values posted to this method) and the database values.

C#

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await SetDbErrorMessage(dbValues, clientValues, _context);

        // Save the current ConcurrencyToken so next postback
        // matches unless an new concurrency issue happens.
        Department.ConcurrencyToken = dbValues.ConcurrencyToken;
        // Clear the model error for the next postback.
        ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
    }
```

The following code adds a custom error message for each column that has database values different from what was posted to `OnPostAsync`:

C#

```
private async Task SetDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database
"
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

The following highlighted code sets the `ConcurrencyToken` value to the new value retrieved from the database. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

C#

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
```

```
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await SetDbErrorMessage(dbValues, clientValues, _context);

        // Save the current ConcurrencyToken so next postback
        // matches unless an new concurrency issue happens.
        Department.ConcurrencyToken = dbValues.ConcurrencyToken;
        // Clear the model error for the next postback.
        ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
    }
```

The ModelState.Remove statement is required because `ModelState` has the previous `ConcurrencyToken` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

SQL Server vs SQLite code differences

The following shows the differences between the SQL Server and SQLite versions:

diff

```
+ using System;    // For GUID on SQLite

+ departmentToUpdate.ConcurrencyToken = Guid.NewGuid();

 _context.Entry(departmentToUpdate)
    .Property(d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;

- Department.ConcurrencyToken = (byte[])dbValues.ConcurrencyToken;
+ Department.ConcurrencyToken = dbValues.ConcurrencyToken;
```

Update the Edit Razor page

Update `Pages/Departments/Edit.cshtml` with the following code:

CSHTML

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.ConcurrencyToken" />
            <div class="form-group">
                <label>Version</label>
                @Utility.GetLastChars(Model.Department.ConcurrencyToken)
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label">
</label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label">
</label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label">
</label>
                <input asp-for="Department.StartDate" class="form-control"
/>
                <span asp-validation-for="Department.StartDate" class="text-
danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
```

```
                <select asp-for="Department.InstructorID" class="form-
control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID"
class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `ConcurrencyToken` must be added so postback binds the value.
- Displays the last byte of `ConcurrencyToken` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

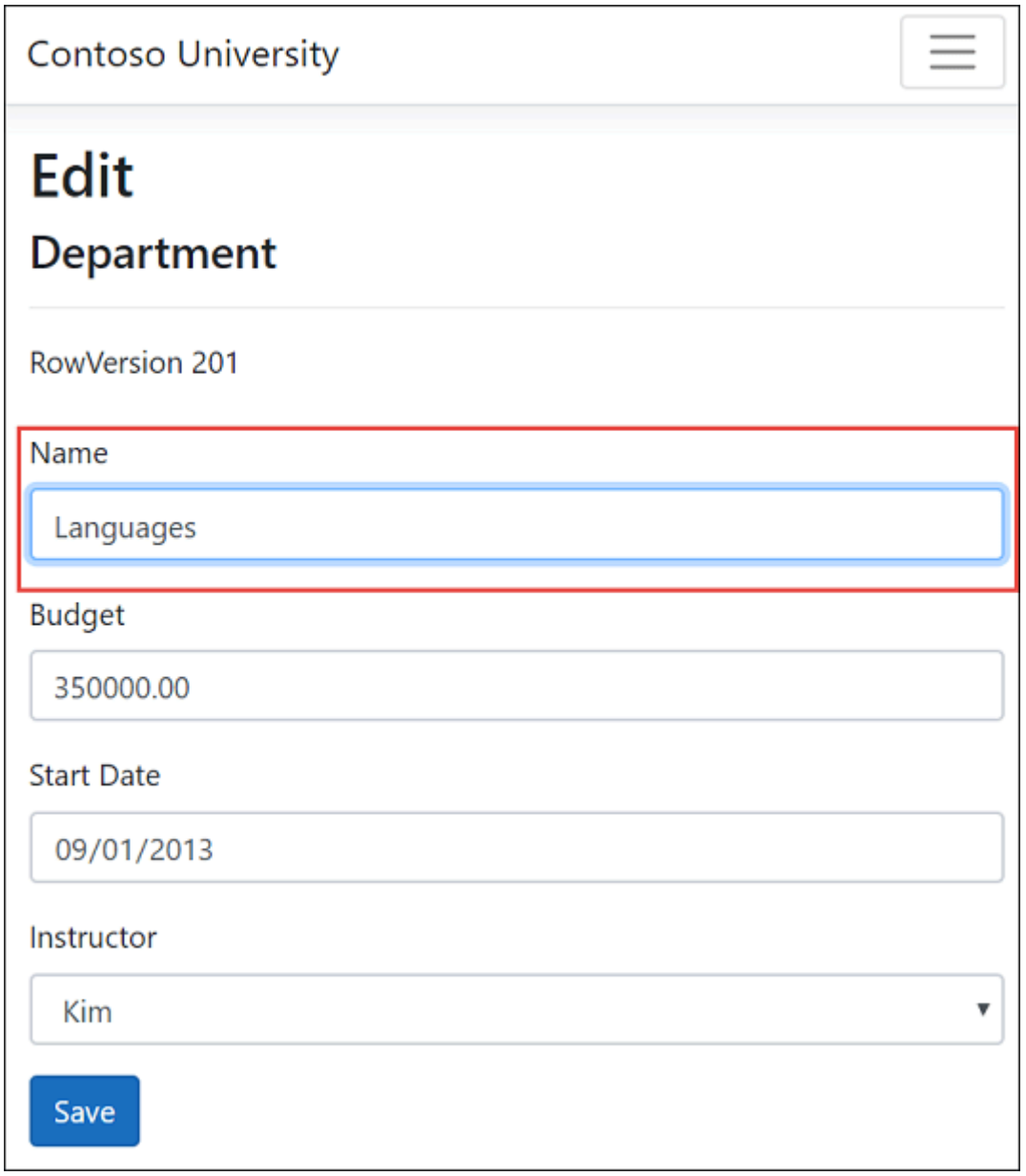
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

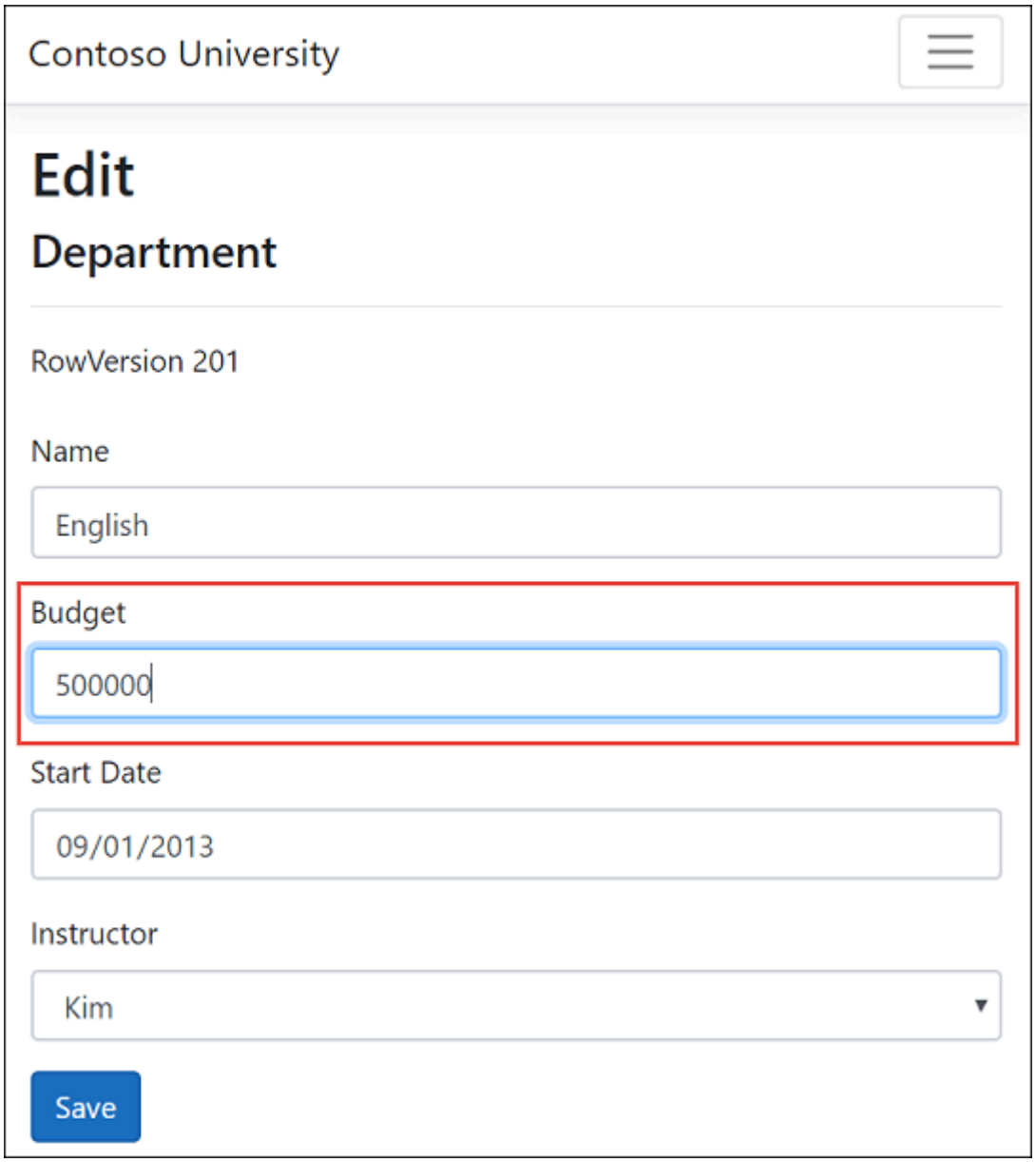
The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.

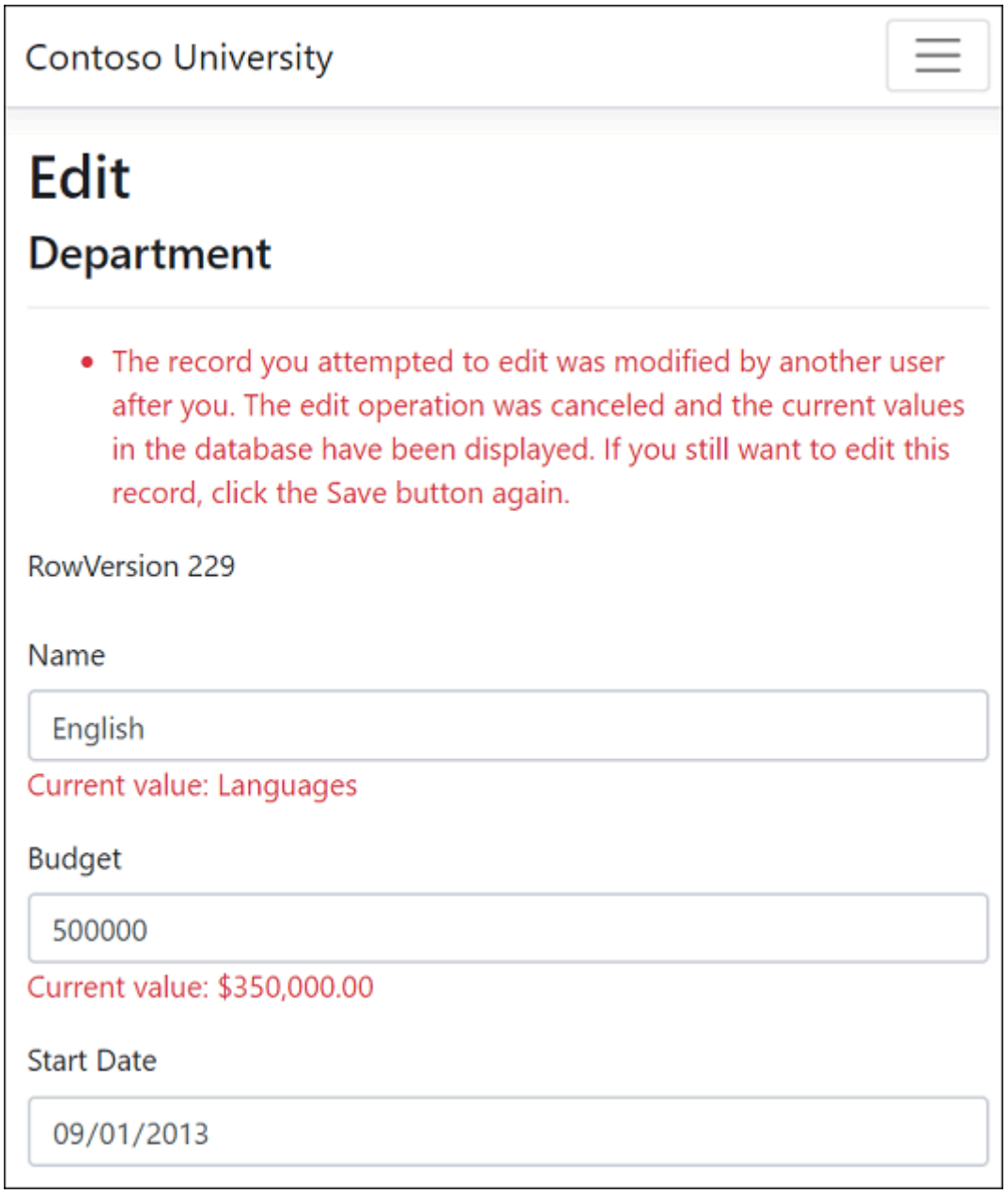


The browser shows the Index page with the changed value and updated `ConcurrencyToken` indicator. Note the updated `ConcurrencyToken` indicator, it's displayed on the second postback in the other tab.

Change a different field in the second browser tab.



Click **Save**. You see error messages for all fields that don't match the database values:



This browser window didn't intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.

Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page model

Update `Pages/Departments/Delete.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
```

```
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool?
concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                 return NotFound();
            }

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to
delete "
                  + "was modified by another user after you selected delete.
"
                  + "The delete operation was canceled and the current
values in the "
                  + "database have been displayed. If you still want to
delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.ConcurrencyToken value is from when the
entity
```

```
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched. `Department.ConcurrencyToken` is the row version when the entity was fetched. When EF Core creates the `SQL DELETE` command, it includes a WHERE clause with `ConcurrencyToken`. If the `SQL DELETE` command results in zero rows affected:

- The `ConcurrencyToken` in the `SQL DELETE` command doesn't match `ConcurrencyToken` in the database.
- A `DbUpdateConcurrencyException` exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete Razor page

Update `Pages/Departments/Delete.cshtml` with the following code:

CSHTML

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h1>Delete</h1>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.ConcurrencyToken)
        </dt>
        <dd class="col-sm-10">
            @Utility.GetLastChars(Model.Department.ConcurrencyToken)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model =>
model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.ConcurrencyToken" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Changes `ConcurrencyToken` to display the last byte.
- Adds a hidden row version. `ConcurrencyToken` must be added so postback binds the value.

Test concurrency conflicts

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated `ConcurrencyToken` indicator. Note the updated `ConcurrencyToken` indicator, it's displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the database. Clicking **Delete** deletes the entity, unless `ConcurrencyToken` has been updated.

Additional resources

- Concurrency Tokens in EF Core
- Handle concurrency in EF Core
- Debugging ASP.NET Core 2.x source

Next steps

This is the last tutorial in the series. Additional topics are covered in the MVC version of this tutorial series.

Previous tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

ASP.NET Core MVC with EF Core - tutorial series

Article • 04/11/2023

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model. For new development, we recommend Razor Pages over MVC with controllers and views. See the Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- Best practices for EF.

1. Get started
2. Create, Read, Update, and Delete operations
3. Sorting, filtering, paging, and grouping
4. Migrations
5. Create a complex data model
6. Reading related data
7. Updating related data
8. Handle concurrency conflicts
9. Inheritance
10. Advanced topics

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Tutorial: Get started with EF Core in an ASP.NET MVC web app

Article • 04/11/2023

By Tom Dykstra and Rick Anderson

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model. For new development, we recommend Razor Pages over MVC with controllers and views. See the Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- Best practices for EF.

The Contoso University sample web app demonstrates how to create an ASP.NET Core MVC web app using Entity Framework (EF) Core and Visual Studio.

The sample app is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This is the first in a series of tutorials that explain how to build the Contoso University sample app.

Prerequisites

- If you're new to ASP.NET Core MVC, go through the Get started with ASP.NET Core MVC tutorial series before starting this one.

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

This tutorial has not been updated for ASP.NET Core 6 or later. The tutorial's instructions will not work correctly if you create a project that targets ASP.NET Core 6 or later. For example, the ASP.NET Core 6 and later web templates use the minimal hosting model, which unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file.

Another difference introduced in .NET 6 is the NRT (nullable reference types) feature. The project templates enable this feature by default. Problems can happen where EF considers a property to be required in .NET 6 which is nullable in .NET 5. For example, the Create Student page will fail silently unless the `Enrollments` property is made nullable or the `asp-validation-summary` helper tag is changed from `ModelOnly` to `All`.

We recommend that you install and use the .NET 5 SDK for this tutorial. Until this tutorial is updated, see Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8 on how to use Entity Framework with ASP.NET Core 6 or later.

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

Solve problems and troubleshoot

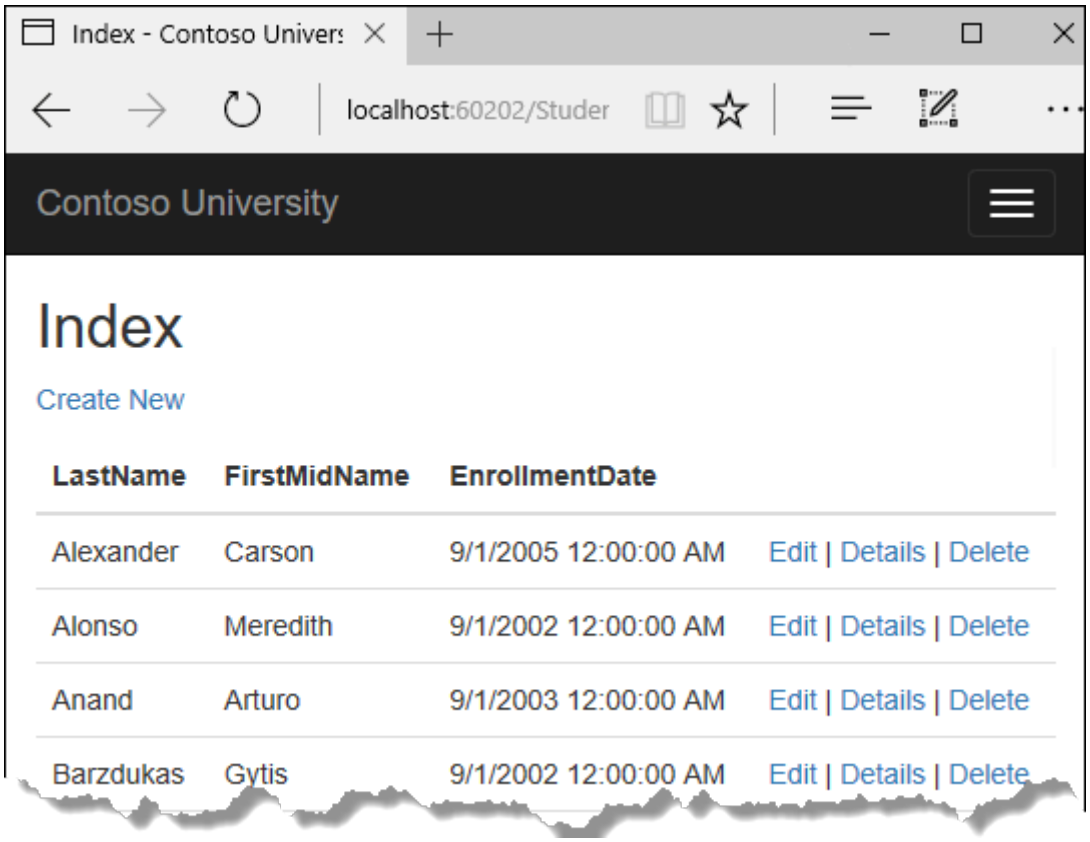
If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the completed project. For a list of common errors and how to solve them, see the Troubleshooting section of the last tutorial in the series. If you don't find what you need there, you can post a question to StackOverflow.com for ASP.NET Core or EF Core.

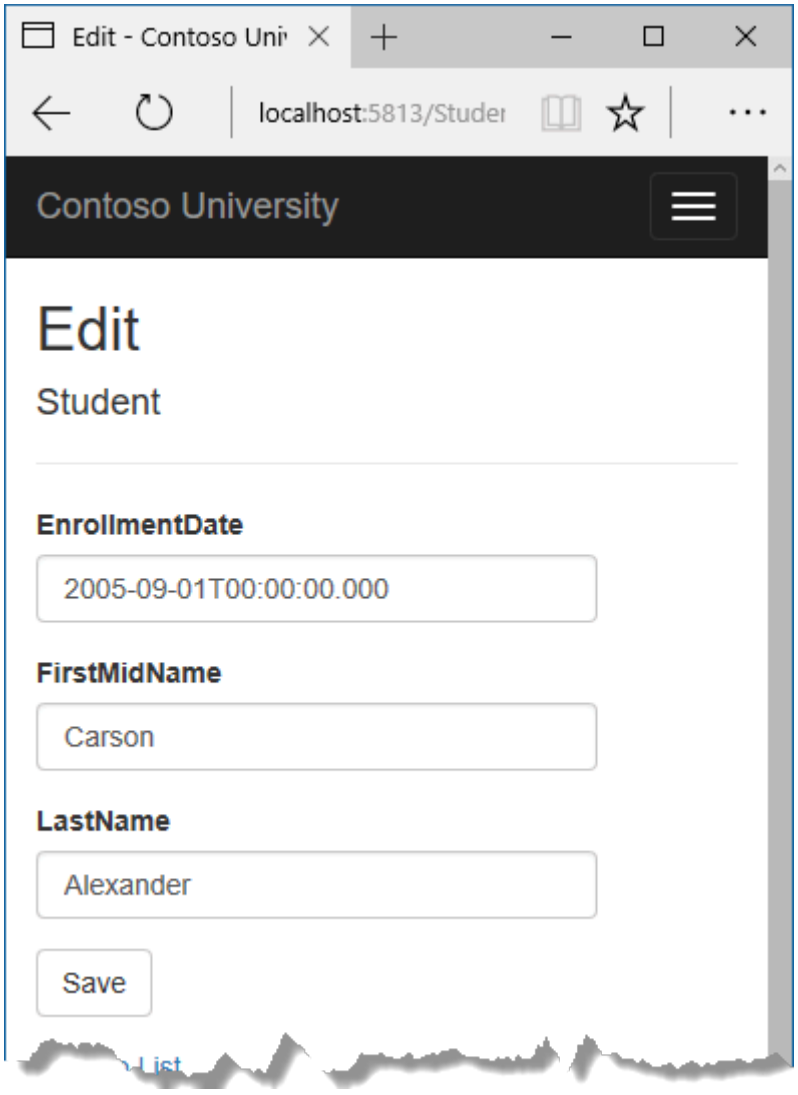
> **Tip**
>
> This is a series of 10 tutorials, each of which builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. Then if you run into problems, you can start over from the previous tutorial instead of going back to the beginning of the whole series.

Contoso University web app

The app built in these tutorials is a basic university web site.

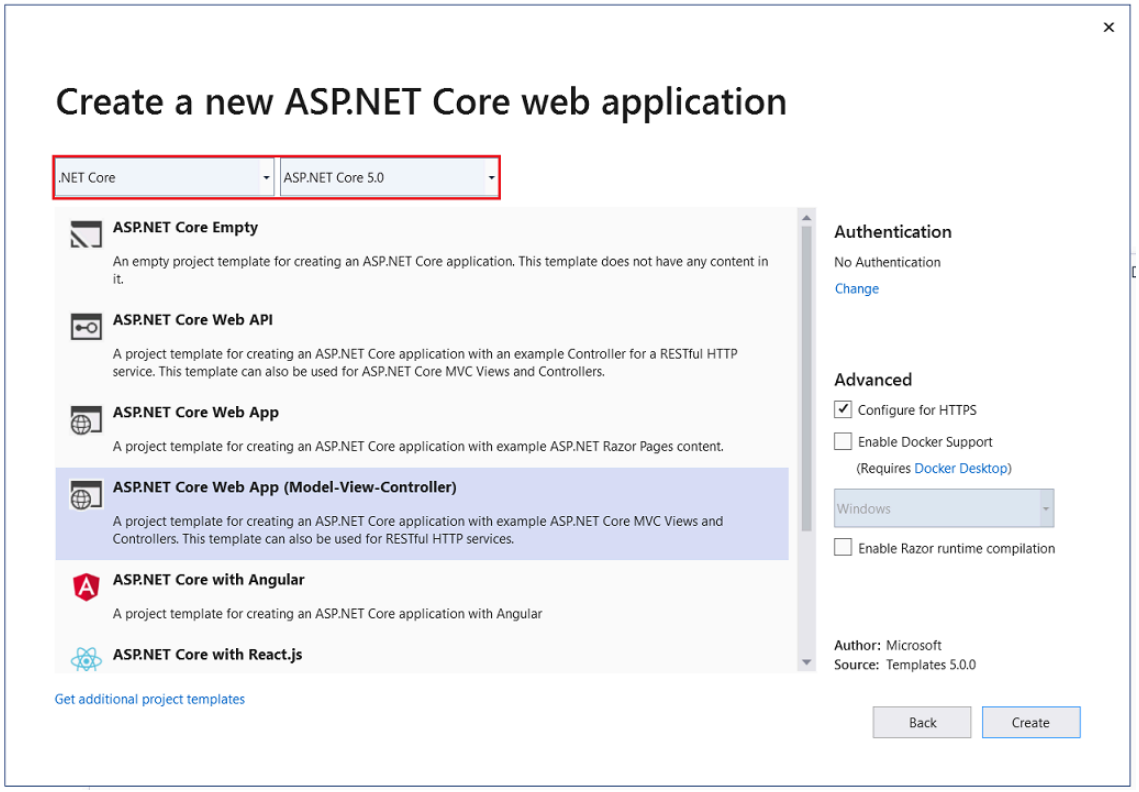
Users can view and update student, course, and instructor information. Here are a few of the screens in the app:


Index - Contoso Univers
localhost:60202/Studer
Contoso University
Index
Create New
LastName
FirstMidName
EnrollmentDate
Alexander
Carson
9/1/2005 12:00:00 AM
Edit | Details | Delete
Alonso
Meredith
9/1/2002 12:00:00 AM
Edit | Details | Delete
Anand
Arturo
9/1/2003 12:00:00 AM
Edit | Details | Delete
Barzdukas
Gytis
9/1/2002 12:00:00 AM
Edit | Details | Delete


Edit - Contoso Uni
localhost:5813/Studer
Contoso University
Edit
Student
EnrollmentDate
2005-09-01T00:00:00.000
FirstMidName
Carson
LastName
Alexander
Save

Create web app

1. Start Visual Studio and select **Create a new project**.
2. In the **Create a new project** dialog, select **ASP.NET Core Web Application** > **Next**.
3. In the **Configure your new project** dialog, enter `ContosoUniversity` for **Project name**. It's important to use this exact name including capitalization, so each `namespace` matches when code is copied.
4. Select **Create**.
5. In the **Create a new ASP.NET Core web application** dialog, select:
   a. **.NET Core** and **ASP.NET Core 5.0** in the dropdowns.
   b. **ASP.NET Core Web App (Model-View-Controller)**.
   c. **Create**



Set up the site style

A few basic changes set up the site menu, layout, and home page.

Open `Views/Shared/_Layout.cshtml` and make the following changes:

- Change each occurrence of `ContosoUniversity` to `Contoso University`. There are three occurrences.
- Add menu entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Privacy** menu entry.

The preceding changes are highlighted in the following code:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home"
asp-action="Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Students" asp-action="Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Courses" asp-action="Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Instructors" asp-action="Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Departments" asp-action="Index">Departments</a>
                        </li>
                    </ul>
                </div>
```

```
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2020 - Contoso University - <a asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

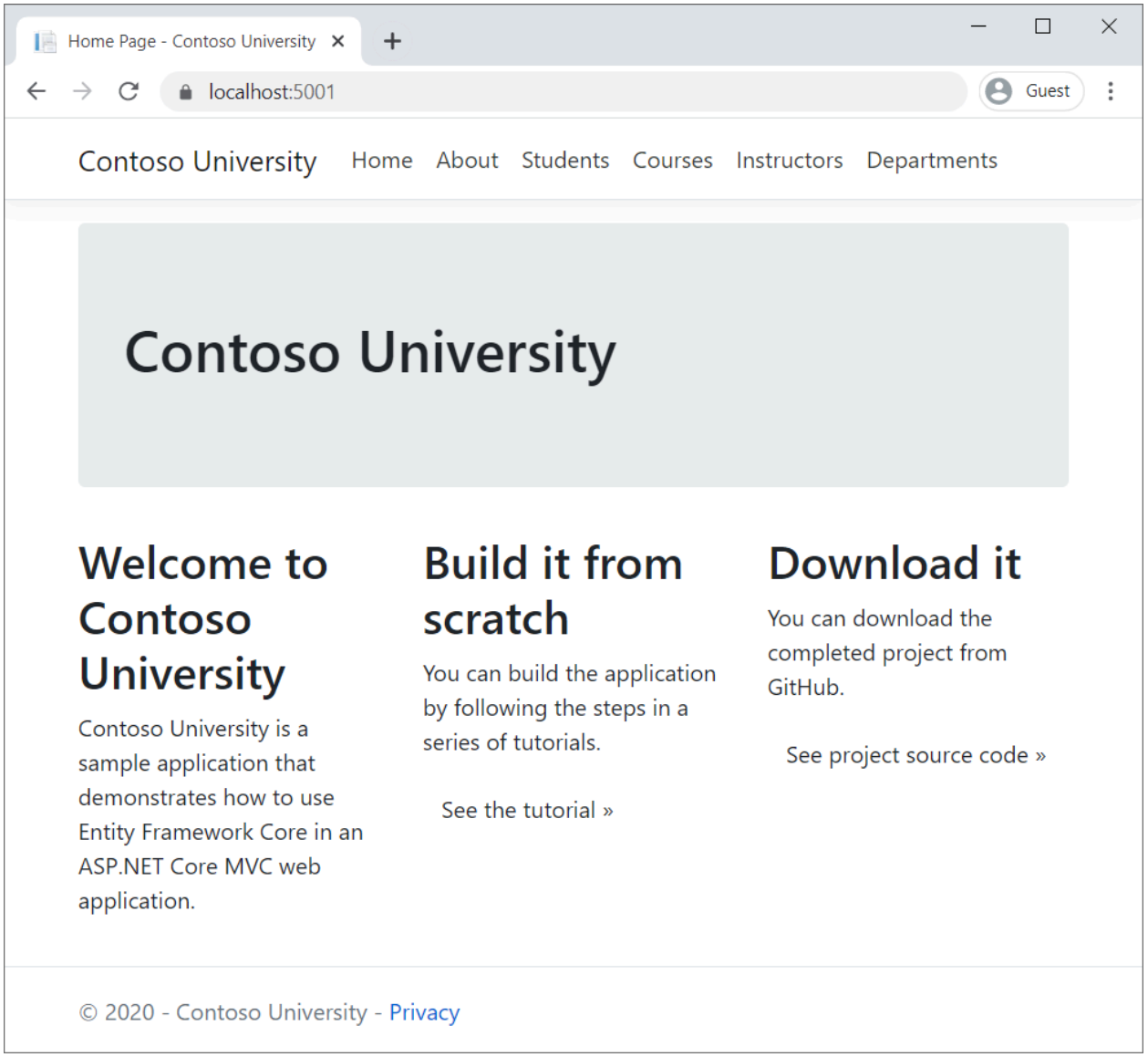
In `Views/Home/Index.cshtml`, replace the contents of the file with the following markup:

CSHTML

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series
of tutorials.</p>
        <p><a class="btn btn-default"
href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
```

```
        <p><a class="btn btn-default"
href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef
-mvc/intro/samples/5cu-final">See project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project or choose **Debug** > **Start Without Debugging** from the menu. The home page is displayed with tabs for the pages created in this tutorial.



EF Core NuGet packages

This tutorial uses SQL Server, and the provider package is Microsoft.EntityFrameworkCore.SqlServer.

The EF SQL Server package and its dependencies, `Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`, provide runtime support for EF.

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package. In the Package Manager Console (PMC), enter the following commands to add the NuGet packages:

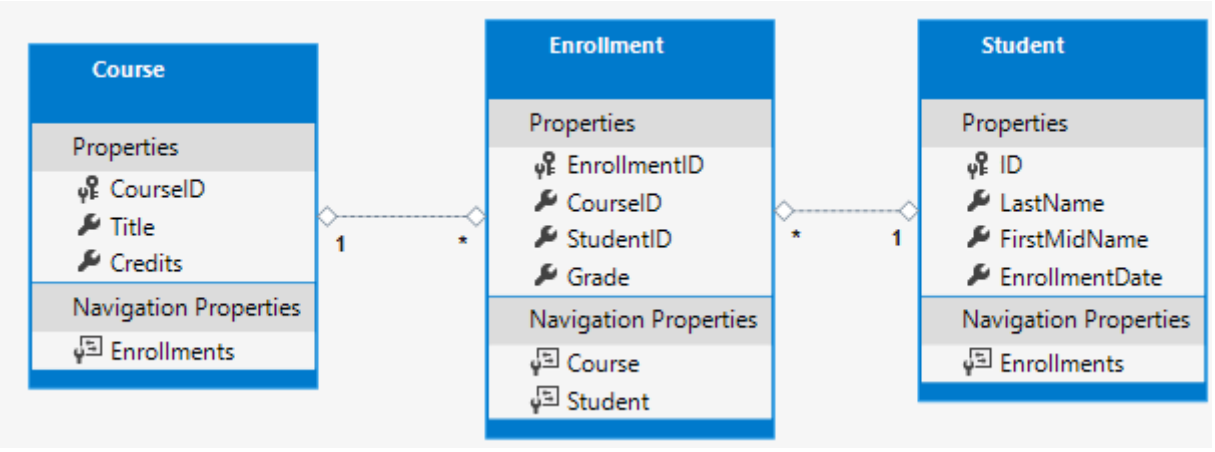
```
PowerShell

Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

The `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` NuGet package provides ASP.NET Core middleware for EF Core error pages. This middleware helps to detect and diagnose errors with EF Core migrations.

For information about other database providers that are available for EF Core, see Database providers.

Create the data model

The following entity classes are created for this app:

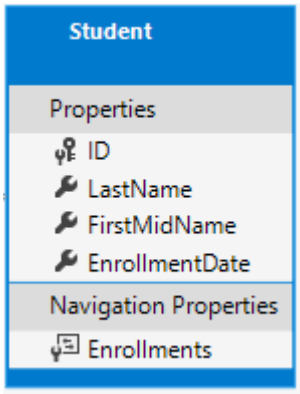


The preceding entities have the following relationships:

- A one-to-many relationship between `Student` and `Enrollment` entities. A student can be enrolled in any number of courses.
- A one-to-many relationship between `Course` and `Enrollment` entities. A course can have any number of students enrolled in it.

In the following sections, a class is created for each of these entities.

The Student entity



In the *Models* folder, create the `Student` class with the following code:

C#

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property is the primary key (**PK**) column of the database table that corresponds to this class. By default, EF interprets a property that's named `ID` or `classnameID` as the primary key. For example, the PK could be named `StudentID` rather than `ID`.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. The `Enrollments` property of a `Student` entity:

- Contains all of the `Enrollment` entities that are related to that `Student` entity.
- If a specific `Student` row in the database has two related `Enrollment` rows:
  - That `Student` entity's `Enrollments` navigation property contains those two `Enrollment` entities.

`Enrollment` rows contain a student's PK value in the `StudentID` foreign key (**FK**) column.

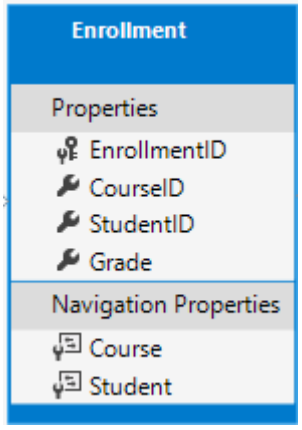
If a navigation property can hold multiple entities:

- The type must be a list, such as `ICollection<T>`, `List<T>`, or `HashSet<T>`.

- Entities can be added, deleted, and updated.

Many-to-many and one-to-many navigation relationships can contain multiple entities. When `ICollection<T>` is used, EF creates a `HashSet<T>` collection by default.

The Enrollment entity



In the *Models* folder, create the `Enrollment` class with the following code:

C#

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the PK. This entity uses the `classnameID` pattern instead of `ID` by itself. The `Student` entity used the `ID` pattern. Some developers prefer to use one pattern throughout the data model. In this tutorial, the variation illustrates that either pattern can be used. A later tutorial shows how using `ID` without classname makes it easier to implement inheritance in the data model.

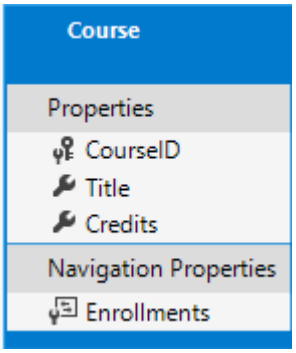
The `Grade` property is an `enum`. The `?` after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's `null` is different from a zero grade. `null` means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key (FK), and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity. This differs from the `Student.Enrollments` navigation property, which can hold multiple `Enrollment` entities.

The `CourseID` property is a FK, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a FK property if it's named `<`navigation property name`><`primary key property name`>`. For example, `StudentID` for the `Student` navigation property since the `Student` entity's PK is `ID`. FK properties can also be named `<`primary key property name`>`. For example, `CourseID` because the `Course` entity's PK is `CourseID`.

The Course entity



In the *Models* folder, create the `Course` class with the following code:

C#

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
```

```
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The DatabaseGenerated attribute is explained in a later tutorial. This attribute allows entering the PK for the course rather than having the database generate it.

Create the database context

The main class that coordinates EF functionality for a given data model is the DbContext database context class. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. The `DbContext` derived class specifies which entities are included in the data model. Some EF behaviors can be customized. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named `Data`.

In the *Data* folder create a `SchoolContext` class with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

The preceding code creates a `DbSet` property for each entity set. In EF terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

The `DbSet<Enrollment>` and `DbSet<Course>` statements could be omitted and it would work the same. EF would include them implicitly because:

- The `Student` entity references the `Enrollment` entity.
- The `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural. For example, `Students` rather than `Student`. Developers disagree about whether table names should be pluralized or not. For these tutorials, the default behavior is overridden by specifying singular table names in the `DbContext`. To do that, add the following highlighted code after the last DbSet property.

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Register the `SchoolContext`

ASP.NET Core includes dependency injection. Services, such as the EF database context, are registered with dependency injection during app startup. Components that require these services, such as MVC controllers, are provided these services via constructor

parameters. The controller constructor code that gets a context instance is shown later in this tutorial.

To register `SchoolContext` as a service, open `Startup.cs`, and add the highlighted lines to the `ConfigureServices` method.

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace ContosoUniversity
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

            services.AddControllersWithViews();
        }
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the ASP.NET Core configuration system reads the connection string from the `appsettings.json` file.

Open the `appsettings.json` file and add a connection string as shown in the following markup:

JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;
MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter to `ConfigureServices` as shown in the following code:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

    services.AddDatabaseDeveloperPageExceptionFilter();

    services.AddControllersWithViews();
}
```

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment.

SQL Server Express LocalDB

The connection string specifies SQL Server LocalDB. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB creates *.mdf* DB files in the `C:/Users/<user>` directory.

Initialize DB with test data

EF creates an empty database. In this section, a method is added that's called after the database is created in order to populate it with test data.

The `EnsureCreated` method is used to automatically create the database. In a later tutorial, you see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the *Data* folder, create a new class named `DbInitializer` with the following code:

C#

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.P
arse("2005-09-01")},
            new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Pa
rse("2002-09-01")},
            new
Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse
("2003-09-01")},
            new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Pa
rse("2002-09-01")},
            new
Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002
-09-01")},
            new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Pars
e("2001-09-01")},
            new
```

```
Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse
("2003-09-01")},
            new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Pars
e("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollments.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

The preceding code checks if the database exists:

- If the database is not found;
  - It is created and loaded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.
- If the database is found, it takes no action.

Update `Program.cs` with the following code:

C#

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>
();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger =
services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the
DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
```

```
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

`Program.cs` does the following on app startup:

- Get a database context instance from the dependency injection container.
- Call the `DbInitializer.Initialize` method.
- Dispose the context when the `Initialize` method completes as shown in the following code:

C#

```
public static void Main(string[] args)
{
     var host = CreateWebHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the
database.");
        }
    }

    host.Run();
}
```

The first time the app is run, the database is created and loaded with test data. Whenever the data model changes:

- Delete the database.
- Update the seed method, and start afresh with a new database.

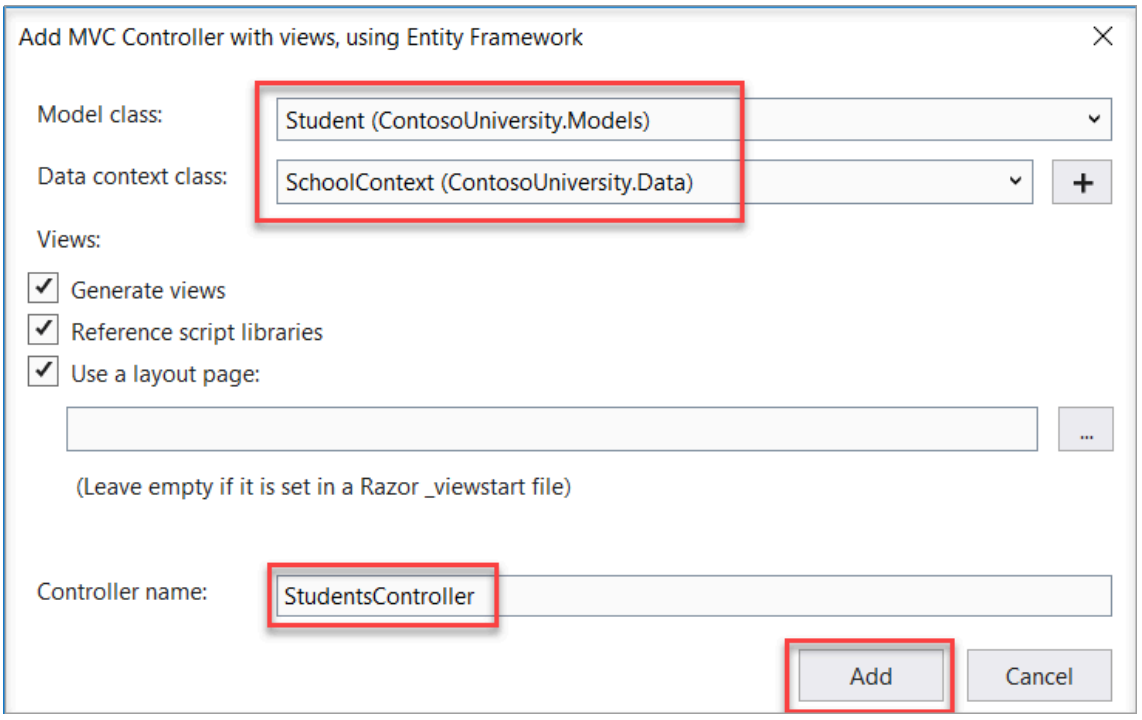
In later tutorials, the database is modified when the data model changes, without deleting and re-creating it. No data is lost when the data model changes.

Create controller and views

Use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding.

- In **Solution Explorer**, right-click the `Controllers` folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog box:
  - Select **MVC controller with views, using Entity Framework**.
  - Click **Add**. The **Add MVC Controller with views, using Entity Framework** dialog box appears:



  - In **Model class**, select **Student**.
  - In **Data context class**, select **SchoolContext**.
  - Accept the default **StudentsController** as the name.
  - Click **Add**.

The Visual Studio scaffolding engine creates a `StudentsController.cs` file and a set of views (`*.cshtml` files) that work with the controller.

Notice the controller takes a `SchoolContext` as a constructor parameter.

C#

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
```

```
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
```

ASP.NET Core dependency injection takes care of passing an instance of `SchoolContext` into the controller. You configured that in the `Startup` class.

The controller contains an `Index` action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the `Students` property of the database context instance:

C#

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

The asynchronous programming elements in this code are explained later in the tutorial.

The `Views/Students/Index.cshtml` view displays this list in a table:

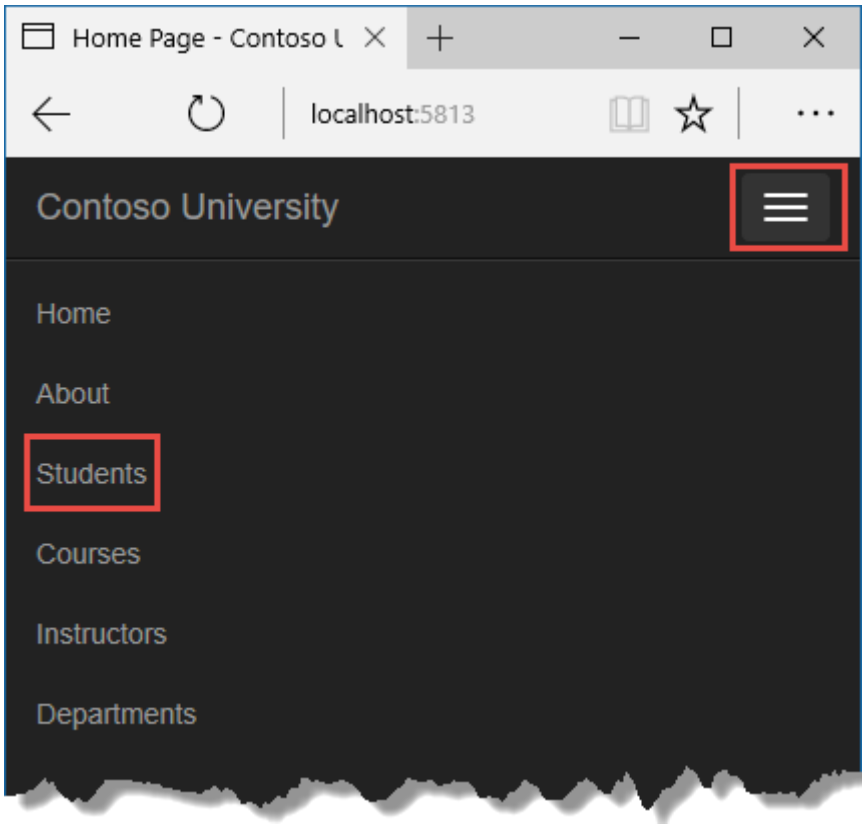
CSHTML

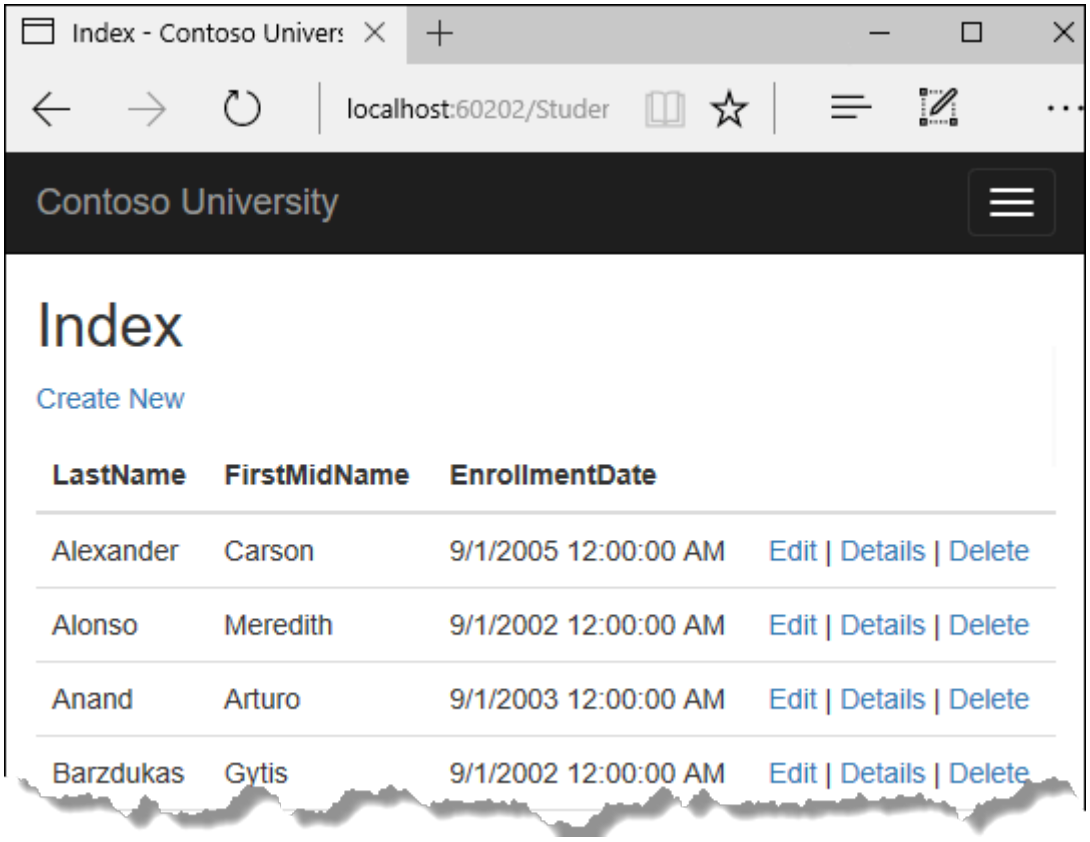
```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.LastName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.EnrollmentDate)
                </th>
```

```
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a>
|
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Press CTRL+F5 to run the project or choose **Debug** > **Start Without Debugging** from the menu.

Click the Students tab to see the test data that the `DbInitializer.Initialize` method inserted. Depending on how narrow your browser window is, you'll see the `Students` tab link at the top of the page or you'll have to click the navigation icon in the upper right corner to see the link.





View the database

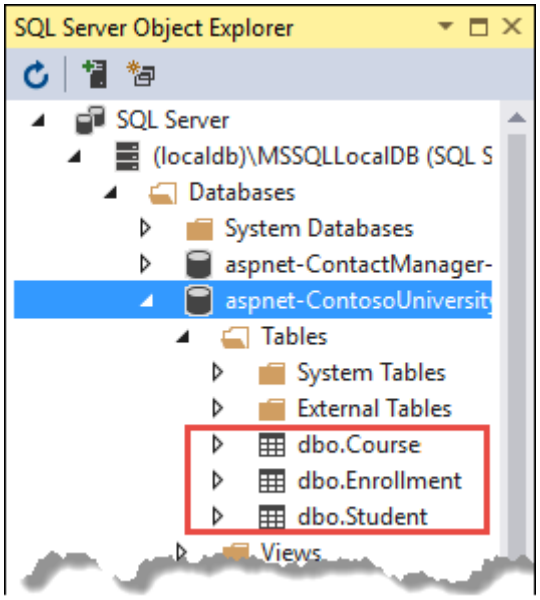
When the app is started, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database:

- So it created a database.

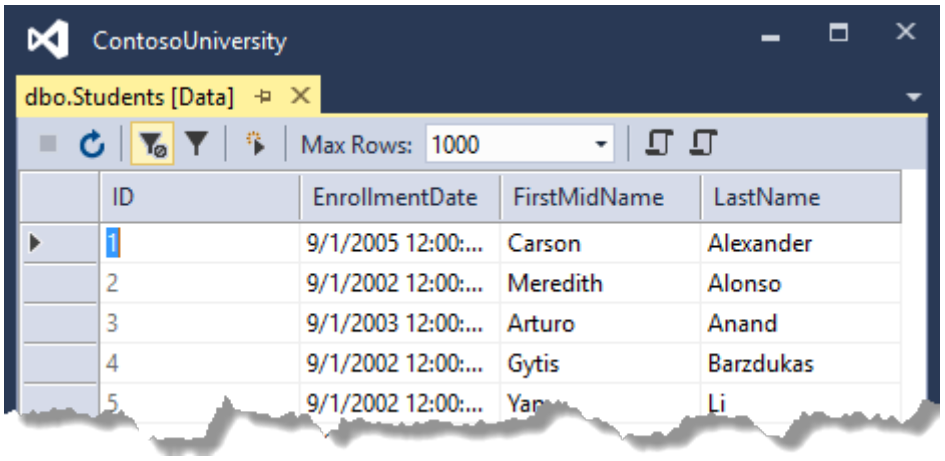
- The `Initialize` method code populated the database with data.

Use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio:

- Select **SQL Server Object Explorer** from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB** > **Databases**.
- Select `ContosoUniversity1`, the entry for the database name that's in the connection string in the `appsettings.json` file.
- Expand the **Tables** node to see the tables in the database.



Right-click the **Student** table and click **View Data** to see the data in the table.



The `*.mdf` and `*.ldf` database files are in the *C:\Users\<username>* folder.

Because `EnsureCreated` is called in the initializer method that runs on app start, you could:

- Make a change to the `Student` class.
- Delete the database.

- Stop, then start the app. The database is automatically re-created to match the change.

For example, if an `EmailAddress` property is added to the `Student` class, a new `EmailAddress` column in the re-created table. The view won't display the new `EmailAddress` property.

Conventions

The amount of code written in order for the EF to create a complete database is minimal because of the use of the conventions EF uses:

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or `classnameID` are recognized as PK properties.
- A property is interpreted as a FK property if it's named `<`navigation property name`><`PK property name`>`. For example, `StudentID` for the `Student` navigation property since the `Student` entity's PK is `ID`. FK properties can also be named `<`primary key property name`>`. For example, `EnrollmentID` since the `Enrollment` entity's PK is `EnrollmentID`.

Conventional behavior can be overridden. For example, table names can be explicitly specified, as shown earlier in this tutorial. Column names and any property can be set as a PK or FK.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, `async`, `Task<T>`, `await`, and `ToListAsync` make the code execute asynchronously.

```
C#

public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that's returned.
- The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes, for example, `ToListAsync`, `SingleOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include, for example, statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF context isn't thread safe: don't try to do multiple operations in parallel. When you call any async EF method, always use the `await` keyword.
- To take advantage of the performance benefits of async code, make sure that any library packages used also use async if they call any EF methods that cause queries to be sent to the database.

For more information about asynchronous programming in .NET, see Async Overview.

Limit entities fetched

See Performance considerations for information on limiting the number of entities returned from a query.

SQL Logging of Entity Framework Core

Logging configuration is commonly provided by the `Logging` section of `appsettings.{Environment}.json` files. To log SQL statements, add `"Microsoft.EntityFrameworkCore.Database.Command": "Information"` to the `appsettings.Development.json` file:

JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDB-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
     ,"Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

With the preceding JSON, SQL statements are displayed on the command line and in the Visual Studio output window.

For more information, see Logging in .NET Core and ASP.NET Core and this GitHub issue.

Advance to the next tutorial to learn how to perform basic CRUD (create, read, update, delete) operations.

Implement basic CRUD functionality

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Tutorial: Implement CRUD Functionality - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial, you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

> **Note**
>
> It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about repositories with EF, see the last tutorial in this series.

In this tutorial, you:

- ✓ Customize the Details page
- ✓ Update the Create page
- ✓ Update the Edit page
- ✓ Update the Delete page
- ✓ Close database connections

Prerequisites

- Get started with EF Core and ASP.NET Core MVC

Customize the Details page

The scaffolded code for the Students Index page left out the `Enrollments` property, because that property holds a collection. In the **Details** page, you'll display the contents of the collection in an HTML table.

In `Controllers/StudentsController.cs`, the action method for the Details view uses the `FirstOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls

`Include`. `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .Include(s => s.Enrollments)
            .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. You'll learn more about these methods in the read related data tutorial.

The `AsNoTracking` method improves performance in scenarios where the entities returned won't be updated in the current context's lifetime. You'll learn more about `AsNoTracking` at the end of this tutorial.

Route data

The key value that's passed to the `Details` method comes from *route data*. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:

C#

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
```

```
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the `Index` method `id` parameter if you pass it as a query string value:

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the `id` parameter matches the default route, so `id` is added to the route data.

HTML

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

HTML

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, `studentID` doesn't match a parameter in the default route, so it's added as a query string.

HTML

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

HTML

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

For more information about tag helpers, see Tag Helpers in ASP.NET Core.

Add enrollments to the Details view

Open `Views/Students/Details.cshtml`. Each field is displayed using `DisplayNameFor` and `DisplayFor` helpers, as shown in the following example:

CSHTML

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd class="col-sm-10">
    @Html.DisplayFor(model => model.LastName)
</dd>
```

After the last field and immediately before the closing `</dl>` tag, add the following code to display a list of enrollments:

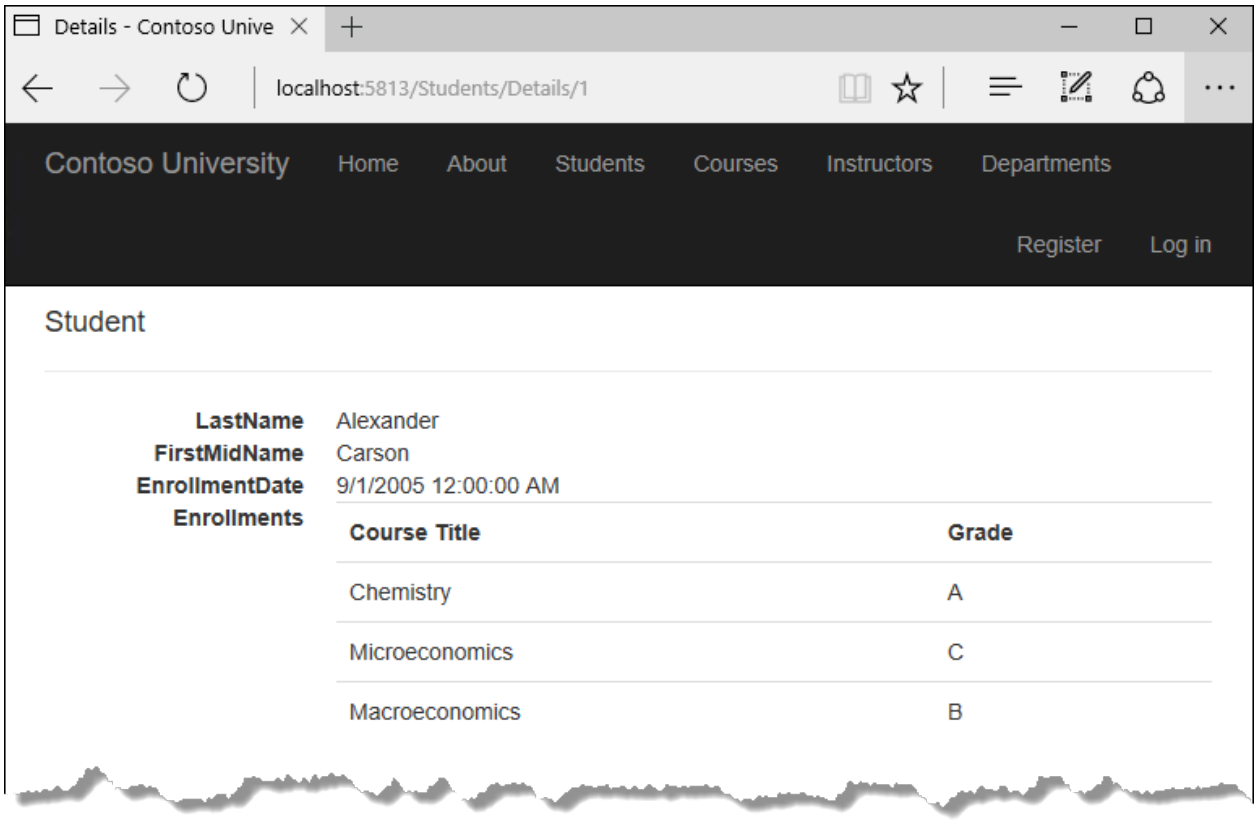
CSHTML

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd class="col-sm-10">
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. You see the list of courses and grades for the selected student:



Update the Create page

In `StudentsController.cs`, modify the HttpPost `Create` method by adding a try-catch block and removing ID from the `Bind` attribute.

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
```

```
            }
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists " +
                "see your system administrator.");
        }
        return View(student);
    }
```

This code adds the Student entity created by the ASP.NET Core MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET Core MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed `ID` from the `Bind` attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user doesn't set the ID value.

Other than the `Bind` attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from `DbUpdateException` is caught while the changes are being saved, a generic error message is displayed. `DbUpdateException` exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in Monitoring and Telemetry (Building Real-World Cloud Apps with Azure).

The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the FormTagHelper and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

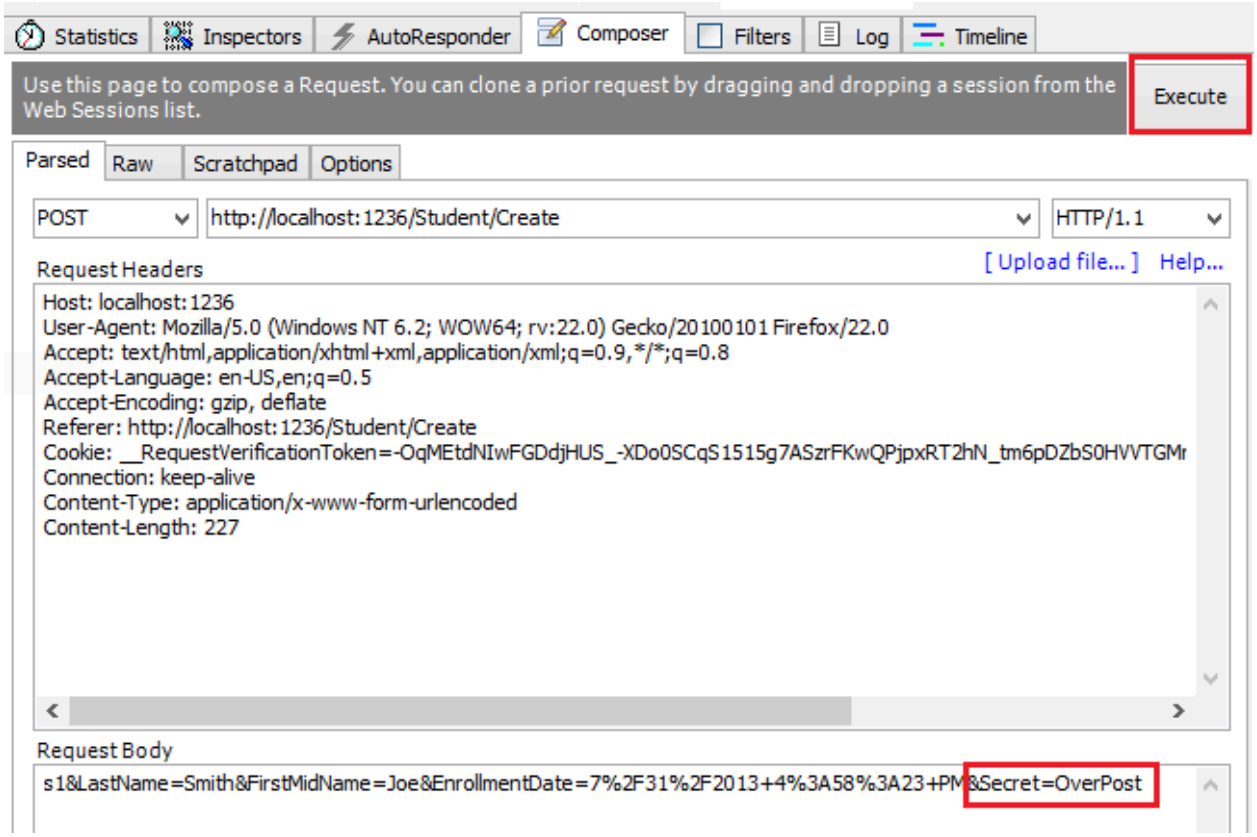
Security note about overposting

The `Bind` attribute that the scaffolded code includes on the `Create` method is one way to protect against overposting in create scenarios. For example, suppose the Student entity includes a `Secret` property that you don't want this web page to set.

C#

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a Student instance, the model binder would pick up that `Secret` form value and use it to create the Student entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to set that property.

You can prevent overposting in edit scenarios by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That's the method used in these tutorials.

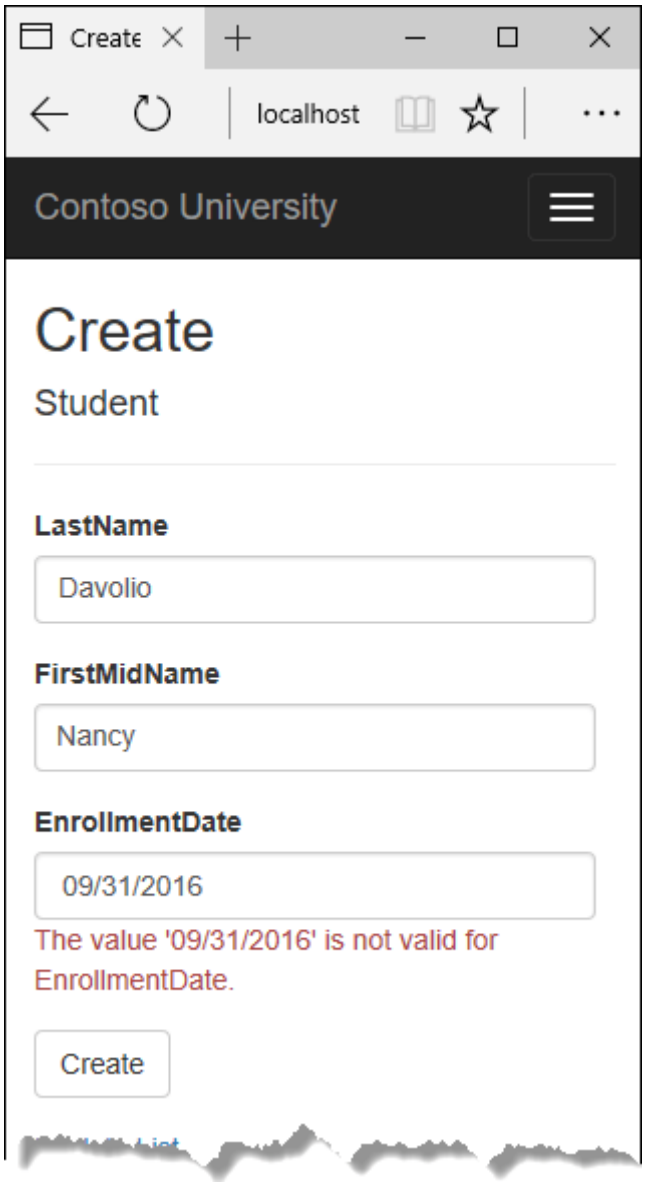
An alternative way to prevent overposting that's preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use `_context.Entry` on the entity instance to set its state to `Unchanged`, and then set `Property("PropertyName").IsModified` to true on each entity property that's included in the view model. This method works in both edit and create scenarios.

Test the Create page

The code in `Views/Students/Create.cshtml` uses `label`, `input`, and `span` (for validation messages) tag helpers for each field.

Run the app, select the **Students** tab, and click **Create New**.

Enter names and a date. Try entering an invalid date if your browser lets you do that. (Some browsers force you to use a date picker.) Then click **Create** to see the error message.



This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the `Create` method.

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
```

```
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

Update the Edit page

In `StudentController.cs`, the HttpGet `Edit` method (the one without the `HttpPost` attribute) uses the `FirstOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the `Details` method. You don't need to change this method.

Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.FirstOrDefaultAsync(s =>
s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
```

```
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}
```

These changes implement a security best practice to prevent overposting. The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a `Modified` flag. That code isn't recommended for many scenarios because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter.

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity based on user input in the posted form data. The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

As a best practice to prevent overposting, the fields that you want to be updateable by the **Edit** page are declared in the `TryUpdateModel` parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the HttpPost `Edit` method is the same as the HttpGet `Edit` method; therefore you've renamed the method `EditPost`.

Alternative HttpPost Edit code: Create and attach

The recommended HttpPost edit code ensures that only changed columns get updated and preserves data in properties that you don't want included for model binding. However, the read-first approach requires an extra database read, and can result in more complex code for handling concurrency conflicts. An alternative is to attach an entity created by the model binder to the EF context and mark it as modified. (Don't update your project with this code, it's only shown to illustrate an optional approach.)

C#

```
public async Task<IActionResult> Edit(int id,
[Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student student)
```

```
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}
```

You can use this approach when the web page UI includes all of the fields in the entity and can update any of them.

The scaffolded code uses the create-and-attach approach but only catches `DbUpdateConcurrencyException` exceptions and returns 404 error codes. The example shown catches any database update exception and displays an error message.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the `Add` method, that entity's state is set to `Added`. Then when you call the `SaveChanges` method, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`. The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.

- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. You read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

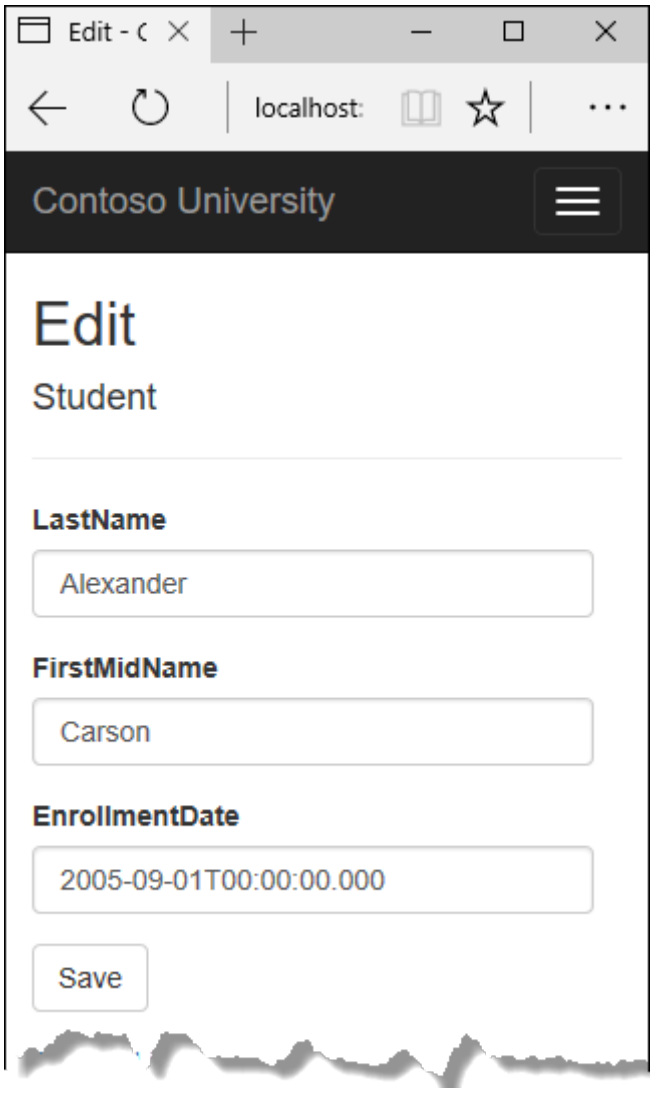
In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the HttpPost `Edit` action method is called, a new web request is made and you have a new instance of the `DbContext`. If you re-read the entity in that new context, you simulate desktop processing.

But if you don't want to do the extra read operation, you have to use the entity object created by the model binder. The simplest way to do this is to set the entity state to Modified as is done in the alternative HttpPost Edit code shown earlier. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want to avoid the read-first approach, but you also want the SQL UPDATE statement to update only the fields that the user actually changed, the code is more complex. You have to save the original values in some way (such as by using hidden fields) so that they're available when the HttpPost `Edit` method is called. Then you can create a Student entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`.

Test the Edit page

Run the app, select the **Students** tab, then click an **Edit** hyperlink.



Change some of the data and click **Save**. The **Index** page opens and you see the changed data.

Update the Delete page

In `StudentController.cs`, the template code for the HttpGet `Delete` method uses the `FirstOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that's called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the HttpPost `Delete` method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the HttpPost `Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the HttpPost Delete

method calls the HttpGet Delete method, passing it a parameter that indicates that an error has occurred. The HttpGet Delete method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

Replace the HttpGet `Delete` action method with the following code, which manages error reporting.

C#

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError =
false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the HttpGet `Delete` method is called without a previous failure. When it's called by the HttpPost `Delete` method in response to a database update error, the parameter is true and an error message is passed to the view.

The read-first approach to HttpPost Delete

Replace the HttpPost `Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

C#

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students.FindAsync(id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id,
saveChangesError = true });
    }
}
```

This code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated.

The create-and-attach approach to HttpPost Delete

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query by instantiating a Student entity using only the primary key value and then setting the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity. (Don't put this code in your project; it's here just to illustrate an alternative.)

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
```

```
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id,
saveChangesError = true });
    }
}
```

If the entity has related data that should also be deleted, make sure that cascade delete is configured in the database. With this approach to entity deletion, EF might not realize there are related entities to be deleted.

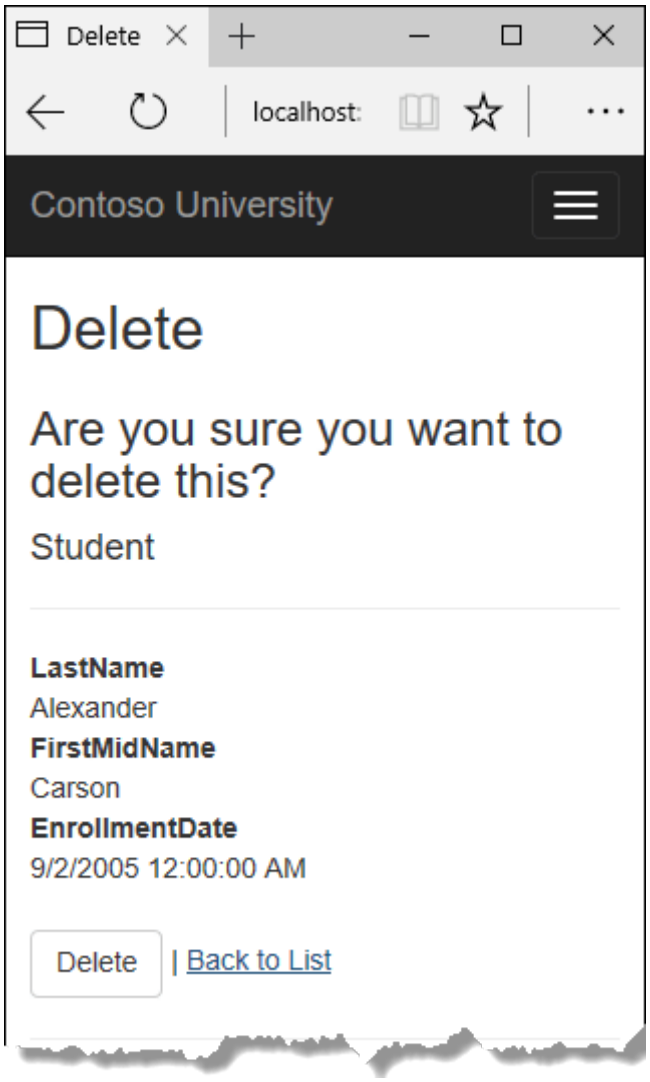
Update the Delete view

In `Views/Student/Delete.cshtml`, add an error message between the h2 heading and the h3 heading, as shown in the following example:

CSHTML

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the app, select the **Students** tab, and click a **Delete** hyperlink:



Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Close database connections

To free up the resources that a database connection holds, the context instance must be disposed as soon as possible when you are done with it. The ASP.NET Core built-in dependency injection takes care of that task for you.

In `Startup.cs`, you call the AddDbContext extension method to provision the `DbContext` class in the ASP.NET Core DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handle transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or they all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by calling the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

- During the context lifetime you don't need to update any entities, and you don't need EF to automatically load navigation properties with entities retrieved by separate queries. Frequently these conditions are met in a controller's HttpGet action methods.
- You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.
- You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see Tracking vs. No-Tracking.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized the Details page
- ✓ Updated the Create page
- ✓ Updated the Edit page
- ✓ Updated the Delete page
- ✓ Closed database connections

Advance to the next tutorial to learn how to expand the functionality of the **Index** page by adding sorting, filtering, and paging.

Next: Sorting, filtering, and paging

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

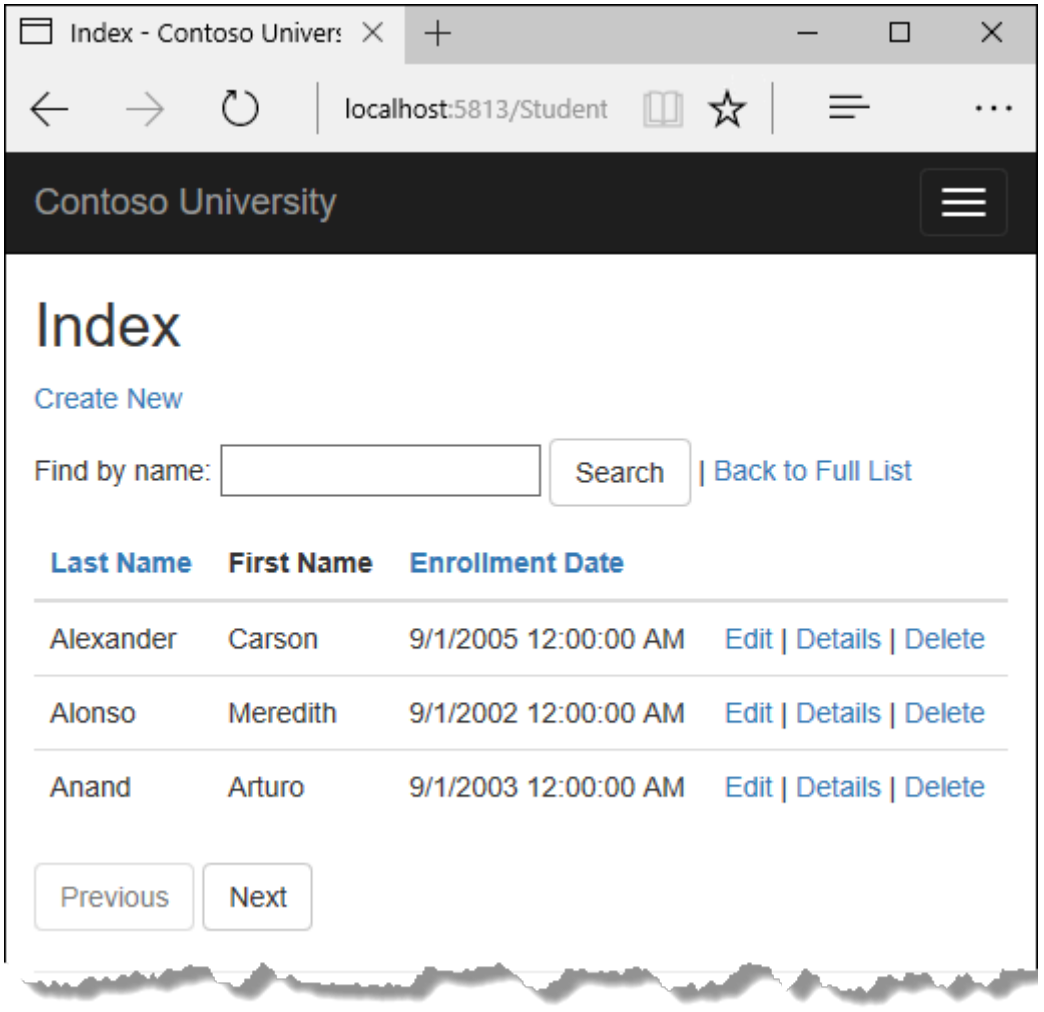
Provide product feedback

Tutorial: Add sorting, filtering, and paging - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for Student entities. In this tutorial you'll add sorting, filtering, and paging functionality to the Students Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



In this tutorial, you:

- ✓ Add column sort links
- ✓ Add a Search box
- ✓ Add paging to Students Index
- ✓ Add paging to Index method
- ✓ Add paging links

✔ Create an About page

Prerequisites

- Implement CRUD Functionality

Add column sort links

To add sorting to the Student Index page, you'll change the `Index` method of the Students controller and add code to the Student Index view.

Add sorting Functionality to the Index method

In `StudentsController.cs`, replace the `Index` method with the following code:

C#

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an

underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewData` elements (NameSortParm and DateSortParm) are used by the view to configure the column heading hyperlinks with the appropriate query string values.

C#

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, NameSortParm should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

Expand table

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an `IQueryable` variable before the switch statement, modifies it in the switch statement, and calls the `ToListAsync` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query isn't executed until you convert the `IQueryable` object into a collection by calling a method such as `ToListAsync`. Therefore, this code results in a single query that's not executed until the `return View` statement.

This code could get verbose with a large number of columns. The last tutorial in this series shows how to write code that lets you pass the name of the `OrderBy` column in a string variable.

Add column heading hyperlinks to the Student Index view

Replace the code in `Views/Students/Index.cshtml`, with the following code to add column heading hyperlinks. The changed lines are highlighted.

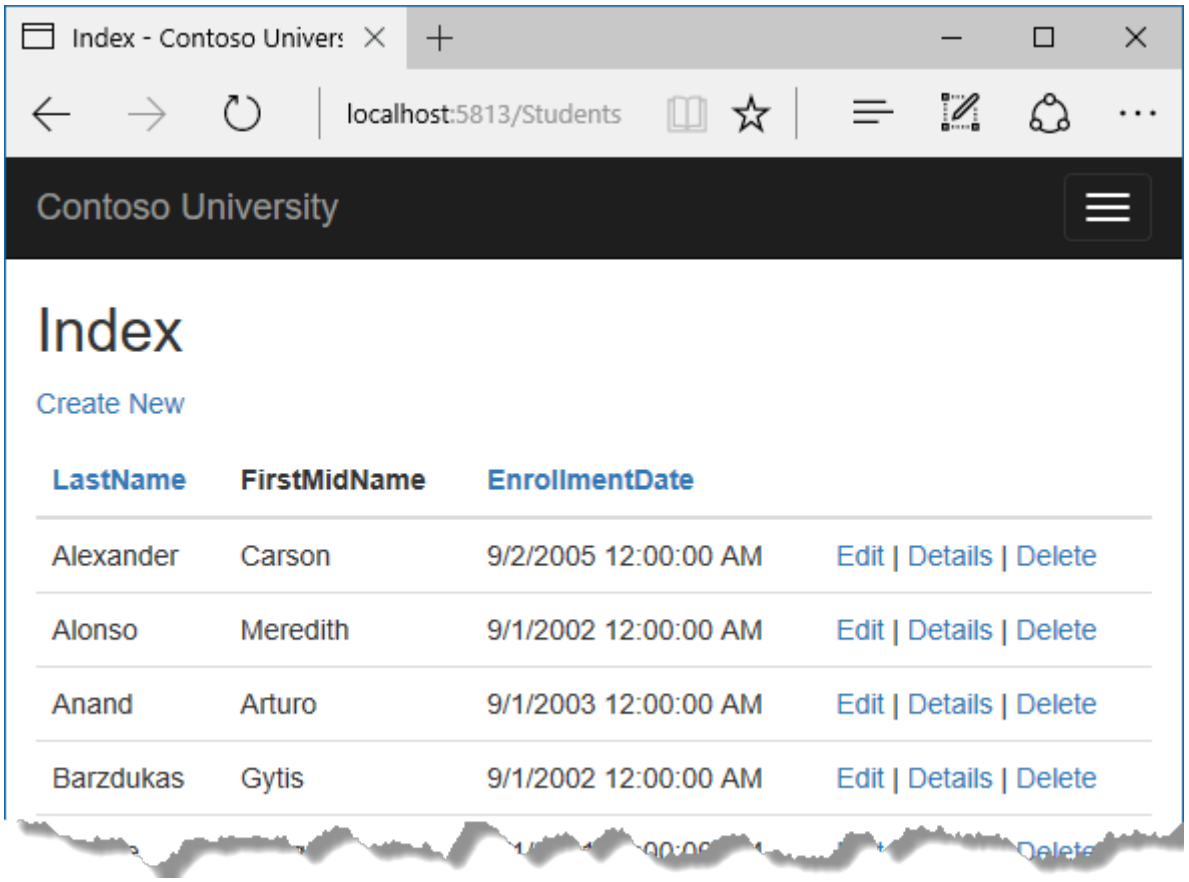
CSHTML

```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model =>
model.LastName)</a>
                </th>
```

```
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model =>
model.EnrollmentDate)</a>
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a>
|
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

This code uses the information in `ViewData` properties to set up hyperlinks with the appropriate query string values.

Run the app, select the **Students** tab, and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Add a Search box

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

In `StudentsController.cs`, replace the `Index` method with the following code (the changes are highlighted).

C#

```
public async Task<IActionResult> Index(string sortOrder, string
searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
```

```
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

You've added a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

> ⓘ **Note**
>
> Here you are calling the `Where` method on an `IQueryable` object, and the filter will be processed on the server. In some scenarios you might be calling the `Where` method as an extension method on an in-memory collection. (For example, suppose you change the reference to `_context.Students` so that instead of an EF `DbSet` it references a repository method that returns an `IEnumerable` collection.) The result would normally be the same but in some cases may be different.
>
> For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the

.NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there's a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

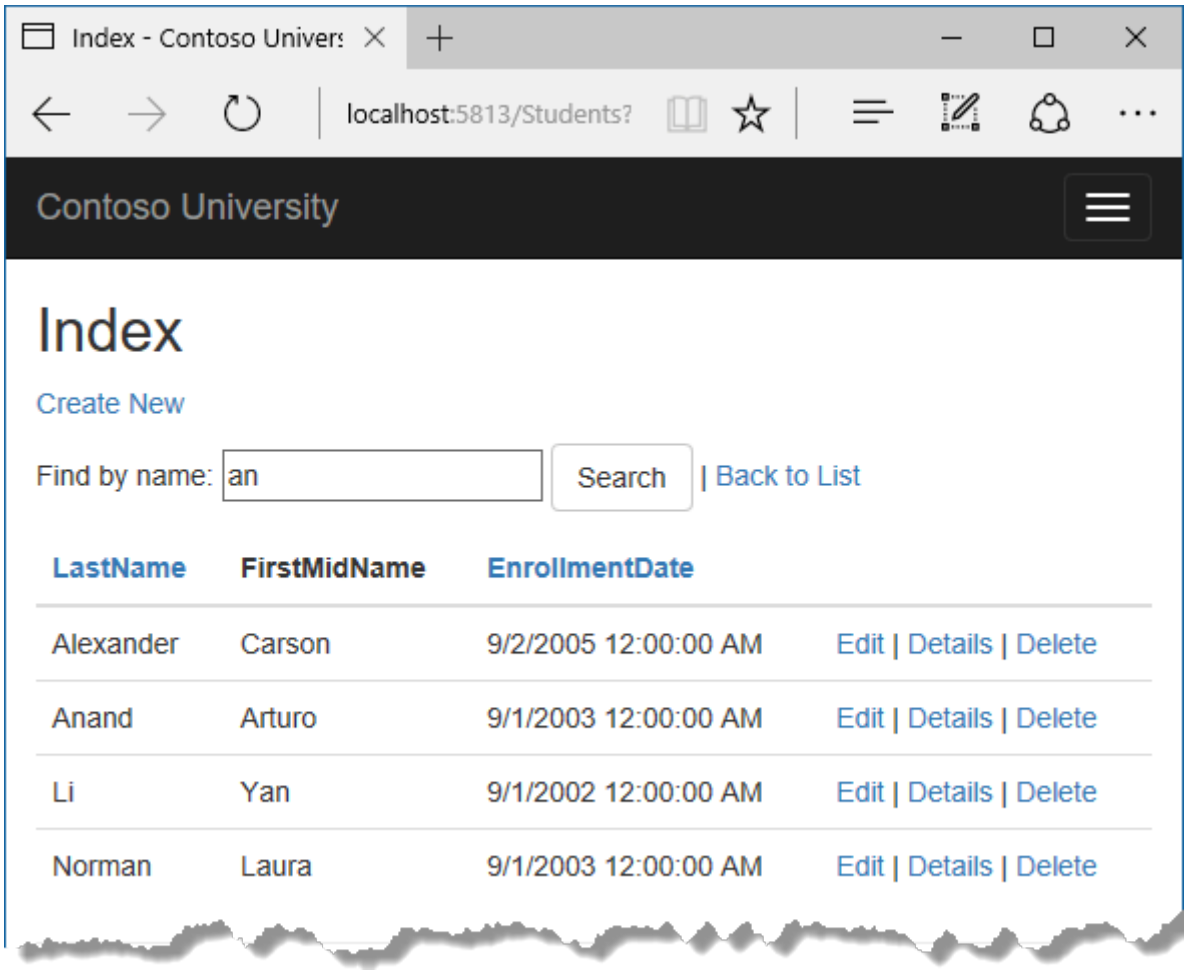
In `Views/Student/Index.cshtml`, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

CSHTML

```
<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString"
value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
```

This code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action doesn't result in an update.

Run the app, select the **Students** tab, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

```
HTML
```

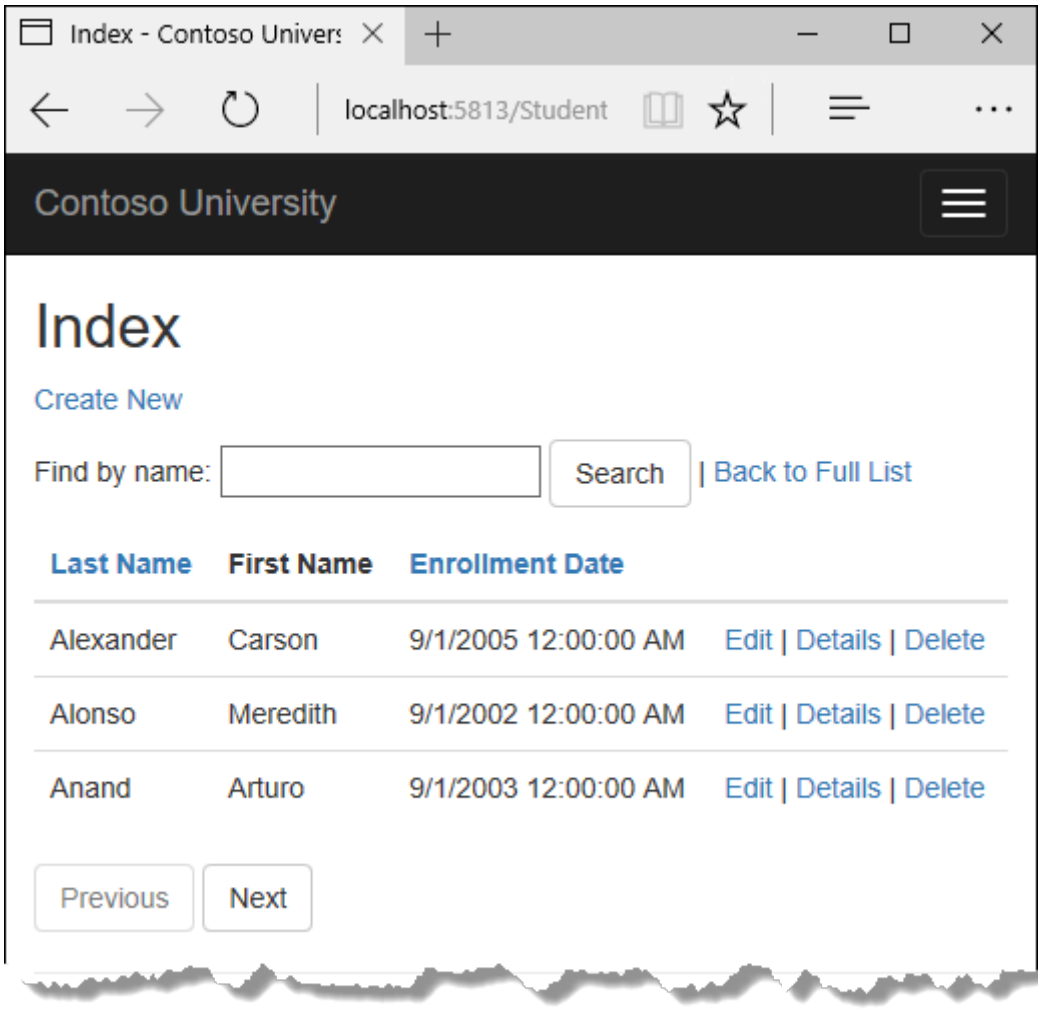
```
http://localhost:5813/Students?SearchString=an
```

If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding `method="get"` to the `form` tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the **Search** box. You'll fix that in the next section.

Add paging to Students Index

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs`, and then replace the template code with the following code.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int
pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }
```

```
        public bool HasPreviousPage => PageIndex > 1;

        public bool HasNextPage => PageIndex < TotalPages;

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T>
source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) *
pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in this code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it will return a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` can be used to enable or disable **Previous** and **Next** paging buttons.

A `CreateAsync` method is used instead of a constructor to create the `PaginatedList<T>` object because constructors can't run asynchronous code.

Add paging to Index method

In `StudentsController.cs`, replace the `Index` method with the following code.

C#

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
```

```
    if (searchString != null)
    {
        pageNumber = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1,
pageSize));
}
```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

C#

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The `ViewData` element named CurrentSort provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The `ViewData` element named CurrentFilter provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the `searchString` parameter isn't null.

C#

```
if (searchString != null)
{
    pageNumber = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the `Index` method, the `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

C#

```
return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1,
pageSize));
```

The `PaginatedList.CreateAsync` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(pageNumber ?? 1)` means return the value of `pageNumber` if it has a value, or return 1 if `pageNumber` is null.

Add paging links

In `Views/Students/Index.cshtml`, replace the existing code with the following code. The changes are highlighted.

CSHTML

```
@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString"
value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
```

```
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

```
@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex + 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

The `@model` statement at the top of the page specifies that the view now gets a `PaginatedList<T>` object instead of a `List<T>` object.

The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

HTML

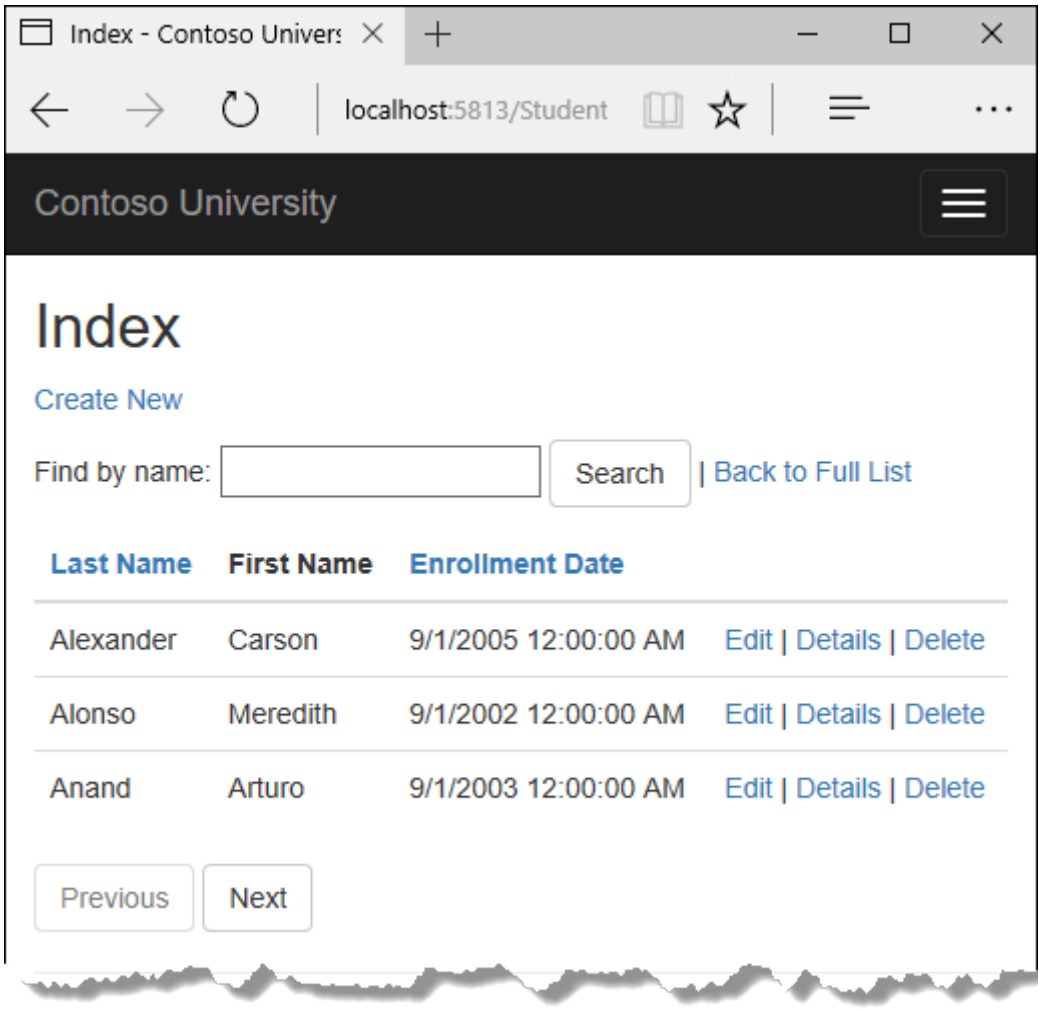
```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-
route-currentFilter ="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

The paging buttons are displayed by tag helpers:

HTML

```
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
   Previous
</a>
```

Run the app and go to the Students page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

Create an About page

For the Contoso University website's **About** page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations

on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Create the About method in the Home controller.
- Create the About view.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the new folder, add a class file `EnrollmentDateGroup.cs` and replace the template code with the following code:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Modify the Home Controller

In `HomeController.cs`, add the following using statements at the top of the file:

C#

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.Extensions.Logging;
```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

C#

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly SchoolContext _context;

    public HomeController(ILogger<HomeController> logger, SchoolContext
context)
    {
        _logger = logger;
        _context = context;
    }
```

Add an `About` method with the following code:

C#

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Create the About View

Add a `Views/Home/About.cshtml` file with the following code:

CSHTML

```
@model
IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>
```

```
<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and go to the About page. The count of students for each enrollment date is displayed in a table.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Added column sort links
- ✓ Added a Search box
- ✓ Added paging to Students Index
- ✓ Added paging to Index method
- ✓ Added paging links
- ✓ Created an About page

Advance to the next tutorial to learn how to handle data model changes by using migrations.

Next: Handle data model changes

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Tutorial: Part 5, apply migrations to the Contoso University sample

Article • 07/28/2023

In this tutorial, you start using the EF Core migrations feature for managing data model changes. In later tutorials, you'll add more migrations as you change the data model.

In this tutorial, you:

- ✓ Learn about migrations
- ✓ Create an initial migration
- ✓ Examine Up and Down methods
- ✓ Learn about the data model snapshot
- ✓ Apply the migration

Prerequisites

- Sorting, filtering, and paging

About migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You started these tutorials by configuring the Entity Framework to create the database if it doesn't exist. Then each time you change the data model -- add, remove, or change entity classes or change your DbContext class -- you can delete the database and EF creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it's usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The EF Core Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.

To work with migrations, you can use the **Package Manager Console** (PMC) or the CLI. These tutorials show how to use CLI commands. Information about the PMC is at the end of this tutorial.

Drop the database

Install EF Core tools as a global tool and delete the database:

.NET CLI

```
dotnet tool install --global dotnet-ef
dotnet ef database drop
```

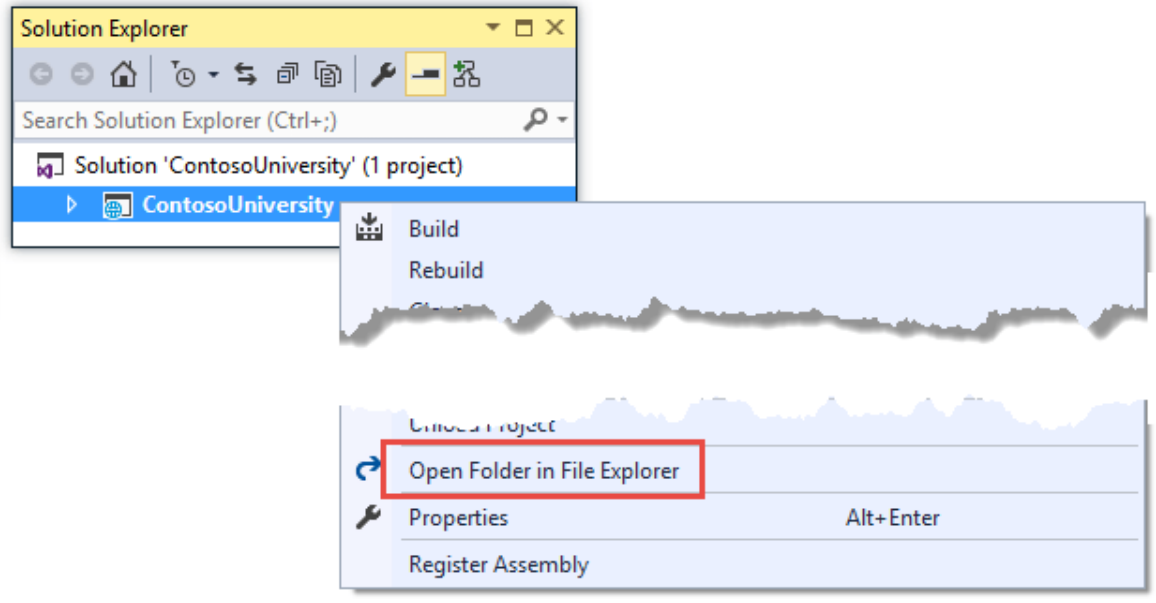
> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

The following section explains how to run CLI commands.

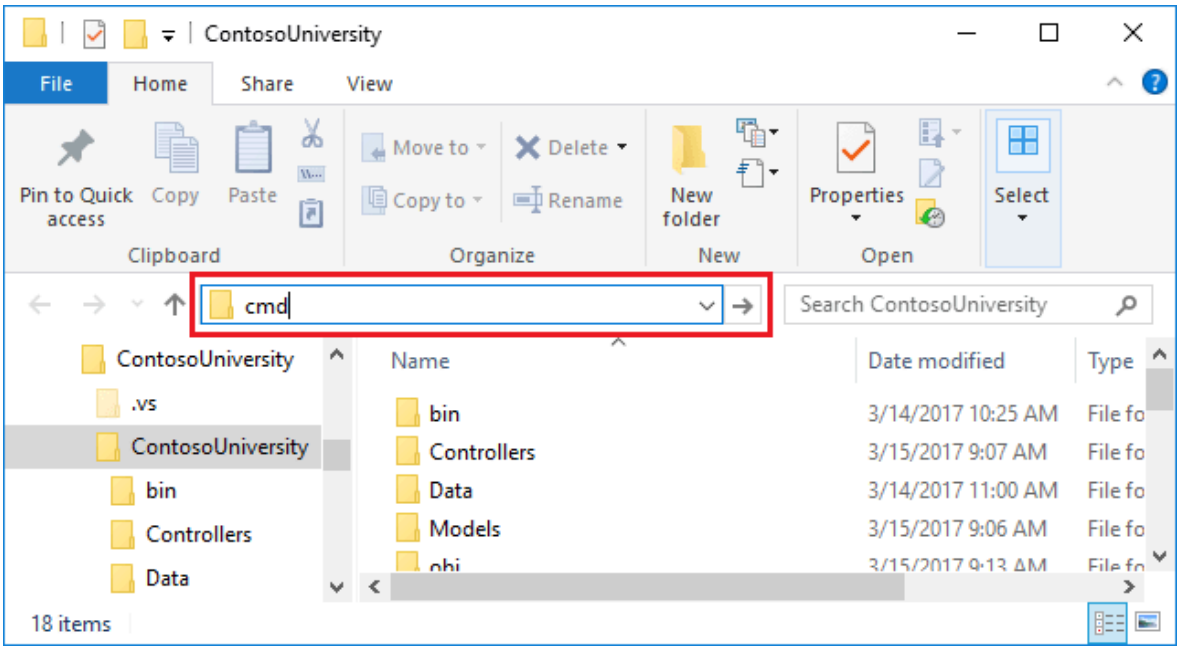
Create an initial migration

Save your changes and build the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

- In **Solution Explorer**, right-click the project and choose **Open Folder in File Explorer** from the context menu.



- Enter "cmd" in the address bar and press Enter.



Enter the following command in the command window:

.NET CLI

```
dotnet ef migrations add InitialCreate
```

In the preceding commands, output similar to the following is displayed:

Console

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

If you see an error message "*cannot access the file ... ContosoUniversity.dll because it is being used by another process.*", find the IIS Express icon in the Windows System Tray, and right-click it, then click **ContosoUniversity** > **Stop Site**.

Examine Up and Down methods

When you executed the `migrations add` command, EF generated the code that will create the database from scratch. This code is in the *Migrations* folder, in the file named `<timestamp>_InitialCreate.cs`. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

C#

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");
        // Additional code not shown
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the `migrations add InitialCreate` command. The migration name parameter ("InitialCreate" in the example) is used for the file name and can be whatever you want. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you dropped the database earlier -- so that migrations can create a new one from scratch.

The data model snapshot

Migrations creates a *snapshot* of the current database schema in `Migrations/SchoolContextModelSnapshot.cs`. When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

Use the dotnet ef migrations remove command to remove a migration. `dotnet ef migrations remove` deletes the migration and ensures the snapshot is correctly reset. If `dotnet ef migrations remove` fails, use `dotnet ef migrations remove -v` to get more information on the failure.

See EF Core Migrations in Team Environments for more information about how the snapshot file is used.

Apply the migration

In the command window, enter the following command to create the database and tables in it.

.NET CLI

```
dotnet ef database update
```

The output from the command is similar to the `migrations add` command, except that you see logs for the SQL commands that set up the database. Most of the logs are omitted in the following sample output. If you prefer not to see this level of detail in log messages, you can change the log level in the `appsettings.Development.json` file. For more information, see Logging in .NET Core and ASP.NET Core.

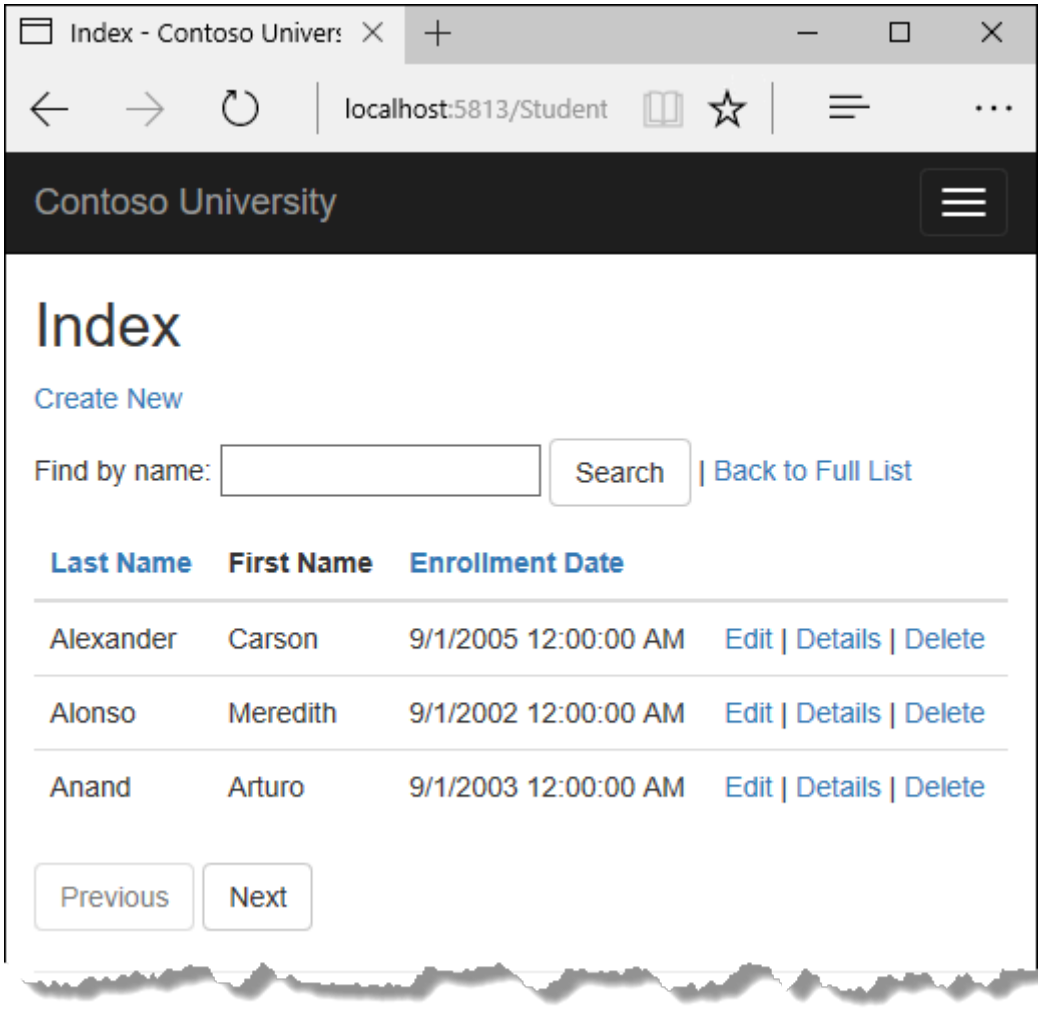
text

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (274ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (60ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [ContosoUniversity2] SET READ_COMMITTED_SNAPSHOT
ON;
      END;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[], CommandType='Text',
```

```
CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );

<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20190327172701_InitialCreate', N'5.0-rtm');
Done.
```

Use **SQL Server Object Explorer** to inspect the database as you did in the first tutorial. You'll notice the addition of an __EFMigrationsHistory table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one row for the first migration. (The last log in the preceding CLI output example shows the INSERT statement that creates this row.)

Run the application to verify that everything still works the same as before.



Compare CLI and PMC

The EF tooling for managing migrations is available from .NET CLI commands or from PowerShell cmdlets in the Visual Studio **Package Manager Console** (PMC) window. This tutorial shows how to use the CLI, but you can use the PMC if you prefer.

The EF commands for the PMC commands are in the Microsoft.EntityFrameworkCore.Tools package. This package is included in the Microsoft.AspNetCore.App metapackage, so you don't need to add a package reference if your app has a package reference for `Microsoft.AspNetCore.App`.

Important: This isn't the same package as the one you install for the CLI by editing the `.csproj` file. The name of this one ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see .NET CLI.

For more information about the PMC commands, see Package Manager Console (Visual Studio).

Get the code

Download or view the completed application.

Next step

Advance to the next tutorial to begin looking at more advanced topics about expanding the data model. Along the way you'll create and apply additional migrations.

Create and apply additional migrations

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

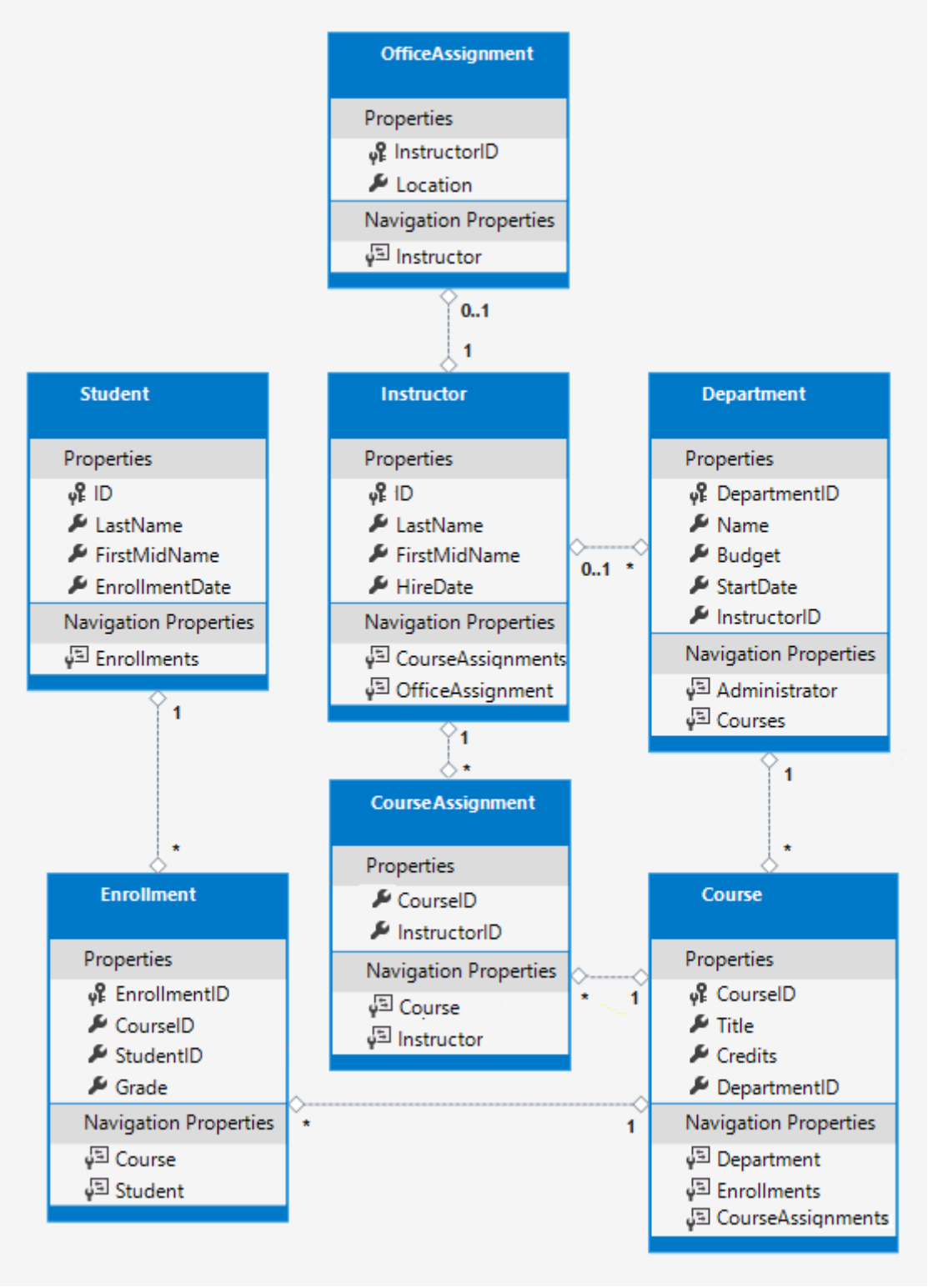
Provide product feedback

Tutorial: Create a complex data model - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorials, you worked with a simple data model that was composed of three entities. In this tutorial, you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



In this tutorial, you:

- ✓ Customize the Data model
- ✓ Make changes to Student entity
- ✓ Create Instructor entity
- ✓ Create OfficeAssignment entity
- ✓ Modify Course entity
- ✓ Create Department entity
- ✓ Modify Enrollment entity
- ✓ Update the database context

- ✓ Seed database with test data
- ✓ Add a migration
- ✓ Change the connection string
- ✓ Update the database

Prerequisites

- Using EF Core migrations

Customize the Data model

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete School data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In `Models/Student.cs`, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
```

```
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes don't provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

C#

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

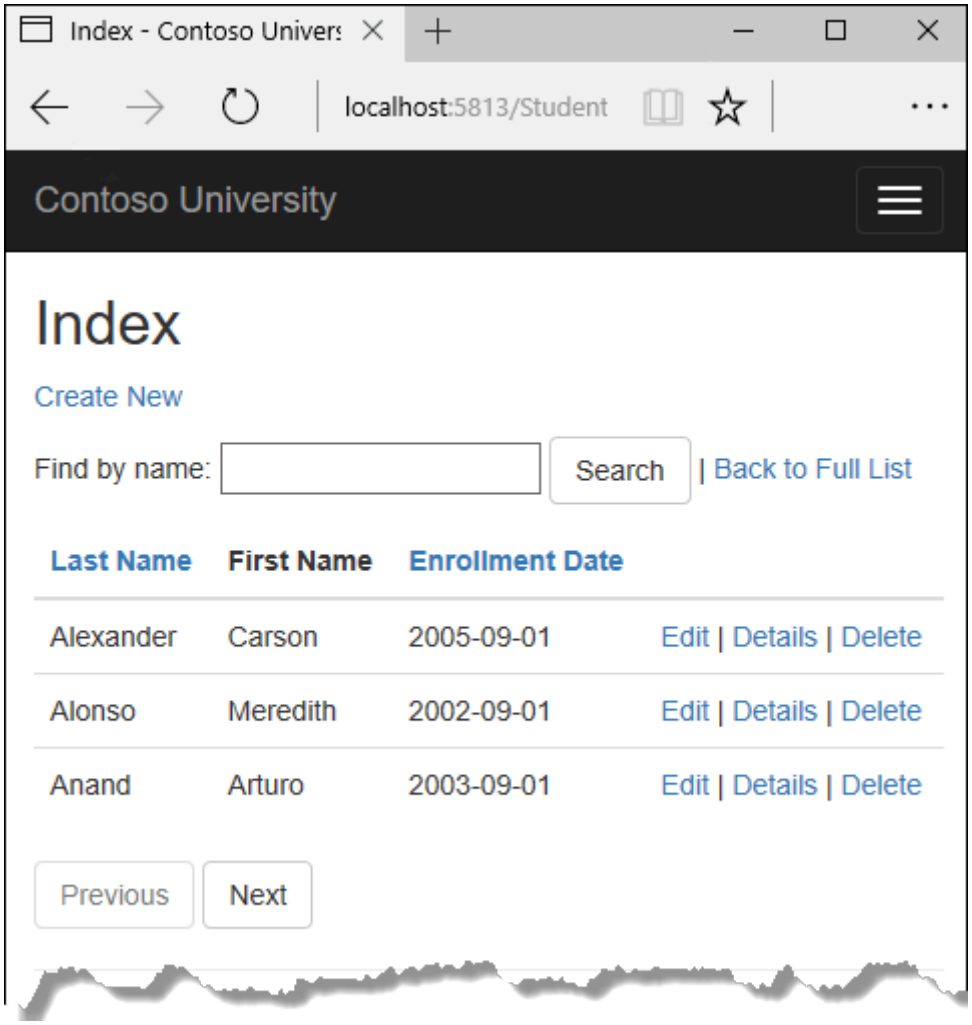
You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).

- By default, the browser will render data using the correct format based on your locale.

For more information, see the <input> tag helper documentation.

Run the app, go to the Students Index page and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the Student model.



The StringLength attribute

You can also specify data validation rules and validation error messages using attributes. The `StringLength` attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET Core MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `StringLength` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `StringLength` attribute won't prevent a user from entering white space for a name. You can use the `RegularExpression` attribute to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

C#

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

The `MaxLength` attribute provides functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

.NET CLI

```
dotnet ef migrations add MaxLengthOnNames
```

.NET CLI

```
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations creates a file named `<timeStamp>_MaxLengthOnNames.cs`. This file contains code in the `Up` method that will update the database to match the current data model. The `database update` command ran that code.

The timestamp prefixed to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the update-database command, and then all of the migrations are applied in the order in which they were created.

Run the app, select the **Students** tab, click **Create New**, and try to enter either name longer than 50 characters. The application should prevent you from doing this.

The Column attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they're given the same name as the property name.

In the `Student.cs` file, add a `using` statement for `System.ComponentModel.DataAnnotations.Schema` and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

```
namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The addition of the `Column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

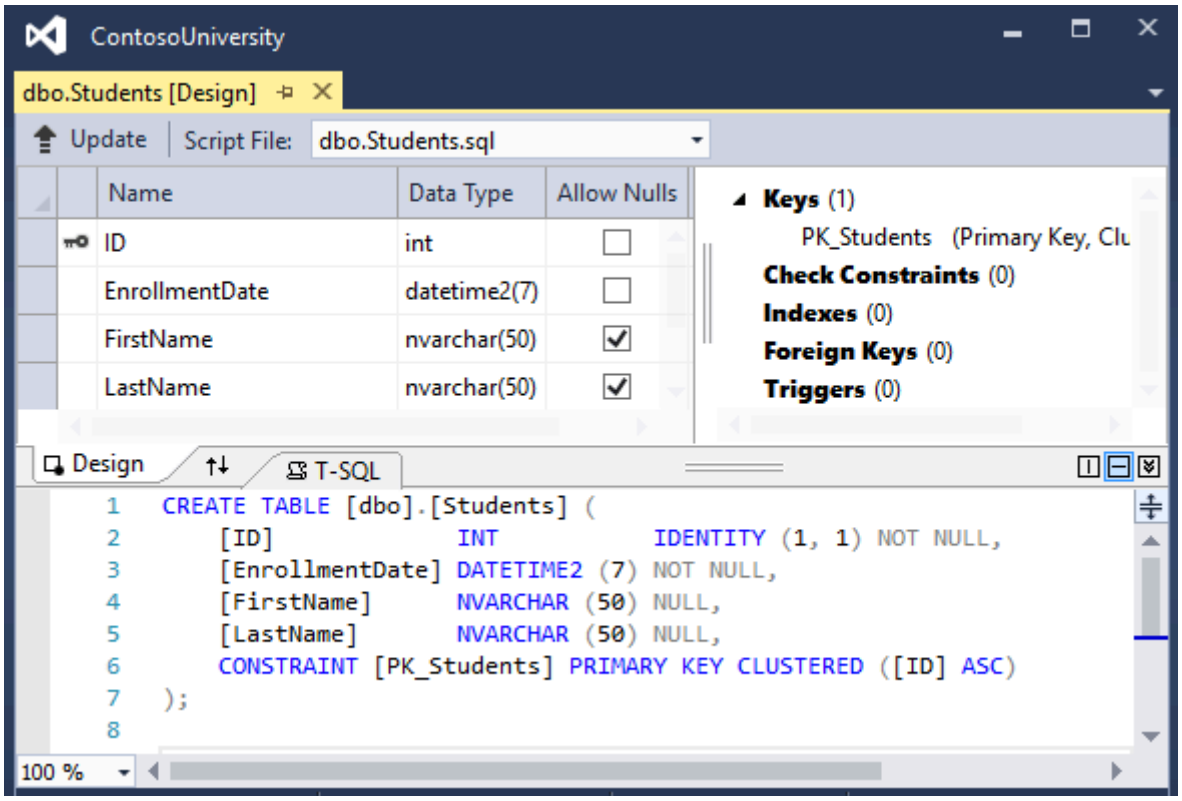
.NET CLI

```
dotnet ef migrations add ColumnFirstName
```

.NET CLI

```
dotnet ef database update
```

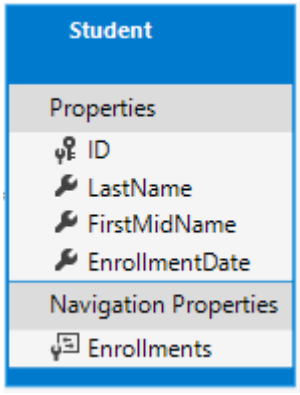
In **SQL Server Object Explorer**, open the Student table designer by double-clicking the **Student** table.



Before you applied the first two migrations, the name columns were of type nvarchar(MAX). They're now nvarchar(50) and the column name has changed from FirstMidName to FirstName.

> ⓘ Note
>
> If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Changes to Student entity



In `Models/Student.cs`, replace the code you added earlier with the following code. The changes are highlighted.

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50)]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

C#

```
[Display(Name = "Last Name")]
[Required]
```

```
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

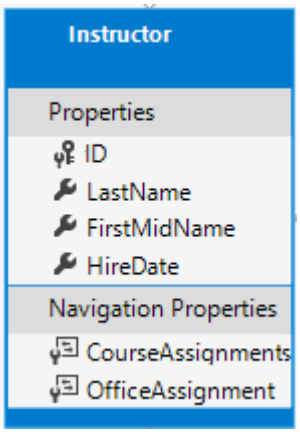
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a get accessor, and no `FullName` column will be generated in the database.

Create Instructor entity



Create `Models/Instructor.cs`, replacing the template code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
```

```
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the Student and Instructor entities. In the Implementing Inheritance tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the `HireDate` attributes as follows:

C#

```
[DataType(DataType.Date),Display(Name = "Hire
Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

C#

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

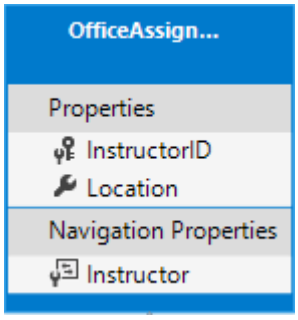
The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single OfficeAssignment entity (which may be null if no office is assigned).

C#

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }
```

```
        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

There's a one-to-zero-or-one relationship between the `Instructor` and the `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the `Instructor` entity. But the Entity Framework can't automatically recognize `InstructorID` as the primary key of this entity because its name doesn't follow the `ID` or `classnameID` naming convention. Therefore, the `Key` attribute is used to identify it as the key:

C#

```
[Key]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something other than classnameID or ID.

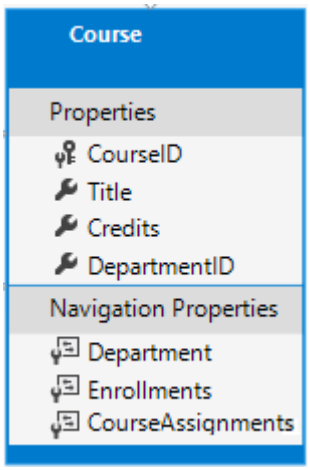
By default, EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The Instructor entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the OfficeAssignment entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an Instructor entity has a related OfficeAssignment entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the `InstructorID` foreign key (which is also the key to this table) is non-nullable.

Modify Course entity



In `Models/Course.cs`, replace the code you added earlier with the following code. The changes are highlighted.

C#

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related Department entity and it has a `Department` navigation property.

The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they're needed and creates shadow properties for

them. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a `Course` entity to edit, the `Department` entity is null if you don't load it, so when you update the `Course` entity, you would have to first fetch the `Department` entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the `Department` entity before you update.

The DatabaseGenerated attribute

The `DatabaseGenerated` attribute with the `None` parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

C#

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for `Course` entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

C#

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

C#

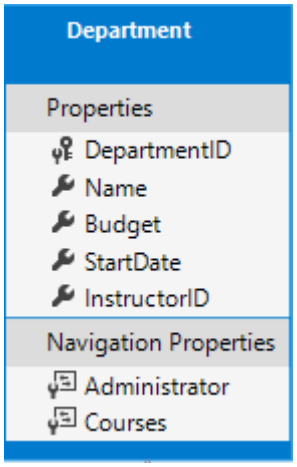
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained later):

C#

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create Department entity



Create `Models/Department.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
```

```
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Earlier you used the `Column` attribute to change column name mapping. In the code for the `Department` entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server `money` type in the database:

C#

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the Instructor entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation property is named `Administrator` but holds an Instructor entity:

C#

```
public int? InstructorID { get; set; }
```

```
public Instructor Administrator { get; set; }
```

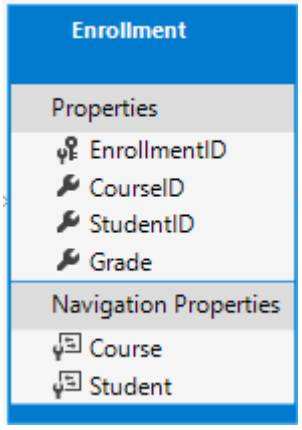
A department may have many courses, so there's a Courses navigation property:

C#

```
public ICollection<Course> Courses { get; set; }
```

> **Note**
>
> By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the `Department.InstructorID` property as nullable, EF would configure a cascade delete rule to delete the department when you delete the instructor, which isn't what you want to have happen. If your business rules required the `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:
>
> C#
>
> ```
> modelBuilder.Entity<Department>()
>     .HasOne(d => d.Administrator)
>     .WithMany()
>     .OnDelete(DeleteBehavior.Restrict)
> ```

Modify Enrollment entity



In `Models/Enrollment.cs`, replace the code you added earlier with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The foreign key properties and navigation properties reflect the following relationships:

An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

C#

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

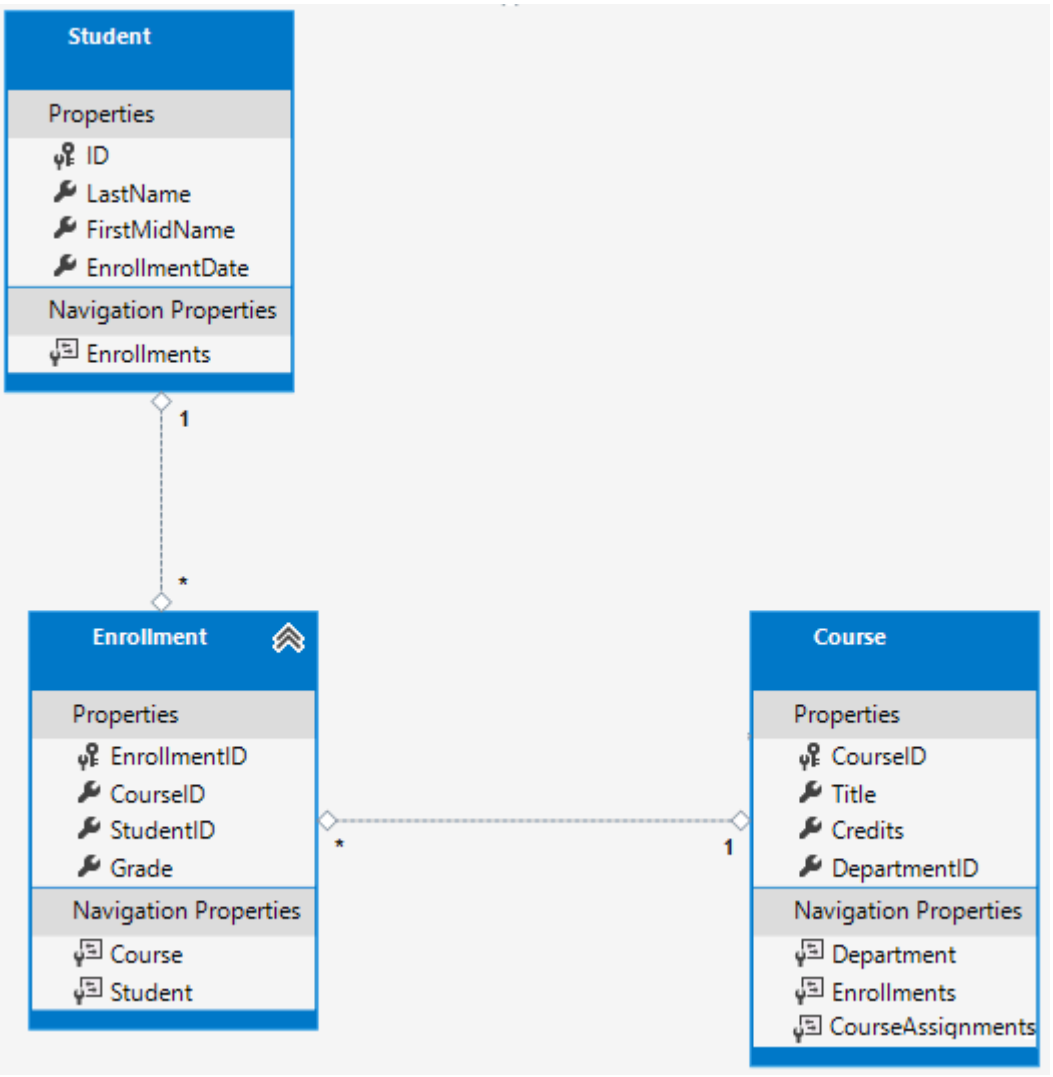
C#

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many relationships

There's a many-to-many relationship between the `Student` and `Course` entities, and the `Enrollment` entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the `Enrollment` table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a `Grade` property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)



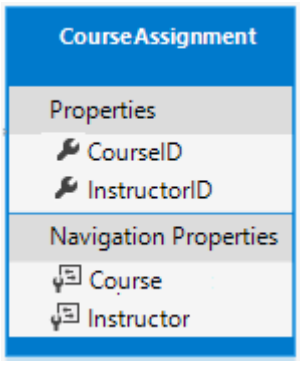
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two foreign keys `CourseID` and `StudentID`. In that case, it would be a many-to-many join table without payload (or a pure join table) in the database. The `Instructor` and `Course` entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

EF Core supports implicit join tables for many-to-many relationships, but this tutoral has not been updated to use an implicit join table. See Many-to-Many Relationships, the

Razor Pages version of this tutorial which has been updated.

The CourseAssignment entity



Create `Models/CourseAssignment.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there's no need for a separate primary key. The `InstructorID` and `CourseID` properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Add the following highlighted code to the `Data/SchoolContext.cs` file:

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
```

```
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>
().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>
().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

About a fluent API alternative

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the EF Core documentation:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

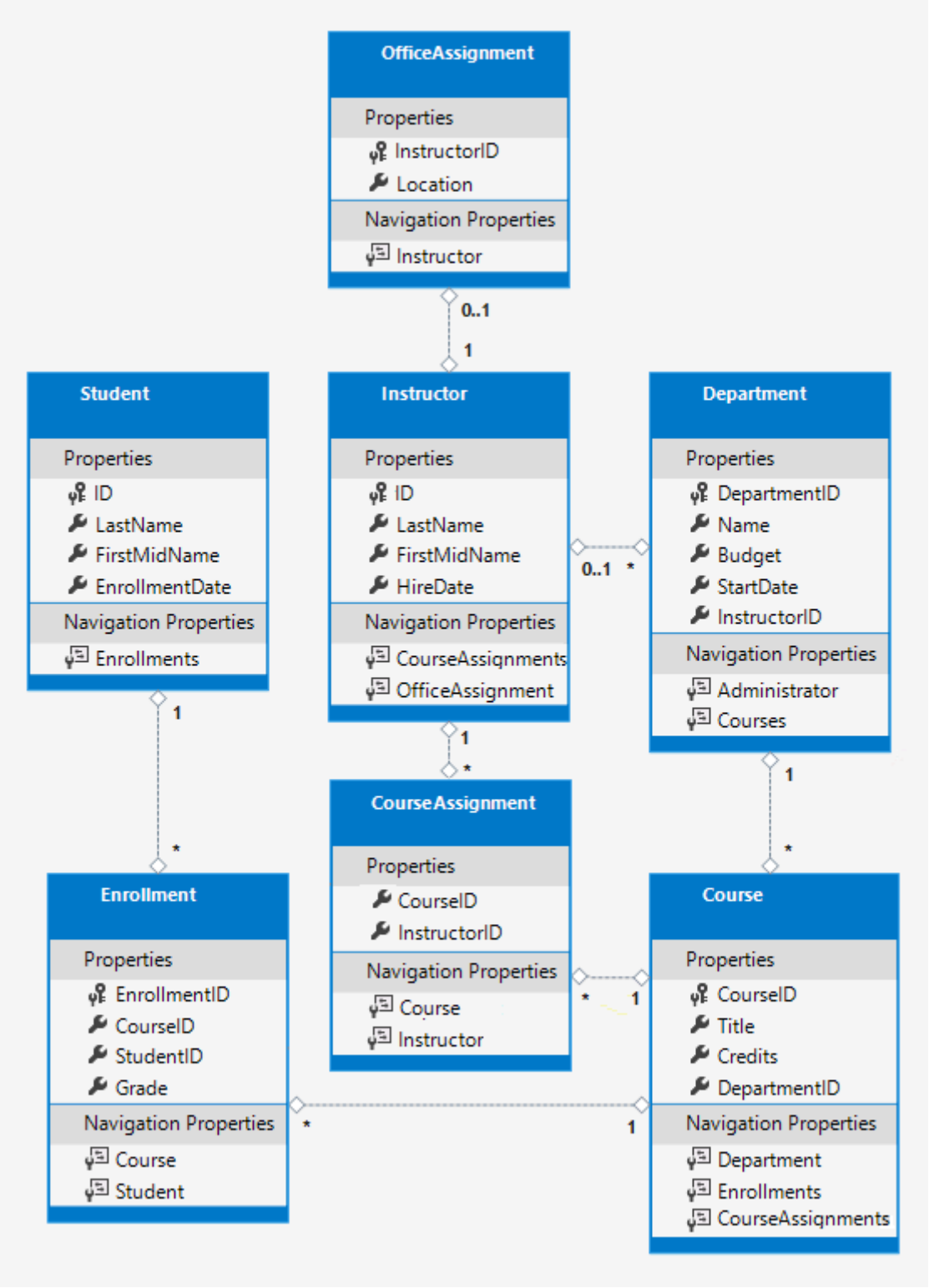
In this tutorial, you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible. If you do use both, note that wherever there's a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment`

entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Seed database with test data

Replace the code in the `Data/DbInitializer.cs` file with the following code in order to provide seed data for the new entities you've created.

C#

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName =
"Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName =
"Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName =
"Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName =
"Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName =
"Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
```

```
                new Student { FirstMidName = "Nino",     LastName =
"Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName =
"Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName =
"Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName =
"Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName =
"Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName =
"Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            foreach (Instructor i in instructors)
            {
                context.Instructors.Add(i);
            }
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Kapoor").ID }
            };
```

```
            foreach (Department d in departments)
            {
                context.Departments.Add(d);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course {CourseID = 1050, Title = "Chemistry",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"Engineering").DepartmentID
                },
                new Course {CourseID = 4022, Title = "Microeconomics",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID
                },
                new Course {CourseID = 4041, Title = "Macroeconomics",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID
                },
                new Course {CourseID = 1045, Title = "Calculus",
Credits = 4,
                    DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID
                },
                new Course {CourseID = 3141, Title = "Trigonometry",
Credits = 4,
                    DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID
                },
                new Course {CourseID = 2021, Title = "Composition",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID
                },
                new Course {CourseID = 2042, Title = "Literature",
Credits = 4,
                    DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID
                },
            };

            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var officeAssignments = new OfficeAssignment[]
            {
                new OfficeAssignment {
```

```
                    InstructorID = instructors.Single( i => i.LastName ==
"Fakhouri").ID,
                    Location = "Smith 17" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName ==
"Harui").ID,
                    Location = "Gowan 27" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName ==
"Kapoor").ID,
                    Location = "Thompson 304" },
            };

            foreach (OfficeAssignment o in officeAssignments)
            {
                context.OfficeAssignments.Add(o);
            }
            context.SaveChanges();

            var courseInstructors = new CourseAssignment[]
            {
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Kapoor").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Harui").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title ==
"Microeconomics" ).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Zheng").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title ==
"Macroeconomics" ).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Zheng").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Calculus"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Fakhouri").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Trigonometry"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
```

```
"Harui").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Composition"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Abercrombie").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Literature"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Abercrombie").ID
                    },
            };

            foreach (CourseAssignment ci in courseInstructors)
            {
                context.CourseAssignments.Add(ci);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID,
                    Grade = Grade.A
                },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title ==
"Microeconomics" ).CourseID,
                    Grade = Grade.C
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title ==
"Macroeconomics" ).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                        StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Calculus"
).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                        StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
```

```
                    CourseID = courses.Single(c => c.Title == "Trigonometry"
).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Composition"
).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Anand").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Anand").ID,
                    CourseID = courses.Single(c => c.Title ==
"Microeconomics").CourseID,
                    Grade = Grade.B
                    },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title ==
"Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title ==
"Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Justice").ID,
                    CourseID = courses.Single(c => c.Title ==
"Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID ==
e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
```

```
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

.NET CLI

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

text

```
An operation was scaffolded that may result in the loss of data. Please
review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If you tried to run the `database update` command at this point (don't do it yet), you would get the following error:

> The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints. The generated code in the `Up` method adds a non-nullable `DepartmentID` foreign key to the `Course` table. If there are already rows in the Course table when the code runs, the `AddColumn` operation fails

because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing Course rows will all be related to the "Temp" department after the `Up` method runs.

- Open the `{timestamp}_ComplexDataModel.cs` file.
- Comment out the line of code that adds the DepartmentID column to the Course table.

```
C#

migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

- Add the following highlighted code after the code that creates the Department table:

```
C#

migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
```

```
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget,
StartDate) VALUES ('Temp', 0.00, GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the `Course.DepartmentID` column.

Save your changes and build the project.

Change the connection string

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in `appsettings.json` to ContosoUniversity3 or some other name that you haven't used on the computer you're using.

JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;
MultipleActiveResultSets=true"
  },
```

Save your change to `appsettings.json`.

> **Note**
>
> As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:
>
> .NET CLI
>
> ```
> dotnet ef database drop
> ```

Update the database

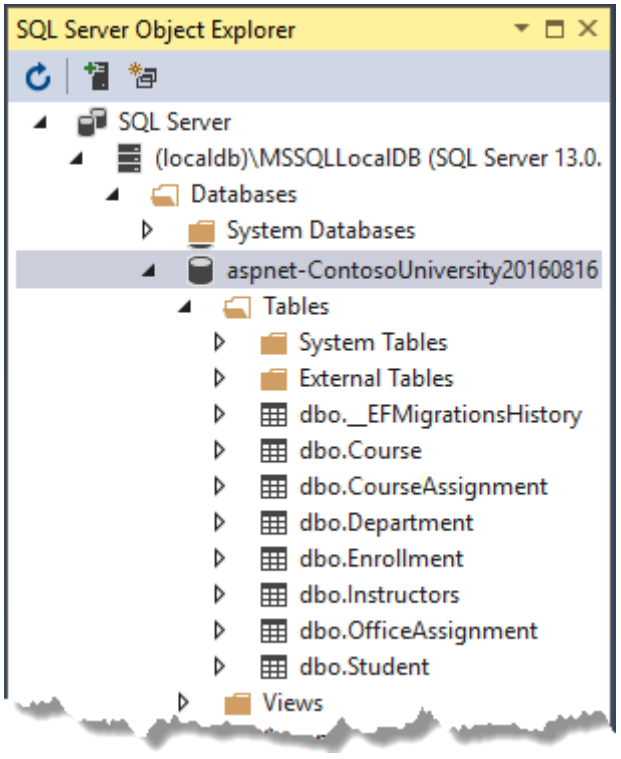
After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

.NET CLI

```
dotnet ef database update
```

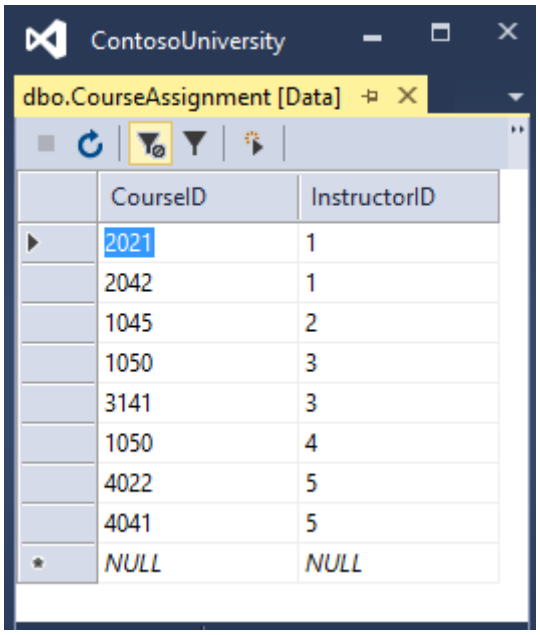
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

Open the database in SSOX as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have SSOX open from the earlier time, click the **Refresh** button.)



Run the app to trigger the initializer code that seeds the database.

Right-click the **CourseAssignment** table and select **View Data** to verify that it has data in it.



Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized the Data model
- ✓ Made changes to Student entity
- ✓ Created Instructor entity
- ✓ Created OfficeAssignment entity
- ✓ Modified Course entity
- ✓ Created Department entity
- ✓ Modified Enrollment entity
- ✓ Updated the database context
- ✓ Seeded database with test data
- ✓ Added a migration
- ✓ Changed the connection string
- ✓ Updated the database

Advance to the next tutorial to learn more about how to access related data.

Next: Access related data

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

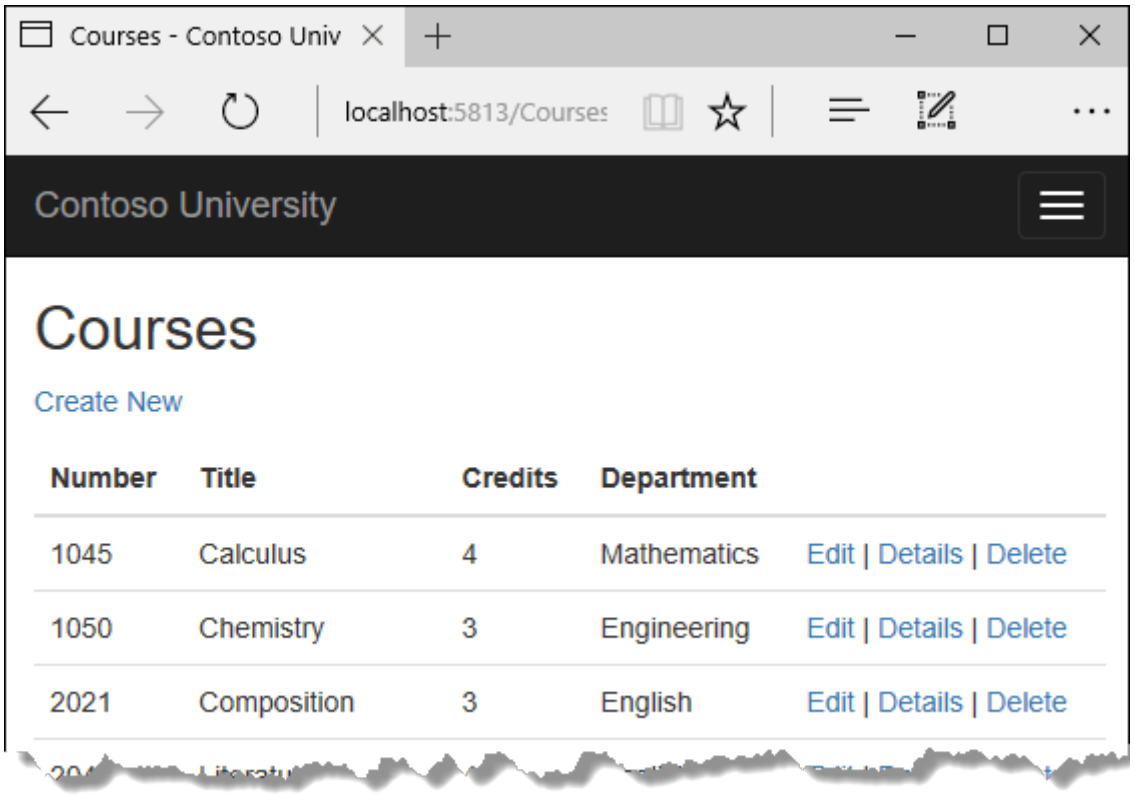
Provide product feedback

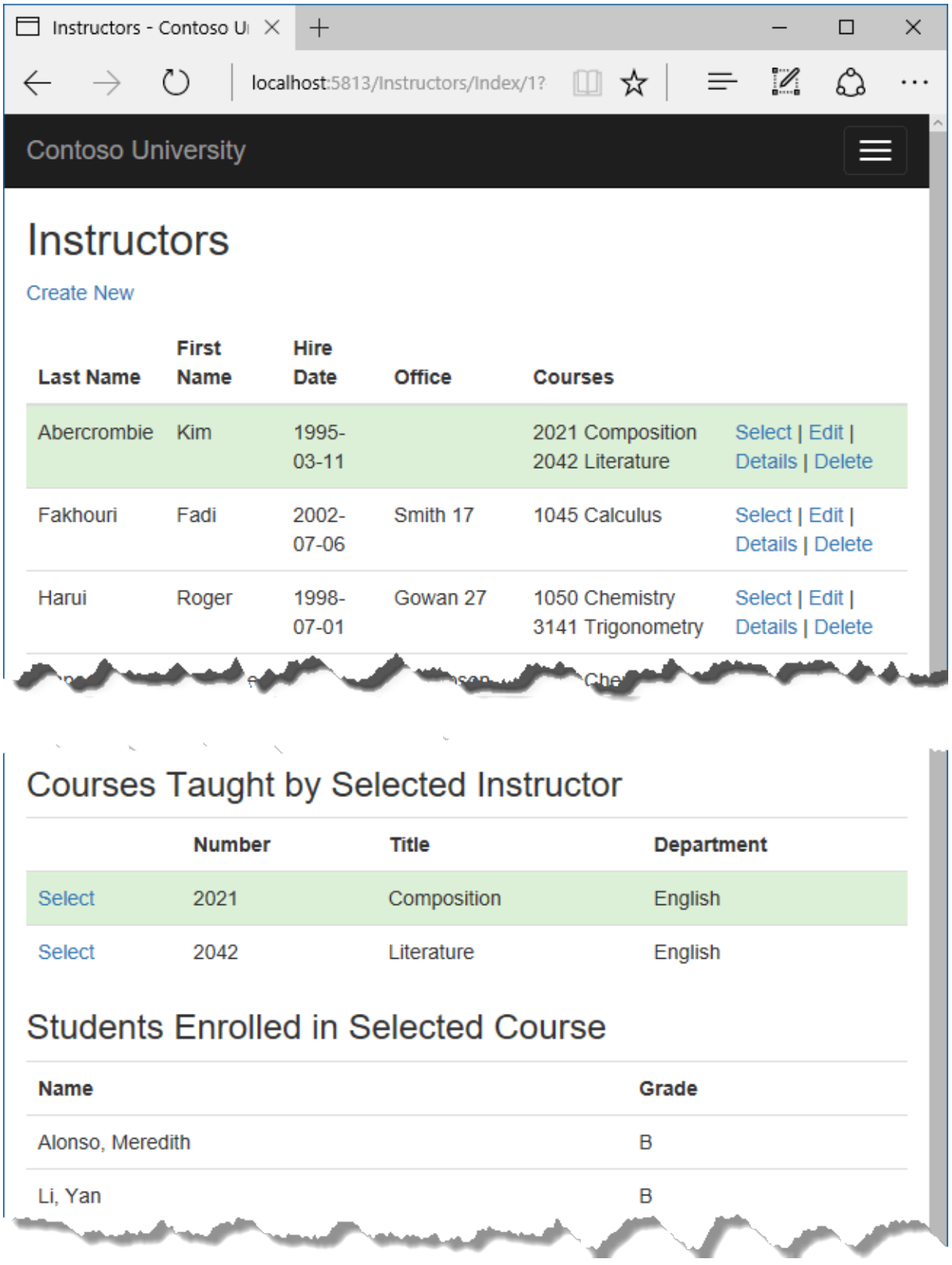
Tutorial: Read related data - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you completed the School data model. In this tutorial, you'll read and display related data -- that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.





In this tutorial, you:

- ✓ Learn how to load related data
- ✓ Create a Courses page
- ✓ Create an Instructors page
- ✓ Learn about explicit loading

Prerequisites

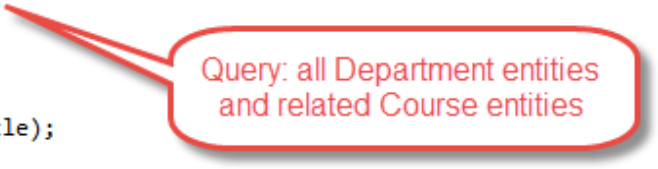
- Create a complex data model

Learn how to load related data

There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

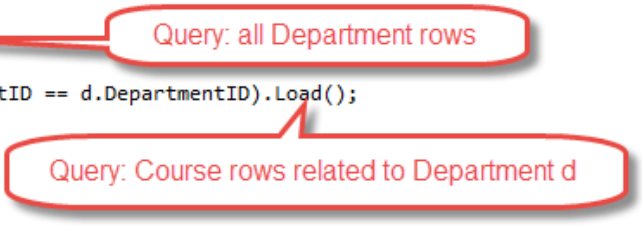
- Eager loading: When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

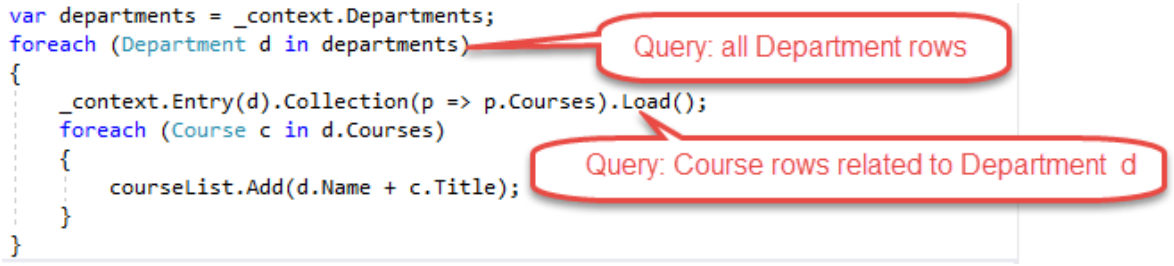


  You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of previously retrieved entities. For the query that retrieves related data, you can use the `Load` method instead of a method that returns a list or object, such as `ToList` or `Single`.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```



- Explicit loading: When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed. As in the case of eager loading with separate queries, explicit loading results in multiple queries sent to the database. The difference is that with explicit loading, the code specifies the navigation properties to be loaded. In Entity Framework Core 1.1 you can use the `Load` method to do explicit loading. For example:



- Lazy loading: When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. A query is sent to the database each time you try to get data from a navigation property for the first time. Entity Framework Core 1.0 doesn't support lazy loading.

Performance considerations

If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager loading of all related data would result in just a single (join) query and a single round trip to the database. A separate query for courses for each department would result in eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

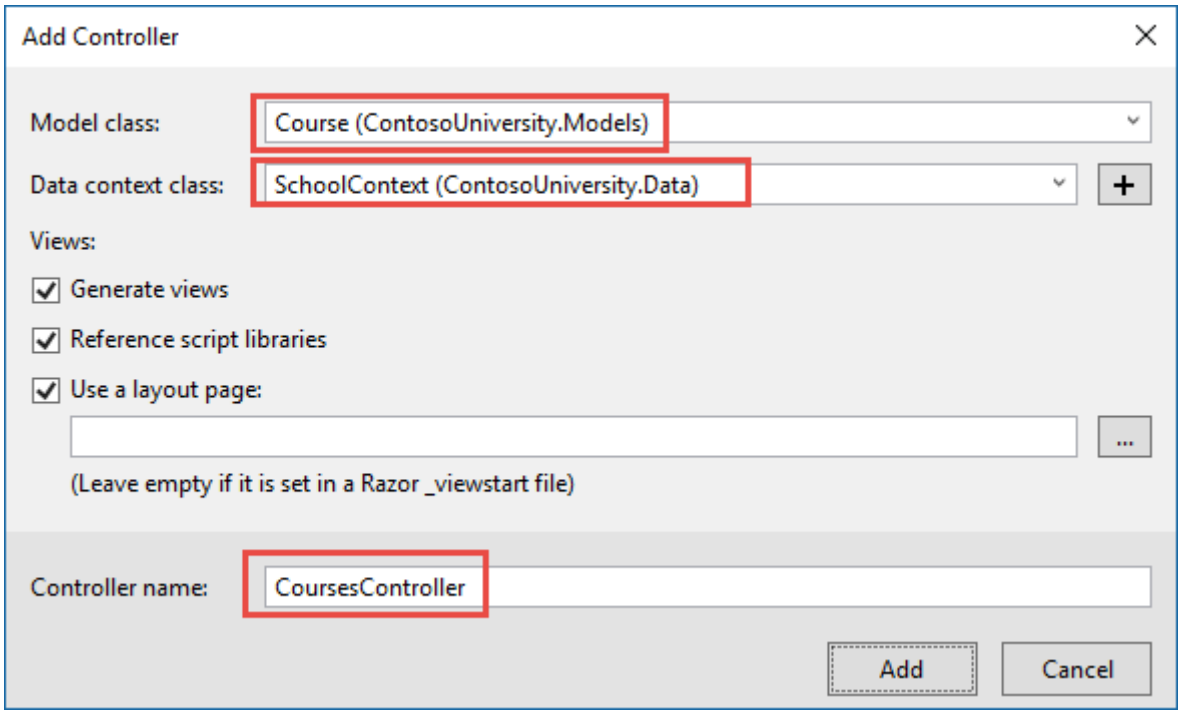
On the other hand, in some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, separate queries might perform better because eager loading of everything up front would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Create a Courses page

The `Course` entity includes a navigation property that contains the `Department` entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the `Name` property from the `Department` entity that's in the `Course.Department` navigation property.

Create a controller named `CoursesController` for the `Course` entity type, using the same options for the **MVC Controller with views, using Entity Framework** scaffolder that you

did earlier for the `StudentsController`, as shown in the following illustration:



Open `CoursesController.cs` and examine the `Index` method. The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Replace the `Index` method with the following code that uses a more appropriate name for the `IQueryable` that returns Course entities (`courses` instead of `schoolContext`):

C#

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Open `Views/Courses/Index.cshtml` and replace the template code with the following code. The changes are highlighted:

CSHTML

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewData["Title"] = "Courses";
}
```

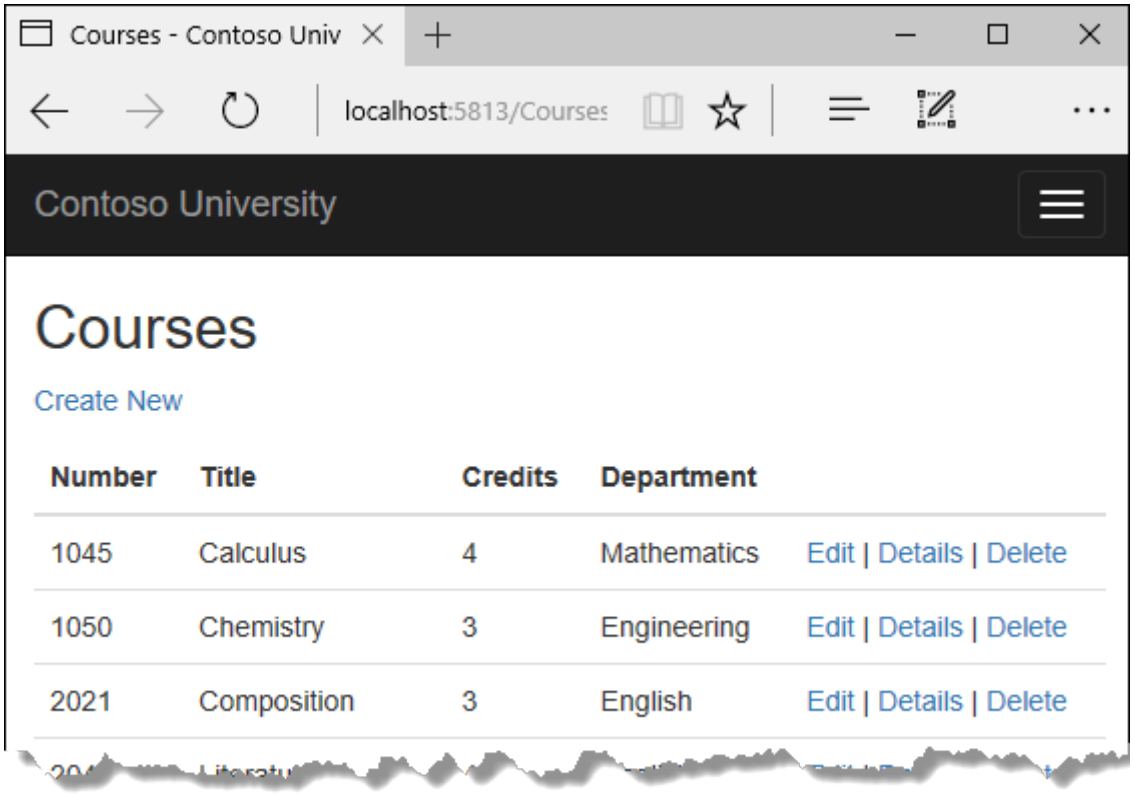
```
<h2>Courses</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

You've made the following changes to the scaffolded code:

- Changed the heading from **Index** to **Courses**.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

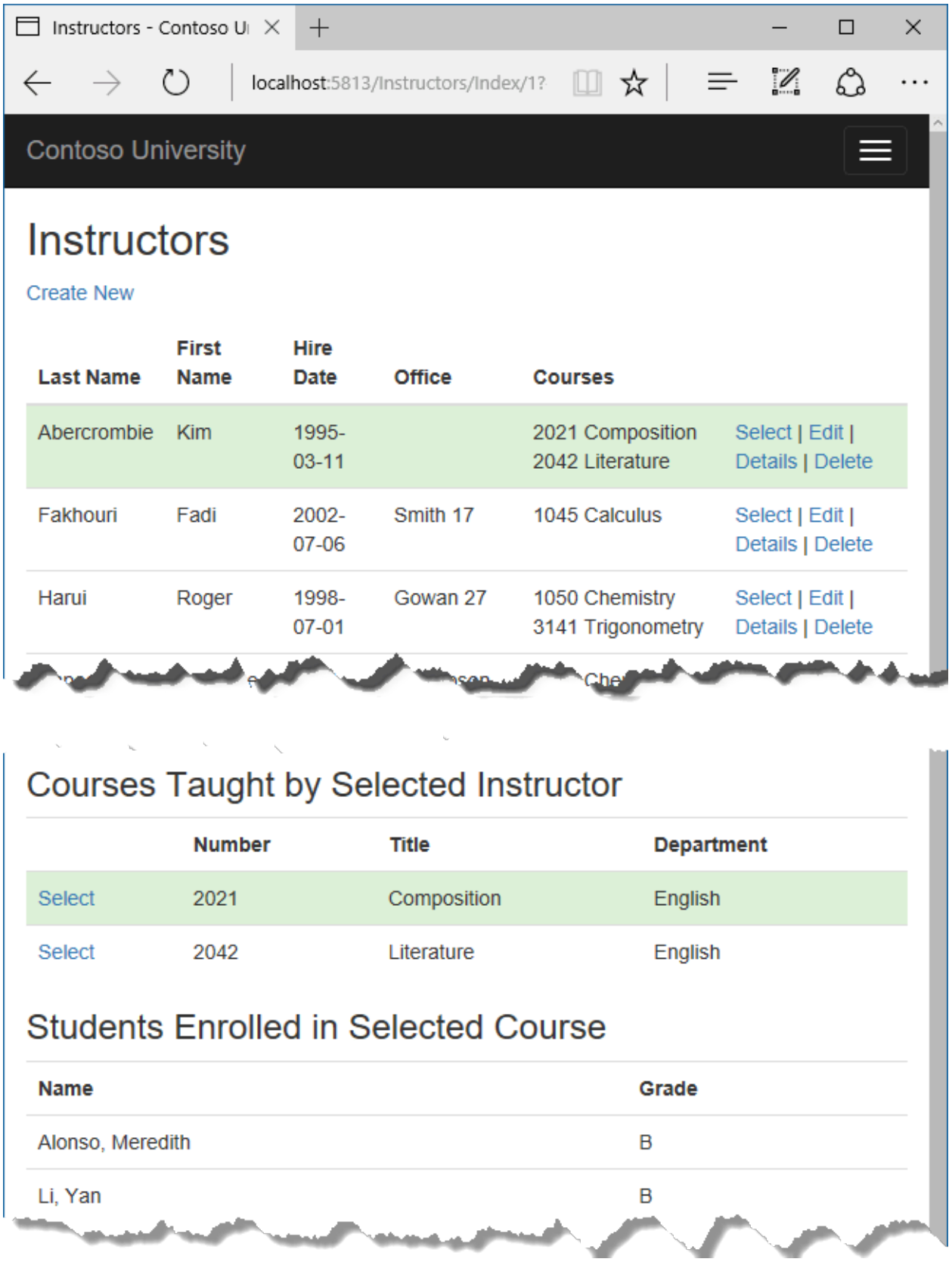
```
HTML

@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Create an Instructors page

In this section, you'll create a controller and view for the `Instructor` entity in order to display the Instructors page:



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity. The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. You'll use eager loading for the `OfficeAssignment` entities. As explained earlier, eager loading is typically more efficient when you need the

related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.

- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. You'll use eager loading for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity set is displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship. You'll use separate queries for `Enrollment` entities and their related `Student` entities.

Create a view model for the Instructor Index view

The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

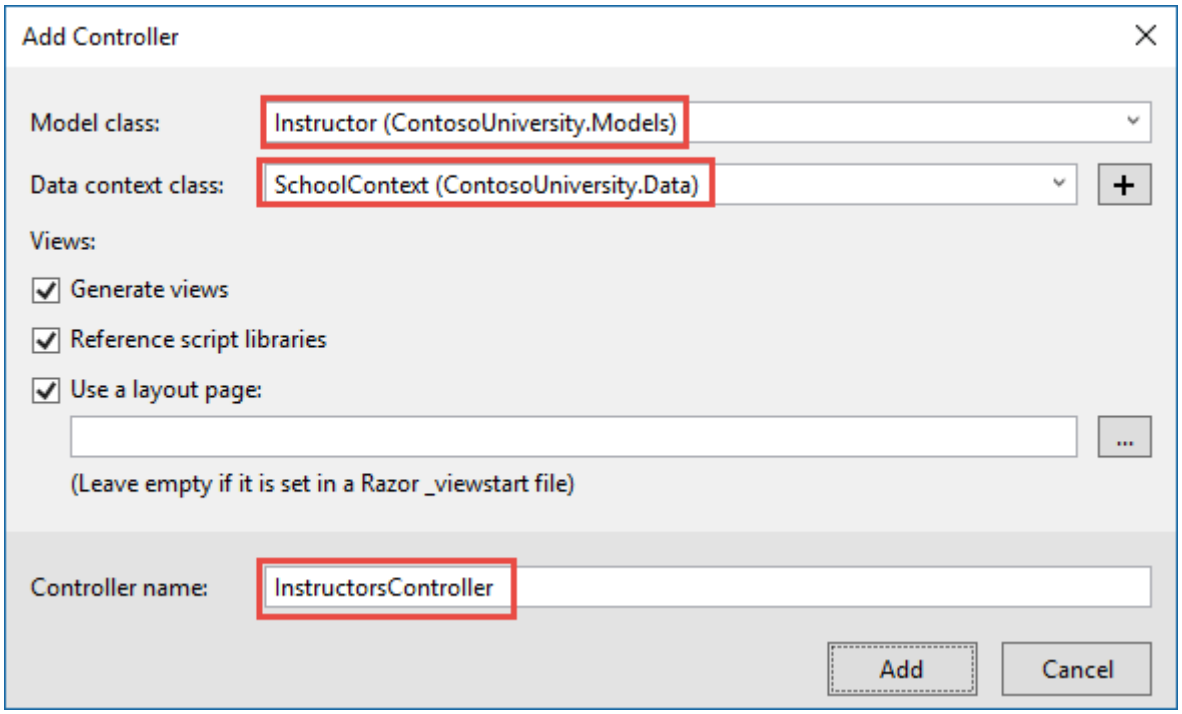
In the *SchoolViewModels* folder, create `InstructorIndexData.cs` and replace the existing code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:



Open `InstructorsController.cs` and add a using statement for the ViewModels namespace:

C#

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

C#

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
```

```
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s =>
s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Since the view always requires the `OfficeAssignment` entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `Enrollment.Student`.

You can read more about including multiple levels of related data here.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `Course.Enrollments`.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

C#

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single Instructor entity being returned. The `Single` method converts the collection into a single `Instructor` entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it's empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the Where condition instead of calling the `Where` method separately:

C#

```
.Single(i => i.ID == id.Value)
```

Instead of:

C#

```
.Where(i => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

C#

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
```

```
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Tracking vs no-tracking

No-tracking queries are useful when the results are used in a read-only scenario. They're generally quicker to execute because there's no need to set up the change tracking information. If the entities retrieved from the database don't need to be updated, then a no-tracking query is likely to perform better than a tracking query.

In some cases a tracking query is more efficient than a no-tracking query. For more information, see Tracking vs. No-Tracking Queries.

Modify the Instructor Index view

In `Views/Instructors/Index.cshtml`, replace the template code with the following code. The changes are highlighted.

CSHTML

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
```

```
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @foreach (var course in item.CourseAssignments)
                    {
                        @course.Course.CourseID @course.Course.Title <br />
                    }
                </td>
                <td>
                    <a asp-action="Index" asp-route-id="@item.ID">Select</a>
|
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

CSHTML

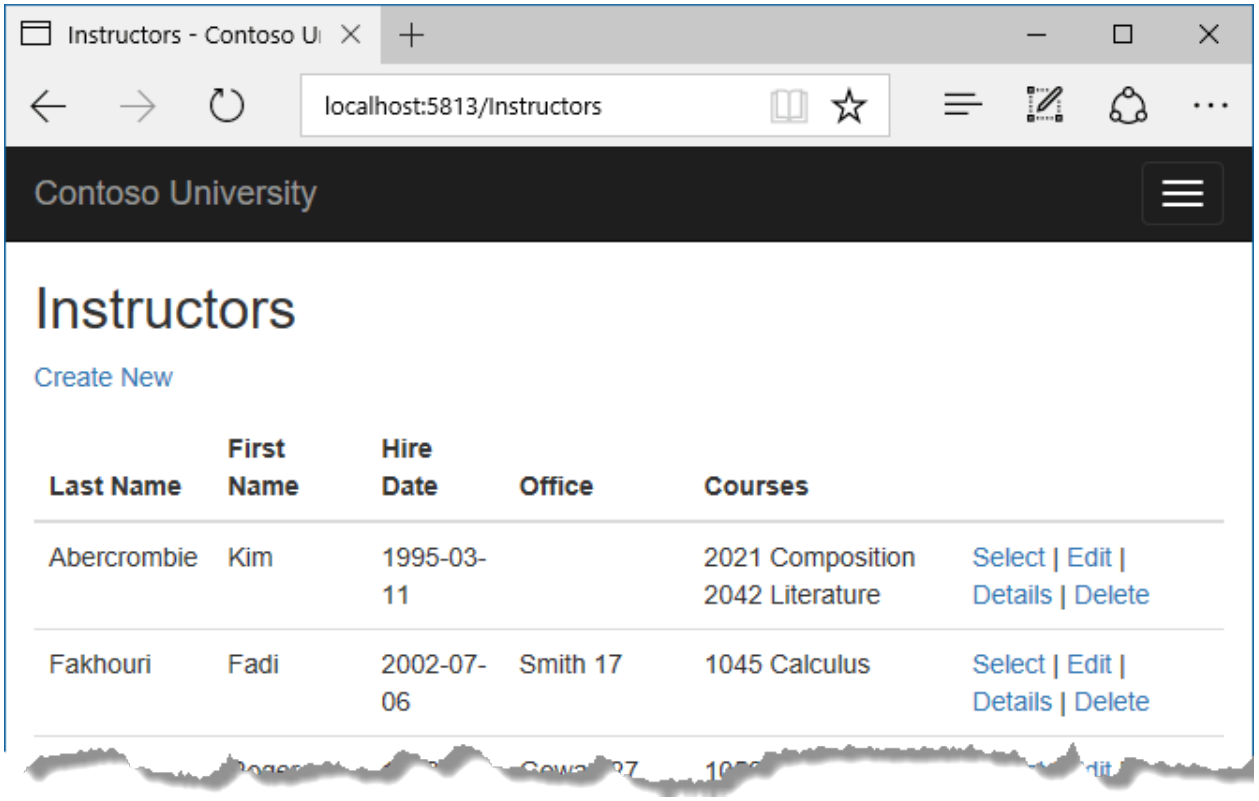
```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- Added a **Courses** column that displays courses taught by each instructor. For more information, see the Explicit line transition section of the Razor syntax article.
- Added code that conditionally adds a Bootstrap CSS class to the `tr` element of the selected instructor. This class sets a background color for the selected row.
- Added a new hyperlink labeled **Select** immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

CSHTML

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.



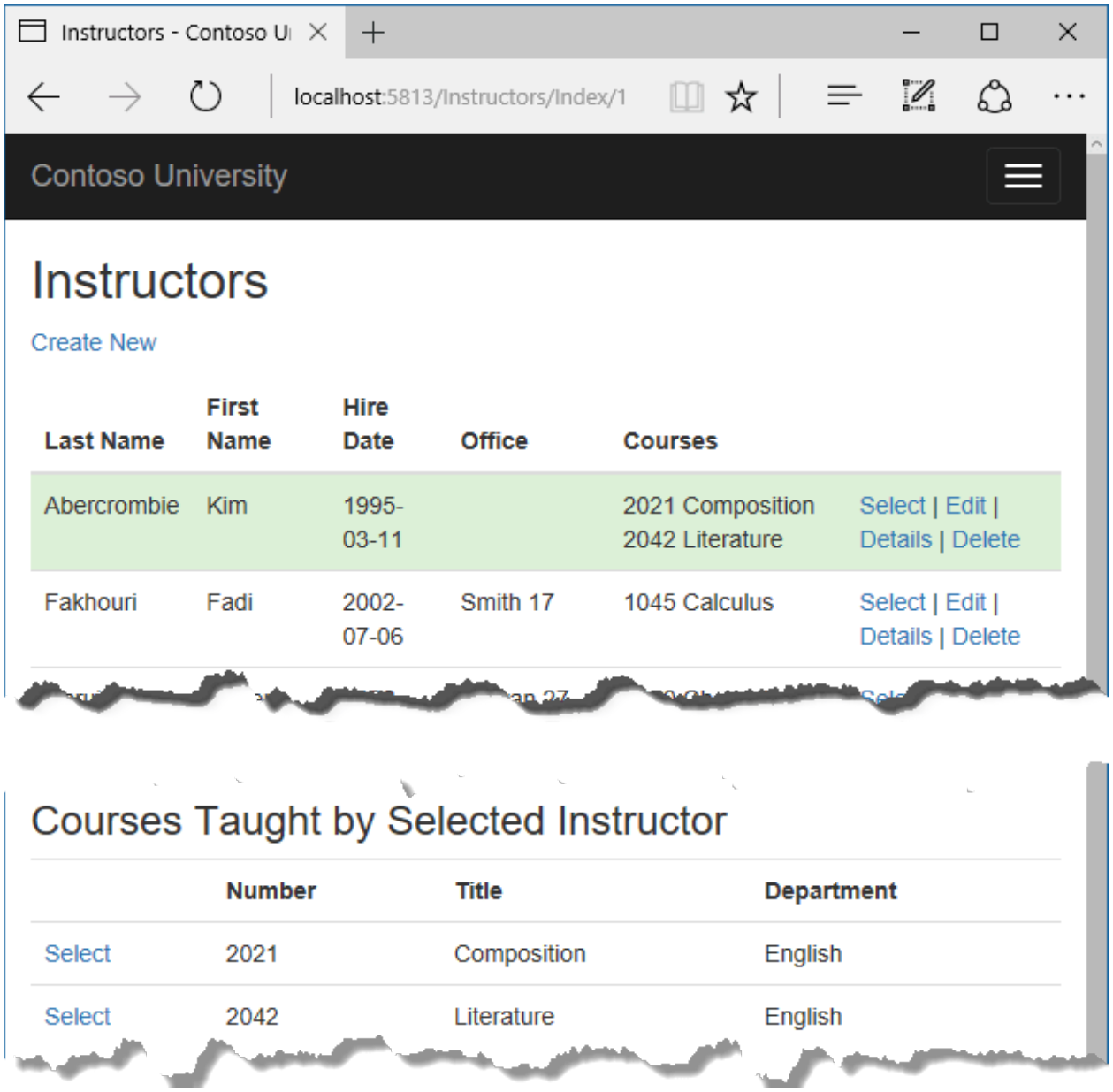
In the `Views/Instructors/Index.cshtml` file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

CSHTML

```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID =
item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a **Select** hyperlink that sends the ID of the selected course to the `Index` action method.

Refresh the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

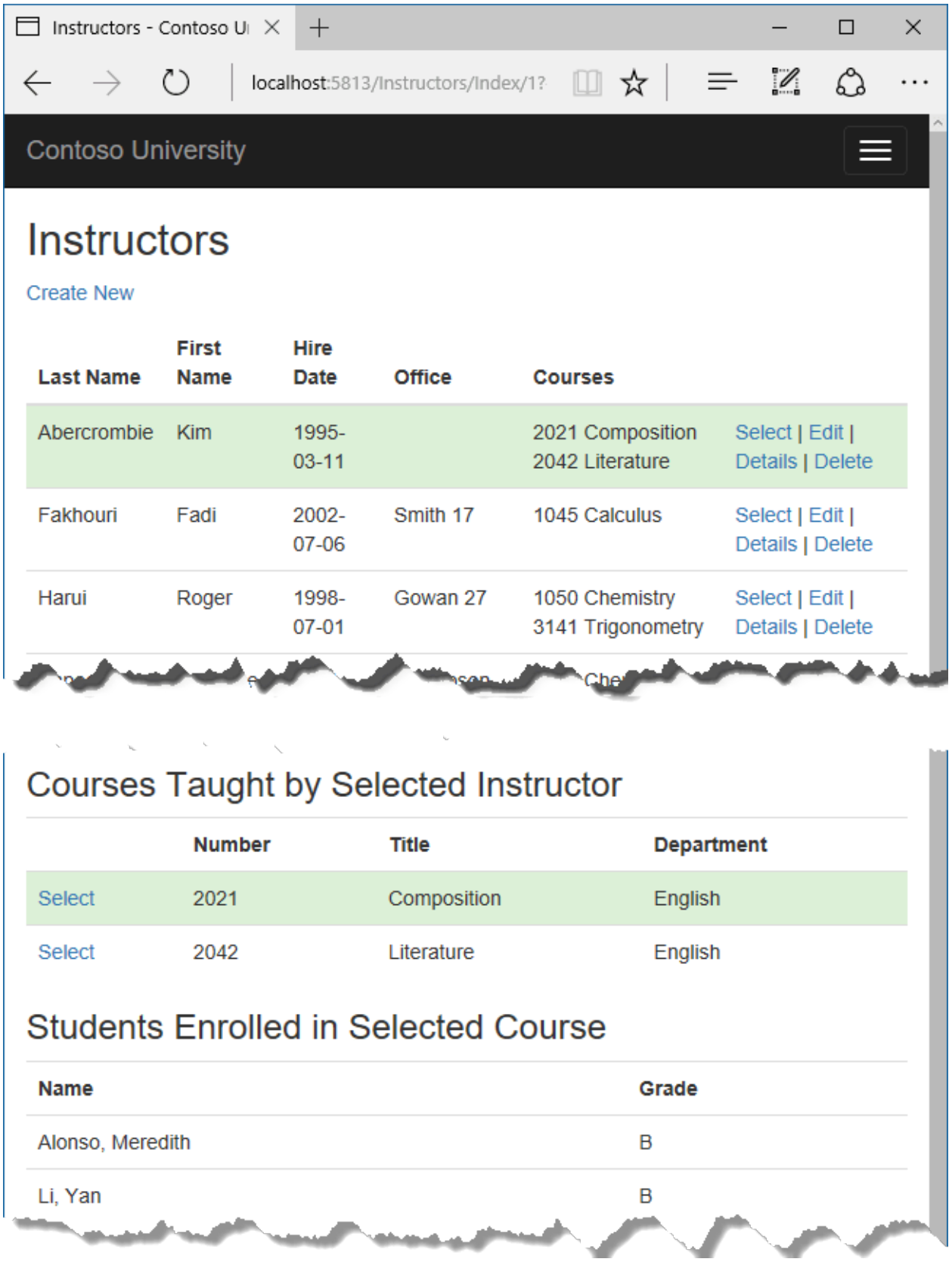
CSHTML

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
```

```
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the `Enrollments` property of the view model in order to display a list of students enrolled in the course.

Refresh the page again and select an instructor. Then select a course to see the list of enrolled students and their grades.



About explicit loading

When you retrieved the list of instructors in `InstructorsController.cs`, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if

it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for `Enrollments` and loads that property explicitly. The code changes are highlighted.

C#

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s =>
s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID ==
courseID).Single();
        await _context.Entry(selectedCourse).Collection(x =>
x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x =>
x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

The new code drops the `ThenInclude` method calls for enrollment data from the code that retrieves instructor entities. It also drops `AsNoTracking`. If an instructor and course are selected, the highlighted code retrieves `Enrollment` entities for the selected course, and `Student` entities for each `Enrollment`.

Run the app, go to the Instructors Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Learned how to load related data
- ✓ Created a Courses page
- ✓ Created an Instructors page
- ✓ Learned about explicit loading

Advance to the next tutorial to learn how to update related data.

Update related data

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

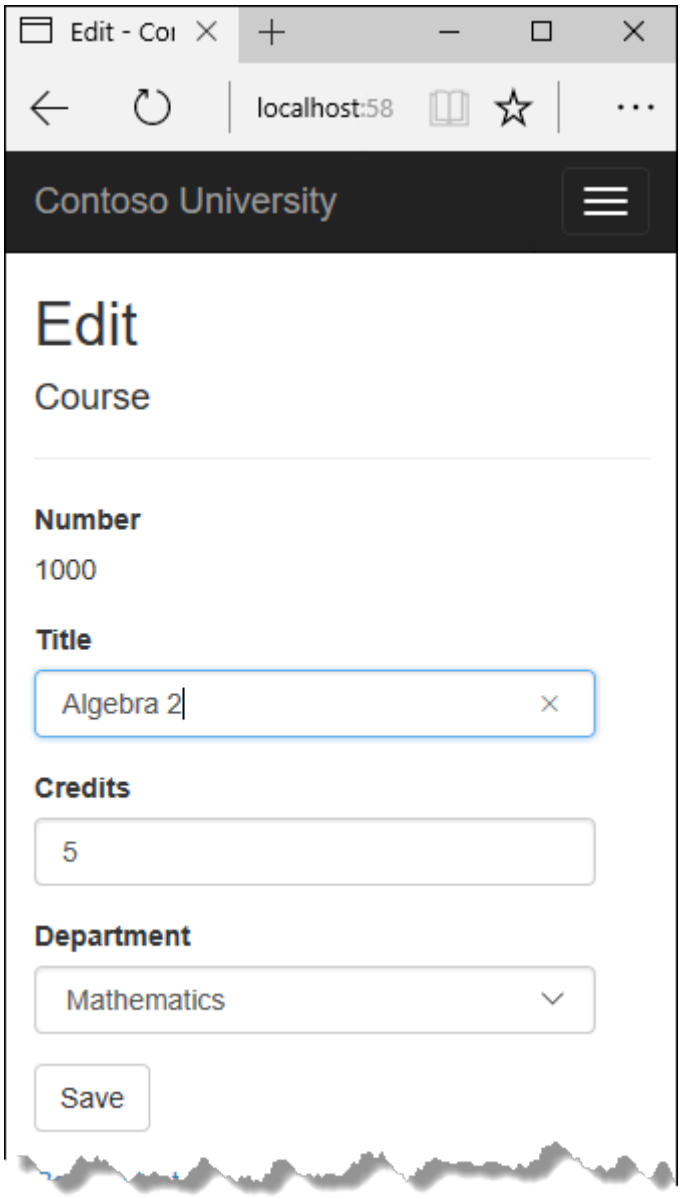
Provide product feedback

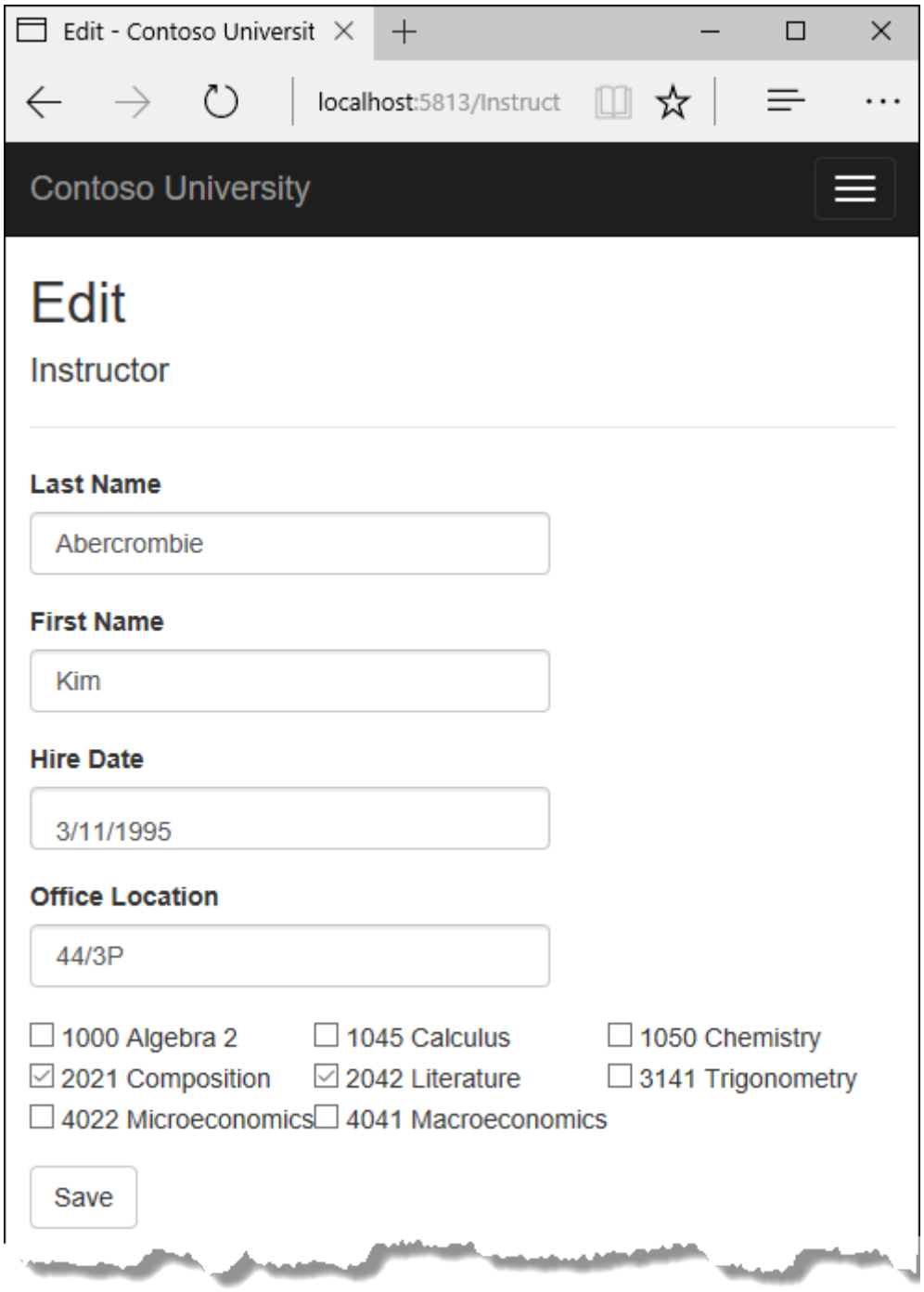
Tutorial: Update related data - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.





In this tutorial, you:

- ✓ Customize Courses pages
- ✓ Add Instructors Edit page
- ✓ Add courses to Edit page
- ✓ Update Delete page
- ✓ Add office location and courses to Create page

Prerequisites

- Read related data

Customize Courses pages

When a new `Course` entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate `Department` entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In `CoursesController.cs`, delete the four Create and Edit methods and replace them with the following code:

C#

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
```

```
    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .FirstOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}
```

After the `Edit` HttpPost method, create a new method that loads department info for the drop-down list.

C#

```
private void PopulateDepartmentsDropDownList(object selectedDepartment =
null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(),
"DepartmentID", "Name", selectedDepartment);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The HttpGet `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department isn't established yet:

C#

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The HttpGet `Edit` method sets the selected item, based on the ID of the department that's already assigned to the course being edited:

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
```

```
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The HttpPost methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add .AsNoTracking to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and HttpGet `Delete` methods.

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

C#

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
```

```
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

Modify the Course views

In `Views/Courses/Create.cshtml`, add a "Select Department" option to the **Department** drop-down list, change the caption from **DepartmentID** to **Department**, and add a validation message.

CSHTML

```
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-
items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />
</div>
```

In `Views/Courses/Edit.cshtml`, make the same change for the Department field that you just did in `Create.cshtml`.

Also in `Views/Courses/Edit.cshtml`, add a course number field before the **Title** field. Because the course number is the primary key, it's displayed, but it can't be changed.

CSHTML

```
<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>
```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the **Edit** page.

In `Views/Courses/Delete.cshtml`, add a course number field at the top and change department ID to department name.

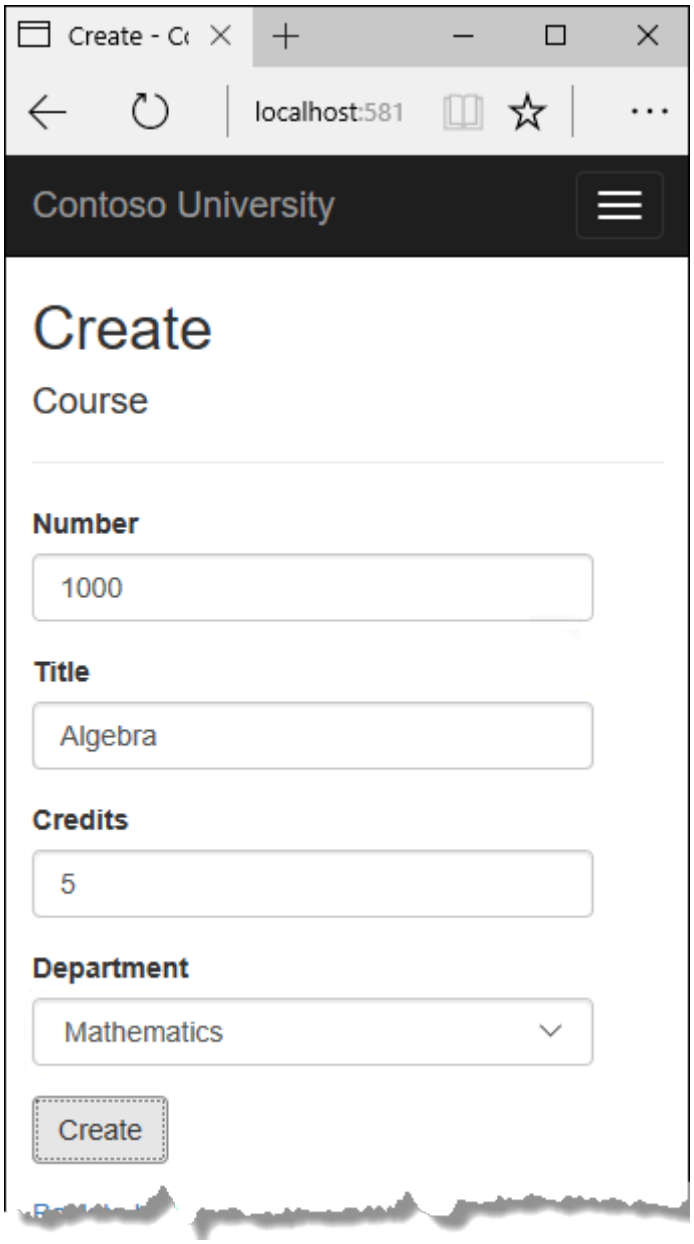
CSHTML

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

In `Views/Courses/Details.cshtml`, make the same change that you just did for `Delete.cshtml`.

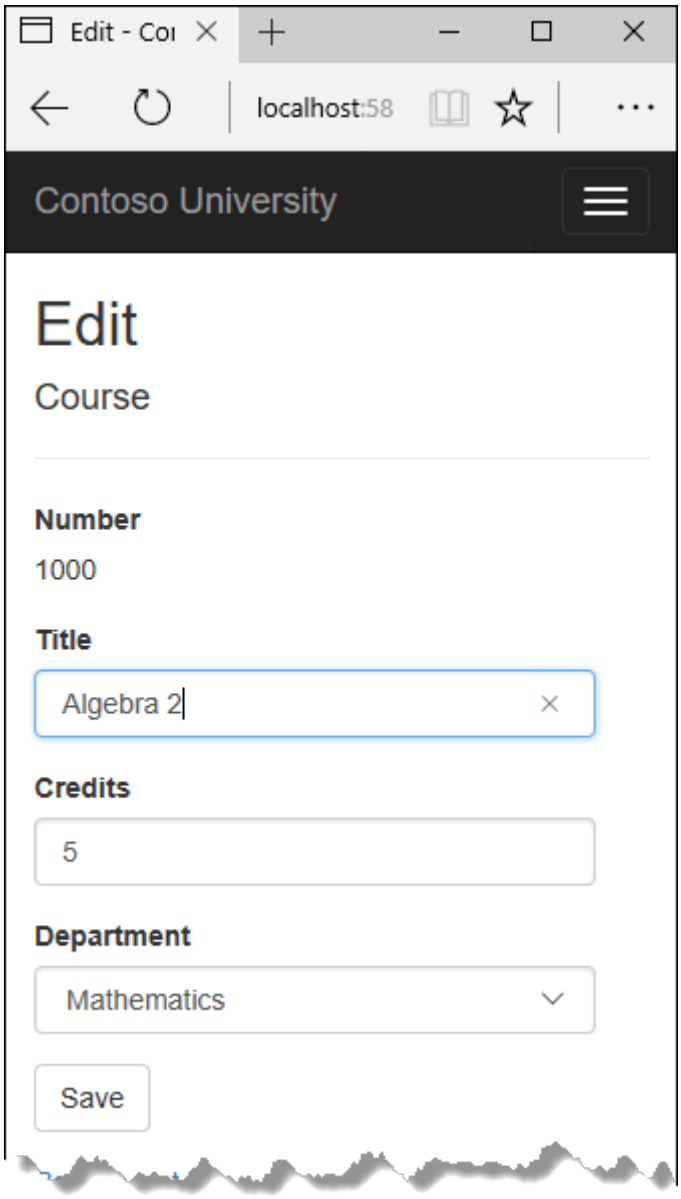
Test the Course pages

Run the app, select the **Courses** tab, click **Create New**, and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Click **Edit** on a course in the Courses Index page.



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add Instructors Edit page

When you edit an instructor record, you want to be able to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the `OfficeAssignment` entity.
- If the user enters an office assignment value and it originally was empty, create a new `OfficeAssignment` entity.
- If the user changes the value of an office assignment, change the value in an existing `OfficeAssignment` entity.

Update the Instructors controller

In `InstructorsController.cs`, change the code in the HttpGet `Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Replace the HttpPost `Edit` method with the following code to handle office assignment updates:

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .FirstOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
    {
```

```
        if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}
```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the HttpGet `Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the HttpGet `Edit` method.
- Updates the retrieved `Instructor` entity with values from the model binder. The `TryUpdateModel` overload enables you to declare the properties you want to include. This prevents over-posting, as explained in the second tutorial.

  C#

  ```
  if (await TryUpdateModelAsync<Instructor>(
      instructorToUpdate,
      "",
      i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
  i.OfficeAssignment))
  ```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

  C#

```
if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Locatio
n))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- Saves the changes to the database.

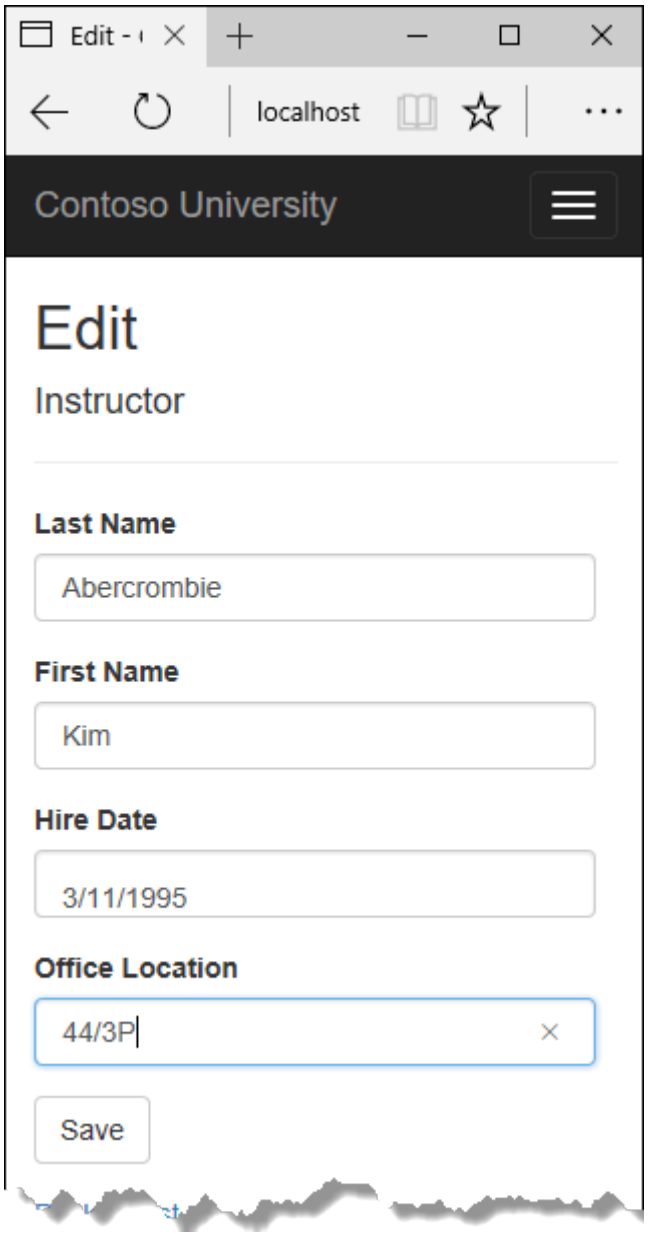
Update the Instructor Edit view

In `Views/Instructors/Edit.cshtml`, add a new field for editing the office location, at the end before the **Save** button:

CSHTML

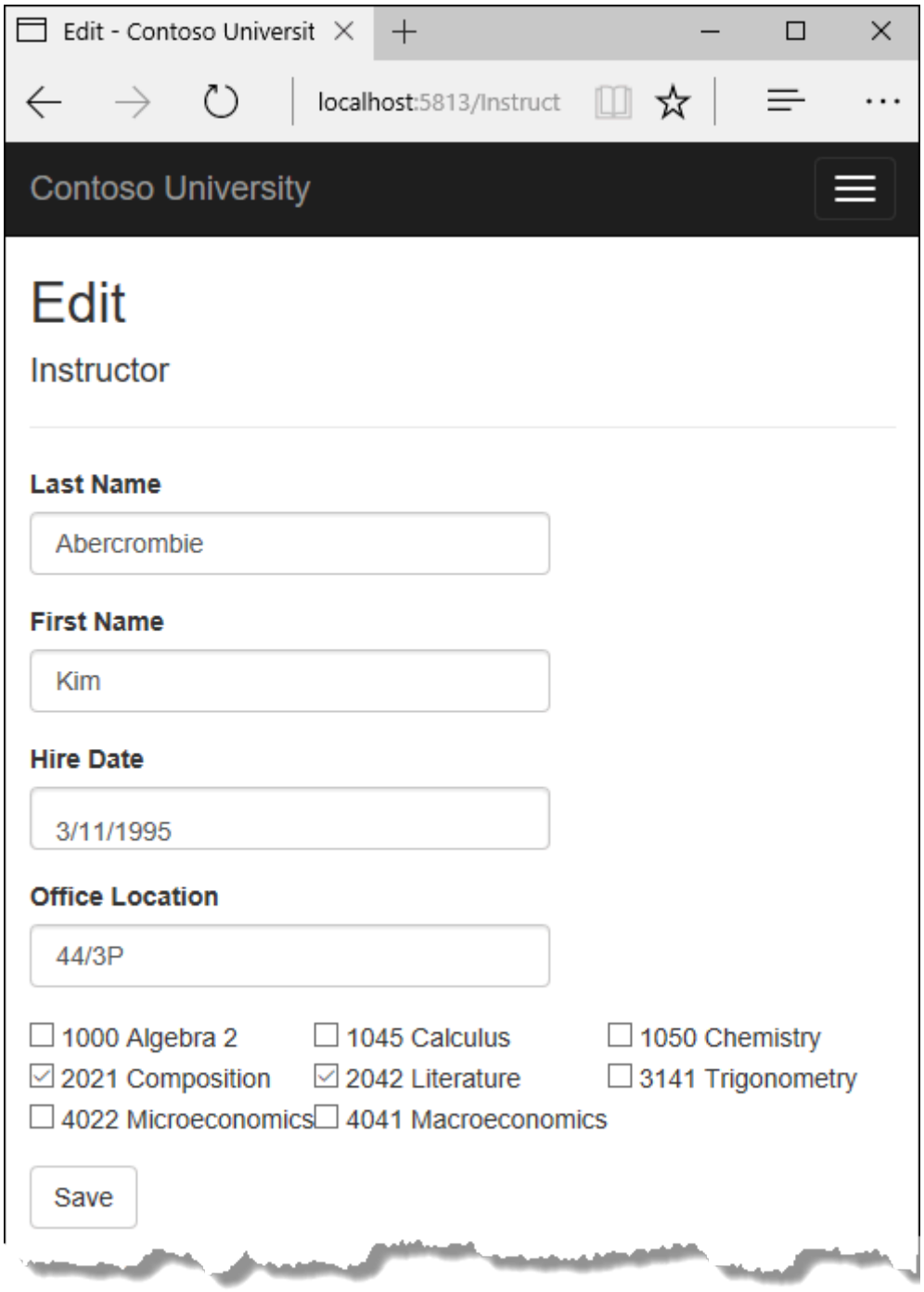
```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label">
</label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger"
/>
</div>
```

Run the app, select the **Instructors** tab, and then click **Edit** on an instructor. Change the **Office Location** and click **Save**.



Add courses to Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of checkboxes, as shown in the following screen shot:



The relationship between the `Course` and `Instructor` entities is many-to-many. To add and remove relationships, you add and remove entities to and from the `CourseAssignments` join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of checkboxes. A checkbox for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of checkboxes, you'll use a view model class.

Create `AssignedCourseData.cs` in the *SchoolViewModels* folder and replace the existing code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In `InstructorsController.cs`, replace the HttpGet `Edit` method with the following code. The changes are highlighted.

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
```

```
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>
(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the checkbox array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set to true for courses the instructor is assigned to. The view will use this property to determine which checkboxes must be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
```

```
            .FirstOrDefaultAsync(m => m.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "",
            i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
        {
            if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(selectedCourses, instructorToUpdate);
            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateException /* ex */)
            {
                //Log the error (uncomment ex variable name and write a log.)
                ModelState.AddModelError("", "Unable to save changes. " +
                    "Try again, and if the problem persists, " +
                    "see your system administrator.");
            }
            return RedirectToAction(nameof(Index));
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(instructorToUpdate);
        return View(instructorToUpdate);
    }
```

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The method signature is now different from the HttpGet `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore, you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload that requires explicit approval and `CourseAssignments` isn't in the include list.

If no checkboxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the checkbox for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection

in the navigation property.

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

If the checkbox for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
```

```
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

Update the Instructor views

In `Views/Instructors/Edit.cshtml`, add a **Courses** field with an array of checkboxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

> ⓘ Note
>
> When you paste the code in Visual Studio, line breaks might be changed in a way that breaks the code. If the code looks different after pasting, press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab three

times to line up the new code with the existing code. This problem is fixed in Visual Studio 2019.

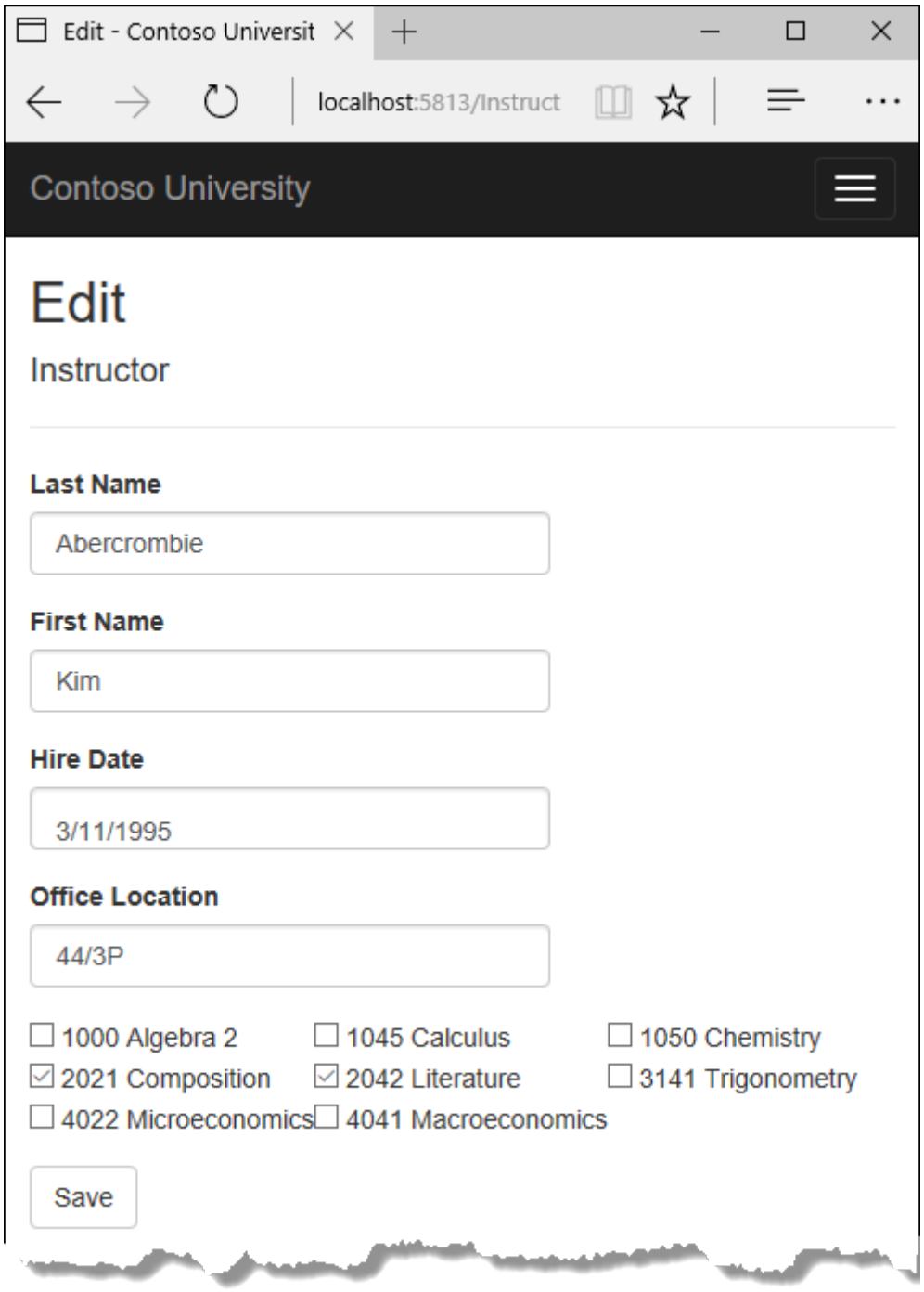
CSHTML

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

This code creates an HTML table that has three columns. In each column is a checkbox followed by a caption that consists of the course number and title. The checkboxes all have the same name ("selectedCourses"), which informs the model binder that they're to be treated as a group. The value attribute of each checkbox is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the checkboxes which are selected.

When the checkboxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the app, select the **Instructors** tab, and click **Edit** on an instructor to see the **Edit** page.



Change some course assignments and click Save. The changes you make are reflected on the Index page.

> ⓘ **Note**
>
> The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update Delete page

In `InstructorsController.cs`, delete the `DeleteConfirmed` method and insert the following code in its place.

```
C#

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to Create page

In `InstructorsController.cs`, delete the HttpGet and HttpPost `Create` methods, and then add the following code in their place:

```
C#

public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}
```

```
// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID =
instructor.ID, CourseID = int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The HttpGet `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost `Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

C#

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the `Instructor` model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

C#

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new
List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In `Views/Instructor/Create.cshtml`, add an office location text box and checkboxes for courses before the Submit button. As in the case of the Edit page, fix the formatting if Visual Studio reformats the code when you paste it.

CSHTML

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label">
</label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger"
/>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
```

```
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                            @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

Test by running the app and creating an instructor.

Handling Transactions

As explained in the CRUD tutorial, the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized Courses pages
- ✓ Added Instructors Edit page
- ✓ Added courses to Edit page
- ✓ Updated Delete page
- ✓ Added office location and courses to Create page

Advance to the next tutorial to learn how to handle concurrency conflicts.

Handle concurrency conflicts

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

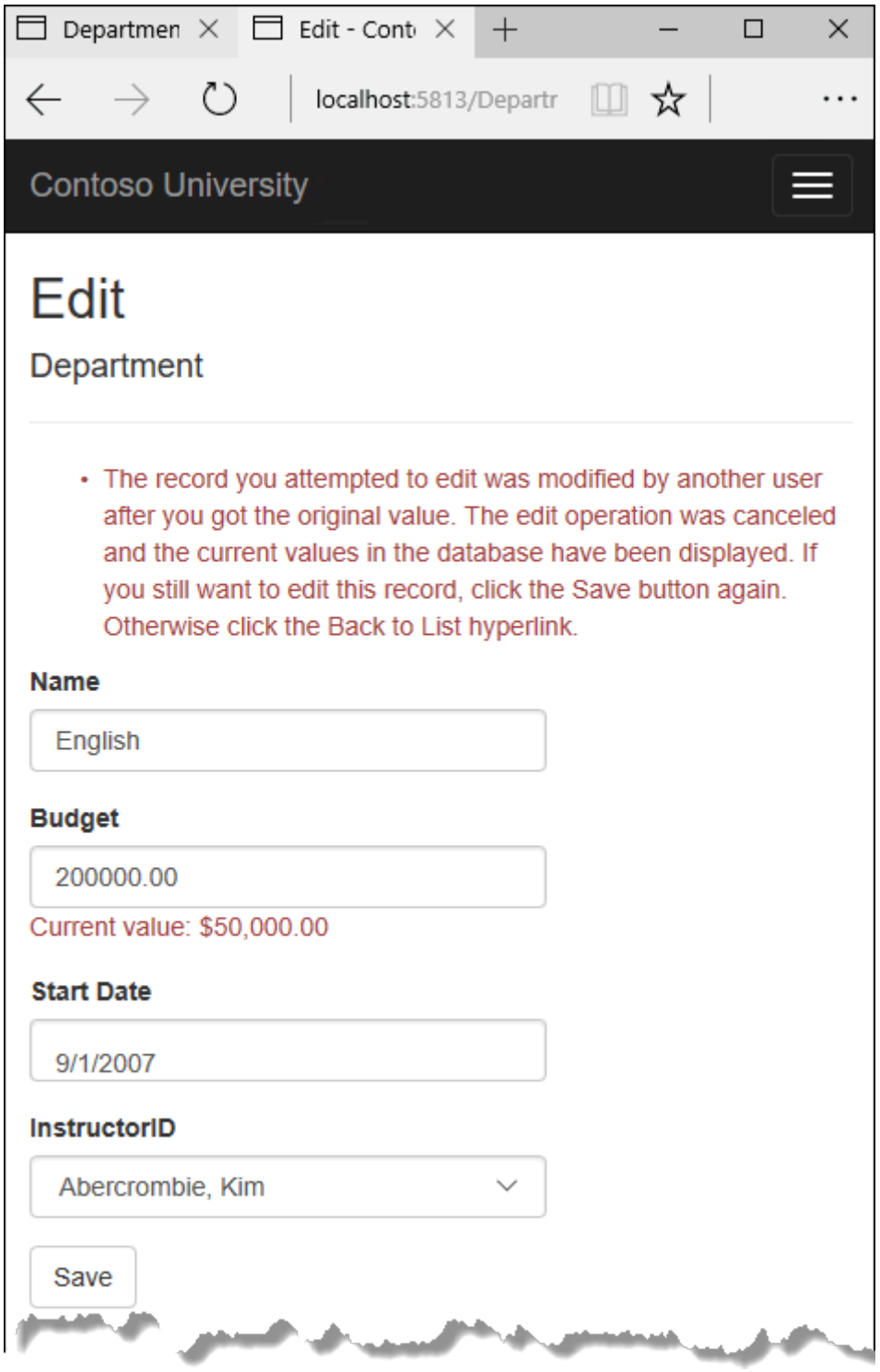
- Open a documentation issue
- Provide product feedback

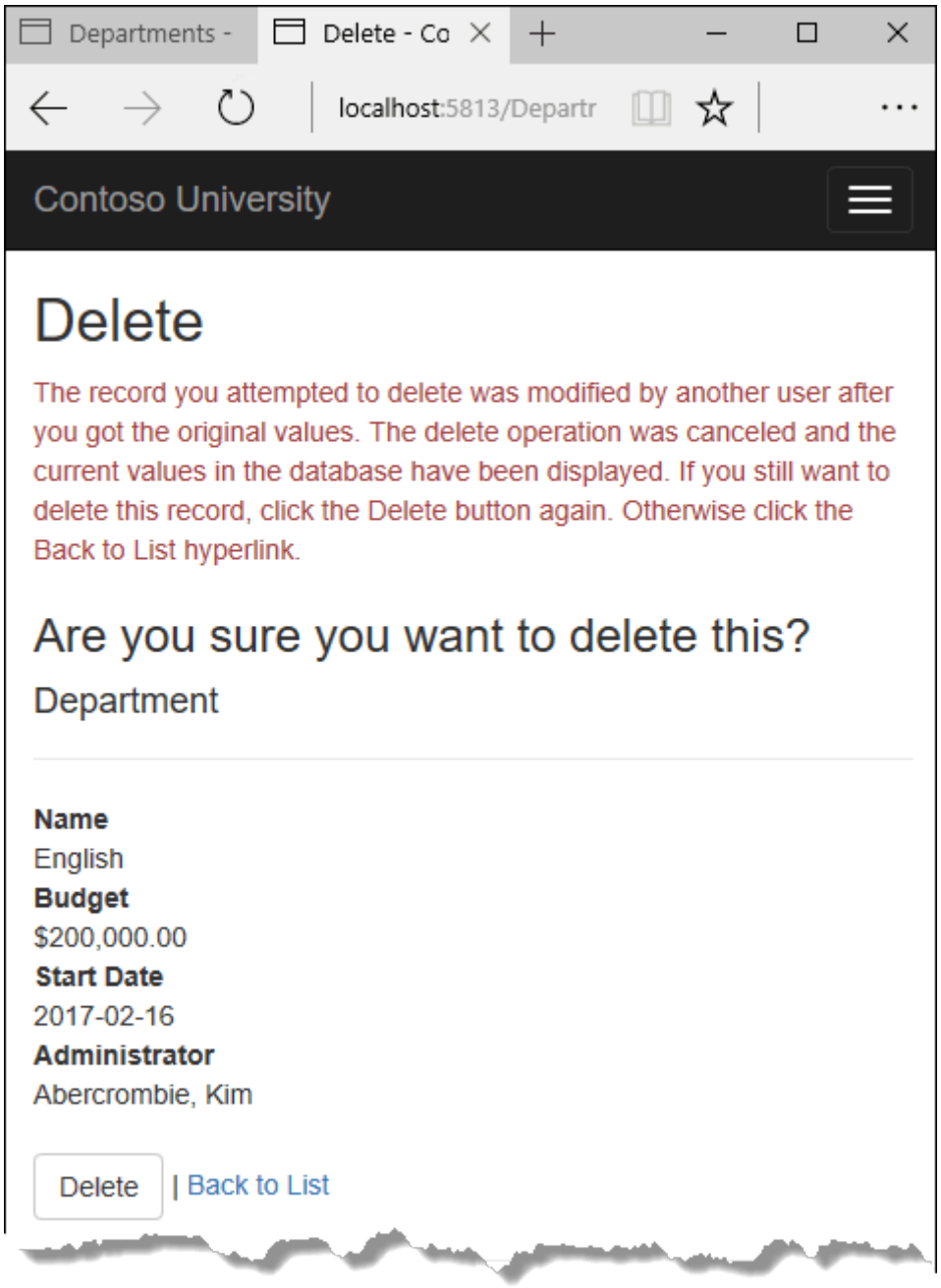
Tutorial: Handle concurrency - ASP.NET MVC with EF Core

Article • 04/11/2023

In earlier tutorials, you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the `Department` entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.


Departmen
Edit - Cont
localhost:5813/Departr
Contoso University
Edit
Department
The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.
Name
English
Budget
200000.00
Current value: $50,000.00
Start Date
9/1/2007
InstructorID
Abercrombie, Kim
Save



In this tutorial, you:

- ✓ Learn about concurrency conflicts
- ✓ Add a tracking property
- ✓ Create Departments controller and views
- ✓ Update Index view
- ✓ Update Edit methods
- ✓ Update Edit view
- ✓ Test concurrency conflicts
- ✓ Update the Delete page
- ✓ Update Details and Create views

Prerequisites

- Update related data

Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

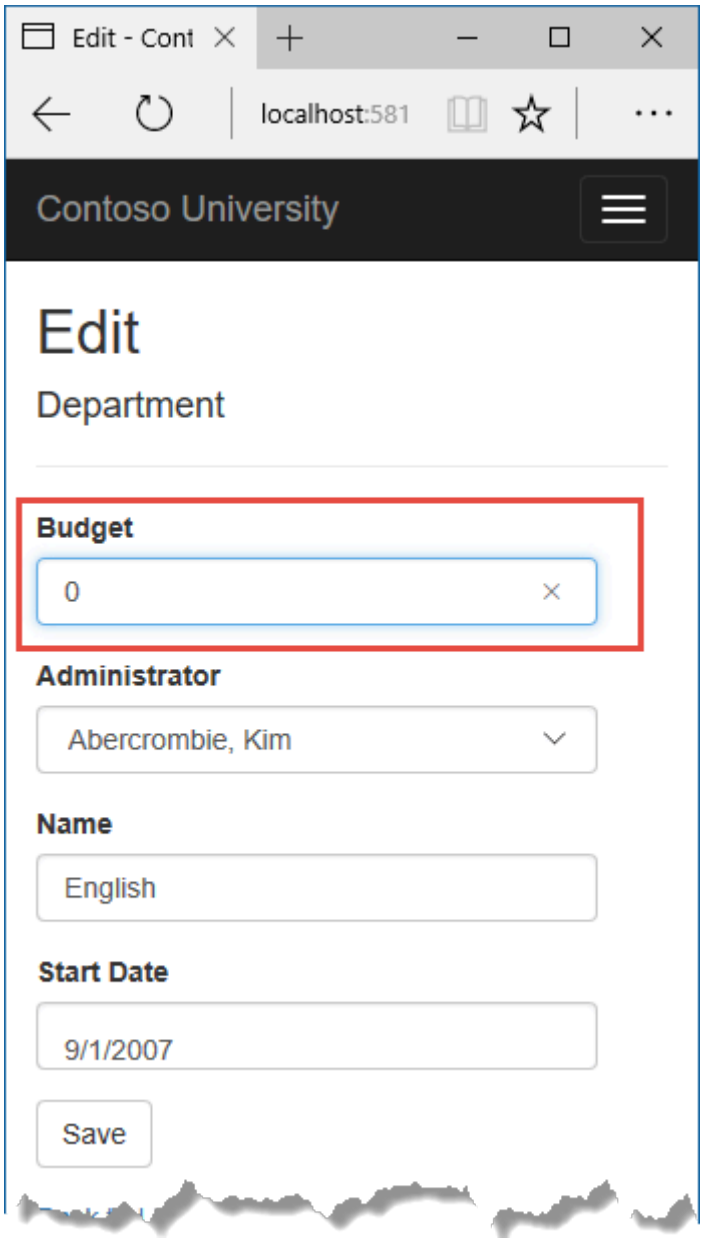
Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

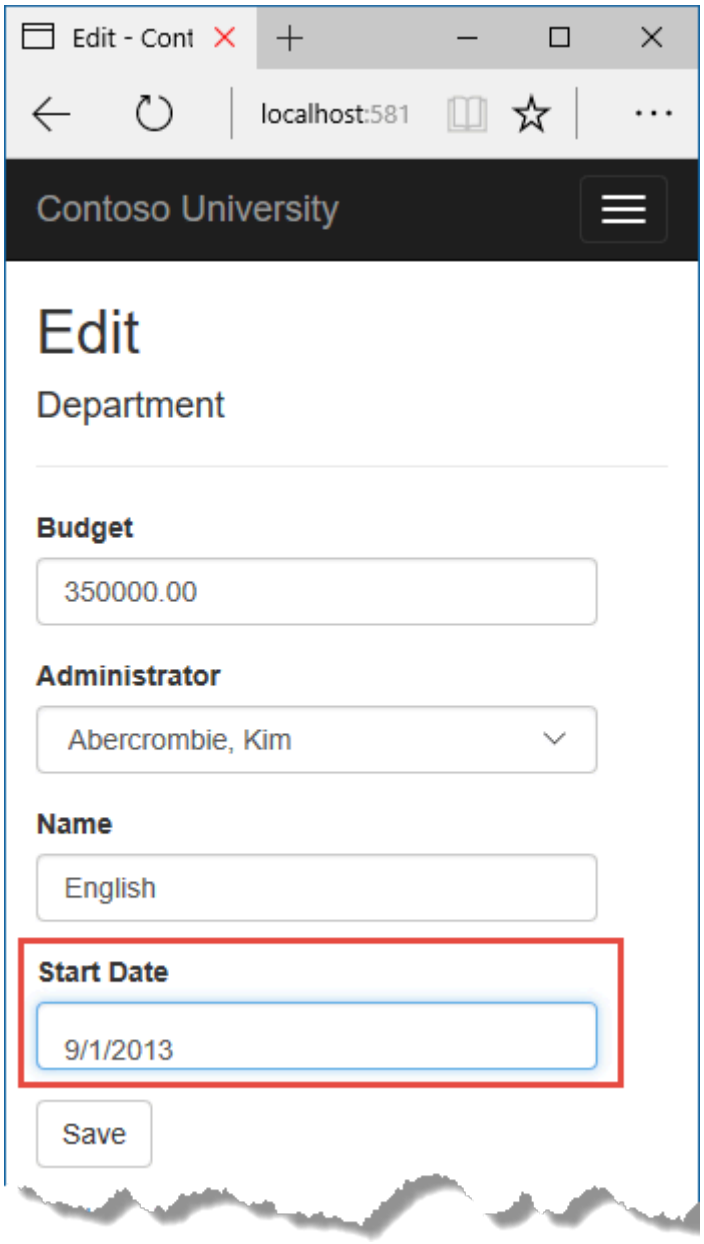
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

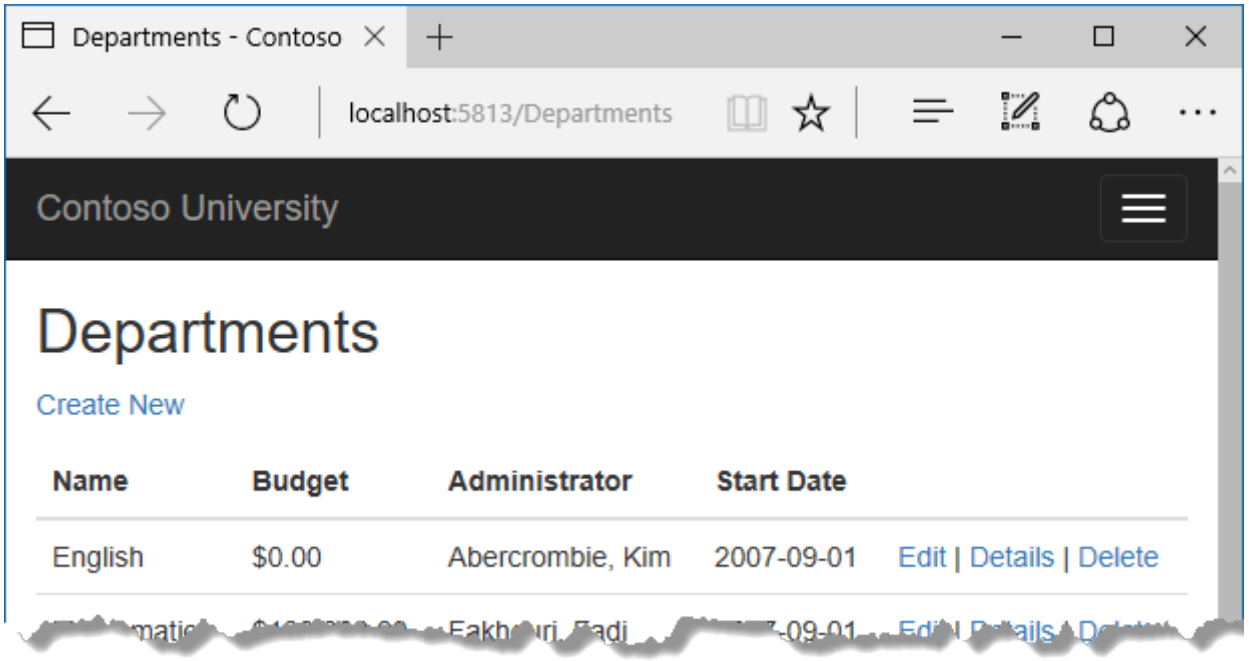
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



Then John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

  In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent John's change from being updated in the database.

  Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database

and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

  The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version) . If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

  As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

  If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

Add a tracking property

In `Models/Department.cs`, add a tracking property named RowVersion:

```
```

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsRowVersion()` method (in `Data/SchoolContext.cs`) to specify the tracking property, as shown in the following example:

C#

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsRowVersion();
```

By adding a property you changed the database model, so you need to do another migration.

Save your changes and build the project, and then enter the following commands in the command window:

.NET CLI

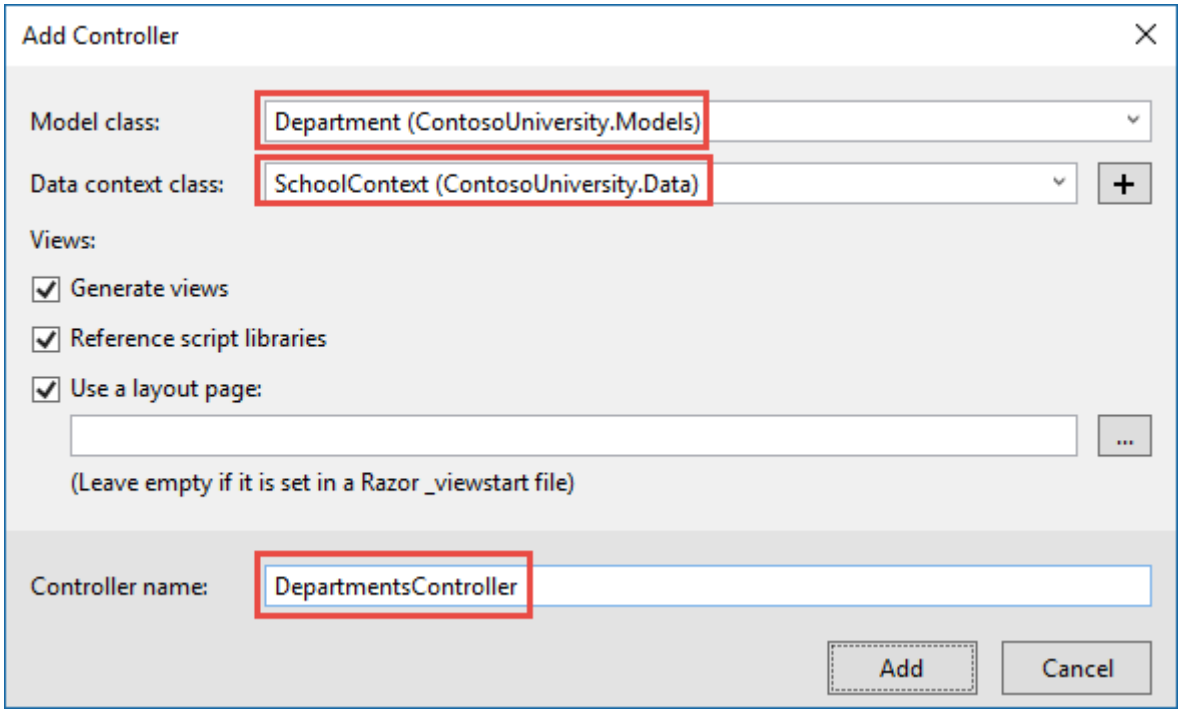
```
dotnet ef migrations add RowVersion
```

.NET CLI

```
dotnet ef database update
```

Create Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



In the `DepartmentsController.cs` file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

C#

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID",
"FullName", department.InstructorID);
```

Update Index view

The scaffolding engine created a `RowVersion` column in the Index view, but that field shouldn't be displayed.

Replace the code in `Views/Departments/Index.cshtml` with the following code.

CSHTML

```
@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
```

```
                    @Html.DisplayFor(modelItem =>
item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

This changes the heading to "Departments", deletes the `RowVersion` column, and shows full name instead of first name for the administrator.

Update Edit methods

In both the HttpGet `Edit` method and the `Details` method, add `AsNoTracking`. In the HttpGet `Edit` method, add eager loading for the Administrator.

C#

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .FirstOrDefaultAsync(m => m.DepartmentID == id);
```

Replace the existing code for the HttpPost `Edit` method with the following code:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i =>
i.Administrator).FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
```

```
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another
user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors,
"ID", "FullName", deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue
= rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by
another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value:
{databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value:
{databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {
                    ModelState.AddModelError("StartDate", $"Current value:
{databaseValues.StartDate:d}");
                }
                if (databaseValues.InstructorID !=
clientValues.InstructorID)
```

```
                {
                    Instructor databaseInstructor = await
_context.Instructors.FirstOrDefaultAsync(i => i.ID ==
databaseValues.InstructorID);
                    ModelState.AddModelError("InstructorID", $"Current
value: {databaseInstructor?.FullName}");
                }

                ModelState.AddModelError(string.Empty, "The record you
attempted to edit "
                        + "was modified by another user after you got the
original value. The "
                        + "edit operation was canceled and the current
values in the database "
                        + "have been displayed. If you still want to edit
this record, click "
                        + "the Save button again. Otherwise click the Back
to List hyperlink.");
                departmentToUpdate.RowVersion =
(byte[])databaseValues.RowVersion;
                ModelState.Remove("RowVersion");
            }
        }
    }
    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID",
"FullName", departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}
```

The code begins by trying to read the department to be updated. If the `FirstOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a `Department` entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the `Department` entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

C#

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue =
rowVersion;
```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no

rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

C#

```
var exceptionEntry = ex.Entries.Single();
```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.

C#

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

C#

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value:
{databaseValues.Name}");
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

C#

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update Edit view

In `Views/Departments/Edit.cshtml`, make the following changes:

- Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.
- Add a "Select Administrator" option to the drop-down list.

CSHTML

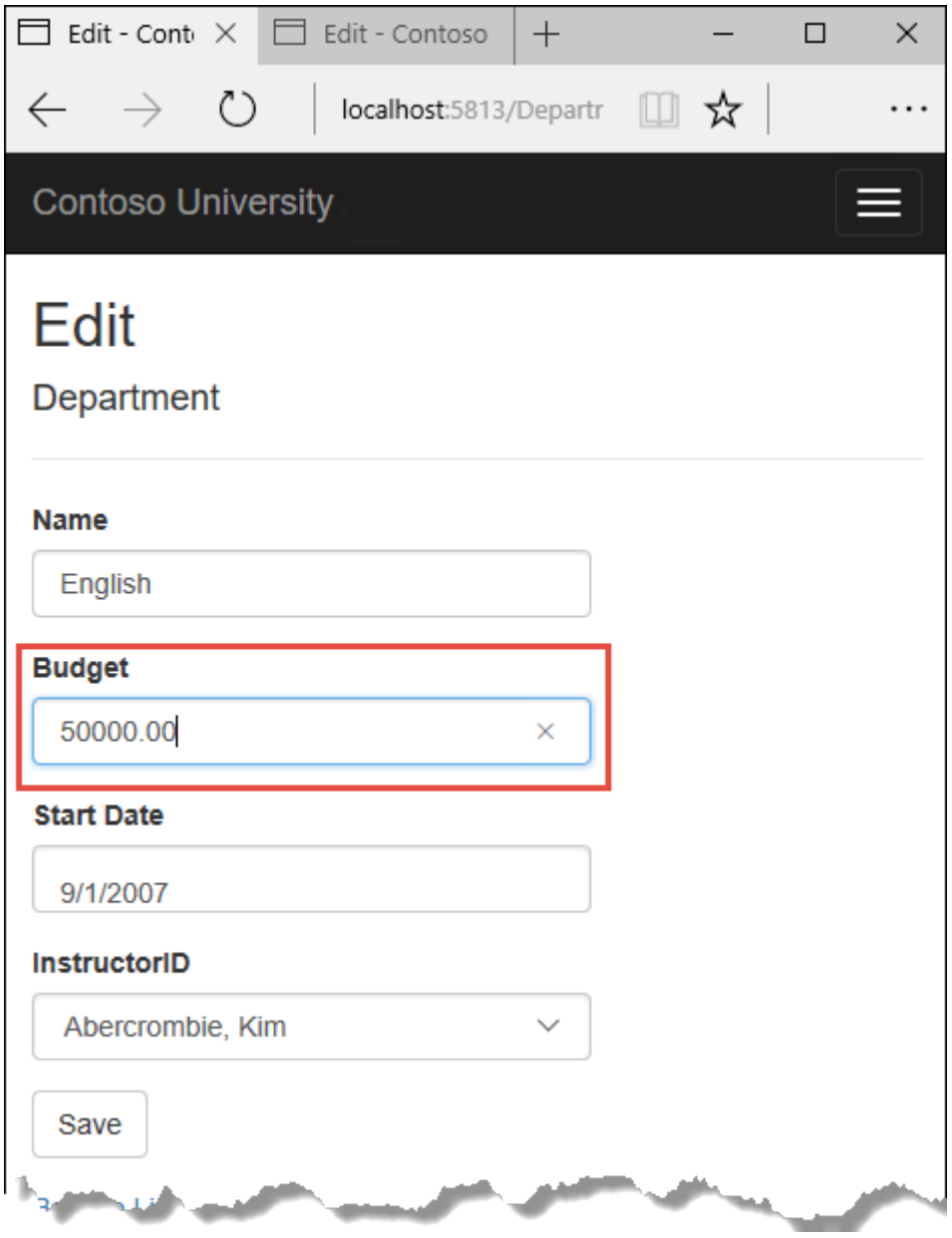
```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger">
```

```
</span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test concurrency conflicts

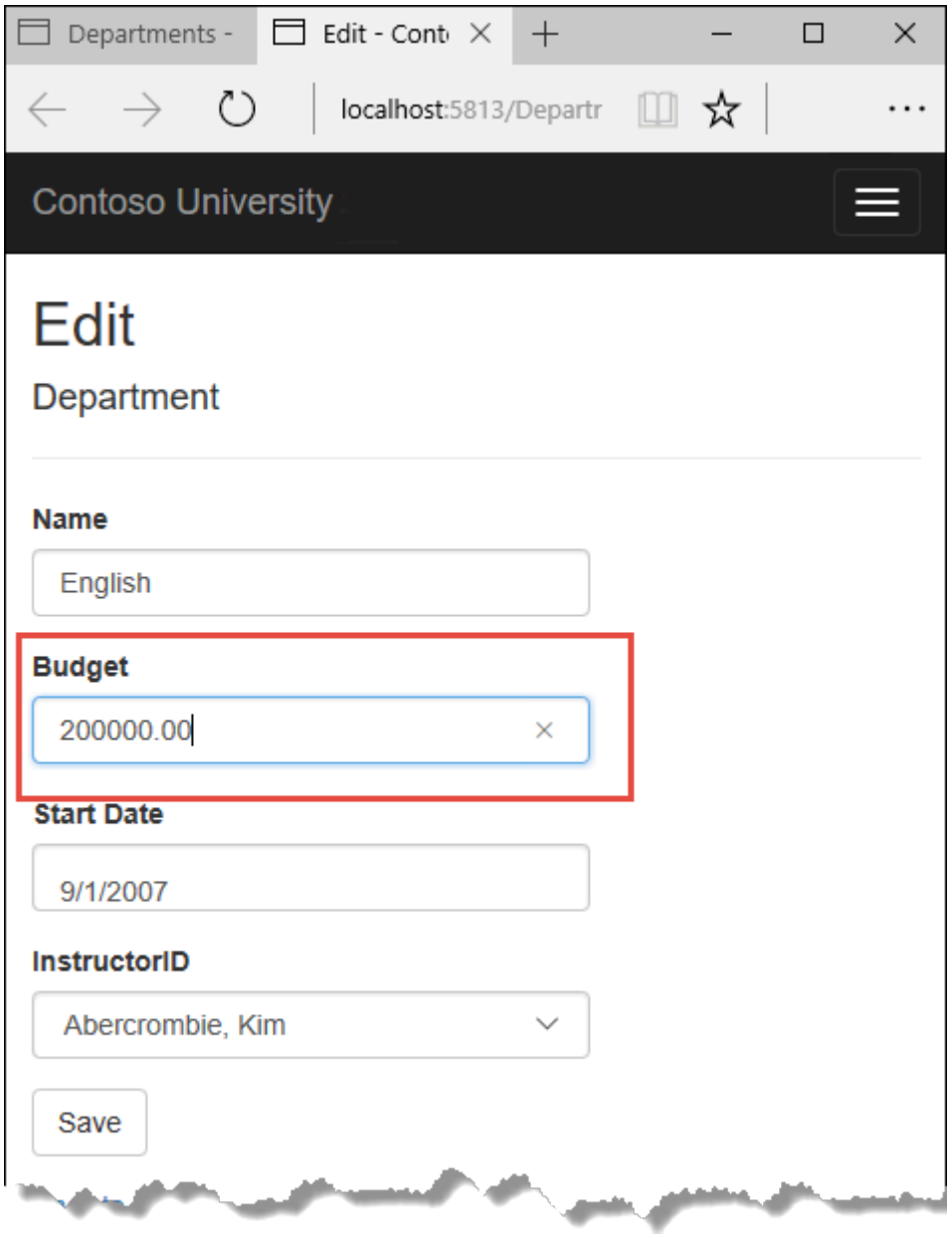
Run the app and go to the Departments Index page. Right-click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.

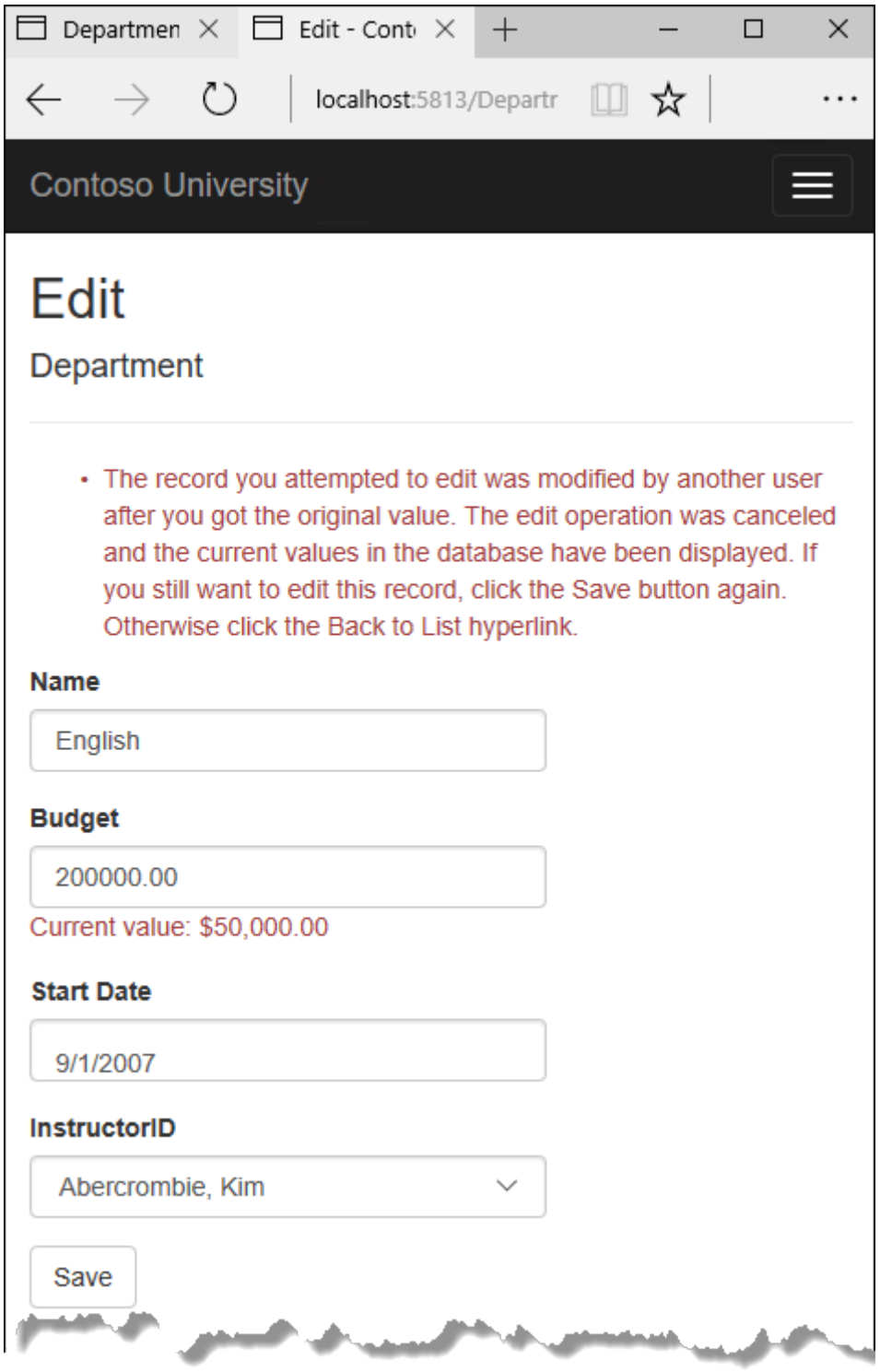


The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the HttpGet `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the HttpPost `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL

DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the HttpGet `Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In `DepartmentsController.cs`, replace the HttpGet `Delete` method with the following code:

C#

```
public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to
delete "
            + "was modified by another user after you got the original
values. "
            + "The delete operation was canceled and the current values in
the "
            + "database have been displayed. If you still want to delete
this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }
```

```
    return View(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the department does exist, it was changed by another user. In that case, the code sends an error message to the view using `ViewData`.

Replace the code in the HttpPost `Delete` method (named `DeleteConfirmed`) with the following code:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID ==
department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError =
true, id = department.DepartmentID });
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

C#

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a `Department` entity instance created by the model binder. This gives EF access to the RowVers`ion property value in addition to the record key.

C#

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In `Views/Departments/Delete.cshtml`, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
```

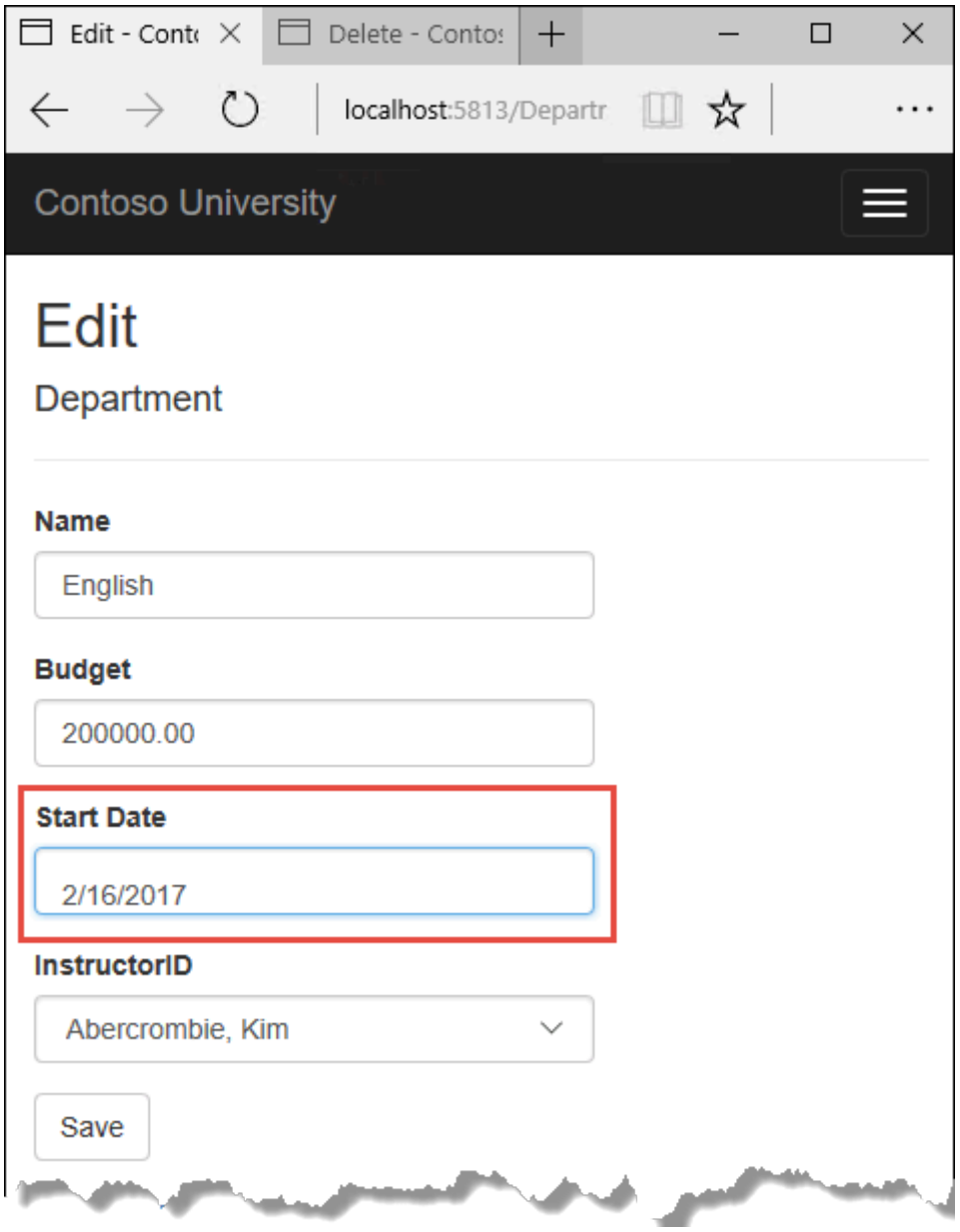
```
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

This makes the following changes:

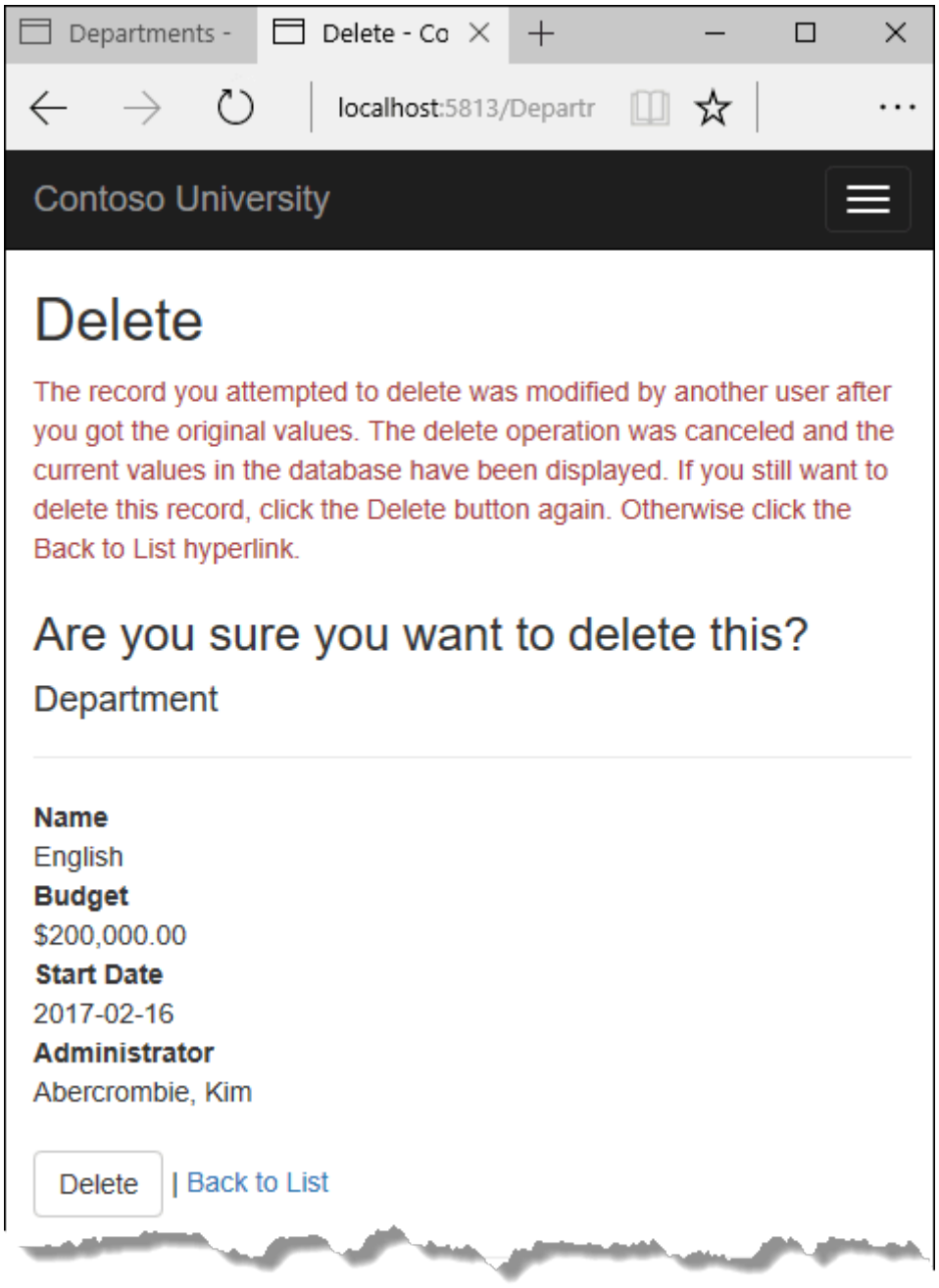
- Adds an error message between the `h2` and `h3` headings.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Removes the RowVersion field.
- Adds a hidden field for the `RowVersion` property.

Run the app and go to the Departments Index page. Right-click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in `Views/Departments/Details.cshtml` to delete the RowVersion column and show the full name of the Administrator.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
```

```
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Replace the code in `Views/Departments/Create.cshtml` to add a Select option to the drop-down list.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
```

```
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Get the code

Download or view the completed application.

Additional resources

For more information about how to handle concurrency in EF Core, see Concurrency conflicts.

Next steps

In this tutorial, you:

- ✓ Learned about concurrency conflicts
- ✓ Added a tracking property
- ✓ Created Departments controller and views
- ✓ Updated Index view
- ✓ Updated Edit methods
- ✓ Updated Edit view
- ✓ Tested concurrency conflicts
- ✓ Updated the Delete page
- ✓ Updated Details and Create views

Advance to the next tutorial to learn how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

Next: Implement table-per-hierarchy inheritance

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Implement inheritance - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

In this tutorial, you:

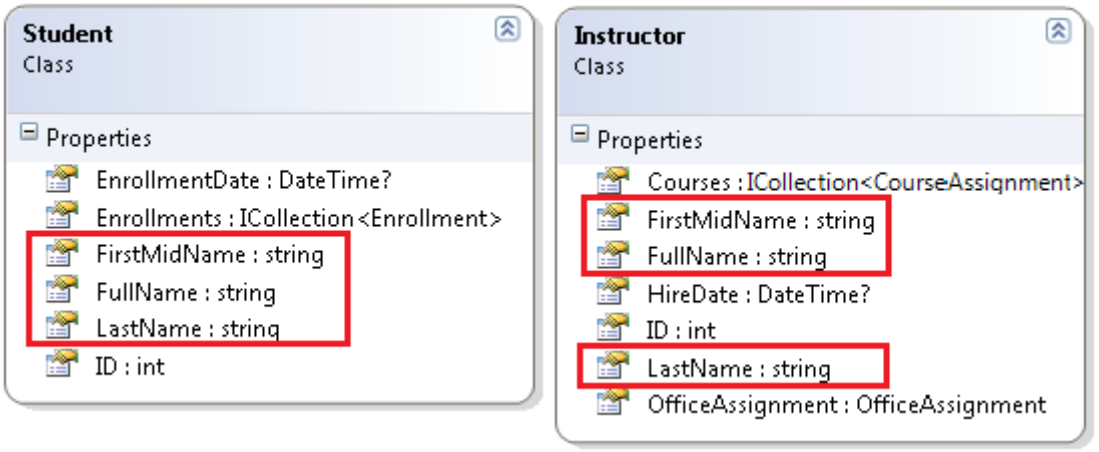
- ✓ Map inheritance to database
- ✓ Create the Person class
- ✓ Update Instructor and Student
- ✓ Add Person to the model
- ✓ Create and update migrations
- ✓ Test the implementation

Prerequisites

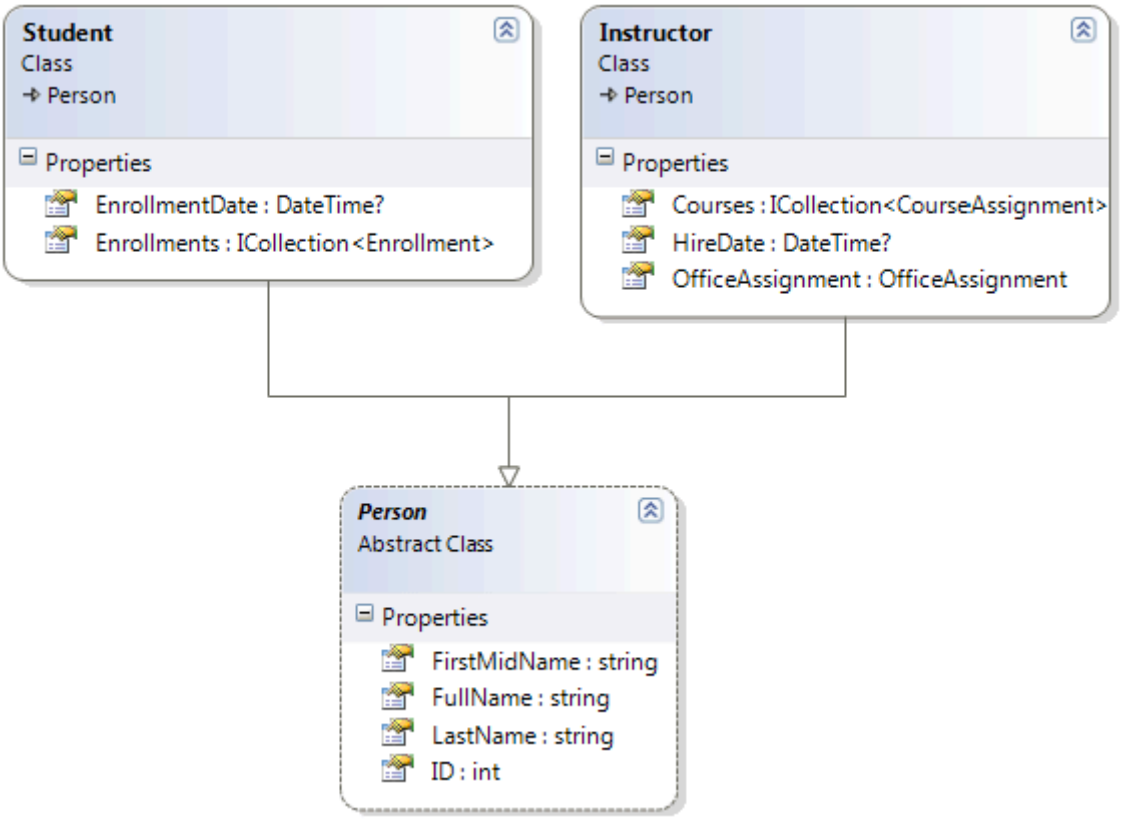
- Handle Concurrency

Map inheritance to database

The `Instructor` and `Student` classes in the School data model have several properties that are identical:

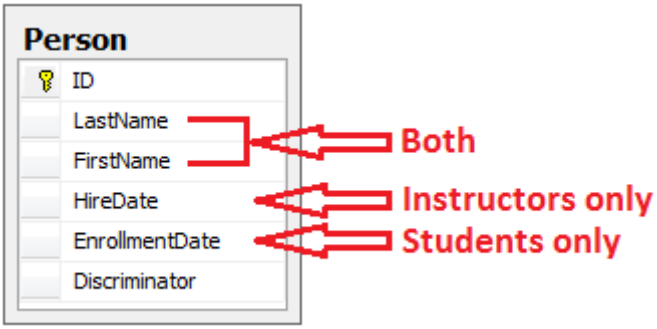


Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:



There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row

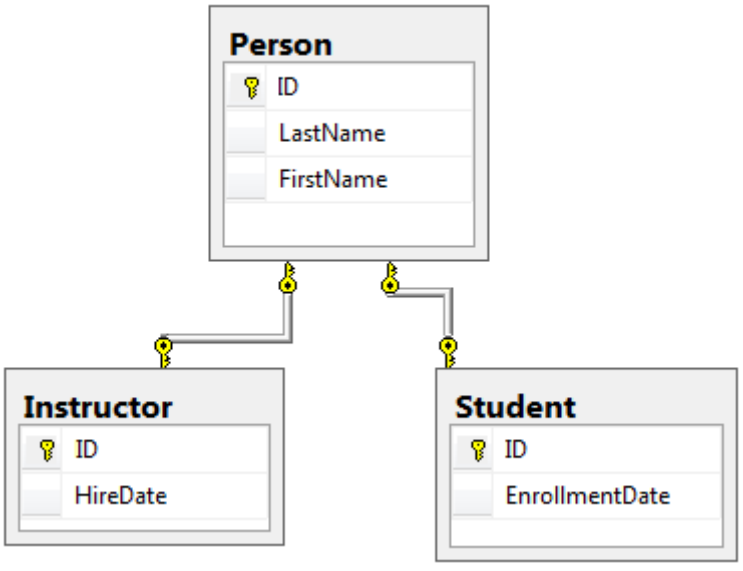
represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy (TPH)* inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the `Person` table and have separate `Instructor` and `Student` tables with the date fields.

> ⚠ **Warning**
>
> Table-Per-Type (TPT) is not supported by EF Core 3.x, however it is has been implemented in EF Core 5.0.



This pattern of making a database table for each entity class is called *table-per-type (TPT)* inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called *Table-per-Concrete Class (TPC)* inheritance. If you implemented TPC

inheritance for the `Person`, `Student`, and `Instructor` classes as shown earlier, the `Student` and `Instructor` tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

> **Tip**
>
> Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

Create the Person class

In the Models folder, create Person.cs and replace the template code with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than
50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
```

```
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

Update Instructor and Student

In `Instructor.cs`, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Make the same changes in `Student.cs`.

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
```

```
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }


        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Add Person to the model

Add the Person entity type to `SchoolContext.cs`. The new lines are highlighted.

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>
().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>
().ToTable("CourseAssignment");
```

```
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and update migrations

Save your changes and build the project. Then open the command window in the project folder and enter the following command:

.NET CLI

```
dotnet ef migrations add Inheritance
```

Don't run the `database update` command yet. That command will result in lost data because it will drop the Instructor table and rename the Student table to Person. You need to provide custom code to preserve existing data.

Open `Migrations/<timestamp>_Inheritance.cs` and replace the `Up` method with the following code:

C#

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table:
"Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table:
"Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table:
"Person", nullable: false, maxLength: 128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table:
"Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person",
```

```
nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName,
HireDate, EnrollmentDate, Discriminator, OldId) SELECT LastName, FirstName,
null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, ID AS OldId
FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID
FROM dbo.Person WHERE OldId = Enrollment.StudentId AND Discriminator =
'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
         name: "IX_Enrollment_StudentID",
         table: "Enrollment",
         column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
- Adds nullable EnrollmentDate for students.
- Adds Discriminator column to indicate whether a row is for a student or an instructor.
- Makes HireDate nullable since student rows won't have hire dates.
- Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they will get new primary key values.

- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command:

.NET CLI

```
dotnet ef database update
```

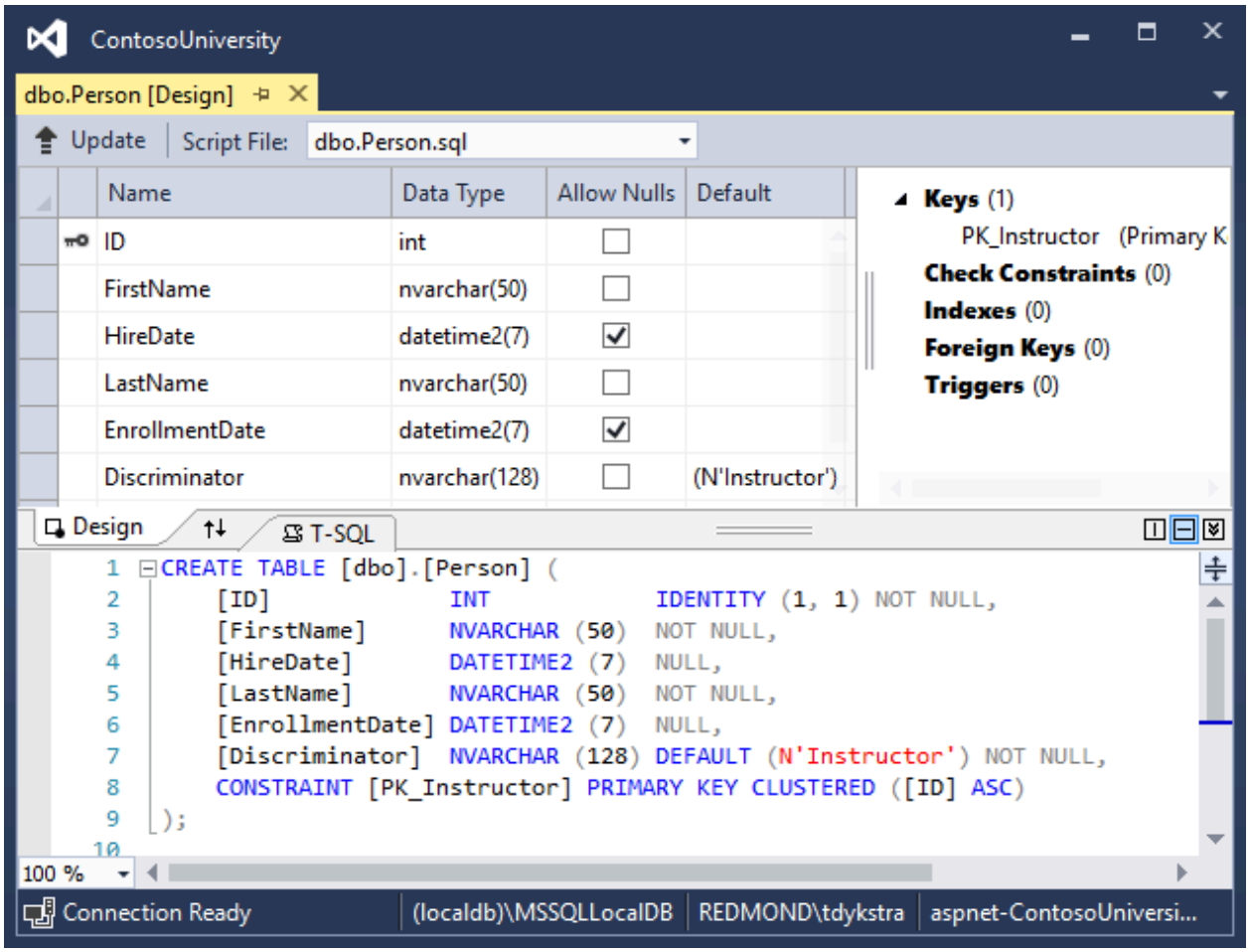
(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

> **Note**
>
> It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `database drop` CLI command.

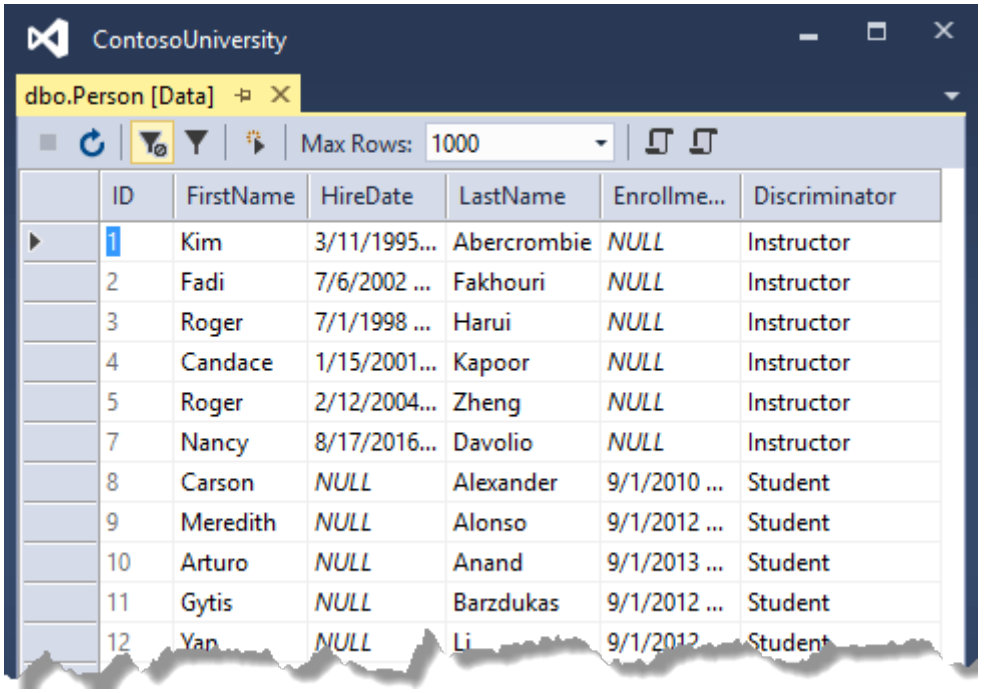
Test the implementation

Run the app and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.



Get the code

Download or view the completed application.

Additional resources

For more information about inheritance in Entity Framework Core, see Inheritance.

Next steps

In this tutorial, you:

- ✓ Mapped inheritance to database
- ✓ Created the Person class
- ✓ Updated Instructor and Student
- ✓ Added Person to the model
- ✓ Created and update migrations
- ✓ Tested the implementation

Advance to the next tutorial to learn how to handle a variety of relatively advanced Entity Framework scenarios.

Next: Advanced topics

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Learn about advanced scenarios - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET Core web applications that use Entity Framework Core.

In this tutorial, you:

- ✓ Perform raw SQL queries
- ✓ Call a query to return entities
- ✓ Call a query to return other types
- ✓ Call an update query
- ✓ Examine SQL queries
- ✓ Create an abstraction layer
- ✓ Learn about Automatic change detection
- ✓ Learn about EF Core source code and development plans
- ✓ Learn how to use dynamic LINQ to simplify code

Prerequisites

- Implement Inheritance

Perform raw SQL queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

- Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they're automatically tracked by the database context unless you turn tracking off.
- Use the `Database.ExecuteSqlCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Call a query to return entities

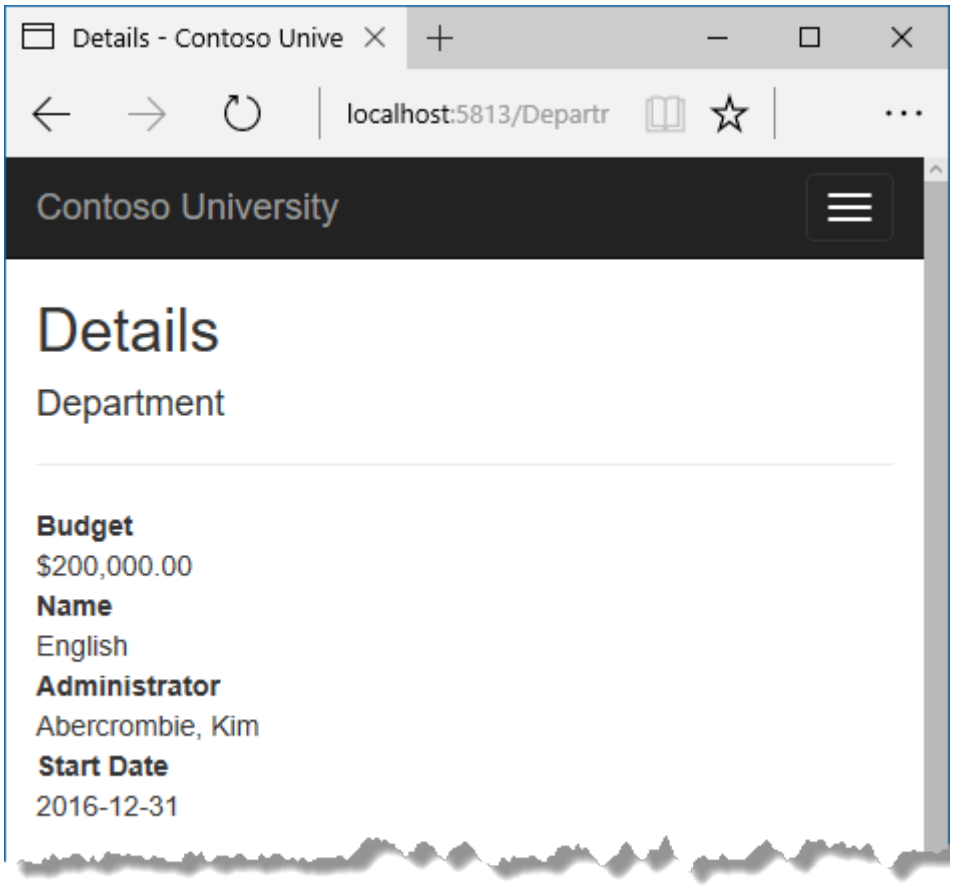
The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In `DepartmentsController.cs`, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Call a query to return other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is to write ADO.NET code and get the database connection from EF.

In `HomeController.cs`, replace the `About` method with the following code:

C#

```
public async Task<ActionResult> About()
{
```

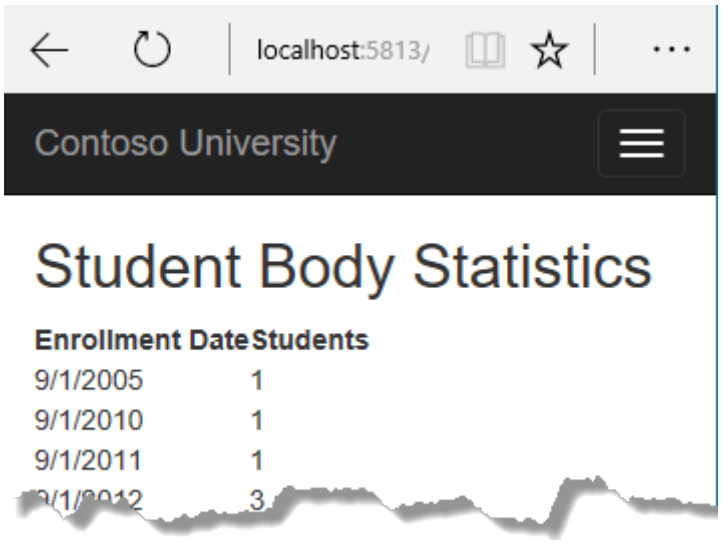
```
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount
"
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate =
reader.GetDateTime(0), StudentCount = reader.GetInt32(1) };
                    groups.Add(row);
                }
            }
            reader.Dispose();
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}
```

Add a using statement:

C#

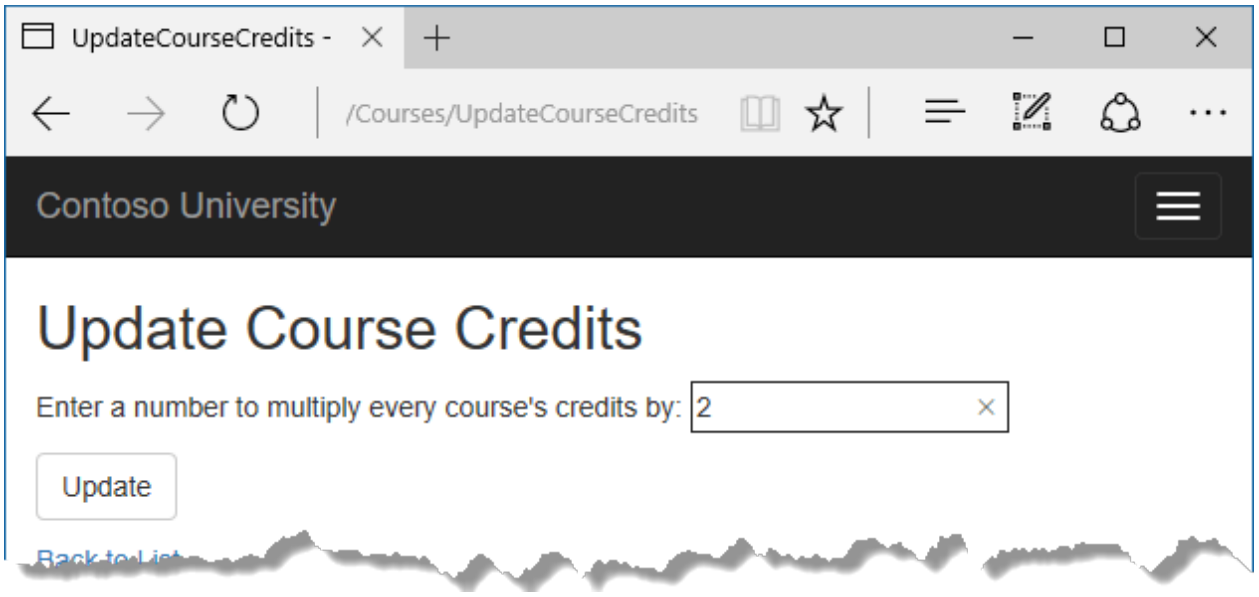
```
using System.Data.Common;
```

Run the app and go to the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In `CoursesController.cs`, add UpdateCourseCredits methods for HttpGet and HttpPost:

C#

```
public IActionResult UpdateCourseCredits()
{
```

```
    return View();
}
```

C#

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

When the controller processes an HttpGet request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the HttpPost method is called, and multiplier has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

In **Solution Explorer**, right-click the *Views/Courses* folder, and then click **Add** > **New Item**.

In the **Add New Item** dialog, click **ASP.NET Core** under **Installed** in the left pane, click **Razor View**, and name the new view `UpdateCourseCredits.cshtml`.

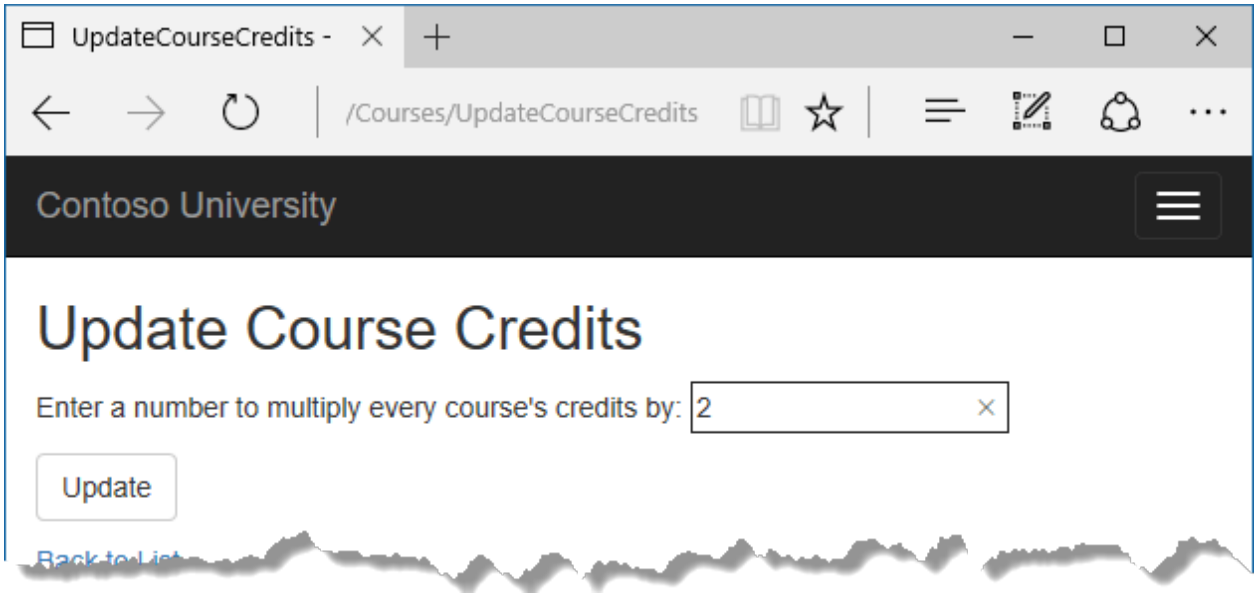
In `Views/Courses/UpdateCourseCredits.cshtml`, replace the template code with the following code:

CSHTML

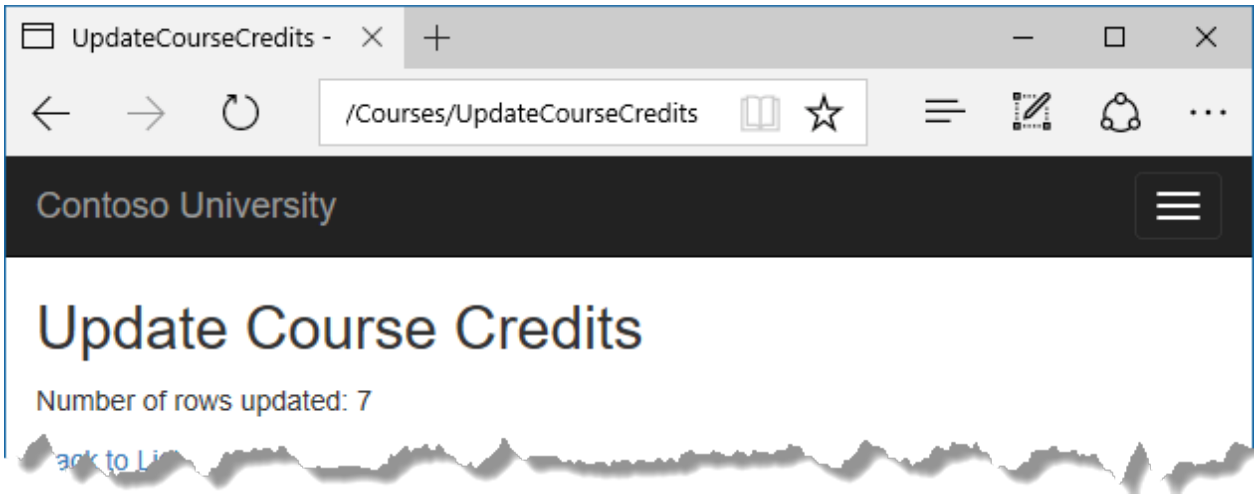
```
@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
```

```
            <p>
                Enter a number to multiply every course's credits by:
@Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default"
/>
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Courses/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see Raw SQL Queries.

Examine SQL queries

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open `StudentsController.cs` and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the app in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed
DbCommand (56ms) [Parameters=[@__id_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].
[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
```

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed
DbCommand (122ms) [Parameters=[@__id_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID],
[s.Enrollments].[Grade], [s.Enrollments].[StudentID], [e.Course].[CourseID],
[e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] =
[e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`) from the Person table. The `SingleOrDefaultAsync` method doesn't resolve to 1 row on the server. Here's why:

- If the query would return multiple rows, the method returns null.
- To determine whether the query would return multiple rows, EF has to check if it returns at least 2.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Create an abstraction layer

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns isn't always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series.

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see Test with InMemory.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- DbContext.SaveChanges
- DbContext.Entry
- ChangeTracker.Entries

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
C#

_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

EF Core source code and development plans

The Entity Framework Core source is at https://github.com/dotnet/efcore. The EF Core repository contains nightly builds, issue tracking, feature specs, design meeting notes, and the roadmap for future development. You can file or find bugs, and contribute.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the scaffold-dbcontext command. See the getting-started tutorial.

Use dynamic LINQ to simplify code

The third tutorial in this series shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

C#

```
 public async Task<IActionResult> Index(
     string sortOrder,
     string currentFilter,
     string searchString,
     int? pageNumber)
 {
     ViewData["CurrentSort"] = sortOrder;
     ViewData["NameSortParm"] =
         String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
     ViewData["DateSortParm"] =
         sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" :
"EnrollmentDate";

     if (searchString != null)
     {
         pageNumber = 1;
     }
     else
     {
         searchString = currentFilter;
     }

     ViewData["CurrentFilter"] = searchString;

     var students = from s in _context.Students
                    select s;

     if (!String.IsNullOrEmpty(searchString))
     {
         students = students.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
     }

     if (string.IsNullOrEmpty(sortOrder))
     {
         sortOrder = "LastName";
     }
```

```
      bool descending = false;
      if (sortOrder.EndsWith("_desc"))
      {
          sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
          descending = true;
      }

      if (descending)
      {
          students = students.OrderByDescending(e => EF.Property<object>(e,
sortOrder));
      }
      else
      {
          students = students.OrderBy(e => EF.Property<object>(e,
sortOrder));
      }

      int pageSize = 3;
      return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
          pageNumber ?? 1, pageSize));
  }
```

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials. John Parente and Paul Goldman worked on updating the tutorial for ASP.NET Core 2.2.

Troubleshoot common errors

ContosoUniversity.dll used by another process

Error message:

> Cannot open '...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in `appsettings.json`. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the `database drop` CLI command:

.NET CLI

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

> A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify

> that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

Get the code

Download or view the completed application.

Additional resources

For more information about EF Core, see the Entity Framework Core documentation. A book is also available: Entity Framework Core in Action.

For information on how to deploy a web app, see Host and deploy ASP.NET Core.

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see Overview of ASP.NET Core.

Next steps

In this tutorial, you:

- ✓ Performed raw SQL queries
- ✓ Called a query to return entities
- ✓ Called a query to return other types
- ✓ Called an update query
- ✓ Examined SQL queries
- ✓ Created an abstraction layer
- ✓ Learned about Automatic change detection
- ✓ Learned about EF Core source code and development plans
- ✓ Learned how to use dynamic LINQ to simplify code

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET Core MVC application. This series worked with a new database; an alternative is to reverse engineer a model from an existing database.

Tutorial: EF Core with MVC, existing database

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core fundamentals overview

Article • 11/14/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article provides an overview of the fundamentals for building ASP.NET Core apps, including dependency injection (DI), configuration, middleware, and more.

For Blazor fundamentals guidance, which adds to or supersedes the guidance in this node, see ASP.NET Core Blazor fundamentals.

Program.cs

ASP.NET Core apps created with the web templates contain the application startup code in the `Program.cs` file. The `Program.cs` file is where:

- Services required by the app are configured.
- The app's request handling pipeline is defined as a series of middleware components.

The following app startup code supports:

- Razor Pages
- MVC controllers with views
- Web API with controllers
- Minimal web APIs

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
```

```
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

Dependency injection (services)

ASP.NET Core includes dependency injection (DI) that makes configured services available throughout an app. Services are added to the DI container with WebApplicationBuilder.Services, `builder.Services` in the preceding code. When the WebApplicationBuilder is instantiated, many framework-provided services are added. `builder` is a `WebApplicationBuilder` in the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

In the preceding highlighted code, `builder` has configuration, logging, and many other services added to the DI container.

The following code adds Razor Pages, MVC controllers with views, and a custom DbContext to the DI container:

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
```

```
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RPMovieConte
xt")));

var app = builder.Build();
```

Services are typically resolved from DI using constructor injection. The DI framework provides an instance of this service at runtime.

The following code uses constructor injection to resolve the database context and logger from DI:

C#

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(RazorPagesMovieContext context, ILogger<IndexModel>
logger)
    {
        _context = context;
        _logger = logger;
    }

    public IList<Movie> Movie { get;set; }

    public async Task OnGetAsync()
    {
        _logger.LogInformation("IndexModel OnGetAsync.");
        Movie = await _context.Movie.ToListAsync();
    }
}
```

Middleware

The request handling pipeline is composed as a series of middleware components. Each component performs operations on an HttpContext and either invokes the next middleware in the pipeline or terminates the request.

By convention, a middleware component is added to the pipeline by invoking a `Use{Feature}` extension method. Middleware added to the app is highlighted in the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

For more information, see ASP.NET Core Middleware.

Host

On startup, an ASP.NET Core app builds a *host*. The host encapsulates all of the app's resources, such as:

- An HTTP server implementation
- Middleware components
- Logging
- Dependency injection (DI) services
- Configuration

There are three different hosts capable of running an ASP.NET Core app:

- ASP.NET Core WebApplication, also known as the Minimal Host
- .NET Generic Host combined with ASP.NET Core's ConfigureWebHostDefaults
- ASP.NET Core WebHost

The ASP.NET Core WebApplication and WebApplicationBuilder types are recommended and used in all the ASP.NET Core templates. `WebApplication` behaves similarly to the .NET Generic Host and exposes many of the same interfaces but requires less callbacks to configure. The ASP.NET Core WebHost is available only for backward compatibility.

The following example instantiates a `WebApplication`:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

The WebApplicationBuilder.Build method configures a host with a set of default options, such as:

- Use Kestrel as the web server and enable IIS integration.
- Load configuration from `appsettings.json`, environment variables, command line arguments, and other configuration sources.
- Send logging output to the console and debug providers.

Non-web scenarios

The Generic Host allows other types of apps to use cross-cutting framework extensions, such as logging, dependency injection (DI), configuration, and app lifetime management. For more information, see .NET Generic Host in ASP.NET Core and Background tasks with hosted services in ASP.NET Core.

Servers

An ASP.NET Core app uses an HTTP server implementation to listen for HTTP requests. The server surfaces requests to the app as a set of request features composed into an `HttpContext`.

Windows

ASP.NET Core provides the following server implementations:

- *Kestrel* is a cross-platform web server. Kestrel is often run in a reverse proxy configuration using IIS. In ASP.NET Core 2.0 or later, Kestrel can be run as a public-facing edge server exposed directly to the Internet.
- *IIS HTTP Server* is a server for Windows that uses IIS. With this server, the ASP.NET Core app and IIS run in the same process.
- *HTTP.sys* is a server for Windows that isn't used with IIS.

For more information, see Web server implementations in ASP.NET Core.

Configuration

ASP.NET Core provides a configuration framework that gets settings as name-value pairs from an ordered set of configuration providers. Built-in configuration providers are available for a variety of sources, such as `.json` files, `.xml` files, environment variables, and command-line arguments. Write custom configuration providers to support other sources.

By default, ASP.NET Core apps are configured to read from `appsettings.json`, environment variables, the command line, and more. When the app's configuration is loaded, values from environment variables override values from `appsettings.json`.

For managing confidential configuration data such as passwords, .NET Core provides the Secret Manager. For production secrets, we recommend Azure Key Vault.

For more information, see Configuration in ASP.NET Core.

Environments

Execution environments, such as `Development`, `Staging`, and `Production`, are available in ASP.NET Core. Specify the environment an app is running in by setting the `ASPNETCORE_ENVIRONMENT` environment variable. ASP.NET Core reads that environment variable at app startup and stores the value in an `IWebHostEnvironment` implementation. This implementation is available anywhere in an app via dependency injection (DI).

The following example configures the exception handler and HTTP Strict Transport Security Protocol (HSTS) middleware when ***not*** running in the `Development` environment:

C#

```
var builder = WebApplication.CreateBuilder(args);
```

```
// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

For more information, see Use multiple environments in ASP.NET Core.

Logging

ASP.NET Core supports a logging API that works with a variety of built-in and third-party logging providers. Available providers include:

- Console
- Debug
- Event Tracing on Windows
- Windows Event Log
- TraceSource
- Azure App Service
- Azure Application Insights

To create logs, resolve an ILogger<TCategoryName> service from dependency injection (DI) and call logging methods such as LogInformation. For example:

C#

```
public class IndexModel : PageModel
{
```

```
    private readonly RazorPagesMovieContext _context;
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(RazorPagesMovieContext context, ILogger<IndexModel>
logger)
    {
        _context = context;
        _logger = logger;
    }

    public IList<Movie> Movie { get;set; }

    public async Task OnGetAsync()
    {
        _logger.LogInformation("IndexModel OnGetAsync.");
        Movie = await _context.Movie.ToListAsync();
    }
}
```

For more information, see Logging in .NET Core and ASP.NET Core.

Routing

A *route* is a URL pattern that is mapped to a handler. The handler is typically a Razor page, an action method in an MVC controller, or a middleware. ASP.NET Core routing gives you control over the URLs used by your app.

The following code, generated by the ASP.NET Core web application template, calls UseRouting:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

```
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

For more information, see Routing in ASP.NET Core.

Error handling

ASP.NET Core has built-in features for handling errors, such as:

- A developer exception page
- Custom error pages
- Static status code pages
- Startup exception handling

For more information, see Handle errors in ASP.NET Core.

Make HTTP requests

An implementation of `IHttpClientFactory` is available for creating `HttpClient` instances. The factory:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, register and configure a *github* client for accessing GitHub. Register and configure a default client for other purposes.
- Supports registration and chaining of multiple delegating handlers to build an outgoing request middleware pipeline. This pattern is similar to ASP.NET Core's inbound middleware pipeline. The pattern provides a mechanism to manage cross-cutting concerns for HTTP requests, including caching, error handling, serialization, and logging.
- Integrates with *Polly*, a popular third-party library for transient fault handling.
- Manages the pooling and lifetime of underlying `HttpClientHandler` instances to avoid common DNS problems that occur when managing `HttpClient` lifetimes manually.
- Adds a configurable logging experience via ILogger for all requests sent through clients created by the factory.

For more information, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

Content root

The content root is the base path for:

- The executable hosting the app (*.exe*).
- Compiled assemblies that make up the app (*.dll*).
- Content files used by the app, such as:
  - Razor files (`.cshtml`, `.razor`)
  - Configuration files (`.json`, `.xml`)
  - Data files (`.db`)
- The Web root, typically the *wwwroot* folder.

During development, the content root defaults to the project's root directory. This directory is also the base path for both the app's content files and the Web root. Specify a different content root by setting its path when building the host. For more information, see Content root.

Web root

The web root is the base path for public, static resource files, such as:

- Stylesheets (`.css`)
- JavaScript (`.js`)
- Images (`.png`, `.jpg`)

By default, static files are served only from the web root directory and its sub-directories. The web root path defaults to *{content root}/wwwroot*. Specify a different web root by setting its path when building the host. For more information, see Web root.

Prevent publishing files in *wwwroot* with the <Content> project item in the project file. The following example prevents publishing content in *wwwroot/local* and its sub-directories:

XML

```
<ItemGroup>
  <Content Update="wwwroot\local\**\*.*" CopyToPublishDirectory="Never" />
</ItemGroup>
```

In Razor `.cshtml` files, `~/` points to the web root. A path beginning with `~/` is referred to as a *virtual path*.

For more information, see Static files in ASP.NET Core.

Additional resources

- WebApplicationBuilder source code

App startup in ASP.NET Core

Article • 12/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Rick Anderson

ASP.NET Core apps created with the web templates contain the application startup code in the `Program.cs` file.

For Blazor startup guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor startup.

The following app startup code supports several app types:

- Blazor Web Apps
- Razor Pages
- MVC controllers with views
- Web API with controllers
- Minimal APIs

C#

```
using WebAll.Components;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
```

```
app.UseStaticFiles();

app.UseAuthorization();

app.MapGet("/hi", () => "Hello!");

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.UseAntiforgery();

app.Run();
```

Apps that use EventSource can measure the startup time to understand and optimize startup performance. The ServerReady event in Microsoft.AspNetCore.Hosting represents the point where the server is ready to respond to requests.

Extend Startup with startup filters

Use IStartupFilter:

- To configure middleware at the beginning or end of an app's middleware pipeline without an explicit call to `Use{Middleware}`. Use `IStartupFilter` to add defaults to the beginning of the pipeline without explicitly registering the default middleware. `IStartupFilter` allows a different component to call `Use{Middleware}` on behalf of the app author.
- To create a pipeline of `Configure` methods. IStartupFilter.Configure can set a middleware to run before or after middleware added by libraries.

An `IStartupFilter` implementation implements Configure, which receives and returns an `Action<IApplicationBuilder>`. An IApplicationBuilder defines a class to configure an app's request pipeline. For more information, see Create a middleware pipeline with IApplicationBuilder.

Each `IStartupFilter` implementation can add one or more middlewares in the request pipeline. The filters are invoked in the order they were added to the service container. Filters can add middleware before or after passing control to the next filter, thus they append to the beginning or end of the app pipeline.

The following example demonstrates how to register a middleware with `IStartupFilter`. The `RequestSetOptionsMiddleware` middleware sets an options value from a query string parameter:

C#

```
public class RequestSetOptionsMiddleware
{
    private readonly RequestDelegate _next;

    public RequestSetOptionsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    // Test with https://localhost:5001/Privacy/?option=Hello
    public async Task Invoke(HttpContext httpContext)
    {
        var option = httpContext.Request.Query["option"];

        if (!string.IsNullOrWhiteSpace(option))
        {
            httpContext.Items["option"] = WebUtility.HtmlEncode(option);
        }

        await _next(httpContext);
    }
}
```

The `RequestSetOptionsMiddleware` is configured in the `RequestSetOptionsStartupFilter` class:

C#

```
public class RequestSetOptionsStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder>
next)
    {
        return builder =>
        {
            builder.UseMiddleware<RequestSetOptionsMiddleware>();
            next(builder);
        };
    }
}
```

The `IStartupFilter` implementation is registered in `Program.cs`:

C#

```
using WebStartup.Middleware;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddRazorPages();
builder.Services.AddTransient<IStartupFilter,
                        RequestSetOptionsStartupFilter>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

When a query string parameter for `option` is provided, the middleware processes the value assignment before the ASP.NET Core middleware renders the response:

CSHTML

```
@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<p> Append query string ?option=hello</p>
Option String: @HttpContext.Items["option"];
```

Middleware execution order is set by the order of `IStartupFilter` registrations:

- Multiple `IStartupFilter` implementations might interact with the same objects. If ordering is important, order their `IStartupFilter` service registrations to match the order that their middlewares should run.
- Libraries can add middleware with one or more `IStartupFilter` implementations that run before or after other app middleware registered with `IStartupFilter`. To invoke an `IStartupFilter` middleware before a middleware added by a library's `IStartupFilter`:

    - Position the service registration before the library is added to the service container.
    - To invoke afterward, position the service registration after the library is added.

You can't extend the ASP.NET Core app when you override `Configure`. For more information, see this GitHub issue.

Add configuration at startup from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Program.cs` file. For more information, see Use hosting startup assemblies in ASP.NET Core.

Startup, ConfigureServices, and Configure

For information on using the ConfigureServices and Configure methods with the minimal hosting model, see:

- Use Startup with the minimal hosting model
- The ASP.NET Core 5.0 version of this article:
    - The ConfigureServices method
    - The Configure method

Dependency injection in ASP.NET Core

Article • 09/10/2024

By Kirk Larkin, Steve Smith, and Brandon Dahler

ASP.NET Core supports the dependency injection (DI) software design pattern, which is a technique for achieving Inversion of Control (IoC) between classes and their dependencies.

For more information specific to dependency injection within MVC controllers, see Dependency injection into controllers in ASP.NET Core.

For information on using dependency injection in applications other than web apps, see Dependency injection in .NET.

For more information on dependency injection of options, see Options pattern in ASP.NET Core.

This topic provides information on dependency injection in ASP.NET Core. The primary documentation on using dependency injection is contained in Dependency injection in .NET.

View or download sample code (how to download)

Overview of dependency injection

A *dependency* is an object that another object depends on. Examine the following `MyDependency` class with a `WriteMessage` method that other classes depend on:

C#

```
public class MyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage called. Message:
{message}");
    }
}
```

A class can create an instance of the `MyDependency` class to make use of its `WriteMessage` method. In the following example, the `MyDependency` class is a dependency of the `IndexModel` class:

C#

```
public class IndexModel : PageModel
{
    private readonly MyDependency _dependency = new MyDependency();

    public void OnGet()
    {
        _dependency.WriteMessage("IndexModel.OnGet");
    }
}
```

The class creates and directly depends on the `MyDependency` class. Code dependencies, such as in the previous example, are problematic and should be avoided for the following reasons:

- To replace `MyDependency` with a different implementation, the `IndexModel` class must be modified.
- If `MyDependency` has dependencies, they must also be configured by the `IndexModel` class. In a large project with multiple classes depending on `MyDependency`, the configuration code becomes scattered across the app.
- This implementation is difficult to unit test.

Dependency injection addresses these problems through:

- The use of an interface or base class to abstract the dependency implementation.
- Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, IServiceProvider. Services are typically registered in the app's `Program.cs` file.
- *Injection* of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

In the sample app, the `IMyDependency` interface defines the `WriteMessage` method:

C#

```
public interface IMyDependency
{
    void WriteMessage(string message);
}
```

This interface is implemented by a concrete type, `MyDependency`:

C#

```
public class MyDependency : IMyDependency
{
    public void WriteMessage(string message)
    {
        Console.WriteLine($"MyDependency.WriteMessage Message: {message}");
    }
}
```

The sample app registers the `IMyDependency` service with the concrete type `MyDependency`. The AddScoped method registers the service with a scoped lifetime, the lifetime of a single request. Service lifetimes are described later in this topic.

C#

```
using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<IMyDependency, MyDependency>();

var app = builder.Build();
```

In the sample app, the `IMyDependency` service is requested and used to call the `WriteMessage` method:

C#

```
public class Index2Model : PageModel
{
    private readonly IMyDependency _myDependency;

    public Index2Model(IMyDependency myDependency)
    {
        _myDependency = myDependency;
    }

    public void OnGet()
    {
        _myDependency.WriteMessage("Index2Model.OnGet");
    }
}
```

By using the DI pattern, the controller or Razor Page:

- Doesn't use the concrete type `MyDependency`, only the `IMyDependency` interface it implements. That makes it easy to change the implementation without modifying the controller or Razor Page.
- Doesn't create an instance of `MyDependency`, it's created by the DI container.

The implementation of the `IMyDependency` interface can be improved by using the built-in logging API:

C#

```
public class MyDependency2 : IMyDependency
{
    private readonly ILogger<MyDependency2> _logger;

    public MyDependency2(ILogger<MyDependency2> logger)
    {
        _logger = logger;
    }

    public void WriteMessage(string message)
    {
        _logger.LogInformation( $"MyDependency2.WriteMessage Message:
{message}");
    }
}
```

The updated `Program.cs` registers the new `IMyDependency` implementation:

C#

```
using DependencyInjectionSample.Interfaces;
using DependencyInjectionSample.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<IMyDependency, MyDependency2>();

var app = builder.Build();
```

`MyDependency2` depends on ILogger<TCategoryName>, which it requests in the constructor. `ILogger<TCategoryName>` is a framework-provided service.

It's not unusual to use dependency injection in a chained fashion. Each requested dependency in turn requests its own dependencies. The container resolves the dependencies in the graph and returns the fully resolved service. The collective set of

dependencies that must be resolved is typically referred to as a *dependency tree*, *dependency graph*, or *object graph*.

The container resolves `ILogger<TCategoryName>` by taking advantage of (generic) open types, eliminating the need to register every (generic) constructed type.

In dependency injection terminology, a service:

- Is typically an object that provides a service to other objects, such as the `IMyDependency` service.
- Is not related to a web service, although the service may use a web service.

The framework provides a robust logging system. The `IMyDependency` implementations shown in the preceding examples were written to demonstrate basic DI, not to implement logging. Most apps shouldn't need to write loggers. The following code demonstrates using the default logging, which doesn't require any services to be registered:

C#

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }

    public string Message { get; set; } = string.Empty;

    public void OnGet()
    {
        Message = $"About page visited at
{DateTime.UtcNow.ToLongTimeString()}";
        _logger.LogInformation(Message);
    }
}
```

Using the preceding code, there is no need to update `Program.cs`, because logging is provided by the framework.

Register groups of services with extension methods

The ASP.NET Core framework uses a convention for registering a group of related services. The convention is to use a single `Add{GROUP_NAME}` extension method to register all of the services required by a framework feature. For example, the AddControllers extension method registers the services required for MVC controllers.

The following code is generated by the Razor Pages template using individual user accounts and shows how to add additional services to the container using the extension methods AddDbContext and AddDefaultIdentity:

C#

```
using DependencyInjectionSample.Data;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();
```

Consider the following which registers services and configures options:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));
builder.Services.Configure<ColorOptions>(
    builder.Configuration.GetSection(ColorOptions.Color));

builder.Services.AddScoped<IMyDependency, MyDependency>();
builder.Services.AddScoped<IMyDependency2, MyDependency2>();
```

```
var app = builder.Build();
```

Related groups of registrations can be moved to an extension method to register services. For example, the configuration services are added to the following class:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.Configuration;

namespace Microsoft.Extensions.DependencyInjection
{
    public static class MyConfigServiceCollectionExtensions
    {
        public static IServiceCollection AddConfig(
             this IServiceCollection services, IConfiguration config)
        {
            services.Configure<PositionOptions>(
                config.GetSection(PositionOptions.Position));
            services.Configure<ColorOptions>(
                config.GetSection(ColorOptions.Color));

            return services;
        }

        public static IServiceCollection AddMyDependencyGroup(
             this IServiceCollection services)
        {
            services.AddScoped<IMyDependency, MyDependency>();
            services.AddScoped<IMyDependency2, MyDependency2>();

            return services;
        }
    }
}
```

The remaining services are registered in a similar class. The following code uses the new extension methods to register the services:

C#

```
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddConfig(builder.Configuration)
    .AddMyDependencyGroup();

builder.Services.AddRazorPages();
```

```
var app = builder.Build();
```

Note: Each `services.Add{GROUP_NAME}` extension method adds and potentially configures services. For example, AddControllersWithViews adds the services MVC controllers with views require, and AddRazorPages adds the services Razor Pages requires.

Service lifetimes

See Service lifetimes in Dependency injection in .NET

To use scoped services in middleware, use one of the following approaches:

- Inject the service into the middleware's `Invoke` or `InvokeAsync` method. Using constructor injection throws a runtime exception because it forces the scoped service to behave like a singleton. The sample in the Lifetime and registration options section demonstrates the `InvokeAsync` approach.
- Use Factory-based middleware. Middleware registered using this approach is activated per client request (connection), which allows scoped services to be injected into the middleware's constructor.

For more information, see Write custom ASP.NET Core middleware.

Service registration methods

See Service registration methods in Dependency injection in .NET

It's common to use multiple implementations when mocking types for testing.

Registering a service with only an implementation type is equivalent to registering that service with the same implementation and service type. This is why multiple implementations of a service cannot be registered using the methods that don't take an explicit service type. These methods can register multiple *instances* of a service, but they will all have the same *implementation* type.

Any of the above service registration methods can be used to register multiple service instances of the same service type. In the following example, `AddSingleton` is called twice with `IMyDependency` as the service type. The second call to `AddSingleton` overrides the previous one when resolved as `IMyDependency` and adds to the previous one when multiple services are resolved via `IEnumerable<IMyDependency>`. Services appear in the order they were registered when resolved via `IEnumerable<{SERVICE}>`.

```
```

C#

```
services.AddSingleton<IMyDependency, MyDependency>();
services.AddSingleton<IMyDependency, DifferentDependency>();

public class MyService
{
    public MyService(IMyDependency myDependency,
       IEnumerable<IMyDependency> myDependencies)
    {
        Trace.Assert(myDependency is DifferentDependency);

        var dependencyArray = myDependencies.ToArray();
        Trace.Assert(dependencyArray[0] is MyDependency);
        Trace.Assert(dependencyArray[1] is DifferentDependency);
    }
}
```

Keyed services

Keyed services refers to a mechanism for registering and retrieving Dependency Injection (DI) services using keys. A service is associated with a key by calling AddKeyedSingleton (or `AddKeyedScoped` or `AddKeyedTransient`) to register it. Access a registered service by specifying the key with the [FromKeyedServices] attribute. The following code shows how to use keyed services:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.SignalR;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
builder.Services.AddControllers();

var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));
app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>

smallCache.Get("date"));

app.MapControllers();

app.Run();

public interface ICache
```

```
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

[ApiController]
[Route("/cache")]
public class CustomServicesApiController : Controller
{
    [HttpGet("big-cache")]
    public ActionResult<object> GetOk([FromKeyedServices("big")] ICache
cache)
    {
        return cache.Get("data-mvc");
    }
}

public class MyHub : Hub
{
    public void Method([FromKeyedServices("small")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }
}
```

Constructor injection behavior

See Constructor injection behavior in Dependency injection in .NET

Entity Framework contexts

By default, Entity Framework contexts are added to the service container using the scoped lifetime because web app database operations are normally scoped to the client request. To use a different lifetime, specify the lifetime by using an AddDbContext overload. Services of a given lifetime shouldn't use a database context with a lifetime that's shorter than the service's lifetime.

Lifetime and registration options

To demonstrate the difference between service lifetimes and their registration options, consider the following interfaces that represent a task as an operation with an identifier, `OperationId`. Depending on how the lifetime of an operation's service is configured for the following interfaces, the container provides either the same or different instances of the service when requested by a class:

C#

```
public interface IOperation
{
    string OperationId { get; }
}

public interface IOperationTransient : IOperation { }
public interface IOperationScoped : IOperation { }
public interface IOperationSingleton : IOperation { }
```

The following `Operation` class implements all of the preceding interfaces. The `Operation` constructor generates a GUID and stores the last 4 characters in the `OperationId` property:

C#

```
public class Operation : IOperationTransient, IOperationScoped,
IOperationSingleton
{
    public Operation()
    {
        OperationId = Guid.NewGuid().ToString()[^4..];
    }

    public string OperationId { get; }
}
```

The following code creates multiple registrations of the `Operation` class according to the named lifetimes:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddTransient<IOperationTransient, Operation>();
builder.Services.AddScoped<IOperationScoped, Operation>();
builder.Services.AddSingleton<IOperationSingleton, Operation>();

var app = builder.Build();
```

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseMyMiddleware();
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The sample app demonstrates object lifetimes both within and between requests. The `IndexModel` and the middleware request each kind of `IOperation` type and log the `OperationId` for each:

C#

```
public class IndexModel : PageModel
{
    private readonly ILogger _logger;
    private readonly IOperationTransient _transientOperation;
    private readonly IOperationSingleton _singletonOperation;
    private readonly IOperationScoped _scopedOperation;

    public IndexModel(ILogger<IndexModel> logger,
                      IOperationTransient transientOperation,
                      IOperationScoped scopedOperation,
                      IOperationSingleton singletonOperation)
    {
        _logger = logger;
        _transientOperation = transientOperation;
        _scopedOperation    = scopedOperation;
        _singletonOperation = singletonOperation;
    }

    public void  OnGet()
    {
        _logger.LogInformation("Transient: " +
_transientOperation.OperationId);
        _logger.LogInformation("Scoped: "    +
_scopedOperation.OperationId);
        _logger.LogInformation("Singleton: " +
_singletonOperation.OperationId);
```

```
    }
}
```

Similar to the `IndexModel`, the middleware resolves the same services:

C#

```
public class MyMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger _logger;

    private readonly IOperationSingleton _singletonOperation;

    public MyMiddleware(RequestDelegate next, ILogger<MyMiddleware> logger,
        IOperationSingleton singletonOperation)
    {
        _logger = logger;
        _singletonOperation = singletonOperation;
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context,
        IOperationTransient transientOperation, IOperationScoped
scopedOperation)
    {
        _logger.LogInformation("Transient: " +
transientOperation.OperationId);
        _logger.LogInformation("Scoped: " + scopedOperation.OperationId);
        _logger.LogInformation("Singleton: " +
_singletonOperation.OperationId);

        await _next(context);
    }
}

public static class MyMiddlewareExtensions
{
    public static IApplicationBuilder UseMyMiddleware(this
IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyMiddleware>();
    }
}
```

Scoped and transient services must be resolved in the `InvokeAsync` method:

C#

```
public async Task InvokeAsync(HttpContext context,
    IOperationTransient transientOperation, IOperationScoped
scopedOperation)
{
    _logger.LogInformation("Transient: " + transientOperation.OperationId);
    _logger.LogInformation("Scoped: " + scopedOperation.OperationId);
    _logger.LogInformation("Singleton: " + _singletonOperation.OperationId);

    await _next(context);
}
```

The logger output shows:

- *Transient* objects are always different. The transient `OperationId` value is different in the `IndexModel` and in the middleware.
- *Scoped* objects are the same for a given request but differ across each new request.
- *Singleton* objects are the same for every request.

To reduce the logging output, set "Logging:LogLevel:Microsoft:Error" in the `appsettings.Development.json` file:

JSON

```
{
  "MyKey": "MyKey from appsettings.Developement.json",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "System": "Debug",
      "Microsoft": "Error"
    }
  }
}
```

Resolve a service at app start up

The following code shows how to resolve a scoped service for a limited duration when the app starts:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMyDependency, MyDependency>();

var app = builder.Build();
```

```
using (var serviceScope = app.Services.CreateScope())
{
    var services = serviceScope.ServiceProvider;

    var myDependency = services.GetRequiredService<IMyDependency>();
    myDependency.WriteMessage("Call services from main");
}

app.MapGet("/", () => "Hello World!");

app.Run();
```

Scope validation

See Constructor injection behavior in Dependency injection in .NET

For more information, see Scope validation.

Request Services

Services and their dependencies within an ASP.NET Core request are exposed through HttpContext.RequestServices.

The framework creates a scope per request, and `RequestServices` exposes the scoped service provider. All scoped services are valid for as long as the request is active.

> **Note**
>
> Prefer requesting dependencies as constructor parameters over resolving services from `RequestServices`. Requesting dependencies as constructor parameters yields classes that are easier to test.

Design services for dependency injection

When designing services for dependency injection:

- Avoid stateful, static classes and members. Avoid creating global state by designing apps to use singleton services instead.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make services small, well-factored, and easily tested.

If a class has a lot of injected dependencies, it might be a sign that the class has too many responsibilities and violates the Single Responsibility Principle (SRP). Attempt to refactor the class by moving some of its responsibilities into new classes. Keep in mind that Razor Pages page model classes and MVC controller classes should focus on UI concerns.

Disposal of services

The container calls Dispose for the IDisposable types it creates. Services resolved from the container should never be disposed by the developer. If a type or factory is registered as a singleton, the container disposes the singleton automatically.

In the following example, the services are created by the service container and disposed automatically: dependency-injection\samples\6.x\DIsample2\DIsample2\Services\Service1.cs

C#

```
public class Service1 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
        Console.WriteLine($"Service1: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service1.Dispose");
        _disposed = true;
    }
}

public class Service2 : IDisposable
{
    private bool _disposed;

    public void Write(string message)
    {
        Console.WriteLine($"Service2: {message}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;
```

```
        Console.WriteLine("Service2.Dispose");
        _disposed = true;
    }
}

public interface IService3
{
    public void Write(string message);
}

public class Service3 : IService3, IDisposable
{
    private bool _disposed;

    public Service3(string myKey)
    {
        MyKey = myKey;
    }

    public string MyKey { get; }

    public void Write(string message)
    {
        Console.WriteLine($"Service3: {message}, MyKey = {MyKey}");
    }

    public void Dispose()
    {
        if (_disposed)
            return;

        Console.WriteLine("Service3.Dispose");
        _disposed = true;
    }
}
```

C#

```
using DIsample2.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddScoped<Service1>();
builder.Services.AddSingleton<Service2>();

var myKey = builder.Configuration["MyKey"];
builder.Services.AddSingleton<IService3>(sp => new Service3(myKey));

var app = builder.Build();
```

C#

```
public class IndexModel : PageModel
{
    private readonly Service1 _service1;
    private readonly Service2 _service2;
    private readonly IService3 _service3;

    public IndexModel(Service1 service1, Service2 service2, IService3
service3)
    {
        _service1 = service1;
        _service2 = service2;
        _service3 = service3;
    }

    public void OnGet()
    {
        _service1.Write("IndexModel.OnGet");
        _service2.Write("IndexModel.OnGet");
        _service3.Write("IndexModel.OnGet");
    }
}
```

The debug console shows the following output after each refresh of the Index page:

Console

```
Service1: IndexModel.OnGet
Service2: IndexModel.OnGet
Service3: IndexModel.OnGet, MyKey = MyKey from appsettings.Developement.json
Service1.Dispose
```

Services not created by the service container

Consider the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddSingleton(new Service1());
builder.Services.AddSingleton(new Service2());
```

In the preceding code:

- The service instances aren't created by the service container.
- The framework doesn't dispose of the services automatically.
- The developer is responsible for disposing the services.

IDisposable guidance for Transient and shared instances

See IDisposable guidance for Transient and shared instance in Dependency injection in .NET

Default service container replacement

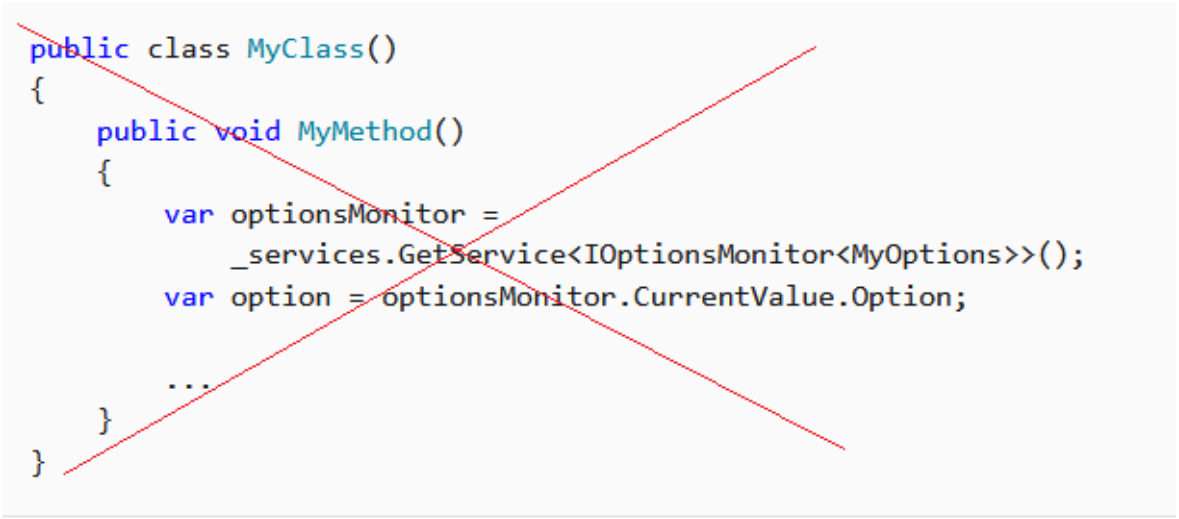
See Default service container replacement in Dependency injection in .NET

Recommendations

See Recommendations in Dependency injection in .NET

- Avoid using the *service locator pattern*. For example, don't invoke GetService to obtain a service instance when you can use DI instead:

  Incorrect:



  Correct:

```C#
public class MyClass
{
    private readonly IOptionsMonitor<MyOptions> _optionsMonitor;

    public MyClass(IOptionsMonitor<MyOptions> optionsMonitor)
    {
```

```
            _optionsMonitor = optionsMonitor;
        }

        public void MyMethod()
        {
            var option = _optionsMonitor.CurrentValue.Option;

            ...
        }
    }
```

- Another service locator variation to avoid is injecting a factory that resolves dependencies at runtime. Both of these practices mix Inversion of Control strategies.
- Avoid static access to `HttpContext` (for example, IHttpContextAccessor.HttpContext).

DI is an *alternative* to static/global object access patterns. You may not be able to realize the benefits of DI if you mix it with static object access.

Recommended patterns for multi-tenancy in DI

Orchard Core is an application framework for building modular, multi-tenant applications on ASP.NET Core. For more information, see the Orchard Core Documentation.

See the Orchard Core samples for examples of how to build modular and multi-tenant apps using just the Orchard Core Framework without any of its CMS-specific features.

Framework-provided services

`Program.cs` registers services that the app uses, including platform features, such as Entity Framework Core and ASP.NET Core MVC. Initially, the `IServiceCollection` provided to `Program.cs` has services defined by the framework depending on how the host was configured. For apps based on the ASP.NET Core templates, the framework registers more than 250 services.

The following table lists a small sample of these framework-registered services:

Expand table

Service Type	Lifetime
Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory	Transient
IHostApplicationLifetime	Singleton
IWebHostEnvironment	Singleton
Microsoft.AspNetCore.Hosting.IStartup	Singleton
Microsoft.AspNetCore.Hosting.IStartupFilter	Transient
Microsoft.AspNetCore.Hosting.Server.IServer	Singleton
Microsoft.AspNetCore.Http.IHttpContextFactory	Transient
Microsoft.Extensions.Logging.ILogger<TCategoryName>	Singleton
Microsoft.Extensions.Logging.ILoggerFactory	Singleton
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	Singleton
Microsoft.Extensions.Options.IConfigureOptions<TOptions>	Transient
Microsoft.Extensions.Options.IOptions<TOptions>	Singleton
System.Diagnostics.DiagnosticSource	Singleton
System.Diagnostics.DiagnosticListener	Singleton

Additional resources

- Dependency injection into views in ASP.NET Core
- Dependency injection into controllers in ASP.NET Core
- Dependency injection in requirement handlers in ASP.NET Core
- ASP.NET Core Blazor dependency injection
- NDC Conference Patterns for DI app development
- App startup in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Understand dependency injection basics in .NET
- Dependency injection guidelines
- Tutorial: Use dependency injection in .NET
- .NET dependency injection
- ASP.NET CORE DEPENDENCY INJECTION: WHAT IS THE ISERVICECOLLECTION?
- Four ways to dispose IDisposables in ASP.NET Core
- Writing Clean Code in ASP.NET Core with Dependency Injection (MSDN)
- Explicit Dependencies Principle

- Inversion of Control Containers and the Dependency Injection Pattern (Martin Fowler)
- How to register a service with multiple interfaces in ASP.NET Core DI

ASP.NET Core support for Native AOT

Article • 08/14/2024

By Mitch Denny

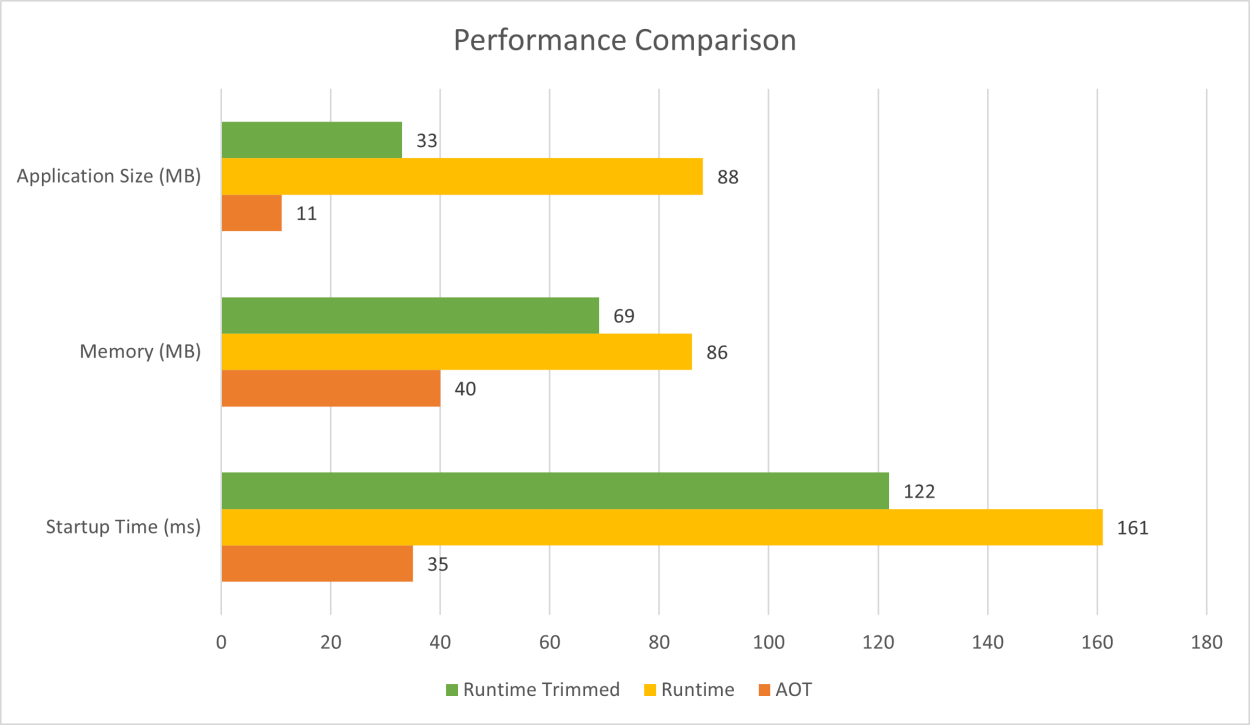
ASP.NET Core 8.0 introduces support for .NET native ahead-of-time (AOT).

Why use Native AOT with ASP.NET Core

Publishing and deploying a Native AOT app provides the following benefits:

- **Minimized disk footprint**: When publishing using Native AOT, a single executable is produced containing just the code from external dependencies that is needed to support the program. Reduced executable size can lead to:
  - Smaller container images, for example in containerized deployment scenarios.
  - Reduced deployment time from smaller images.
- **Reduced startup time**: Native AOT applications can show reduced start-up times, which means
  - The app is ready to service requests quicker.
  - Improved deployment where container orchestrators need to manage transition from one version of the app to another.
- **Reduced memory demand**: Native AOT apps can have reduced memory demands, depending on the work done by the app. Reduced memory consumption can lead to greater deployment density and improved scalability.

The template app was run in our benchmarking lab to compare performance of an AOT published app, a trimmed runtime app, and an untrimmed runtime app. The following chart shows the results of the benchmarking:



The preceding chart shows that Native AOT has lower app size, memory usage, and startup time.

ASP.NET Core and Native AOT compatibility

Not all features in ASP.NET Core are currently compatible with Native AOT. The following table summarizes ASP.NET Core feature compatibility with Native AOT:

Expand table

Feature	Fully Supported	Partially Supported	Not Supported
gRPC	✔		
Minimal APIs		✔	
MVC			✘
Blazor Server			✘
SignalR			✘
JWT Authentication	✔		
Other Authentication			✘
CORS	✔		
HealthChecks	✔		

Feature	Fully Supported	Partially Supported	Not Supported
HttpLogging	✔		
Localization	✔		
OutputCaching	✔		
RateLimiting	✔		
RequestDecompression	✔		
ResponseCaching	✔		
ResponseCompression	✔		
Rewrite	✔		
Session			✘
Spa			✘
StaticFiles	✔		
WebSockets	✔		

For more information on limitations, see:

- Limitations of Native AOT deployment
- Introduction to AOT warnings
- Known trimming incompatibilities
- Introduction to trim warnings
- GitHub issue dotnet/core #8288

It's important to test an app thoroughly when moving to a Native AOT deployment model. The AOT deployed app should be tested to verify functionality hasn't changed from the untrimmed and JIT-compiled app. When building the app, review and correct AOT warnings. An app that issues AOT warnings during publishing may not work correctly. If no AOT warnings are issued at publish time, the published AOT app should work the same as the untrimmed and JIT-compiled app.

Native AOT publishing

Native AOT is enabled with the `PublishAot` MSBuild property. The following example shows how to enable Native AOT in a project file:

```
XML
```

```
<PropertyGroup>
  <PublishAot>true</PublishAot>
</PropertyGroup>
```

This setting enables Native AOT compilation during publish and enables dynamic code usage analysis during build and editing. A project that uses Native AOT publishing uses JIT compilation when running locally. An AOT app has the following differences from a JIT-compiled app:

- Features that aren't compatible with Native AOT are disabled and throw exceptions at run time.
- A source analyzer is enabled to highlight code that isn't compatible with Native AOT. At publish time, the entire app, including NuGet packages, are analyzed for compatibility again.

Native AOT analysis includes all of the app's code and the libraries the app depends on. Review Native AOT warnings and take corrective steps. It's a good idea to publish apps frequently to discover issues early in the development lifecycle.

In .NET 8, Native AOT is supported by the following ASP.NET Core app types:

- minimal APIs - For more information, see the The Web API (Native AOT) template section later in this article.
- gRPC - For more information, see gRPC and Native AOT.
- Worker services - For more information, see AOT in Worker Service templates.

The Web API (Native AOT) template

The **ASP.NET Core Web API (Native AOT)** template (short name `webapiaot`) creates a project with AOT enabled. The template differs from the **Web API** project template in the following ways:

- Uses minimal APIs only, as MVC isn't yet compatible with Native AOT.
- Uses the CreateSlimBuilder() API to ensure only the essential features are enabled by default, minimizing the app's deployed size.
- Is configured to listen on HTTP only, as HTTPS traffic is commonly handled by an ingress service in cloud-native deployments.
- Doesn't include a launch profile for running under IIS or IIS Express.
- Creates an .http file configured with sample HTTP requests that can be sent to the app's endpoints.
- Includes a sample `Todo` API instead of the weather forecast sample.
- Adds `PublishAot` to the project file, as shown earlier in this article.

- Enables the JSON serializer source generators. The source generator is used to generate serialization code at build time, which is required for Native AOT compilation.

Changes to support source generation

The following example shows the code added to the `Program.cs` file to support JSON serialization source generation:

diff

```
using MyFirstAotWebApi;
+using System.Text.Json.Serialization;

-var builder = WebApplication.CreateBuilder();
+var builder = WebApplication.CreateSlimBuilder(args);

+builder.Services.ConfigureHttpJsonOptions(options =>
+{
+  options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
+});

var app = builder.Build();

var sampleTodos = TodoGenerator.GenerateTodos().ToArray();

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());

app.Run();

+[JsonSerializable(typeof(Todo[]))]
+internal partial class AppJsonSerializerContext : JsonSerializerContext
+{
+
+}
```

Without this added code, `System.Text.Json` uses reflection to serialize and deserialize JSON. Reflection isn't supported in Native AOT.

For more information, see:

- Combine source generators
- TypeInfoResolverChain

Changes to `launchSettings.json`

The `launchSettings.json` file created by the **Web API (Native AOT)** template has the `iisSettings` section and `IIS Express` profile removed:

diff

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
-  "iisSettings": {
-    "windowsAuthentication": false,
-    "anonymousAuthentication": true,
-    "iisExpress": {
-      "applicationUrl": "http://localhost:11152",
-      "sslPort": 0
-    }
-  },
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "todos",
      "applicationUrl": "http://localhost:5102",
        "environmentVariables": {
          "ASPNETCORE_ENVIRONMENT": "Development"
        }
      },
-     "IIS Express": {
-       "commandName": "IISExpress",
-       "launchBrowser": true,
-       "launchUrl": "todos",
-      "environmentVariables": {
-       "ASPNETCORE_ENVIRONMENT": "Development"
-      }
-    }
  }
}
```

The `CreateSlimBuilder` method

The template uses the CreateSlimBuilder() method instead of the CreateBuilder() method.

C#

```
using System.Text.Json.Serialization;
using MyFirstAotWebApi;

var builder = WebApplication.CreateSlimBuilder(args);
```

```
builder.Logging.AddConsole();

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

var sampleTodos = TodoGenerator.GenerateTodos().ToArray();

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());

app.Run();

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

The `CreateSlimBuilder` method initializes the WebApplicationBuilder with the minimum ASP.NET Core features necessary to run an app.

As noted earlier, the `CreateSlimBuilder` method doesn't include support for HTTPS or HTTP/3. These protocols typically aren't required for apps that run behind a TLS termination proxy. For example, see TLS termination and end to end TLS with Application Gateway. HTTPS can be enabled by calling builder.WebHost.UseKestrelHttpsConfiguration HTTP/3 can be enabled by calling builder.WebHost.UseQuic.

`CreateSlimBuilder` vs `CreateBuilder`

The `CreateSlimBuilder` method doesn't support the following features that are supported by the `CreateBuilder` method:

- Hosting startup assemblies
- UseStartup
- The following logging providers:
  - Windows EventLog
  - Debug

    - Event Source
- Web hosting features:
    - UseStaticWebAssets
    - IIS Integration
- Kestrel configuration
    - HTTPS endpoints in Kestrel
    - Quic (HTTP/3)
- Regex and alpha constraints used in routing

The `CreateSlimBuilder` method includes the following features needed for an efficient development experience:

- JSON file configuration for `appsettings.json` and `appsettings.{EnvironmentName}.json`.
- User secrets configuration.
- Console logging.
- Logging configuration.

For a builder that omits the preceding features, see The CreateEmptyBuilder method.

Including minimal features has benefits for trimming as well as AOT. For more information, see Trim self-contained deployments and executables.

For more detailed information, see Comparing WebApplication.CreateBuilder to CreateSlimBuilder

Source generators

Because unused code is trimmed during publishing for Native AOT, the app can't use unbounded reflection at runtime. Source generators are used to produce code that avoids the need for reflection. In some cases, source generators produce code optimized for AOT even when a generator isn't required.

To view the source code that is generated, add the EmitCompilerGeneratedFiles property to an app's `.csproj` file, as shown in the following example:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <!-- Other properties omitted for brevity -->
    <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
  </PropertyGroup>
```

```
</Project>
```

Run the `dotnet build` command to see the generated code. The output includes an `obj/Debug/net8.0/generated/` directory that contains all the generated files for the project.

The `dotnet publish` command also compiles the source files and generates files that are compiled. In addition, `dotnet publish` passes the generated assemblies to a native IL compiler. The IL compiler produces the native executable. The native executable contains the native machine code.

Libraries and Native AOT

Many of the popular libraries used in ASP.NET Core projects currently have some compatibility issues when used in a project targeting Native AOT, such as:

- Use of reflection to inspect and discover types.
- Conditionally loading libraries at runtime.
- Generating code on the fly to implement functionality.

Libraries using these dynamic features need to be updated in order to work with Native AOT. They can be updated using tools like Roslyn source generators.

Library authors hoping to support Native AOT are encouraged to:

- Read about Native AOT compatibility requirements.
- Prepare the library for trimming.

Minimal APIs and JSON payloads

The Minimal API framework is optimized for receiving and returning JSON payloads using System.Text.Json. `System.Text.Json`:

- Imposes compatibility requirements for JSON and Native AOT.
- Requires the use of the System.Text.Json source generator.

All types that are transmitted as part of the HTTP body or returned from request delegates in Minimal APIs apps must be configured on a JsonSerializerContext that is registered via ASP.NET Core's dependency injection:

```
C#
```

```
using System.Text.Json.Serialization;
using MyFirstAotWebApi;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Logging.AddConsole();

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

var sampleTodos = TodoGenerator.GenerateTodos().ToArray();

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());

app.Run();

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

In the preceding highlighted code:

- The JSON serializer context is registered with the DI container. For more information, see:
  - Combine source generators
  - TypeInfoResolverChain
- The custom `JsonSerializerContext` is annotated with the [JsonSerializable] attribute to enable source generated JSON serializer code for the `ToDo` type.

A parameter on the delegate that isn't bound to the body and does ***not*** need to be serializable. For example, a query string parameter that is a rich object type and implements `IParsable<T>`.

C#

```
public class Todo
{
    public int Id { get; set; }
```

```
    public string? Title { get; set; }
    public DateOnly? DueBy { get; set; }
    public bool IsComplete { get; set; }
}

static class TodoGenerator
{
    private static readonly (string[] Prefixes, string[] Suffixes)[] _parts
= new[]
        {
            (new[] { "Walk the", "Feed the" }, new[] { "dog", "cat", "goat"
}),
            (new[] { "Do the", "Put away the" }, new[] { "groceries",
"dishes", "laundry" }),
            (new[] { "Clean the" }, new[] { "bathroom", "pool", "blinds",
"car" })
        };
    // Remaining code omitted for brevity.
```

Known issues

See this GitHub issue to report or review issues with Native AOT support in ASP.NET Core.

See also

- Tutorial: Publish an ASP.NET Core app using Native AOT
- Native AOT deployment
- Optimize AOT deployments
- Configuration-binding source generator
- Using the configuration binder source generator
- The minimal API AOT compilation template
- Comparing WebApplication.CreateBuilder to CreateSlimBuilder
- Exploring the new minimal API source generator
- Replacing method calls with Interceptors
- Behind [LogProperties] and the new telemetry logging source generator

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Tutorial: Publish an ASP.NET Core app using Native AOT

Article • 07/26/2024

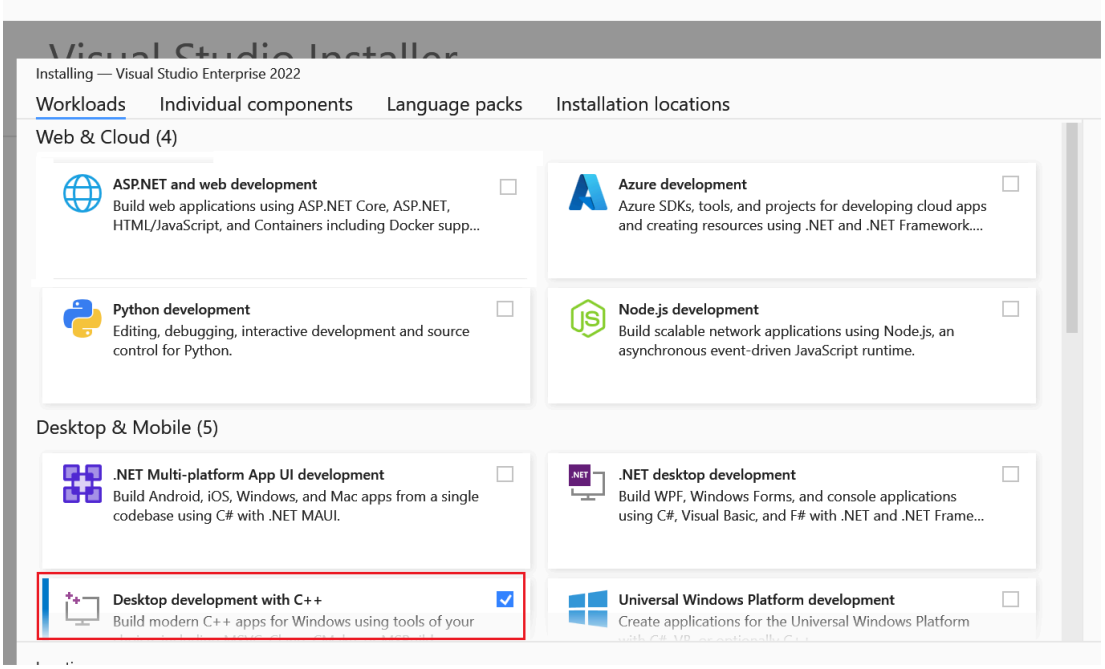
ASP.NET Core 8.0 introduces support for .NET native ahead-of-time (AOT).

> **Note**
>
> - The Native AOT feature is currently in preview.
> - In .NET 8, not all ASP.NET Core features are compatible with Native AOT.
> - Tabs are provided for the **.NET CLI** and **Visual Studio** instructions:
>   - Visual Studio is a prerequisite even if the CLI tab is selected.
>   - The CLI must be used to publish even if the Visual Studio tab is selected.

Prerequisites

.NET CLI

- .NET 8.0 SDK
- On Linux, see Prerequisites for Native AOT deployment.
- Visual Studio 2022 with the **Desktop development with C++** workload installed.



> **Note**
>
> Visual Studio 2022 is required because Native AOT requires link.exe and the Visual C++ static runtime libraries. There are no plans to support Native AOT ***without*** Visual Studio.

Create a web app with Native AOT

Create an ASP.NET Core API app that is configured to work with Native AOT:

.NET CLI

Run the following commands:

.NET CLI

```
dotnet new webapiaot -o MyFirstAotWebApi && cd MyFirstAotWebApi
```

Output similar to the following example is displayed:

Output

```
The template "ASP.NET Core Web API (Native AOT)" was created
successfully.

Processing post-creation actions...
Restoring C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj:
```

```
  Determining projects to restore...
  Restored C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj (in
302 ms).
Restore succeeded.
```

Publish the Native AOT app

Verify the app can be published using Native AOT:

.NET CLI

.NET CLI

```
dotnet publish
```

The `dotnet publish` command:

- Compiles the source files.
- Generates source code files that are compiled.
- Passes generated assemblies to a native IL compiler. The IL compiler produces the native executable. The native executable contains the native machine code.

Output similar to the following example is displayed:

Output

```
MSBuild version 17.<version> for .NET
  Determining projects to restore...
  Restored C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj (in 241
ms).
C:\Code\dotnet\aspnetcore\.dotnet\sdk\8.0.
<version>\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.RuntimeIde
ntifierInference.targets(287,5): message NETSDK1057: You are using a preview
version of .NET. See: https://aka.ms/dotne
t-support-policy [C:\Code\Demos\MyFirstAotWebApi\MyFirstAotWebApi.csproj]
  MyFirstAotWebApi -> C:\Code\Demos\MyFirstAotWebApi\bin\Release\net8.0\win-
x64\MyFirstAotWebApi.dll
  Generating native code
  MyFirstAotWebApi -> C:\Code\Demos\MyFirstAotWebApi\bin\Release\net8.0\win-
x64\publish\
```

The output may differ from the preceding example depending on the version of .NET 8 used, directory used, and other factors.

Review the contents of the output directory:

```
dir bin\Release\net8.0\win-x64\publish
```

Output similar to the following example is displayed:

Output

```
    Directory: C:\Code\Demos\MyFirstAotWebApi\bin\Release\net8.0\win-
x64\publish

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          30/03/2023  1:41 PM        9480704 MyFirstAotWebApi.exe
-a---          30/03/2023  1:41 PM       43044864 MyFirstAotWebApi.pdb
```

The executable is self-contained and doesn't require a .NET runtime to run. When launched, it behaves the same as the app run in the development environment. Run the AOT app:

```
.\bin\Release\net8.0\win-x64\publish\MyFirstAotWebApi.exe
```

Output similar to the following example is displayed:

Output

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Code\Demos\MyFirstAotWebApi
```

Libraries and Native AOT

Many of the popular libraries used in ASP.NET Core projects currently have some compatibility issues when used in a project targeting Native AOT, such as:

- Use of reflection to inspect and discover types.

- Conditionally loading libraries at runtime.
- Generating code on the fly to implement functionality.

Libraries using these dynamic features need to be updated in order to work with Native AOT. They can be updated using tools like Roslyn source generators.

Library authors hoping to support Native AOT are encouraged to:

- Read about Native AOT compatibility requirements.
- Prepare the library for trimming.

See also

- ASP.NET Core support for Native AOT
- Native AOT deployment
- Using the configuration binder source generator
- The minimal API AOT compilation template
- Comparing WebApplication.CreateBuilder to CreateSlimBuilder
- Exploring the new minimal API source generator
- Replacing method calls with Interceptors
- Configuration-binding source generator

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Turn Map methods into request delegates with the ASP.NET Core Request Delegate Generator

Article • 05/02/2024

The ASP.NET Core Request Delegate Generator (RDG) is a compile-time source generator that compiles route handlers provided to a minimal API to request delegates that can be processed by ASP.NET Core's routing infrastructure. The RDG is implicitly enabled when applications are published with AoT enabled or when trimming is enabled. The RDG generates trim and native AoT-friendly code.

> **Note**
>
> - The Native AOT feature is currently in preview.
> - In .NET 8, not all ASP.NET Core features are compatible with Native AOT.

The RDG:

- Is a source generator.
- Turns each `Map` method into a RequestDelegate associated with the specific route. `Map` methods include the methods in the EndpointRouteBuilderExtensions, such as MapGet, MapPost, MapPatch, MapPut, and MapDelete.

When publishing and Native AOT is ***not*** enabled:

- `Map` methods associated with a specific route are compiled in memory into a request delegate when the app starts, not when the app is built.
- The request delegates are generated at runtime.

When publishing with Native AOT enabled:

- `Map` methods associated with a specific route are compiled when the app is built. The RDG creates the request delegate for the route and the request delegate is compiled into the app's native image.
- Eliminates the need to generate the request delegate at runtime.
- Ensures:
  - The types used in the app's APIs are rooted in the app code in a way that is statically analyzable by the Native AOT tool-chain.
  - The required code isn't trimmed away.

The RDG:

- Is enabled automatically in projects when publishing with Native AOT is enabled or when trimming is enabled.
- Can be manually enabled even when not using Native AOT by setting `<EnableRequestDelegateGenerator>true</EnableRequestDelegateGenerator>` in the project file:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <EnableRequestDelegateGenerator>true</EnableRequestDelegateGenerator>
  </PropertyGroup>

</Project>
```

Manually enabling RDG can be useful for:

- Evaluating a project's compatibility with Native AOT.
- Reducing the app's startup time by pregenerating the request delegates.

Minimal APIs are optimized for using System.Text.Json, which requires using the System.Text.Json source generator. All types accepted as parameters to or returned from request delegates in Minimal APIs must be configured on a JsonSerializerContext that's registered via ASP.NET Core's dependency injection:

C#

```csharp
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(
                        0, AppJsonSerializerContext.Default);
});

var app = builder.Build();

var sampleTodos = new Todo[] {
    new(1, "Walk the dog"),
    new(2, "Do the dishes", DateOnly.FromDateTime(DateTime.Now)),
    new(3, "Do the laundry",
```

```
DateOnly.FromDateTime(DateTime.Now.AddDays(1))),
    new(4, "Clean the bathroom"),
    new(5, "Clean the car", DateOnly.FromDateTime(DateTime.Now.AddDays(2)))
};

var todosApi = app.MapGroup("/todos");
todosApi.MapGet("/", () => sampleTodos);
todosApi.MapGet("/{id}", (int id) =>
    sampleTodos.FirstOrDefault(a => a.Id == id) is { } todo
        ? Results.Ok(todo)
        : Results.NotFound());

app.Run();

public record Todo(int Id, string? Title, DateOnly? DueBy = null, bool
IsComplete = false);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

Diagnostics for unsupported RDG scenarios

When the app is built, the RDG emits diagnostics for scenarios that aren't supported by Native AOT. The diagnostics are emitted as warnings and don't prevent the app from building. For the list of diagnostics, see ASP.NET Core Request Delegate Generator diagnostics.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core Middleware

Article • 11/24/2023

By Rick Anderson and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

Request delegates are used to build the request pipeline. The request delegates handle each HTTP request.

Request delegates are configured using Run, Map, and Use extension methods. An individual request delegate can be specified in-line as an anonymous method (called in-line middleware), or it can be defined in a reusable class. These reusable classes and in-line anonymous methods are *middleware*, also called *middleware components*. Each middleware component in the request pipeline is responsible for invoking the next component in the pipeline or short-circuiting the pipeline. When a middleware short-circuits, it's called a *terminal middleware* because it prevents further middleware from processing the request.

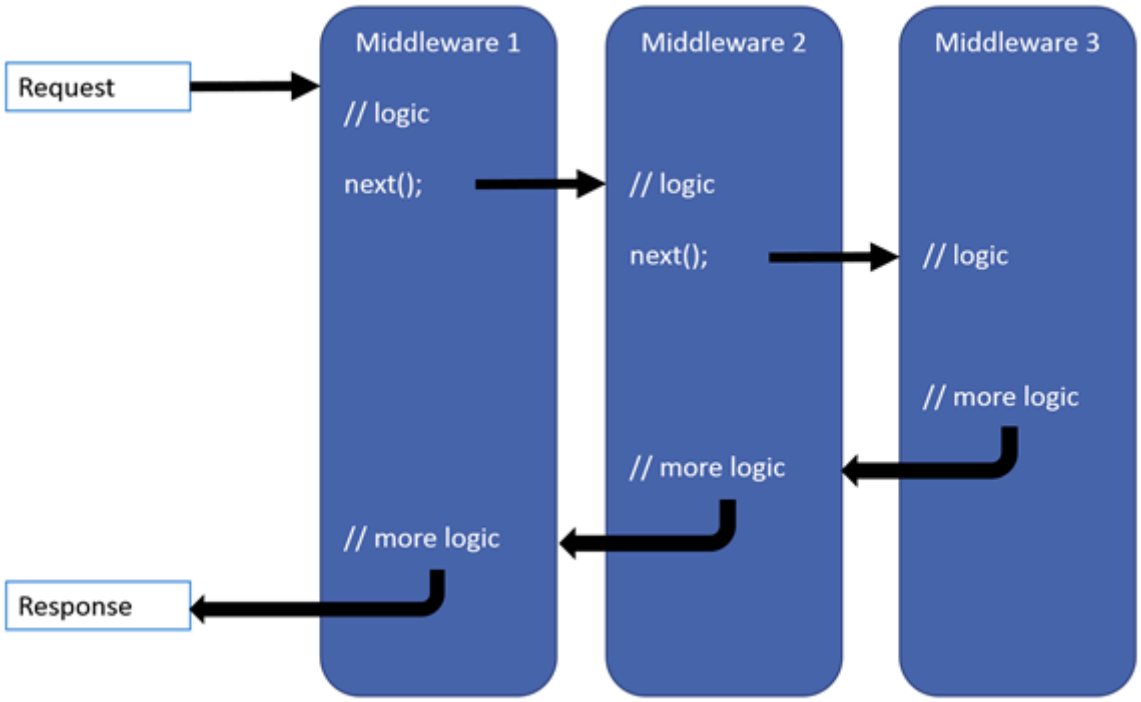
Migrate HTTP handlers and modules to ASP.NET Core middleware explains the difference between request pipelines in ASP.NET Core and ASP.NET 4.x and provides additional middleware samples.

Middleware code analysis

ASP.NET Core includes many compiler platform analyzers that inspect application code for quality. For more information, see Code analysis in ASP.NET Core apps

Create a middleware pipeline with `WebApplication`

The ASP.NET Core request pipeline consists of a sequence of request delegates, called one after the other. The following diagram demonstrates the concept. The thread of execution follows the black arrows.



Each delegate can perform operations before and after the next delegate. Exception-handling delegates should be called early in the pipeline, so they can catch exceptions that occur in later stages of the pipeline.

The simplest possible ASP.NET Core app sets up a single request delegate that handles all requests. This case doesn't include an actual request pipeline. Instead, a single anonymous function is called in response to every HTTP request.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello world!");
});

app.Run();
```

Chain multiple request delegates together with Use. The `next` parameter represents the next delegate in the pipeline. You can short-circuit the pipeline by *not* calling the `next` parameter. You can typically perform actions both before and after the `next` delegate, as the following example demonstrates:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) =>
{
    // Do work that can write to the Response.
    await next.Invoke();
    // Do logging or other work that doesn't write to the Response.
});

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from 2nd delegate.");
});

app.Run();
```

Short-circuiting the request pipeline

When a delegate doesn't pass a request to the next delegate, it's called *short-circuiting the request pipeline*. Short-circuiting is often desirable because it avoids unnecessary work. For example, Static File Middleware can act as a *terminal middleware* by processing a request for a static file and short-circuiting the rest of the pipeline. Middleware added to the pipeline before the middleware that terminates further processing still processes code after their `next.Invoke` statements. However, see the following warning about attempting to write to a response that has already been sent.

> **⚠ Warning**
>
> Don't call `next.Invoke` during or after the response has been sent to the client. After an **HttpResponse** has started, changes result in an exception. For example, **setting headers and a status code throw an exception** after the response starts. Writing to the response body after calling `next`:
>
> - May cause a protocol violation, such as writing more than the stated `Content-Length`.
> - May corrupt the body format, such as writing an HTML footer to a CSS file.
>
> **HasStarted** is a useful hint to indicate if headers have been sent or the body has been written to.

For more information, see Short-circuit middleware after routing.

`Run` delegates

Run delegates don't receive a `next` parameter. The first `Run` delegate is always terminal and terminates the pipeline. `Run` is a convention. Some middleware components may expose `Run[Middleware]` methods that run at the end of the pipeline:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) =>
{
    // Do work that can write to the Response.
    await next.Invoke();
    // Do logging or other work that doesn't write to the Response.
});

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from 2nd delegate.");
});

app.Run();
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

In the preceding example, the `Run` delegate writes `"Hello from 2nd delegate."` to the response and then terminates the pipeline. If another `Use` or `Run` delegate is added after the `Run` delegate, it's not called.

Prefer app.Use overload that requires passing the context to next

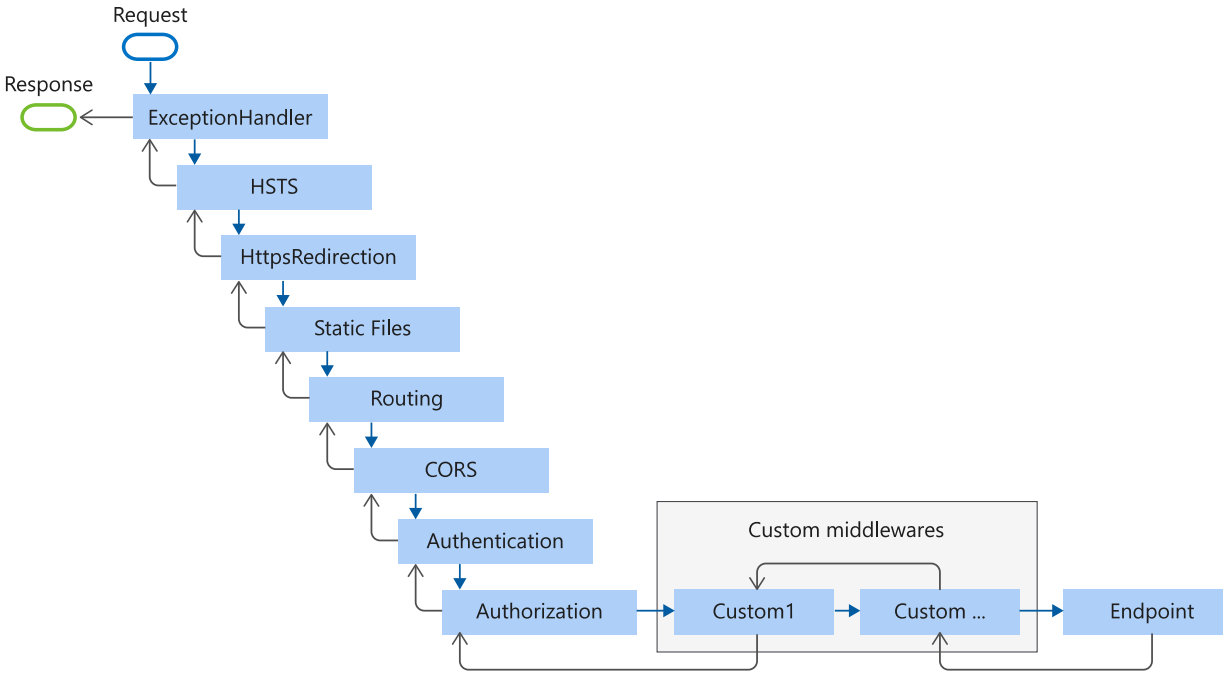
The non-allocating app.Use extension method:

- Requires passing the context to `next`.
- Saves two internal per-request allocations that are required when using the other overload.

For more information, see this GitHub issue.

Middleware order

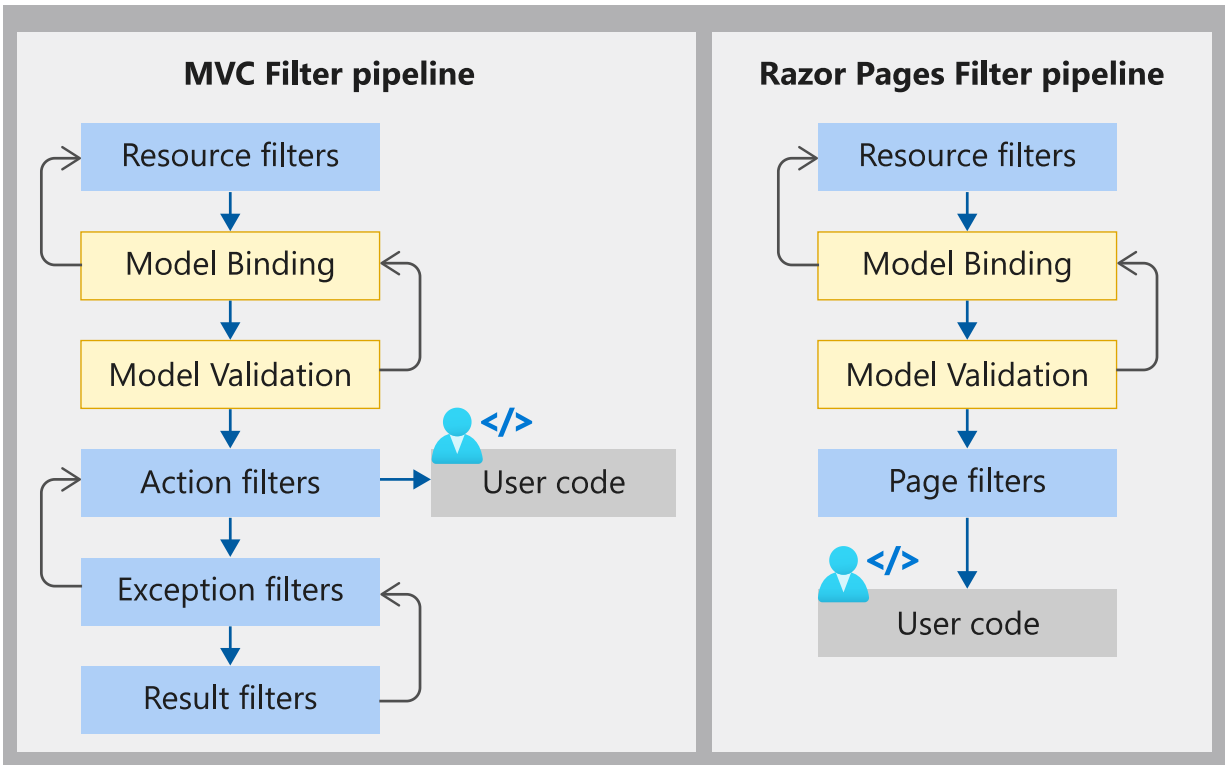
The following diagram shows the complete request processing pipeline for ASP.NET Core MVC and Razor Pages apps. You can see how, in a typical app, existing middlewares are ordered and where custom middlewares are added. You have full control over how to reorder existing middlewares or inject new custom middlewares as necessary for your scenarios.



The **Endpoint** middleware in the preceding diagram executes the filter pipeline for the corresponding app type—MVC or Razor Pages.

The **Routing** middleware in the preceding diagram is shown following **Static Files**. This is the order that the project templates implement by explicitly calling app.UseRouting. If you don't call `app.UseRouting`, the **Routing** middleware runs at the beginning of the pipeline by default. For more information, see Routing.

MVC Endpoint
(called by the Endpoint Middleware)



The order that middleware components are added in the `Program.cs` file defines the order in which the middleware components are invoked on requests and the reverse order for the response. The order is **critical** for security, performance, and functionality.

The following highlighted code in `Program.cs` adds security-related middleware components in the typical recommended order:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebMiddleware.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new InvalidOperationException("Connection string
'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();
```

```
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
// app.UseCookiePolicy();

app.UseRouting();
// app.UseRateLimiter();
// app.UseRequestLocalization();
// app.UseCors();

app.UseAuthentication();
app.UseAuthorization();
// app.UseSession();
// app.UseResponseCompression();
// app.UseResponseCaching();
app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

In the preceding code:

- Middleware that is not added when creating a new web app with individual users accounts is commented out.
- Not every middleware appears in this exact order, but many do. For example:
  - `UseCors`, `UseAuthentication`, and `UseAuthorization` must appear in the order shown.
  - `UseCors` currently must appear before `UseResponseCaching`. This requirement is explained in GitHub issue dotnet/aspnetcore #23218.
  - `UseRequestLocalization` must appear before any middleware that might check the request culture, for example, `app.UseStaticFiles()`.
  - UseRateLimiter must be called after `UseRouting` when rate limiting endpoint specific APIs are used. For example, if the [EnableRateLimiting] attribute is used, `UseRateLimiter` must be called after `UseRouting`. When calling only global limiters, `UseRateLimiter` can be called before `UseRouting`.

In some scenarios, middleware has different ordering. For example, caching and compression ordering is scenario specific, and there are multiple valid orderings. For example:

```
C#

app.UseResponseCaching();
app.UseResponseCompression();
```

With the preceding code, CPU usage could be reduced by caching the compressed response, but you might end up caching multiple representations of a resource using different compression algorithms such as Gzip or Brotli.

The following ordering combines static files to allow caching compressed static files:

```
C#

app.UseResponseCaching();
app.UseResponseCompression();
app.UseStaticFiles();
```

The following `Program.cs` code adds middleware components for common app scenarios:

1. Exception/error handling

    - When the app runs in the Development environment:
        - Developer Exception Page Middleware (UseDeveloperExceptionPage) reports app runtime errors.
        - Database Error Page Middleware (UseDatabaseErrorPage) reports database runtime errors.
    - When the app runs in the Production environment:
        - Exception Handler Middleware (UseExceptionHandler) catches exceptions thrown in the following middlewares.
        - HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) adds the `Strict-Transport-Security` header.

2. HTTPS Redirection Middleware (UseHttpsRedirection) redirects HTTP requests to HTTPS.
3. Static File Middleware (UseStaticFiles) returns static files and short-circuits further request processing.
4. Cookie Policy Middleware (UseCookiePolicy) conforms the app to the EU General Data Protection Regulation (GDPR) regulations.

5. Routing Middleware (UseRouting) to route requests.
6. Authentication Middleware (UseAuthentication) attempts to authenticate the user before they're allowed access to secure resources.
7. Authorization Middleware (UseAuthorization) authorizes a user to access secure resources.
8. Session Middleware (UseSession) establishes and maintains session state. If the app uses session state, call Session Middleware after Cookie Policy Middleware and before MVC Middleware.
9. Endpoint Routing Middleware (UseEndpoints with MapRazorPages) to add Razor Pages endpoints to the request pipeline.

C#

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseDatabaseErrorPage();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseCookiePolicy();
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.UseSession();
app.MapRazorPages();
```

In the preceding example code, each middleware extension method is exposed on WebApplicationBuilder through the Microsoft.AspNetCore.Builder namespace.

UseExceptionHandler is the first middleware component added to the pipeline. Therefore, the Exception Handler Middleware catches any exceptions that occur in later calls.

Static File Middleware is called early in the pipeline so that it can handle requests and short-circuit without going through the remaining components. The Static File Middleware provides **no** authorization checks. Any files served by Static File Middleware, including those under *wwwroot*, are publicly available. For an approach to secure static files, see Static files in ASP.NET Core.

If the request isn't handled by the Static File Middleware, it's passed on to the Authentication Middleware (UseAuthentication), which performs authentication. Authentication doesn't short-circuit unauthenticated requests. Although Authentication Middleware authenticates requests, authorization (and rejection) occurs only after MVC selects a specific Razor Page or MVC controller and action.

The following example demonstrates a middleware order where requests for static files are handled by Static File Middleware before Response Compression Middleware. Static files aren't compressed with this middleware order. The Razor Pages responses can be compressed.

C#

```
// Static files aren't compressed by Static File Middleware.
app.UseStaticFiles();

app.UseRouting();

app.UseResponseCompression();

app.MapRazorPages();
```

For information about Single Page Applications, see Overview of Single Page Apps (SPAs) in ASP.NET Core.

UseCors and UseStaticFiles order

The order for calling `UseCors` and `UseStaticFiles` depends on the app. For more information, see UseCors and UseStaticFiles order

Forwarded Headers Middleware order

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Branch the middleware pipeline

Map extensions are used as a convention for branching the pipeline. `Map` branches the request pipeline based on matches of the given request path. If the request path starts with the given path, the branch is executed.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Map("/map1", HandleMapTest1);

app.Map("/map2", HandleMapTest2);

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

static void HandleMapTest1(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test 1");
    });
}

static void HandleMapTest2(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test 2");
    });
}
```

The following table shows the requests and responses from `http://localhost:1234` using the preceding code.

Expand table

Request	Response
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

When `Map` is used, the matched path segments are removed from `HttpRequest.Path` and appended to `HttpRequest.PathBase` for each request.

`Map` supports nesting, for example:

C#

```
app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a" processing
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b" processing
    });
});
```

`Map` can also match multiple segments at once:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Map("/map1/seg1", HandleMultiSeg);

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

static void HandleMultiSeg(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test 1");
    });
}
```

MapWhen branches the request pipeline based on the result of the given predicate. Any predicate of type `Func<HttpContext, bool>` can be used to map requests to a new branch of the pipeline. In the following example, a predicate is used to detect the presence of a query string variable `branch`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
HandleBranch);
```

```
app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

static void HandleBranch(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        var branchVer = context.Request.Query["branch"];
        await context.Response.WriteAsync($"Branch used = {branchVer}");
    });
}
```

The following table shows the requests and responses from `http://localhost:1234` using the previous code:

Expand table

Request	Response
`localhost:1234`	`Hello from non-Map delegate.`
`localhost:1234/?branch=main`	`Branch used = main`

UseWhen also branches the request pipeline based on the result of the given predicate. Unlike with `MapWhen`, this branch is rejoined to the main pipeline if it doesn't short-circuit or contain a terminal middleware:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseWhen(context => context.Request.Query.ContainsKey("branch"),
    appBuilder => HandleBranchAndRejoin(appBuilder));

app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from non-Map delegate.");
});

app.Run();

void HandleBranchAndRejoin(IApplicationBuilder app)
{
    var logger =
```

```
app.ApplicationServices.GetRequiredService<ILogger<Program>>();

    app.Use(async (context, next) =>
    {
        var branchVer = context.Request.Query["branch"];
        logger.LogInformation("Branch used = {branchVer}", branchVer);

        // Do work that doesn't write to the Response.
        await next();
        // Do other work that doesn't write to the Response.
    });
}
```

In the preceding example, a response of `Hello from non-Map delegate.` is written for all requests. If the request includes a query string variable `branch`, its value is logged before the main pipeline is rejoined.

Built-in middleware

ASP.NET Core ships with the following middleware components. The *Order* column provides notes on middleware placement in the request processing pipeline and under what conditions the middleware may terminate request processing. When a middleware short-circuits the request processing pipeline and prevents further downstream middleware from processing a request, it's called a *terminal middleware*. For more information on short-circuiting, see the Create a middleware pipeline with WebApplication section.

Expand table

Middleware	Description	Order
Authentication	Provides authentication support.	Before `HttpContext.User` is needed. Terminal for OAuth callbacks.
Authorization	Provides authorization support.	Immediately after the Authentication Middleware.
Cookie Policy	Tracks consent from users for storing personal information and enforces minimum standards for cookie fields, such	Before middleware that issues cookies. Examples: Authentication, Session, MVC (TempData).

Middleware	Description	Order
	as `secure` and `SameSite`.	
CORS	Configures Cross-Origin Resource Sharing.	Before components that use CORS. `UseCors` currently must go before `UseResponseCaching` due to this bug.
DeveloperExceptionPage	Generates a page with error information that is intended for use only in the Development environment.	Before components that generate errors. The project templates automatically register this middleware as the first middleware in the pipeline when the environment is Development.
Diagnostics	Several separate middlewares that provide a developer exception page, exception handling, status code pages, and the default web page for new apps.	Before components that generate errors. Terminal for exceptions or serving the default web page for new apps.
Forwarded Headers	Forwards proxied headers onto the current request.	Before components that consume the updated fields. Examples: scheme, host, client IP, method.
Health Check	Checks the health of an ASP.NET Core app and its dependencies, such as checking database availability.	Terminal if a request matches a health check endpoint.
Header Propagation	Propagates HTTP headers from the incoming request to the outgoing HTTP Client requests.	
HTTP Logging	Logs HTTP Requests and Responses.	At the beginning of the middleware pipeline.
HTTP Method Override	Allows an incoming POST request to override the method.	Before components that consume the updated method.
HTTPS Redirection	Redirect all HTTP requests to HTTPS.	Before components that consume the URL.

Middleware	Description	Order
HTTP Strict Transport Security (HSTS)	Security enhancement middleware that adds a special response header.	Before responses are sent and after components that modify requests. Examples: Forwarded Headers, URL Rewriting.
MVC	Processes requests with MVC/Razor Pages.	Terminal if a request matches a route.
OWIN	Interop with OWIN-based apps, servers, and middleware.	Terminal if the OWIN Middleware fully processes the request.
Output Caching	Provides support for caching responses based on configuration.	Before components that require caching. `UseRouting` must come before `UseOutputCaching`. `UseCORS` must come before `UseOutputCaching`.
Response Caching	Provides support for caching responses. This requires client participation to work. Use output caching for complete server control.	Before components that require caching. `UseCORS` must come before `UseResponseCaching`. Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. Output caching benefits UI apps.
Request Decompression	Provides support for decompressing requests.	Before components that read the request body.
Response Compression	Provides support for compressing responses.	Before components that require compression.
Request Localization	Provides localization support.	Before localization sensitive components. Must appear after Routing Middleware when using RouteDataRequestCultureProvider.
Request Timeouts	Provides support for configuring request timeouts, global and per endpoint.	`UseRequestTimeouts` must come after `UseExceptionHandler`, `UseDeveloperExceptionPage`, and `UseRouting`.
Endpoint Routing	Defines and constrains request routes.	Terminal for matching routes.

Middleware	Description	Order
SPA	Handles all requests from this point in the middleware chain by returning the default page for the Single Page Application (SPA)	Late in the chain, so that other middleware for serving static files, MVC actions, etc., takes precedence.
Session	Provides support for managing user sessions.	Before components that require Session.
Static Files	Provides support for serving static files and directory browsing.	Terminal if a request matches a file.
URL Rewrite	Provides support for rewriting URLs and redirecting requests.	Before components that consume the URL.
W3CLogging	Generates server access logs in the W3C Extended Log File Format.	At the beginning of the middleware pipeline.
WebSockets	Enables the WebSockets protocol.	Before components that are required to accept WebSocket requests.

Additional resources

- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- Write custom ASP.NET Core middleware
- Test ASP.NET Core middleware
- Configure gRPC-Web in ASP.NET Core
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Rate limiting middleware in ASP.NET Core

Article • 10/03/2023

By Arvin Kahbazi, Maarten Balliauw, and Rick Anderson

The `Microsoft.AspNetCore.RateLimiting` middleware provides rate limiting middleware. Apps configure rate limiting policies and then attach the policies to endpoints. Apps using rate limiting should be carefully load tested and reviewed before deploying. See Testing endpoints with rate limiting in this article for more information.

For an introduction to rate limiting, see Rate limiting middleware.

Rate limiter algorithms

The RateLimiterOptionsExtensions class provides the following extension methods for rate limiting:

- Fixed window
- Sliding window
- Token bucket
- Concurrency

Fixed window limiter

The AddFixedWindowLimiter method uses a fixed time window to limit requests. When the time window expires, a new time window starts and the request limit is reset.

Consider the following code:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: "fixed", options =>
    {
        options.PermitLimit = 4;
        options.Window = TimeSpan.FromSeconds(12);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
```

```
        options.QueueLimit = 2;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"))
                          .RequireRateLimiting("fixed");

app.Run();
```

The preceding code:

- Calls AddRateLimiter to add a rate limiting service to the service collection.
- Calls `AddFixedWindowLimiter` to create a fixed window limiter with a policy name of `"fixed"` and sets:
- PermitLimit to 4 and the time Window to 12. A maximum of 4 requests per each 12-second window are allowed.
- QueueProcessingOrder to OldestFirst.
- QueueLimit to 2.
- Calls UseRateLimiter to enable rate limiting.

Apps should use Configuration to set limiter options. The following code updates the preceding code using MyRateLimitOptions for configuration:

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
```

```
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Fixed Window Limiter {GetTicks()}"))
                            .RequireRateLimiting(fixedPolicy);

app.Run();
```

UseRateLimiter must be called after `UseRouting` when rate limiting endpoint specific APIs are used. For example, if the [EnableRateLimiting] attribute is used, `UseRateLimiter` must be called after `UseRouting`. When calling only global limiters, `UseRateLimiter` can be called before `UseRouting`.

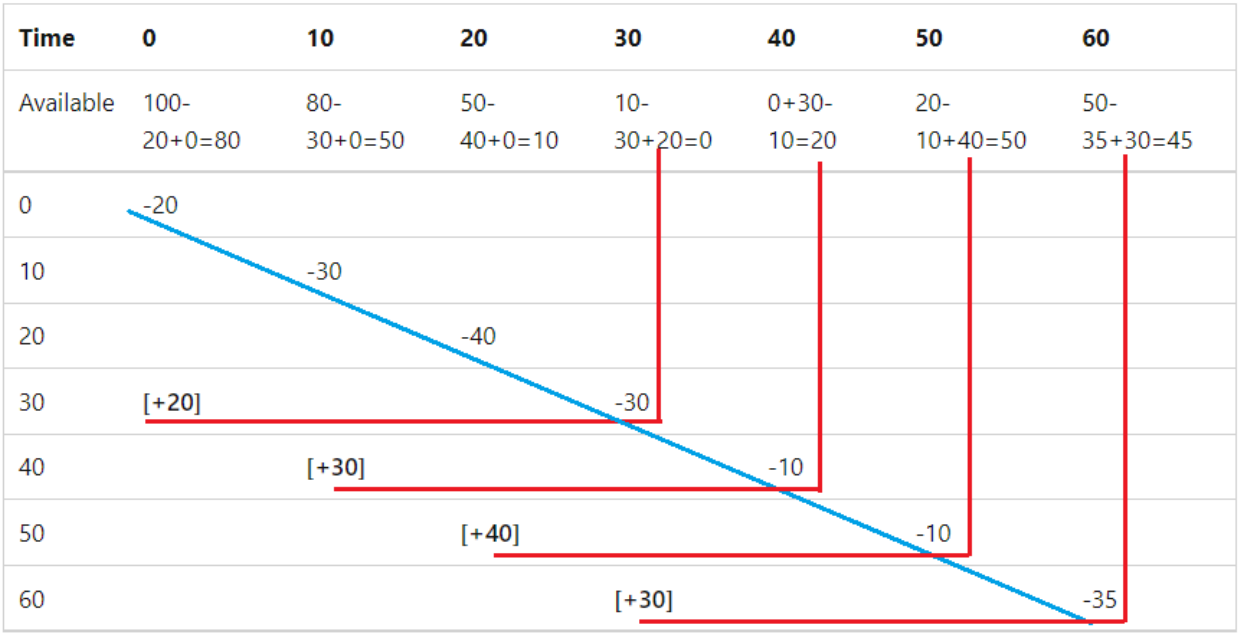
Sliding window limiter

A sliding window algorithm:

- Is similar to the fixed window limiter but adds segments per window. The window slides one segment each segment interval. The segment interval is (window time)/(segments per window).
- Limits the requests for a window to `permitLimit` requests.
- Each time window is divided in `n` segments per window.
- Requests taken from the expired time segment one window back (`n` segments prior to the current segment) are added to the current segment. We refer to the most expired time segment one window back as the expired segment.

Consider the following table that shows a sliding window limiter with a 30-second window, three segments per window, and a limit of 100 requests:

- The top row and first column shows the time segment.
- The second row shows the remaining requests available. The remaining requests are calculated as the available requests minus the processed requests plus the recycled requests.
- Requests at each time moves along the diagonal blue line.
- From time 30 on, the request taken from the expired time segment are added back to the request limit, as shown in the red lines.



The following table shows the data in the previous graph in a different format. The **Available** column shows the requests available from the previous segment (The **Carry over** from the previous row). The first row shows 100 available requests because there's no previous segment.

Expand table

Time	Available	Taken	Recycled from expired	Carry over
0	100	20	0	80
10	80	30	0	50
20	50	40	0	10
30	10	30	20	0
40	0	10	30	20
50	20	10	40	50
60	50	35	30	45

The following code uses the sliding window rate limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);
```

```
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Sliding Window Limiter {GetTicks()}"))
                          .RequireRateLimiting(slidingPolicy);

app.Run();
```

Token bucket limiter

The token bucket limiter is similar to the sliding window limiter, but rather than adding back the requests taken from the expired segment, a fixed number of tokens are added each replenishment period. The tokens added each segment can't increase the available tokens to a number higher than the token bucket limit. The following table shows a token bucket limiter with a limit of 100 tokens and a 10-second replenishment period.

Expand table

Time	Available	Taken	Added	Carry over
0	100	20	0	80
10	80	10	20	90
20	90	5	15	100
30	100	30	20	90
40	90	6	16	100
50	100	40	20	80

Time	Available	Taken	Added	Carry over
60	80	50	20	50

The following code uses the token bucket limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var tokenPolicy = "token";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddTokenBucketLimiter(policyName: tokenPolicy, options =>
    {
        options.TokenLimit = myOptions.TokenLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
        options.ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod);
        options.TokensPerPeriod = myOptions.TokensPerPeriod;
        options.AutoReplenishment = myOptions.AutoReplenishment;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Token Limiter {GetTicks()}"))
                            .RequireRateLimiting(tokenPolicy);

app.Run();
```

When AutoReplenishment is set to `true`, an internal timer replenishes the tokens every ReplenishmentPeriod; when set to `false`, the app must call TryReplenish on the limiter.

Concurrency limiter

The concurrency limiter limits the number of concurrent requests. Each request reduces the concurrency limit by one. When a request completes, the limit is increased by one.

Unlike the other requests limiters that limit the total number of requests for a specified period, the concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period.

The following code uses the concurrency limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var concurrencyPolicy = "Concurrency";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: concurrencyPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", async () =>
{
    await Task.Delay(500);
    return Results.Ok($"Concurrency Limiter {GetTicks()}");

}).RequireRateLimiting(concurrencyPolicy);

app.Run();
```

Create chained limiters

The CreateChained API allows passing in multiple PartitionedRateLimiter which are combined into one `PartitionedRateLimiter`. The combined limiter runs all the input limiters in sequence.

The following code uses `CreateChained`:

C#

```
using System.Globalization;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ =>
{
    _.OnRejected = (context, _) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.Response.WriteAsync("Too many requests. Please
try again later.");

        return new ValueTask();
    };
    _.GlobalLimiter = PartitionedRateLimiter.CreateChained(
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
            (userAgent, _ =>
                new FixedWindowRateLimiterOptions
                {
                    AutoReplenishment = true,
                    PermitLimit = 4,
                    Window = TimeSpan.FromSeconds(2)
                });
        }),
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
            (userAgent, _ =>
                new FixedWindowRateLimiterOptions
                {
                    AutoReplenishment = true,
                    PermitLimit = 20,
                    Window = TimeSpan.FromSeconds(30)
                });
```

```
        }));
});

var app = builder.Build();
app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"));

app.Run();
```

For more information, see the CreateChained source code

`EnableRateLimiting` and `DisableRateLimiting` attributes

The [EnableRateLimiting] and [DisableRateLimiting] attributes can be applied to a Controller, action method, or Razor Page. For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, `[EnableRateLimiting]` can't be applied to `OnGet`, `OnPost`, or any other page handler.

The `[DisableRateLimiting]` attribute ***disables*** rate limiting to the Controller, action method, or Razor Page regardless of named rate limiters or global limiters applied. For example, consider the following code which calls RequireRateLimiting to apply the `fixedPolicy` rate limiting to all controller endpoints:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
```

```
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.SlidingPermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();
app.UseRateLimiter();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.MapRazorPages().RequireRateLimiting(slidingPolicy);
app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy);

app.Run();
```

In the following code, `[DisableRateLimiting]` disables rate limiting and overrides `[EnableRateLimiting("fixed")]` applied to the `Home2Controller` and `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy)` called in `Program.cs`:

C#

```
[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {
        _logger = logger;
```

```
    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
    }
}
```

In the preceding code, the `[EnableRateLimiting("sliding")]` is ***not*** applied to the `Privacy` action method because `Program.cs` called `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy)`.

Consider the following code which doesn't call `RequireRateLimiting` on `MapRazorPages` or `MapDefaultControllerRoute`:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
```

```
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.SlidingPermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.MapRazorPages();
app.MapDefaultControllerRoute();  // RequireRateLimiting not called

app.Run();
```

Consider the following controller:

C#

```
[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {
```

```
        _logger = logger;
    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
    }
}
```

In the preceding controller:

- The `"fixed"` policy rate limiter is applied to all action methods that don't have `EnableRateLimiting` and `DisableRateLimiting` attributes.
- The `"sliding"` policy rate limiter is applied to the `Privacy` action.
- Rate limiting is disabled on the `NoLimit` action method.

Applying attributes to Razor Pages

For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, `[EnableRateLimiting]` can't be applied to `OnGet`, `OnPost`, or any other page handler.

The `DisableRateLimiting` attribute disables rate limiting on a Razor Page. `EnableRateLimiting` is only applied to a Razor Page if `MapRazorPages().RequireRateLimiting(Policy)` has ***not*** been called.

Limiter algorithm comparison

The fixed, sliding, and token limiters all limit the maximum number of requests in a time period. The concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period. The cost of an endpoint should be considered when selecting a limiter. The cost of an endpoint includes the resources used, for example, time, data access, CPU, and I/O.

Rate limiter samples

The following samples aren't meant for production code but are examples on how to use the limiters.

Limiter with `OnRejected`, `RetryAfter`, and `GlobalLimiter`

The following sample:

- Creates a RateLimiterOptions.OnRejected callback that is called when a request exceeds the specified limit. `retryAfter` can be used with the TokenBucketRateLimiter, FixedWindowLimiter, and SlidingWindowLimiter because these algorithms are able to estimate when more permits will be added. The `ConcurrencyLimiter` has no way of calculating when permits will be available.
- Adds the following limiters:
  - A `SampleRateLimiterPolicy` which implements the `IRateLimiterPolicy<TPartitionKey>` interface. The `SampleRateLimiterPolicy` class is shown later in this article.
  - A `SlidingWindowLimiter`:
    - With a partition for each authenticated user.
    - One shared partition for all anonymous users.
  - A GlobalLimiter that is applied to all requests. The global limiter will be executed first, followed by the endpoint-specific limiter, if one exists. The `GlobalLimiter` creates a partition for each IPAddress.

C#

```
// Preceding code removed for brevity.
using System.Globalization;
using System.Net;
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRateLimitAuth;
using WebRateLimitAuth.Data;
using WebRateLimitAuth.Models;
```

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection") ??
    throw new InvalidOperationException("Connection string
'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var userPolicyName = "user";
var helloPolicy = "hello";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.OnRejected = (context, cancellationToken) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.RequestServices.GetService<ILoggerFactory>()?
            .CreateLogger("Microsoft.AspNetCore.RateLimitingMiddleware")
            .LogWarning("OnRejected: {GetUserEndPoint}",
GetUserEndPoint(context.HttpContext));

        return new ValueTask();
    };

    limiterOptions.AddPolicy<string, SampleRateLimiterPolicy>(helloPolicy);
    limiterOptions.AddPolicy(userPolicyName, context =>
    {
        var username = "anonymous user";
        if (context.User.Identity?.IsAuthenticated is true)
        {
```

```
            username = context.User.ToString()!;
        }

        return RateLimitPartition.GetSlidingWindowLimiter(username,
            _ => new SlidingWindowRateLimiterOptions
            {
                PermitLimit = myOptions.PermitLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                Window = TimeSpan.FromSeconds(myOptions.Window),
                SegmentsPerWindow = myOptions.SegmentsPerWindow
            });

    });

    limiterOptions.GlobalLimiter =
PartitionedRateLimiter.Create<HttpContext, IPAddress>(context =>
    {
        IPAddress? remoteIpAddress = context.Connection.RemoteIpAddress;

        if (!IPAddress.IsLoopback(remoteIpAddress!))
        {
            return RateLimitPartition.GetTokenBucketLimiter
            (remoteIpAddress!, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetNoLimiter(IPAddress.Loopback);
    });
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseRouting();
app.UseRateLimiter();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages().RequireRateLimiting(userPolicyName);
app.MapDefaultControllerRoute();

static string GetUserEndPoint(HttpContext context) =>
   $"User {context.User.Identity?.Name ?? "Anonymous"} endpoint:
{context.Request.Path}"
   + $" {context.Connection.RemoteIpAddress}";
static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/a", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(userPolicyName);

app.MapGet("/b", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(helloPolicy);

app.MapGet("/c", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}");

app.Run();
```

> ⚠ **Warning**
>
> Creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks which employ IP Source Address Spoofing. For more information, see BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing.

See the samples repository for the complete Program.cs file.

The `SampleRateLimiterPolicy` class

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using Microsoft.Extensions.Options;
using WebRateLimitAuth.Models;

namespace WebRateLimitAuth;

public class SampleRateLimiterPolicy : IRateLimiterPolicy<string>
```

```
{
    private Func<OnRejectedContext, CancellationToken, ValueTask>?
_onRejected;
    private readonly MyRateLimitOptions _options;

    public SampleRateLimiterPolicy(ILogger<SampleRateLimiterPolicy> logger,
                                   IOptions<MyRateLimitOptions> options)
    {
        _onRejected = (ctx, token) =>
        {
            ctx.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
            logger.LogWarning($"Request rejected by
{nameof(SampleRateLimiterPolicy)}");
            return ValueTask.CompletedTask;
        };
        _options = options.Value;
    }

    public Func<OnRejectedContext, CancellationToken, ValueTask>? OnRejected
=> _onRejected;

    public RateLimitPartition<string> GetPartition(HttpContext httpContext)
    {
        return RateLimitPartition.GetSlidingWindowLimiter(string.Empty,
            _ => new SlidingWindowRateLimiterOptions
            {
                PermitLimit = _options.PermitLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = _options.QueueLimit,
                Window = TimeSpan.FromSeconds(_options.Window),
                SegmentsPerWindow = _options.SegmentsPerWindow
            });
    }
}
```

In the preceding code, OnRejected uses OnRejectedContext to set the response status to 429 Too Many Requests. The default rejected status is 503 Service Unavailable.

Limiter with authorization

The following sample uses JSON Web Tokens (JWT) and creates a partition with the JWT access token. In a production app, the JWT would typically be provided by a server acting as a Security token service (STS). For local development, the dotnet user-jwts command line tool can be used to create and manage app-specific local JWTs.

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Authentication;
```

```
using Microsoft.Extensions.Primitives;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var jwtPolicyName = "jwt";

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.RejectionStatusCode =
StatusCodes.Status429TooManyRequests;
    limiterOptions.AddPolicy(policyName: jwtPolicyName, partitioner:
httpContext =>
    {
        var accessToken =
httpContext.Features.Get<IAuthenticateResultFeature>()?

.AuthenticateResult?.Properties?.GetTokenValue("access_token")?.ToString()
                           ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(accessToken))
        {
            return RateLimitPartition.GetTokenBucketLimiter(accessToken, _
=>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    });
});
```

```
var app = builder.Build();

app.UseAuthorization();
app.UseRateLimiter();

app.MapGet("/", () => "Hello, World!");

app.MapGet("/jwt", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.MapPost("/post", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.Run();

static string GetUserEndPointMethod(HttpContext context) =>
    $"Hello {context.User.Identity?.Name ?? "Anonymous"} " +
    $"Endpoint:{context.Request.Path} Method: {context.Request.Method}";
```

Limiter with `ConcurrencyLimiter`, `TokenBucketRateLimiter`, and authorization

The following sample:

- Adds a `ConcurrencyLimiter` with a policy name of `"get"` that is used on the Razor Pages.
- Adds a `TokenBucketRateLimiter` with a partition for each authorized user and a partition for all anonymous users.
- Sets RateLimiterOptions.RejectionStatusCode to 429 Too Many Requests.

C#

```
var getPolicyName = "get";
var postPolicyName = "post";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: getPolicyName, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
```

```
    })
    .AddPolicy(policyName: postPolicyName, partitioner: httpContext =>
    {
        string userName = httpContext.User.Identity?.Name ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(userName))
        {
            return RateLimitPartition.GetTokenBucketLimiter(userName, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    }));
```

See the samples repository for the complete Program.cs file.

Testing endpoints with rate limiting

Before deploying an app using rate limiting to production, stress test the app to validate the rate limiters and options used. For example, create a JMeter script with a tool like BlazeMeter or Apache JMeter HTTP(S) Test Script Recorder and load the script to Azure Load Testing.

Creating partitions with user input makes the app vulnerable to Denial of Service (DoS) Attacks. For example, creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks that employ IP Source Address Spoofing. For more information, see BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks that employ IP Source Address Spoofing.

Additional resources

- Rate limiting middleware by Maarten Balliauw provides an excellent introduction and overview to rate limiting.
- Rate limit an HTTP handler in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Middleware in Minimal API apps

Article • 07/26/2024

WebApplication automatically adds the following middleware in Minimal API applications depending on certain conditions:

- UseDeveloperExceptionPage is added first when the HostingEnvironment is `"Development"`.
- UseRouting is added second if user code didn't already call `UseRouting` and if there are endpoints configured, for example `app.MapGet`.
- UseEndpoints is added at the end of the middleware pipeline if any endpoints are configured.
- UseAuthentication is added immediately after `UseRouting` if user code didn't already call `UseAuthentication` and if IAuthenticationSchemeProvider can be detected in the service provider. `IAuthenticationSchemeProvider` is added by default when using AddAuthentication, and services are detected using IServiceProviderIsService.
- UseAuthorization is added next if user code didn't already call `UseAuthorization` and if IAuthorizationHandlerProvider can be detected in the service provider. `IAuthorizationHandlerProvider` is added by default when using AddAuthorization, and services are detected using `IServiceProviderIsService`.
- User configured middleware and endpoints are added between `UseRouting` and `UseEndpoints`.

The following code is effectively what the automatic middleware being added to the app produces:

C#

```
if (isDevelopment)
{
    app.UseDeveloperExceptionPage();
}

app.UseRouting();

if (isAuthenticationConfigured)
{
    app.UseAuthentication();
}

if (isAuthorizationConfigured)
{
    app.UseAuthorization();
```

```
}

// user middleware/endpoints
app.CustomMiddleware(...);
app.MapGet("/", () => "hello world");
// end user middleware/endpoints

app.UseEndpoints(e => {});
```

In some cases, the default middleware configuration isn't correct for the app and requires modification. For example, UseCors should be called before UseAuthentication and UseAuthorization. The app needs to call `UseAuthentication` and `UseAuthorization` if `UseCors` is called:

C#

```
app.UseCors();
app.UseAuthentication();
app.UseAuthorization();
```

If middleware should be run before route matching occurs, UseRouting should be called and the middleware should be placed before the call to `UseRouting`. UseEndpoints isn't required in this case as it is automatically added as described previously:

C#

```
app.Use((context, next) =>
{
    return next(context);
});

app.UseRouting();

// other middleware and endpoints
```

When adding a terminal middleware:

- The middleware must be added after `UseEndpoints`.
- The app needs to call `UseRouting` and `UseEndpoints` so that the terminal middleware can be placed at the correct location.

C#

```
app.UseRouting();

app.MapGet("/", () => "hello world");
```

```
app.UseEndpoints(e => {});

app.Run(context =>
{
    context.Response.StatusCode = 404;
    return Task.CompletedTask;
});
```

Terminal middleware is middleware that runs if no endpoint handles the request.

For information on antiforgery middleware in Minimal APIs, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

For more information about middleware see ASP.NET Core Middleware, and the list of built-in middleware that can be added to applications.

For more information about Minimal APIs see Minimal APIs overview.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Test ASP.NET Core middleware

Article • 01/11/2024

By Chris Ross

Middleware can be tested in isolation with TestServer. It allows you to:

- Instantiate an app pipeline containing only the components that you need to test.
- Send custom requests to verify middleware behavior.

Advantages:

- Requests are sent in-memory rather than being serialized over the network.
- This avoids additional concerns, such as port management and HTTPS certificates.
- Exceptions in the middleware can flow directly back to the calling test.
- It's possible to customize server data structures, such as HttpContext, directly in the test.

Set up the TestServer

In the test project, create a test:

- Build and start a host that uses TestServer.
- Add any required services that the middleware uses.
- Add a package reference to the project for the Microsoft.AspNetCore.TestHost NuGet package.
- Configure the processing pipeline to use the middleware for the test.

```
C#

[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
```

```
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    ...
}
```

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at **Package consumption workflow (NuGet documentation)**. Confirm correct package versions at **NuGet.org**.

Send requests with HttpClient

Send a request using HttpClient:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();
```

```
    var response = await host.GetTestClient().GetAsync("/");

    ...
}
```

Assert the result. First, make an assertion the opposite of the result that you expect. An initial run with a false positive assertion confirms that the test fails when the middleware is performing correctly. Run the test and confirm that the test fails.

In the following example, the middleware should return a 404 status code (*Not Found*) when the root endpoint is requested. Make the first test run with `Assert.NotEqual( ... );`, which should fail:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.NotEqual(HttpStatusCode.NotFound, response.StatusCode);
}
```

Change the assertion to test the middleware under normal operating conditions. The final test uses `Assert.Equal( ... );`. Run the test again to confirm that it passes.

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
```

```
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Send requests with HttpContext

A test app can also send a request using SendAsync(Action<HttpContext>, CancellationToken). In the following example, several checks are made when `https://example.com/A/Path/?and=query` is processed by the middleware:

C#

```
[Fact]
public async Task TestMiddleware_ExpectedResponse()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var server = host.GetTestServer();
    server.BaseAddress = new Uri("https://example.com/A/Path/");

    var context = await server.SendAsync(c =>
```

```
    {
        c.Request.Method = HttpMethods.Post;
        c.Request.Path = "/and/file.txt";
        c.Request.QueryString = new QueryString("?and=query");
    });

    Assert.True(context.RequestAborted.CanBeCanceled);
    Assert.Equal(HttpProtocol.Http11, context.Request.Protocol);
    Assert.Equal("POST", context.Request.Method);
    Assert.Equal("https", context.Request.Scheme);
    Assert.Equal("example.com", context.Request.Host.Value);
    Assert.Equal("/A/Path", context.Request.PathBase.Value);
    Assert.Equal("/and/file.txt", context.Request.Path.Value);
    Assert.Equal("?and=query", context.Request.QueryString.Value);
    Assert.NotNull(context.Request.Body);
    Assert.NotNull(context.Request.Headers);
    Assert.NotNull(context.Response.Headers);
    Assert.NotNull(context.Response.Body);
    Assert.Equal(404, context.Response.StatusCode);
    Assert.Null(context.Features.Get<IHttpResponseFeature>().ReasonPhrase);
}
```

SendAsync permits direct configuration of an HttpContext object rather than using the HttpClient abstractions. Use SendAsync to manipulate structures only available on the server, such as HttpContext.Items or HttpContext.Features.

As with the earlier example that tested for a *404 - Not Found* response, check the opposite for each `Assert` statement in the preceding test. The check confirms that the test fails correctly when the middleware is operating normally. After you've confirmed that the false positive test works, set the final `Assert` statements for the expected conditions and values of the test. Run it again to confirm that the test passes.

Add request routes

Additional routes can be added by configuration using the test `HttpClient`:

C#

```
    [Fact]
    public async Task TestWithEndpoint_ExpectedResponse ()
    {
        using var host = await new HostBuilder()
            .ConfigureWebHost(webBuilder =>
            {
                webBuilder
                    .UseTestServer()
                    .ConfigureServices(services =>
                    {
                        services.AddRouting();
```

```
                    })
                    .Configure(app =>
                    {
                        app.UseRouting();
                        app.UseMiddleware<MyMiddleware>();
                        app.UseEndpoints(endpoints =>
                        {
                            endpoints.MapGet("/hello", () =>
                                TypedResults.Text("Hello Tests"));
                        });
                    });
            })
            .StartAsync();

        var client = host.GetTestClient();

        var response = await client.GetAsync("/hello");

        Assert.True(response.IsSuccessStatusCode);
        var responseBody = await response.Content.ReadAsStringAsync();
        Assert.Equal("Hello Tests", responseBody);
```

Additional routes can also be added using the approach `server.SendAsync`.

TestServer limitations

TestServer:

- Was created to replicate server behaviors to test middleware.
- Does ***not*** try to replicate all HttpClient behaviors.
- Attempts to give the client access to as much control over the server as possible, and with as much visibility into what's happening on the server as possible. For example it may throw exceptions not normally thrown by `HttpClient` in order to directly communicate server state.
- Doesn't set some transport specific headers by default as those aren't usually relevant to middleware. For more information, see the next section.
- Ignores the `Stream` position passed through StreamContent. HttpClient sends the entire stream from the start position, even when positioning is set. For more information, see this GitHub issue.

Content-Length and Transfer-Encoding headers

TestServer does ***not*** set transport related request or response headers such as Content-Length or Transfer-Encoding. Applications should avoid depending on these headers because their usage varies by client, scenario, and protocol. If `Content-Length`

and `Transfer-Encoding` are necessary to test a specific scenario, they can be specified in the test when composing the HttpRequestMessage or HttpContext. For more information, see the following GitHub issues:

- dotnet/aspnetcore#21677
- dotnet/aspnetcore#18463
- dotnet/aspnetcore#13273

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Response Caching Middleware in ASP.NET Core

Article • 04/11/2023

By John Luo and Rick Anderson

This article explains how to configure Response Caching Middleware in an ASP.NET Core app. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the [ResponseCache] attribute, see Response Caching.

The Response caching middleware:

- Enables caching server responses based on HTTP cache headers. Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. Output caching, which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the Conditions for caching are met.

To test response caching, use Fiddler, or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see Troubleshooting.

Configuration

In `Program.cs`, add the Response Caching Middleware services AddResponseCaching to the service collection and configure the app to use the middleware with the UseResponseCaching extension method. `UseResponseCaching` adds the middleware to the request processing pipeline:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching();

var app = builder.Build();
```

```
app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();
```

> ⚠ **Warning**
>
> UseCors must be called before UseResponseCaching when using CORS middleware.

The sample app adds headers to control caching on subsequent requests:

- Cache-Control: Caches cacheable responses for up to 10 seconds.
- Vary: Configures the middleware to serve a cached response only if the Accept-Encoding header of subsequent requests matches that of the original request.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching();

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next();
});

app.MapGet("/", () => DateTime.Now.Millisecond);
```

```
app.Run();
```

The preceding headers are not written to the response and are overridden when a controller, action, or Razor Page:

- Has a [ResponseCache] attribute. This applies even if a property isn't set. For example, omitting the VaryByHeader property will cause the corresponding header to be removed from the response.

Response Caching Middleware only caches server responses that result in a 200 (OK) status code. Any other responses, including error pages, are ignored by the middleware.

> ⚠ **Warning**
>
> Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See **Conditions for caching** for details on how the middleware determines if a response is cacheable.

The preceding code typically doesn't return a cached value to a browser. Use Fiddler or another tool that can explicitly set request headers and is preferred for testing caching. For more information, see Troubleshooting in this article.

Options

Response caching options are shown in the following table.

Expand table

Option	Description
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is `64 * 1024 * 1024` (64 MB).
SizeLimit	The size limit for the response cache middleware in bytes. The default value is `100 * 1024 * 1024` (100 MB).
UseCaseSensitivePaths	Determines if responses are cached on case-sensitive paths. The default value is `false`.

The following example configures the middleware to:

- Cache responses with a body size smaller than or equal to 1,024 bytes.

- Store the responses by case-sensitive paths. For example, `/page1` and `/Page1` are stored separately.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching(options =>
{
    options.MaximumBodySize = 1024;
    options.UseCaseSensitivePaths = true;
});

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next(context);
});

app.MapGet("/", () => DateTime.Now.Millisecond);

app.Run();
```

VaryByQueryKeys

When using MVC, web API controllers, or Razor Pages page models, the [ResponseCache] attribute specifies the parameters necessary for setting the appropriate headers for response caching. The only parameter of the `[ResponseCache]` attribute that strictly requires the middleware is VaryByQueryKeys, which doesn't correspond to an actual HTTP header. For more information, see Response caching in ASP.NET Core.

When not using the `[ResponseCache]` attribute, response caching can be varied with `VaryByQueryKeys`. Use the ResponseCachingFeature directly from the HttpContext.Features:

```
C#

var responseCachingFeature =
context.HttpContext.Features.Get<IResponseCachingFeature>();

if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

Using a single value equal to `*` in `VaryByQueryKeys` varies the cache by all request query parameters.

HTTP headers used by Response Caching Middleware

The following table provides information on HTTP headers that affect response caching.

Expand table

Header	Details
`Authorization`	The response isn't cached if the header exists.
`Cache-Control`	The middleware only considers caching responses marked with the `public` cache directive. Control caching with the following parameters: • max-age • max-stale† • min-fresh • must-revalidate • no-cache • no-store • only-if-cached • private • public • s-maxage • proxy-revalidate‡ †If no limit is specified to `max-stale`, the middleware takes no action. ‡`proxy-revalidate` has the same effect as `must-revalidate`.

Header	Details
	For more information, see RFC 9111: Request Directives.
`Pragma`	A `Pragma: no-cache` header in the request produces the same effect as `Cache-Control: no-cache`. This header is overridden by the relevant directives in the `Cache-Control` header, if present. Considered for backward compatibility with HTTP/1.0.
`Set-Cookie`	The response isn't cached if the header exists. Any middleware in the request processing pipeline that sets one or more cookies prevents the Response Caching Middleware from caching the response (for example, the cookie-based TempData provider).
`Vary`	The `Vary` header is used to vary the cached response by another header. For example, cache responses by encoding by including the `Vary: Accept-Encoding` header, which caches responses for requests with headers `Accept-Encoding: gzip` and `Accept-Encoding: text/plain` separately. A response with a header value of `*` is never stored.
`Expires`	A response deemed stale by this header isn't stored or retrieved unless overridden by other `Cache-Control` headers.
`If-None-Match`	The full response is served from cache if the value isn't `*` and the `ETag` of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.
`If-Modified-Since`	If the `If-None-Match` header isn't present, a full response is served from cache if the cached response date is newer than the value provided. Otherwise, a *304 - Not Modified* response is served.
`Date`	When serving from cache, the `Date` header is set by the middleware if it wasn't provided on the original response.
`Content-Length`	When serving from cache, the `Content-Length` header is set by the middleware if it wasn't provided on the original response.
`Age`	The `Age` header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

Caching respects request Cache-Control directives

The middleware respects the rules of RFC 9111: HTTP Caching (Section 5.2. Cache-Control). The rules require a cache to honor a valid `Cache-Control` header sent by the client. Under the specification, a client can make requests with a `no-cache` header value

and force the server to generate a new response for every request. Currently, there's no developer control over this caching behavior when using the middleware because the middleware adheres to the official caching specification.

For more control over caching behavior, explore other caching features of ASP.NET Core. See the following topics:

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Troubleshooting

The Response Caching Middleware uses IMemoryCache, which has a limited capacity. When the capacity is exceeded, the memory cache is compacted.

If caching behavior isn't as expected, confirm that responses are cacheable and capable of being served from the cache. Examine the request's incoming headers and the response's outgoing headers. Enable logging to help with debugging.

When testing and troubleshooting caching behavior, a browser typically sets request headers that prevent caching. For example, a browser may set the `Cache-Control` header to `no-cache` or `max-age=0` when refreshing a page. Fiddler and other tools can explicitly set request headers and are preferred for testing caching.

Conditions for caching

- The request must result in a server response with a 200 (OK) status code.
- The request method must be GET or HEAD.
- Response Caching Middleware must be placed before middleware that require caching. For more information, see ASP.NET Core Middleware.
- The `Authorization` header must not be present.
- `Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
- The `Pragma: no-cache` header must not be present if the `Cache-Control` header isn't present, as the `Cache-Control` header overrides the `Pragma` header when present.
- The `Set-Cookie` header must not be present.
- `Vary` header parameters must be valid and not equal to `*`.

- The `Content-Length` header value (if set) must match the size of the response body.
- The IHttpSendFileFeature isn't used.
- The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
- Response buffering must be successful. The size of the response must be smaller than the configured or default SizeLimit. The body size of the response must be smaller than the configured or default MaximumBodySize.
- The response must be cacheable according to RFC 9111: HTTP Caching. For example, the `no-store` directive must not exist in request or response header fields. See RFC 9111: HTTP Caching (Section 3: *Storing Responses in Caches*) for details.

> **Note**
>
> The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks sets the `Cache-Control` and `Pragma` headers to `no-cache` so that responses aren't cached. For information on how to disable antiforgery tokens for HTML form elements, see **Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core**.

Additional resources

- View or download sample code (how to download)
- GitHub source for IResponseCachingPolicyProvider
- GitHub source for IResponseCachingPolicyProvider
- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Collaborate with us on GitHub

.NET ASP.NET Core feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Write custom ASP.NET Core middleware

Article • 07/26/2024

By Fiyaz Hasan, Rick Anderson, and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. ASP.NET Core provides a rich set of built-in middleware components, but in some scenarios you might want to write a custom middleware.

This topic describes how to write *convention-based* middleware. For an approach that uses strong typing and per-request activation, see Factory-based middleware activation in ASP.NET Core.

Middleware class

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following inline middleware, which sets the culture for the current request from a query string:

C#

```
using System.Globalization;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseHttpsRedirection();

app.Use(async (context, next) =>
{
    var cultureQuery = context.Request.Query["culture"];
    if (!string.IsNullOrWhiteSpace(cultureQuery))
    {
        var culture = new CultureInfo(cultureQuery);

        CultureInfo.CurrentCulture = culture;
        CultureInfo.CurrentUICulture = culture;
    }

    // Call the next delegate/middleware in the pipeline.
    await next(context);
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
```

```
{CultureInfo.CurrentCulture.DisplayName}");
});

app.Run();
```

The preceding highlighted inline middleware is used to demonstrate creating a middleware component by calling Microsoft.AspNetCore.Builder.UseExtensions.Use. The preceding `Use` extension method adds a middleware delegate defined in-line to the application's request pipeline.

There are two overloads available for the `Use` extension:

- One takes a HttpContext and a `Func<Task>`. Invoke the `Func<Task>` without any parameters.
- The other takes a `HttpContext` and a RequestDelegate. Invoke the `RequestDelegate` by passing the `HttpContext`.

Prefer using the later overload as it saves two internal per-request allocations that are required when using the other overload.

Test the middleware by passing in the culture. For example, request `https://localhost:5001/?culture=es-es`.

For ASP.NET Core's built-in localization support, see Globalization and localization in ASP.NET Core.

The following code moves the middleware delegate to a class:

C#

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
    private readonly RequestDelegate _next;

    public RequestCultureMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
```

```
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
        await _next(context);
    }
}
```

The middleware class must include:

- A public constructor with a parameter of type RequestDelegate.
- A public method named `Invoke` or `InvokeAsync`. This method must:
  - Return a `Task`.
  - Accept a first parameter of type HttpContext.

Additional parameters for the constructor and `Invoke`/`InvokeAsync` are populated by dependency injection (DI).

Typically, an extension method is created to expose the middleware through IApplicationBuilder:

C#

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
    private readonly RequestDelegate _next;

    public RequestCultureMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
```

```
        await _next(context);
    }
}

public static class RequestCultureMiddlewareExtensions
{
    public static IApplicationBuilder UseRequestCulture(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<RequestCultureMiddleware>();
    }
}
```

The following code calls the middleware from `Program.cs`:

C#

```
using Middleware.Example;
using System.Globalization;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseHttpsRedirection();

app.UseRequestCulture();

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
{CultureInfo.CurrentCulture.DisplayName}");
});

app.Run();
```

Middleware dependencies

Middleware should follow the Explicit Dependencies Principle by exposing its dependencies in its constructor. Middleware is constructed once per *application lifetime*.

Middleware components can resolve their dependencies from dependency injection (DI) through constructor parameters. UseMiddleware can also accept additional parameters directly.

Per-request middleware dependencies

Middleware is constructed at app startup and therefore has application life time. Scoped lifetime services used by middleware constructors aren't shared with other dependency-injected types during each request. To share a *scoped* service between middleware and other types, add these services to the `InvokeAsync` method's signature. The `InvokeAsync` method can accept additional parameters that are populated by DI:

C#

```
namespace Middleware.Example;

public class MyCustomMiddleware
{
    private readonly RequestDelegate _next;

    public MyCustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    // IMessageWriter is injected into InvokeAsync
    public async Task InvokeAsync(HttpContext httpContext, IMessageWriter
svc)
    {
        svc.Write(DateTime.Now.Ticks.ToString());
        await _next(httpContext);
    }
}

public static class MyCustomMiddlewareExtensions
{
    public static IApplicationBuilder UseMyCustomMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyCustomMiddleware>();
    }
}
```

Lifetime and registration options contains a complete sample of middleware with *scoped* lifetime services.

The following code is used to test the preceding middleware:

C#

```
using Middleware.Example;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMessageWriter, LoggingMessageWriter>();

var app = builder.Build();
```

```
app.UseHttpsRedirection();

app.UseMyCustomMiddleware();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The `IMessageWriter` interface and implementation:

C#

```
namespace Middleware.Example;

public interface IMessageWriter
{
    void Write(string message);
}

public class LoggingMessageWriter : IMessageWriter
{

    private readonly ILogger<LoggingMessageWriter> _logger;

    public LoggingMessageWriter(ILogger<LoggingMessageWriter> logger) =>
        _logger = logger;

    public void Write(string message) =>
        _logger.LogInformation(message);
}
```

Additional resources

- Sample code used in this article
- UseExtensions source on GitHub
- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- DEEP DIVE: HOW IS THE ASP.NET CORE MIDDLEWARE PIPELINE BUILT
- ASP.NET Core Middleware
- Test ASP.NET Core middleware
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Request and response operations in ASP.NET Core

Article • 07/26/2024

By Justin Kotalik

This article explains how to read from the request body and write to the response body. Code for these operations might be required when writing middleware. Outside of writing middleware, custom code isn't generally required because the operations are handled by MVC and Razor Pages.

There are two abstractions for the request and response bodies: Stream and Pipe. For request reading, HttpRequest.Body is a Stream, and `HttpRequest.BodyReader` is a PipeReader. For response writing, HttpResponse.Body is a Stream, and `HttpResponse.BodyWriter` is a PipeWriter.

Pipelines are recommended over streams. Streams can be easier to use for some simple operations, but pipelines have a performance advantage and are easier to use in most scenarios. ASP.NET Core is starting to use pipelines instead of streams internally. Examples include:

- `FormReader`
- `TextReader`
- `TextWriter`
- `HttpResponse.WriteAsync`

Streams aren't being removed from the framework. Streams continue to be used throughout .NET, and many stream types don't have pipe equivalents, such as `FileStreams` and `ResponseCompression`.

Stream examples

Suppose the goal is to create a middleware that reads the entire request body as a list of strings, splitting on new lines. A simple stream implementation might look like the following example:

> ⚠ **Warning**
>
> The following code:

- Is used to demonstrate the problems with not using a pipe to read the request body.
- Is not intended to be used in production apps.

C#

```
private async Task<List<string>> GetListOfStringsFromStream(Stream
requestBody)
{
    // Build up the request body in a string builder.
    StringBuilder builder = new StringBuilder();

    // Rent a shared buffer to write the request body into.
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);
        if (bytesRemaining == 0)
        {
            break;
        }

        // Append the encoded string into the string builder.
        var encodedString = Encoding.UTF8.GetString(buffer, 0,
bytesRemaining);
        builder.Append(encodedString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    var entireRequestBody = builder.ToString();

    // Split on \n in the string.
    return new List<string>(entireRequestBody.Split("\n"));
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

This code works, but there are some issues:

- Before appending to the `StringBuilder`, the example creates another string (`encodedString`) that is thrown away immediately. This process occurs for all bytes in the stream, so the result is extra memory allocation the size of the entire request body.

- The example reads the entire string before splitting on new lines. It's more efficient to check for new lines in the byte array.

Here's an example that fixes some of the preceding issues:

> **⚠ Warning**
>
> The following code:
>
> - Is used to demonstrate the solutions to some problems in the preceding code while not solving all the problems.
> - Is not intended to be used in production apps.

C#

```
private async Task<List<string>>
GetListOfStringsFromStreamMoreEfficient(Stream requestBody)
{
    StringBuilder builder = new StringBuilder();
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);
    List<string> results = new List<string>();

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);

        if (bytesRemaining == 0)
        {
            results.Add(builder.ToString());
            break;
        }

        // Instead of adding the entire buffer into the StringBuilder
        // only add the remainder after the last \n in the array.
        var prevIndex = 0;
        int index;
        while (true)
        {
            index = Array.IndexOf(buffer, (byte)'\n', prevIndex);
            if (index == -1)
            {
                break;
            }

            var encodedString = Encoding.UTF8.GetString(buffer, prevIndex,
index - prevIndex);

            if (builder.Length > 0)
            {
```

```
                // If there was a remainder in the string buffer, include it
in the next string.
                results.Add(builder.Append(encodedString).ToString());
                builder.Clear();
            }
            else
            {
                results.Add(encodedString);
            }

            // Skip past last \n
            prevIndex = index + 1;
        }

        var remainingString = Encoding.UTF8.GetString(buffer, prevIndex,
bytesRemaining - prevIndex);
        builder.Append(remainingString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    return results;
}
```

This preceding example:

- Doesn't buffer the entire request body in a `StringBuilder` unless there aren't any newline characters.
- Doesn't call `Split` on the string.

However, there are still a few issues:

- If newline characters are sparse, much of the request body is buffered in the string.
- The code continues to create strings (`remainingString`) and adds them to the string buffer, which results in an extra allocation.

These issues are fixable, but the code is becoming progressively more complicated with little improvement. Pipelines provide a way to solve these problems with minimal code complexity.

Pipelines

The following example shows how the same scenario can be handled using a PipeReader:

```
C#
```

```
private async Task<List<string>> GetListOfStringFromPipe(PipeReader reader)
{
    List<string> results = new List<string>();

    while (true)
    {
        ReadResult readResult = await reader.ReadAsync();
        var buffer = readResult.Buffer;

        SequencePosition? position = null;

        do
        {
            // Look for a EOL in the buffer
            position = buffer.PositionOf((byte)'\n');

            if (position != null)
            {
                var readOnlySequence = buffer.Slice(0, position.Value);
                AddStringToList(results, in readOnlySequence);

                // Skip the line + the \n character (basically position)
                buffer = buffer.Slice(buffer.GetPosition(1,
position.Value));
            }
        }
        while (position != null);


        if (readResult.IsCompleted && buffer.Length > 0)
        {
            AddStringToList(results, in buffer);
        }

        reader.AdvanceTo(buffer.Start, buffer.End);

        // At this point, buffer will be updated to point one byte after the
last
        // \n character.
        if (readResult.IsCompleted)
        {
            break;
        }
    }

    return results;
}

private static void AddStringToList(List<string> results, in
ReadOnlySequence<byte> readOnlySequence)
{
    // Separate method because Span/ReadOnlySpan cannot be used in async
methods
    ReadOnlySpan<byte> span = readOnlySequence.IsSingleSegment ?
```

```
        readOnlySequence.First.Span : readOnlySequence.ToArray().AsSpan();
        results.Add(Encoding.UTF8.GetString(span));
    }
```

This example fixes many issues that the streams implementations had:

- There's no need for a string buffer because the `PipeReader` handles bytes that haven't been used.
- Encoded strings are directly added to the list of returned strings.
- Other than the `ToArray` call, and the memory used by the string, string creation is allocation free.

Adapters

The `Body`, `BodyReader`, and `BodyWriter` properties are available for `HttpRequest` and `HttpResponse`. When you set `Body` to a different stream, a new set of adapters automatically adapt each type to the other. If you set `HttpRequest.Body` to a new stream, `HttpRequest.BodyReader` is automatically set to a new `PipeReader` that wraps `HttpRequest.Body`.

StartAsync

`HttpResponse.StartAsync` is used to indicate that headers are unmodifiable and to run `OnStarting` callbacks. When using Kestrel as a server, calling `StartAsync` before using the `PipeReader` guarantees that memory returned by `GetMemory` belongs to Kestrel's internal Pipe rather than an external buffer.

Additional resources

- System.IO.Pipelines in .NET
- Write custom ASP.NET Core middleware

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Request decompression in ASP.NET Core

Article • 07/26/2024

By David Acker

Request decompression middleware:

- Enables API endpoints to accept requests with compressed content.
- Uses the Content-Encoding HTTP header to automatically identify and decompress requests which contain compressed content.
- Eliminates the need to write code to handle compressed requests.

When the `Content-Encoding` header value on a request matches one of the available decompression providers, the middleware:

- Uses the matching provider to wrap the HttpRequest.Body in an appropriate decompression stream.
- Removes the `Content-Encoding` header, indicating that the request body is no longer compressed.

Requests that don't include a `Content-Encoding` header are ignored by the request decompression middleware.

Decompression:

- Occurs when the body of the request is read. That is, decompression occurs at the endpoint on model binding. The request body isn't decompressed eagerly.
- When attempting to read the decompressed request body with invalid compressed data for the specified `Content-Encoding`, an exception is thrown. Brotli can throw System.InvalidOperationException: Decoder ran into invalid data. Deflate and GZip can throw System.IO.InvalidDataException: The archive entry was compressed using an unsupported compression method.

If the middleware encounters a request with compressed content but is unable to decompress it, the request is passed to the next delegate in the pipeline. For example, a request with an unsupported `Content-Encoding` header value or multiple `Content-Encoding` header values is passed to the next delegate in the pipeline.

Configuration

The following code uses AddRequestDecompression(IServiceCollection) and UseRequestDecompression to enable request decompression for the default `Content-`

`Encoding` types:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRequestDecompression();

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Default decompression providers

The `Content-Encoding` header values that the request decompression middleware supports by default are listed in the following table:

Expand table

Content-Encoding header values	Description
`br`	Brotli compressed data format
`deflate`	DEFLATE compressed data format
`gzip`	Gzip file format

Custom decompression providers

Support for custom encodings can be added by creating custom decompression provider classes that implement IDecompressionProvider:

```
C#

public class CustomDecompressionProvider : IDecompressionProvider
{
    public Stream GetDecompressionStream(Stream stream)
    {
        // Perform custom decompression logic here
        return stream;
    }
}
```

Custom decompression providers are registered with RequestDecompressionOptions along with their corresponding `Content-Encoding` header values:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRequestDecompression(options =>
{
    options.DecompressionProviders.Add("custom", new
CustomDecompressionProvider());
});

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Request size limits

In order to protect against zip bombs or decompression bombs :

- The maximum size of the decompressed request body is limited to the request body size limit enforced by the endpoint or server.
- If the number of bytes read from the decompressed request body stream exceeds the limit, an InvalidOperationException is thrown to prevent additional bytes from being read from the stream.

In order of precedence, the maximum request size for an endpoint is set by:

1. IRequestSizeLimitMetadata.MaxRequestBodySize, such as RequestSizeLimitAttribute or DisableRequestSizeLimitAttribute for MVC endpoints.
2. The global server size limit IHttpMaxRequestBodySizeFeature.MaxRequestBodySize. `MaxRequestBodySize` can be overridden per request with IHttpMaxRequestBodySizeFeature.MaxRequestBodySize, but defaults to the limit configured for the web server implementation.

Expand table

Web server implementation	`MaxRequestBodySize` configuration
HTTP.sys	HttpSysOptions.MaxRequestBodySize
IIS	IISServerOptions.MaxRequestBodySize
Kestrel	KestrelServerLimits.MaxRequestBodySize

> ⚠ **Warning**
>
> Disabling the request body size limit poses a security risk in regards to uncontrolled resource consumption, particularly if the request body is being buffered. Ensure that safeguards are in place to mitigate the risk of denial-of-service (DoS) attacks.

Additional Resources

- ASP.NET Core Middleware
- Mozilla Developer Network: Content-Encoding
- Brotli Compressed Data Format
- DEFLATE Compressed Data Format Specification version 1.3
- GZIP file format specification version 4.3

Factory-based middleware activation in ASP.NET Core

Article • 07/26/2024

IMiddlewareFactory/IMiddleware is an extensibility point for middleware activation that offers the following benefits:

- Activation per client request (injection of scoped services)
- Strong typing of middleware

UseMiddleware extension methods check if a middleware's registered type implements IMiddleware. If it does, the IMiddlewareFactory instance registered in the container is used to resolve the IMiddleware implementation instead of using the convention-based middleware activation logic. The middleware is registered as a scoped or transient service in the app's service container.

IMiddleware is activated per client request (connection), so scoped services can be injected into the middleware's constructor.

IMiddleware

IMiddleware defines middleware for the app's request pipeline. The InvokeAsync(HttpContext, RequestDelegate) method handles requests and returns a Task that represents the execution of the middleware.

Middleware activated by convention:

C#

```
public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
        => _next = next;

    public async Task InvokeAsync(HttpContext context, SampleDbContext
dbContext)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            dbContext.Requests.Add(new Request("Conventional", keyValue));
```

```
            await dbContext.SaveChangesAsync();
        }

        await _next(context);
    }
}
```

Middleware activated by MiddlewareFactory:

C#

```
public class FactoryActivatedMiddleware : IMiddleware
{
    private readonly SampleDbContext _dbContext;

    public FactoryActivatedMiddleware(SampleDbContext dbContext)
        => _dbContext = dbContext;

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _dbContext.Requests.Add(new Request("Factory", keyValue));

            await _dbContext.SaveChangesAsync();
        }

        await next(context);
    }
}
```

Extensions are created for the middleware:

C#

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseConventionalMiddleware(
        this IApplicationBuilder app)
        => app.UseMiddleware<ConventionalMiddleware>();

    public static IApplicationBuilder UseFactoryActivatedMiddleware(
        this IApplicationBuilder app)
        => app.UseMiddleware<FactoryActivatedMiddleware>();
}
```

It isn't possible to pass objects to the factory-activated middleware with UseMiddleware:

```
C#

public static IApplicationBuilder UseFactoryActivatedMiddleware(
    this IApplicationBuilder app, bool option)
{
    // Passing 'option' as an argument throws a NotSupportedException at
runtime.
    return app.UseMiddleware<FactoryActivatedMiddleware>(option);
}
```

The factory-activated middleware is added to the built-in container in `Program.cs`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<SampleDbContext>
    (options => options.UseInMemoryDatabase("SampleDb"));

builder.Services.AddTransient<FactoryActivatedMiddleware>();
```

Both middleware are registered in the request processing pipeline, also in `Program.cs`:

```
C#

var app = builder.Build();

app.UseConventionalMiddleware();
app.UseFactoryActivatedMiddleware();
```

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware. The middleware factory implementation is registered in the container as a scoped service.

The default IMiddlewareFactory implementation, MiddlewareFactory, is found in the Microsoft.AspNetCore.Http package.

Additional resources

- View or download sample code (how to download)
- ASP.NET Core Middleware
- Middleware activation with a third-party container in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Middleware activation with a third-party container in ASP.NET Core

Article • 07/26/2024

This article demonstrates how to use IMiddlewareFactory and IMiddleware as an extensibility point for middleware activation with a third-party container. For introductory information on `IMiddlewareFactory` and `IMiddleware`, see Factory-based middleware activation in ASP.NET Core.

View or download sample code (how to download)

The sample app demonstrates middleware activation by an `IMiddlewareFactory` implementation, `SimpleInjectorMiddlewareFactory`. The sample uses the Simple Injector dependency injection (DI) container.

The sample's middleware implementation records the value provided by a query string parameter (`key`). The middleware uses an injected database context (a scoped service) to record the query string value in an in-memory database.

> **Note**
>
> The sample app uses Simple Injector purely for demonstration purposes. Use of Simple Injector isn't an endorsement. Middleware activation approaches described in the Simple Injector documentation and GitHub issues are recommended by the maintainers of Simple Injector. For more information, see the Simple Injector documentation and Simple Injector GitHub repository.

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware.

In the sample app, a middleware factory is implemented to create a `SimpleInjectorActivatedMiddleware` instance. The middleware factory uses the Simple Injector container to resolve the middleware:

```
C#

public class SimpleInjectorMiddlewareFactory : IMiddlewareFactory
{
    private readonly Container _container;
```

```
    public SimpleInjectorMiddlewareFactory(Container container)
    {
        _container = container;
    }

    public IMiddleware Create(Type middlewareType)
    {
        return _container.GetInstance(middlewareType) as IMiddleware;
    }

    public void Release(IMiddleware middleware)
    {
        // The container is responsible for releasing resources.
    }
}
```

IMiddleware

IMiddleware defines middleware for the app's request pipeline.

Middleware activated by an `IMiddlewareFactory` implementation (`Middleware/SimpleInjectorActivatedMiddleware.cs`):

C#

```
public class SimpleInjectorActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public SimpleInjectorActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation =
"SimpleInjectorActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }
```

```
            await next(context);
        }
    }
```

An extension is created for the middleware (`Middleware/MiddlewareExtensions.cs`):

C#

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleInjectorActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleInjectorActivatedMiddleware>();
    }
}
```

`Startup.ConfigureServices` must perform several tasks:

- Set up the Simple Injector container.
- Register the factory and middleware.
- Make the app's database context available from the Simple Injector container.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    // Replace the default middleware factory with the
    // SimpleInjectorMiddlewareFactory.
    services.AddTransient<IMiddlewareFactory>(_ =>
    {
        return new SimpleInjectorMiddlewareFactory(_container);
    });

    // Wrap ASP.NET Core requests in a Simple Injector execution
    // context.
    services.UseSimpleInjectorAspNetRequestScoping(_container);

    // Provide the database context from the Simple
    // Injector container whenever it's requested from
    // the default service container.
    services.AddScoped<AppDbContext>(provider =>
        _container.GetInstance<AppDbContext>());

    _container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

    _container.Register<AppDbContext>(() =>
    {
        var optionsBuilder = new DbContextOptionsBuilder<DbContext>();
```

```
            optionsBuilder.UseInMemoryDatabase("InMemoryDb");
            return new AppDbContext(optionsBuilder.Options);
        }, Lifestyle.Scoped);

        _container.Register<SimpleInjectorActivatedMiddleware>();

        _container.Verify();
    }
```

The middleware is registered in the request processing pipeline in `Startup.Configure`:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseSimpleInjectorActivatedMiddleware();

    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Additional resources

- Middleware
- Factory-based middleware activation
- Simple Injector GitHub repository
- Simple Injector documentation

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

WebApplication and WebApplicationBuilder in Minimal API apps

Article • 10/25/2023

`WebApplication`

The following code is generated by an ASP.NET Core template:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding code can be created via `dotnet new web` on the command line or selecting the Empty Web template in Visual Studio.

The following code creates a WebApplication (`app`) without explicitly creating a WebApplicationBuilder:

```
C#

var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run();
```

WebApplication.Create initializes a new instance of the WebApplication class with preconfigured defaults.

Working with ports

When a web app is created with Visual Studio or `dotnet new`, a `Properties/launchSettings.json` file is created that specifies the ports the app responds to. In the port setting samples that follow, running the app from Visual Studio returns an

error dialog `Unable to connect to web server 'AppName'`. Visual Studio returns an error because it's expecting the port specified in `Properties/launchSettings.json`, but the app is using the port specified by `app.Run("http://localhost:3000")`. Run the following port changing samples from the command line.

The following sections set the port the app responds to.

C#

```
var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run("http://localhost:3000");
```

In the preceding code, the app responds to port `3000`.

Multiple ports

In the following code, the app responds to port `3000` and `4000`.

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://localhost:3000");
app.Urls.Add("http://localhost:4000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Set the port from the command line

The following command makes the app respond to port `7777`:

.NET CLI

```
dotnet run --urls="https://localhost:7777"
```

If the Kestrel endpoint is also configured in the `appsettings.json` file, the `appsettings.json` file specified URL is used. For more information, see Kestrel endpoint configuration

Read the port from environment

The following code reads the port from the environment:

```
C#
```

```
var app = WebApplication.Create(args);

var port = Environment.GetEnvironmentVariable("PORT") ?? "3000";

app.MapGet("/", () => "Hello World");

app.Run($"http://localhost:{port}");
```

The preferred way to set the port from the environment is to use the `ASPNETCORE_URLS` environment variable, which is shown in the following section.

Set the ports via the ASPNETCORE_URLS environment variable

The `ASPNETCORE_URLS` environment variable is available to set the port:

```

ASPNETCORE_URLS=http://localhost:3000
```

`ASPNETCORE_URLS` supports multiple URLs:

```

ASPNETCORE_URLS=http://localhost:3000;https://localhost:5000
```

Listen on all interfaces

The following samples demonstrate listening on all interfaces

http://*:3000

```
C#
```

```
var app = WebApplication.Create(args);

app.Urls.Add("http://*:3000");

app.MapGet("/", () => "Hello World");
```

```
app.Run();
```

http://+:3000

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://+:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

`http://0.0.0.0:3000`

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://0.0.0.0:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Listen on all interfaces using ASPNETCORE_URLS

The preceding samples can use `ASPNETCORE_URLS`

```
ASPNETCORE_URLS=http://*:3000;https://+:5000;http://0.0.0.0:5005
```

Listen on all interfaces using ASPNETCORE_HTTPS_PORTS

The preceding samples can use `ASPNETCORE_HTTPS_PORTS` and `ASPNETCORE_HTTP_PORTS`.

```
ASPNETCORE_HTTP_PORTS=3000;5005
ASPNETCORE_HTTPS_PORTS=5000
```

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server

Specify HTTPS with development certificate

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

For more information on the development certificate, see Trust the ASP.NET Core HTTPS development certificate on Windows and macOS.

Specify HTTPS using a custom certificate

The following sections show how to specify the custom certificate using the `appsettings.json` file and via configuration.

Specify the custom certificate with `appsettings.json`

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Kestrel": {
    "Certificates": {
      "Default": {
        "Path": "cert.pem",
        "KeyPath": "key.pem"
      }
    }
  }
}
```

Specify the custom certificate via configuration

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure the cert and the key
builder.Configuration["Kestrel:Certificates:Default:Path"] = "cert.pem";
builder.Configuration["Kestrel:Certificates:Default:KeyPath"] = "key.pem";

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Use the certificate APIs

C#

```
using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>
{
    options.ConfigureHttpsDefaults(httpsOptions =>
    {
        var certPath = Path.Combine(builder.Environment.ContentRootPath,
"cert.pem");
        var keyPath = Path.Combine(builder.Environment.ContentRootPath,
"key.pem");

        httpsOptions.ServerCertificate =
X509Certificate2.CreateFromPemFile(certPath,
                                           keyPath);
    });
});

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Read the environment

C#

```
var app = WebApplication.Create(args);

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/oops");
}

app.MapGet("/", () => "Hello World");
app.MapGet("/oops", () => "Oops! An error happened.");

app.Run();
```

For more information using the environment, see Use multiple environments in ASP.NET Core

Configuration

The following code reads from the configuration system:

C#

```
var app = WebApplication.Create(args);

var message = app.Configuration["HelloKey"] ?? "Config failed!";

app.MapGet("/", () => message);

app.Run();
```

For more information, see Configuration in ASP.NET Core

Logging

The following code writes a message to the log on application startup:

C#

```
var app = WebApplication.Create(args);

app.Logger.LogInformation("The app started");

app.MapGet("/", () => "Hello World");

app.Run();
```

For more information, see Logging in .NET Core and ASP.NET Core

Access the Dependency Injection (DI) container

The following code shows how to get services from the DI container during application startup:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddScoped<SampleService>();

var app = builder.Build();

app.MapControllers();

using (var scope = app.Services.CreateScope())
{
    var sampleService =
scope.ServiceProvider.GetRequiredService<SampleService>();
    sampleService.DoSomething();
}

app.Run();
```

The following code shows how to access keys from the DI container using the [FromKeyedServices] attribute:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");

var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));

app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>
smallCache.Get("date"));

app.Run();

public interface ICache
{
```

```
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

For more information on DI, see Dependency injection in ASP.NET Core.

WebApplicationBuilder

This section contains sample code using WebApplicationBuilder.

Change the content root, application name, and environment

The following code sets the content root, application name, and environment:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    ApplicationName = typeof(Program).Assembly.FullName,
    ContentRootPath = Directory.GetCurrentDirectory(),
    EnvironmentName = Environments.Staging,
    WebRootPath = "customwwwroot"
});

Console.WriteLine($"Application Name:
{builder.Environment.ApplicationName}");
Console.WriteLine($"Environment Name:
{builder.Environment.EnvironmentName}");
Console.WriteLine($"ContentRoot Path:
{builder.Environment.ContentRootPath}");
Console.WriteLine($"WebRootPath: {builder.Environment.WebRootPath}");

var app = builder.Build();
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults.

For more information, see ASP.NET Core fundamentals overview

Change the content root, app name, and environment by using environment variables or command line

The following table shows the environment variable and command-line argument used to change the content root, app name, and environment:

Expand table

feature	Environment variable	Command-line argument
Application name	ASPNETCORE_APPLICATIONNAME	--applicationName
Environment name	ASPNETCORE_ENVIRONMENT	--environment
Content root	ASPNETCORE_CONTENTROOT	--contentRoot

Add configuration providers

The following sample adds the INI configuration provider:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddIniFile("appsettings.ini");

var app = builder.Build();
```

For detailed information, see File configuration providers in Configuration in ASP.NET Core.

Read configuration

By default the WebApplicationBuilder reads configuration from multiple sources, including:

- `appSettings.json` and `appSettings.{environment}.json`
- Environment variables
- The command line

For a complete list of configuration sources read, see Default configuration in Configuration in ASP.NET Core.

The following code reads `HelloKey` from configuration and displays the value at the `/` endpoint. If the configuration value is null, "Hello" is assigned to `message`:

C#

```
var builder = WebApplication.CreateBuilder(args);

var message = builder.Configuration["HelloKey"] ?? "Hello";

var app = builder.Build();

app.MapGet("/", () => message);

app.Run();
```

Read the environment

C#

```
var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsDevelopment())
{
    Console.WriteLine($"Running in development.");
}

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Add logging providers

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure JSON logging to the console.
builder.Logging.AddJsonConsole();

var app = builder.Build();

app.MapGet("/", () => "Hello JSON console!");

app.Run();
```

Add services

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add the memory cache services.
builder.Services.AddMemoryCache();

// Add a custom scoped service.
builder.Services.AddScoped<ITodoRepository, TodoRepository>();
var app = builder.Build();
```

Customize the IHostBuilder

Existing extension methods on IHostBuilder can be accessed using the Host property:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Wait 30 seconds for graceful shutdown.
builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout =
TimeSpan.FromSeconds(30));

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Customize the IWebHostBuilder

Extension methods on IWebHostBuilder can be accessed using the WebApplicationBuilder.WebHost property.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Change the HTTP server implemenation to be HTTP.sys based
builder.WebHost.UseHttpSys();

var app = builder.Build();

app.MapGet("/", () => "Hello HTTP.sys");
```

```
app.Run();
```

Change the web root

By default, the web root is relative to the content root in the `wwwroot` folder. Web root is where the static files middleware looks for static files. Web root can be changed with `WebHostOptions`, the command line, or with the UseWebRoot method:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in webroot
    WebRootPath = "webroot"
});

var app = builder.Build();

app.Run();
```

Custom dependency injection (DI) container

The following example uses Autofac:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());

// Register services directly with Autofac here. Don't
// call builder.Populate(), that happens in AutofacServiceProviderFactory.
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
builder.RegisterModule(new MyApplicationModule()));

var app = builder.Build();
```

Add Middleware

Any existing ASP.NET Core middleware can be configured on the `WebApplication`:

C#

```
var app = WebApplication.Create(args);

// Setup the file server to serve static files.
app.UseFileServer();

app.MapGet("/", () => "Hello World!");

app.Run();
```

For more information, see ASP.NET Core Middleware

Developer exception page

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The developer exception page is enabled in the preconfigured defaults. When the following code is run in the development environment, navigating to `/` renders a friendly page that shows the exception.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () =>
{
    throw new InvalidOperationException("Oops, the '/' route has thrown an exception.");
});

app.Run();
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

.NET Generic Host in ASP.NET Core

Article • 08/29/2024

This article provides information on using the .NET Generic Host in ASP.NET Core.

The ASP.NET Core templates create a WebApplicationBuilder and WebApplication, which provide a streamlined way to configure and run web applications without a `Startup` class. For more information on `WebApplicationBuilder` and `WebApplication`, see Migrate from ASP.NET Core 5.0 to 6.0.

For information on using the .NET Generic Host in console apps, see .NET Generic Host.

Host definition

A *host* is an object that encapsulates an app's resources, such as:

- Dependency injection (DI)
- Logging
- Configuration
- `IHostedService` implementations

When a host starts, it calls IHostedService.StartAsync on each implementation of IHostedService registered in the service container's collection of hosted services. In a web app, one of the `IHostedService` implementations is a web service that starts an HTTP server implementation.

Including all of the app's interdependent resources in one object enables control over app startup and graceful shutdown.

Set up a host

The host is typically configured, built, and run by code in the `Program.cs`. The following code creates a host with an `IHostedService` implementation added to the DI container:

C#

```
await Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddHostedService<SampleHostedService>();
    })
```

```
    .Build()
    .RunAsync();
```

For an HTTP workload, call ConfigureWebHostDefaults after CreateDefaultBuilder:

C#

```
await Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    })
    .Build()
    .RunAsync();
```

Default builder settings

The CreateDefaultBuilder method:

- Sets the content root to the path returned by GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with `DOTNET_`.
  - Command-line arguments.
- Loads app configuration from:
  - `appsettings.json`.
  - `appsettings.{Environment}.json`.
  - User secrets when the app runs in the `Development` environment.
  - Environment variables.
  - Command-line arguments.
- Adds the following logging providers:
  - Console
  - Debug
  - EventSource
  - EventLog (only when running on Windows)
- Enables scope validation and dependency validation when the environment is Development.

The ConfigureWebHostDefaults method:

- Loads host configuration from environment variables prefixed with `ASPNETCORE_`.
- Sets Kestrel server as the web server and configures it using the app's hosting configuration providers. For the Kestrel server's default options, see Configure options for the ASP.NET Core Kestrel web server.

- Adds Host Filtering middleware.
- Adds Forwarded Headers middleware if `ASPNETCORE_FORWARDEDHEADERS_ENABLED` equals `true`.
- Enables IIS integration. For the IIS default options, see Host ASP.NET Core on Windows with IIS.

The Settings for all app types and Settings for web apps sections later in this article show how to override default builder settings.

Framework-provided services

The following services are registered automatically:

- IHostApplicationLifetime
- IHostLifetime
- IHostEnvironment / IWebHostEnvironment

For more information on framework-provided services, see Dependency injection in ASP.NET Core.

IHostApplicationLifetime

Inject the IHostApplicationLifetime (formerly `IApplicationLifetime`) service into any class to handle post-startup and graceful shutdown tasks. Three properties on the interface are cancellation tokens used to register app start and app stop event handler methods. The interface also includes a `StopApplication` method, which allows apps to request a graceful shutdown.

When performing a graceful shutdown, the host:

- Triggers the ApplicationStopping event handlers, which allows the app to run logic before the shutdown process begins.
- Stops the server, which disables new connections. The server waits for requests on existing connections to complete, for as long as the shutdown timeout allows. The server sends the connection close header for further requests on existing connections.
- Triggers the ApplicationStopped event handlers, which allows the app to run logic after the application has shutdown.

The following example is an `IHostedService` implementation that registers `IHostApplicationLifetime` event handlers:

C#

```
public class HostApplicationLifetimeEventsHostedService : IHostedService
{
    private readonly IHostApplicationLifetime _hostApplicationLifetime;

    public HostApplicationLifetimeEventsHostedService(
        IHostApplicationLifetime hostApplicationLifetime)
        => _hostApplicationLifetime = hostApplicationLifetime;

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _hostApplicationLifetime.ApplicationStarted.Register(OnStarted);
        _hostApplicationLifetime.ApplicationStopping.Register(OnStopping);
        _hostApplicationLifetime.ApplicationStopped.Register(OnStopped);

        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
        => Task.CompletedTask;

    private void OnStarted()
    {
        // ...
    }

    private void OnStopping()
    {
        // ...
    }

    private void OnStopped()
    {
        // ...
    }
}
```

IHostLifetime

The IHostLifetime implementation controls when the host starts and when it stops. The last implementation registered is used.

`Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` is the default `IHostLifetime` implementation. `ConsoleLifetime`:

- Listens for `Ctrl`+`C`/SIGINT (Windows), `⌘`+`C` (macOS), or SIGTERM and calls StopApplication to start the shutdown process.
- Unblocks extensions such as RunAsync and WaitForShutdownAsync.

IHostEnvironment

Inject the IHostEnvironment service into a class to get information about the following settings:

- ApplicationName
- EnvironmentName
- ContentRootPath

Web apps implement the `IWebHostEnvironment` interface, which inherits `IHostEnvironment` and adds the WebRootPath.

Host configuration

Host configuration is used for the properties of the IHostEnvironment implementation.

Host configuration is available from HostBuilderContext.Configuration inside ConfigureAppConfiguration. After `ConfigureAppConfiguration`, `HostBuilderContext.Configuration` is replaced with the app config.

To add host configuration, call ConfigureHostConfiguration on `IHostBuilder`. `ConfigureHostConfiguration` can be called multiple times with additive results. The host uses whichever option sets a value last on a given key.

The environment variable provider with prefix `DOTNET_` and command-line arguments are included by `CreateDefaultBuilder`. For web apps, the environment variable provider with prefix `ASPNETCORE_` is added. The prefix is removed when the environment variables are read. For example, the environment variable value for `ASPNETCORE_ENVIRONMENT` becomes the host configuration value for the `environment` key.

The following example creates host configuration:

C#

```
Host.CreateDefaultBuilder(args)
    .ConfigureHostConfiguration(hostConfig =>
    {
        hostConfig.SetBasePath(Directory.GetCurrentDirectory());
        hostConfig.AddJsonFile("hostsettings.json", optional: true);
        hostConfig.AddEnvironmentVariables(prefix: "PREFIX_");
        hostConfig.AddCommandLine(args);
    });
```

App configuration

App configuration is created by calling ConfigureAppConfiguration on `IHostBuilder`. `ConfigureAppConfiguration` can be called multiple times with additive results. The app uses whichever option sets a value last on a given key.

The configuration created by `ConfigureAppConfiguration` is available at HostBuilderContext.Configuration for subsequent operations and as a service from DI. The host configuration is also added to the app configuration.

For more information, see Configuration in ASP.NET Core.

Settings for all app types

This section lists host settings that apply to both HTTP and non-HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix, which appear in the following list of settings as the `{PREFIX_}` placeholder. For more information, see the Default builder settings section and Configuration: Environment variables.

ApplicationName

The IHostEnvironment.ApplicationName property is set from host configuration during host construction.

Key: `applicationName`
Type: `string`
Default: The name of the assembly that contains the app's entry point.
Environment variable: `{PREFIX_}APPLICATIONNAME`

To set this value, use the environment variable.

ContentRoot

The IHostEnvironment.ContentRootPath property determines where the host begins searching for content files. If the path doesn't exist, the host fails to start.

Key: `contentRoot`
Type: `string`
Default: The folder where the app assembly resides.
Environment variable: `{PREFIX_}CONTENTROOT`

To set this value, use the environment variable or call `UseContentRoot` on `IHostBuilder`:

```
C#

Host.CreateDefaultBuilder(args)
    .UseContentRoot("/path/to/content/root")
    // ...
```

For more information, see:

- Fundamentals: Content root
- WebRoot

EnvironmentName

The IHostEnvironment.EnvironmentName property can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case-sensitive.

Key: `environment`
Type: `string`
Default: `Production`
Environment variable: `{PREFIX_}ENVIRONMENT`

To set this value, use the environment variable or call `UseEnvironment` on `IHostBuilder`:

```
C#

Host.CreateDefaultBuilder(args)
    .UseEnvironment("Development")
    // ...
```

ShutdownTimeout

HostOptions.ShutdownTimeout sets the timeout for StopAsync. The default value is 30 seconds. During the timeout period, the host:

- Triggers IHostApplicationLifetime.ApplicationStopping.
- Attempts to stop hosted services, logging errors for services that fail to stop.

If the timeout period expires before all of the hosted services stop, any remaining active services are stopped when the app shuts down. The services stop even if they haven't finished processing. If services require more time to stop, increase the timeout.

Key: `shutdownTimeoutSeconds`
Type: `int`
Default: 30 seconds
Environment variable: `{PREFIX_}SHUTDOWNTIMEOUTSECONDS`

To set this value, use the environment variable or configure `HostOptions`. The following example sets the timeout to 20 seconds:

C#

```
Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
    {
        services.Configure<HostOptions>(options =>
        {
            options.ShutdownTimeout = TimeSpan.FromSeconds(20);
        });
    });
```

Disable app configuration reload on change

By default, `appsettings.json` and `appsettings.{Environment}.json` are reloaded when the file changes. To disable this reload behavior in ASP.NET Core 5.0 or later, set the `hostBuilder:reloadConfigOnChange` key to `false`.

Key: `hostBuilder:reloadConfigOnChange`
Type: `bool` (`true` or `false`)
Default: `true`
Command-line argument: `hostBuilder:reloadConfigOnChange`
Environment variable: `{PREFIX_}hostBuilder:reloadConfigOnChange`

> ⚠ **Warning**
>
> The colon (`:`) separator doesn't work with environment variable hierarchical keys on all platforms. For more information, see Environment variables.

Settings for web apps

Some host settings apply only to HTTP workloads. By default, environment variables used to configure these settings can have a `DOTNET_` or `ASPNETCORE_` prefix, which appear in the following list of settings as the `{PREFIX_}` placeholder.

Extension methods on `IWebHostBuilder` are available for these settings. Code samples that show how to call the extension methods assume `webBuilder` is an instance of `IWebHostBuilder`, as in the following example:

```
C#

Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
        // ...
    });
```

CaptureStartupErrors

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

Key: `captureStartupErrors`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.
Environment variable: `{PREFIX_}CAPTURESTARTUPERRORS`

To set this value, use configuration or call `CaptureStartupErrors`:

```
C#

webBuilder.CaptureStartupErrors(true);
```

DetailedErrors

When enabled, or when the environment is `Development`, the app captures detailed errors.

Key: `detailedErrors`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}DETAILEDERRORS`

To set this value, use configuration or call `UseSetting`:

```
C#
```

```
webBuilder.UseSetting(WebHostDefaults.DetailedErrorsKey, "true");
```

HostingStartupAssemblies

A semicolon-delimited string of hosting startup assemblies to load on startup. Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

Key: `hostingStartupAssemblies`
Type: `string`
Default: Empty string
Environment variable: `{PREFIX_}HOSTINGSTARTUPASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

C#

```
webBuilder.UseSetting(
    WebHostDefaults.HostingStartupAssembliesKey, "assembly1;assembly2");
```

HostingStartupExcludeAssemblies

A semicolon-delimited string of hosting startup assemblies to exclude on startup.

Key: `hostingStartupExcludeAssemblies`
Type: `string`
Default: Empty string
Environment variable: `{PREFIX_}HOSTINGSTARTUPEXCLUDEASSEMBLIES`

To set this value, use configuration or call `UseSetting`:

C#

```
webBuilder.UseSetting(
    WebHostDefaults.HostingStartupExcludeAssembliesKey,
"assembly1;assembly2");
```

HTTPS_Port

Set the HTTPS port to redirect to if you get a non-HTTPS connection. Used in enforcing HTTPS. This setting doesn't cause the server to listen on the specified port. That is, it's possible to accidentally redirect requests to an unused port.

Key: `https_port` **Type**: `string`
Default: A default value isn't set.
Environment variable: `{PREFIX_}HTTPS_PORT`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting("https_port", "8080");
```

HTTPS_Ports

The ports to listen on for HTTPS connections.

Key: `https_ports`
Type: `string`
Default: A default value isn't set.
Environment variable: `{PREFIX_}HTTPS_PORTS`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting("https_ports", "8080");
```

PreferHostingUrls

Indicates whether the host should listen on the URLs configured with the `IWebHostBuilder` instead of those URLs configured with the `IServer` implementation.

Key: `preferHostingUrls`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}PREFERHOSTINGURLS`

To set this value, use the environment variable or call `PreferHostingUrls`:

```
C#
```

```
webBuilder.PreferHostingUrls(true);
```

PreventHostingStartup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. For more information, see Use hosting startup assemblies in ASP.NET Core.

Key: `preventHostingStartup`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}PREVENTHOSTINGSTARTUP`

To set this value, use the environment variable or call `UseSetting` :

C#

```
webBuilder.UseSetting(WebHostDefaults.PreventHostingStartupKey, "true");
```

StartupAssembly

The assembly to search for the `Startup` class.

Key: `startupAssembly`
Type: `string`
Default: The app's assembly
Environment variable: `{PREFIX_}STARTUPASSEMBLY`

To set this value, use the environment variable or call `UseStartup`. `UseStartup` can take an assembly name (`string`) or a type (`TStartup`). If multiple `UseStartup` methods are called, the last one takes precedence.

C#

```
webBuilder.UseStartup("StartupAssemblyName");
```

C#

```
webBuilder.UseStartup<Startup>();
```

SuppressStatusMessages

When enabled, suppresses hosting startup status messages.

Key: `suppressStatusMessages`
Type: `bool` (`true`/`1` or `false`/`0`)
Default: `false`
Environment variable: `{PREFIX_}SUPPRESSSTATUSMESSAGES`

To set this value, use configuration or call `UseSetting`:

```
C#

webBuilder.UseSetting(WebHostDefaults.SuppressStatusMessagesKey, "true");
```

URLs

A semicolon-delimited list of IP addresses or host addresses with ports and protocols that the server should listen on for requests. For example, `http://localhost:123`. Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary among servers.

Key: `urls`
Type: `string`
Default: `http://localhost:5000` and `https://localhost:5001`
Environment variable: `{PREFIX_}URLS`

To set this value, use the environment variable or call `UseUrls`:

```
C#

webBuilder.UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002");
```

Kestrel has its own endpoint configuration API. For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

WebRoot

The IWebHostEnvironment.WebRootPath property determines the relative path to the app's static assets. If the path doesn't exist, a no-op file provider is used.

Key: `webroot`
Type: `string`
Default: The default is `wwwroot`. The path to *{content root}/wwwroot* must exist.
Environment variable: `{PREFIX_}WEBROOT`

To set this value, use the environment variable or call `UseWebRoot` on `IWebHostBuilder`:

```
C#

webBuilder.UseWebRoot("public");
```

For more information, see:

- Fundamentals: Web root
- ContentRoot

Manage the host lifetime

Call methods on the built IHost implementation to start and stop the app. These methods affect all IHostedService implementations that are registered in the service container.

The difference between `Run*` and `Start*` methods is that `Run*` methods wait for the host to complete before returning, whereas `Start*` methods return immediately. The `Run*` methods are typically used in console apps, whereas the `Start*` methods are typically used in long-running services.

Run

Run runs the app and blocks the calling thread until the host is shut down.

RunAsync

RunAsync runs the app and returns a Task that completes when the cancellation token or shutdown is triggered.

RunConsoleAsync

RunConsoleAsync enables console support, builds and starts the host, and waits for Ctrl+C/SIGINT (Windows), ⌘+C (macOS), or SIGTERM to shut down.

Start

Start starts the host synchronously.

StartAsync

StartAsync starts the host and returns a Task that completes when the cancellation token or shutdown is triggered.

WaitForStartAsync is called at the start of `StartAsync`, which waits until it's complete before continuing. This method can be used to delay startup until signaled by an external event.

StopAsync

StopAsync attempts to stop the host within the provided timeout.

WaitForShutdown

WaitForShutdown blocks the calling thread until shutdown is triggered by the IHostLifetime, such as via Ctrl+C/SIGINT (Windows), ⌘+C (macOS), or SIGTERM.

WaitForShutdownAsync

WaitForShutdownAsync returns a Task that completes when shutdown is triggered via the given token and calls StopAsync.

Additional resources

- Background tasks with hosted services in ASP.NET Core
- GitHub link to Generic Host source

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags**

dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

ASP.NET Core Web Host

Article • 08/29/2024

ASP.NET Core apps configure and launch a *host*. The host is responsible for app startup and lifetime management. At a minimum, the host configures a server and a request processing pipeline. The host can also set up logging, dependency injection, and configuration.

This article covers the Web Host, which remains available only for backward compatibility. The ASP.NET Core templates create a WebApplicationBuilder and WebApplication, which is recommended for web apps. For more information on `WebApplicationBuilder` and `WebApplication`, see Migrate from ASP.NET Core 5.0 to 6.0

Set up a host

Create a host using an instance of IWebHostBuilder. This is typically performed in the app's entry point, the `Main` method in `Program.cs`. A typical app calls CreateDefaultBuilder to start setting up a host:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

The code that calls `CreateDefaultBuilder` is in a method named `CreateWebHostBuilder`, which separates it from the code in `Main` that calls `Run` on the builder object. This separation is required if you use Entity Framework Core tools. The tools expect to find a `CreateWebHostBuilder` method that they can call at design time to configure the host without running the app. An alternative is to implement `IDesignTimeDbContextFactory`. For more information, see Design-time DbContext Creation.

`CreateDefaultBuilder` performs the following tasks:

- Configures Kestrel server as the web server using the app's hosting configuration providers. For the Kestrel server's default options, see Configure options for the ASP.NET Core Kestrel web server.
- Sets the content root to the path returned by Directory.GetCurrentDirectory.
- Loads host configuration from:
  - Environment variables prefixed with `ASPNETCORE_` (for example, `ASPNETCORE_ENVIRONMENT`).
  - Command-line arguments.
- Loads app configuration in the following order from:
  - `appsettings.json`.
  - `appsettings.{Environment}.json`.
  - User secrets when the app runs in the `Development` environment using the entry assembly.
  - Environment variables.
  - Command-line arguments.
- Configures logging for console and debug output. Logging includes log filtering rules specified in a Logging configuration section of an `appsettings.json` or `appsettings.{Environment}.json` file.
- When running behind IIS with the ASP.NET Core Module, `CreateDefaultBuilder` enables IIS Integration, which configures the app's base address and port. IIS Integration also configures the app to capture startup errors. For the IIS default options, see Host ASP.NET Core on Windows with IIS.
- Sets ServiceProviderOptions.ValidateScopes to `true` if the app's environment is Development. For more information, see Scope validation.

The configuration defined by `CreateDefaultBuilder` can be overridden and augmented by ConfigureAppConfiguration, ConfigureLogging, and other methods and extension methods of IWebHostBuilder. A few examples follow:

- ConfigureAppConfiguration is used to specify additional `IConfiguration` for the app. The following `ConfigureAppConfiguration` call adds a delegate to include app configuration in the `appsettings.xml` file. `ConfigureAppConfiguration` may be called multiple times. Note that this configuration doesn't apply to the host (for example, server URLs or environment). See the Host configuration values section.

  C#

  ```
  WebHost.CreateDefaultBuilder(args)
      .ConfigureAppConfiguration((hostingContext, config) =>
      {
          config.AddXmlFile("appsettings.xml", optional: true,
  reloadOnChange: true);
  ```

```
    })
    ...
```

- The following `ConfigureLogging` call adds a delegate to configure the minimum logging level (SetMinimumLevel) to LogLevel.Warning. This setting overrides the settings in `appsettings.Development.json` (`LogLevel.Debug`) and `appsettings.Production.json` (`LogLevel.Error`) configured by `CreateDefaultBuilder`. `ConfigureLogging` may be called multiple times.

  C#

  ```
  WebHost.CreateDefaultBuilder(args)
      .ConfigureLogging(logging =>
      {
          logging.SetMinimumLevel(LogLevel.Warning);
      })
      ...
  ```

- The following call to `ConfigureKestrel` overrides the default Limits.MaxRequestBodySize of 30,000,000 bytes established when Kestrel was configured by `CreateDefaultBuilder`:

  C#

  ```
  WebHost.CreateDefaultBuilder(args)
      .ConfigureKestrel((context, options) =>
      {
          options.Limits.MaxRequestBodySize = 20000000;
      });
  ```

The content root determines where the host searches for content files, such as MVC view files. When the app is started from the project's root folder, the project's root folder is used as the content root. This is the default used in Visual Studio and the dotnet new templates.

For more information on app configuration, see Configuration in ASP.NET Core.

> **Note**
>
> As an alternative to using the static `CreateDefaultBuilder` method, creating a host from WebHostBuilder is a supported approach with ASP.NET Core 2.x.

When setting up a host, Configure and ConfigureServices methods can be provided. If a `Startup` class is specified, it must define a `Configure` method. For more information, see App startup in ASP.NET Core. Multiple calls to `ConfigureServices` append to one another. Multiple calls to `Configure` or `UseStartup` on the `WebHostBuilder` replace previous settings.

Host configuration values

WebHostBuilder relies on the following approaches to set the host configuration values:

- Host builder configuration, which includes environment variables with the format `ASPNETCORE_{configurationKey}`. For example, `ASPNETCORE_ENVIRONMENT`.
- Extensions such as UseContentRoot and UseConfiguration (see the Override configuration section).
- UseSetting and the associated key. When setting a value with `UseSetting`, the value is set as a string regardless of the type.

The host uses whichever option sets a value last. For more information, see Override configuration in the next section.

Application Key (Name)

The `IWebHostEnvironment.ApplicationName` property is automatically set when UseStartup or Configure is called during host construction. The value is set to the name of the assembly containing the app's entry point. To set the value explicitly, use the WebHostDefaults.ApplicationKey:

Key: applicationName
Type: *string*
Default: The name of the assembly containing the app's entry point.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_APPLICATIONNAME`

```
C#
```

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.ApplicationKey, "CustomApplicationName")
```

Capture Startup Errors

This setting controls the capture of startup errors.

Key: captureStartupErrors
Type: *bool* (`true` or `1`)
Default: Defaults to `false` unless the app runs with Kestrel behind IIS, where the default is `true`.
Set using: `CaptureStartupErrors`
Environment variable: `ASPNETCORE_CAPTURESTARTUPERRORS`

When `false`, errors during startup result in the host exiting. When `true`, the host captures exceptions during startup and attempts to start the server.

```
C#

WebHost.CreateDefaultBuilder(args)
    .CaptureStartupErrors(true)
```

Content root

This setting determines where ASP.NET Core begins searching for content files.

Key: contentRoot
Type: *string*
Default: Defaults to the folder where the app assembly resides.
Set using: `UseContentRoot`
Environment variable: `ASPNETCORE_CONTENTROOT`

The content root is also used as the base path for the web root. If the content root path doesn't exist, the host fails to start.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseContentRoot("c:\\<content-root>")
```

For more information, see:

- Fundamentals: Content root
- Web root

Detailed Errors

Determines if detailed errors should be captured.

Key: detailedErrors
Type: *bool* (`true` or `1`)
Default: false
Set using: `UseSetting`
Environment variable: `ASPNETCORE_DETAILEDERRORS`

When enabled (or when the Environment is set to `Development`), the app captures detailed exceptions.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.DetailedErrorsKey, "true")
```

Environment

Sets the app's environment.

Key: environment
Type: *string*
Default: Production
Set using: `UseEnvironment`
Environment variable: `ASPNETCORE_ENVIRONMENT`

The environment can be set to any value. Framework-defined values include `Development`, `Staging`, and `Production`. Values aren't case sensitive. By default, the *Environment* is read from the `ASPNETCORE_ENVIRONMENT` environment variable. When using Visual Studio, environment variables may be set in the `launchSettings.json` file. For more information, see Use multiple environments in ASP.NET Core.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseEnvironment(EnvironmentName.Development)
```

Hosting Startup Assemblies

Sets the app's hosting startup assemblies.

Key: hostingStartupAssemblies
Type: *string*
Default: Empty string

Set using: `UseSetting`
Environment variable: `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`

A semicolon-delimited string of hosting startup assemblies to load on startup.

Although the configuration value defaults to an empty string, the hosting startup assemblies always include the app's assembly. When hosting startup assemblies are provided, they're added to the app's assembly for loading when the app builds its common services during startup.

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupAssembliesKey,
"assembly1;assembly2")
```

HTTPS Port

Set the HTTPS port to redirect to if you get a non-HTTPS connection. Used in enforcing HTTPS. This setting doesn't cause the server to listen on the specified port. That is, it's possible to accidentally redirect requests to an unused port.

Key: https_port
Type: *string*
Default: A default value isn't set.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HTTPS_PORT`

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting("https_port", "8080")
```

HTTPS Ports

Set the ports to listen on for HTTPS connections.

Key: https_ports **Type**: *string*
Default: A default value isn't set.
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HTTPS_PORTS`

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting("https_ports", "8080")
```

Hosting Startup Exclude Assemblies

A semicolon-delimited string of hosting startup assemblies to exclude on startup.

Key: hostingStartupExcludeAssemblies
Type: *string*
Default: Empty string
Set using: `UseSetting`
Environment variable: `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES`

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.HostingStartupExcludeAssembliesKey,
"assembly1;assembly2")
```

Prefer Hosting URLs

Indicates whether the host should listen on the URLs configured with the `WebHostBuilder` instead of those configured with the `IServer` implementation.

Key: preferHostingUrls
Type: *bool* (`true` or `1`)
Default: false
Set using: `PreferHostingUrls`
Environment variable: `ASPNETCORE_PREFERHOSTINGURLS`

C#

```
WebHost.CreateDefaultBuilder(args)
    .PreferHostingUrls(true)
```

Prevent Hosting Startup

Prevents the automatic loading of hosting startup assemblies, including hosting startup assemblies configured by the app's assembly. For more information, see Use hosting

startup assemblies in ASP.NET Core.

Key: preventHostingStartup
Type: *bool* (`true` or `1`)
Default: false
Set using: `UseSetting`
Environment variable: `ASPNETCORE_PREVENTHOSTINGSTARTUP`

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseSetting(WebHostDefaults.PreventHostingStartupKey, "true")
```

Server URLs

Indicates the IP addresses or host addresses with ports and protocols that the server should listen on for requests.

Key: urls
Type: *string*
Default: http://localhost:5000
Set using: `UseUrls`
Environment variable: `ASPNETCORE_URLS`

Set to a semicolon-separated (;) list of URL prefixes to which the server should respond. For example, `http://localhost:123`. Use "*" to indicate that the server should listen for requests on any IP address or hostname using the specified port and protocol (for example, `http://*:5000`). The protocol (`http://` or `https://`) must be included with each URL. Supported formats vary among servers.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")
```

Kestrel has its own endpoint configuration API. For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

Shutdown Timeout

Specifies the amount of time to wait for Web Host to shut down.

Key: shutdownTimeoutSeconds
Type: *int*
Default: 5
Set using: `UseShutdownTimeout`
Environment variable: `ASPNETCORE_SHUTDOWNTIMEOUTSECONDS`

Although the key accepts an *int* with `UseSetting` (for example, `.UseSetting(WebHostDefaults.ShutdownTimeoutKey, "10")`), the UseShutdownTimeout extension method takes a TimeSpan.

During the timeout period, hosting:

- Triggers IApplicationLifetime.ApplicationStopping.
- Attempts to stop hosted services, logging any errors for services that fail to stop.

If the timeout period expires before all of the hosted services stop, any remaining active services are stopped when the app shuts down. The services stop even if they haven't finished processing. If services require additional time to stop, increase the timeout.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
```

Startup Assembly

Determines the assembly to search for the `Startup` class.

Key: startupAssembly
Type: *string*
Default: The app's assembly
Set using: `UseStartup`
Environment variable: `ASPNETCORE_STARTUPASSEMBLY`

The assembly by name (`string`) or type (`TStartup`) can be referenced. If multiple `UseStartup` methods are called, the last one takes precedence.

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseStartup("StartupAssemblyName")
```

```
C#
```

```
WebHost.CreateDefaultBuilder(args)
    .UseStartup<TStartup>()
```

Web root

Sets the relative path to the app's static assets.

Key: webroot
Type: *string*
Default: The default is `wwwroot`. The path to *{content root}/wwwroot* must exist. If the path doesn't exist, a no-op file provider is used.
Set using: `UseWebRoot`
Environment variable: `ASPNETCORE_WEBROOT`

```
C#

WebHost.CreateDefaultBuilder(args)
    .UseWebRoot("public")
```

For more information, see:

- Fundamentals: Web root
- Content root

Override configuration

Use Configuration to configure Web Host. In the following example, host configuration is optionally specified in a `hostsettings.json` file. Any configuration loaded from the `hostsettings.json` file may be overridden by command-line arguments. The built configuration (in `config`) is used to configure the host with UseConfiguration. `IWebHostBuilder` configuration is added to the app's configuration, but the converse isn't true—`ConfigureAppConfiguration` doesn't affect the `IWebHostBuilder` configuration.

Overriding the configuration provided by `UseUrls` with `hostsettings.json` config first, command-line argument config second:

```
C#

public class Program
{
    public static void Main(string[] args)
    {
```

```
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args)
        {
            var config = new ConfigurationBuilder()
                .SetBasePath(Directory.GetCurrentDirectory())
                .AddJsonFile("hostsettings.json", optional: true)
                .AddCommandLine(args)
                .Build();

            return WebHost.CreateDefaultBuilder(args)
                .UseUrls("http://*:5000")
                .UseConfiguration(config)
                .Configure(app =>
                {
                    app.Run(context =>
                        context.Response.WriteAsync("Hello, World!"));
                });
        }
    }
```

`hostsettings.json`:

JSON

```
{
    urls: "http://*:5005"
}
```

> **Note**
>
> UseConfiguration only copies keys from the provided `IConfiguration` to the host builder configuration. Therefore, setting `reloadOnChange: true` for JSON, INI, and XML settings files has no effect.

To specify the host run on a particular URL, the desired value can be passed in from a command prompt when executing dotnet run. The command-line argument overrides the `urls` value from the `hostsettings.json` file, and the server listens on port 8080:

.NET CLI

```
dotnet run --urls "http://*:8080"
```

Manage the host

Run

The `Run` method starts the web app and blocks the calling thread until the host is shut down:

```
C#

host.Run();
```

Start

Run the host in a non-blocking manner by calling its `Start` method:

```
C#

using (host)
{
    host.Start();
    Console.ReadLine();
}
```

If a list of URLs is passed to the `Start` method, it listens on the URLs specified:

```
C#

var urls = new List<string>()
{
    "http://*:5000",
    "http://localhost:5001"
};

var host = new WebHostBuilder()
    .UseKestrel()
    .UseStartup<Startup>()
    .Start(urls.ToArray());

using (host)
{
    Console.ReadLine();
}
```

The app can initialize and start a new host using the pre-configured defaults of `CreateDefaultBuilder` using a static convenience method. These methods start the server without console output and with WaitForShutdown wait for a break (Ctrl-C/SIGINT or SIGTERM):

Start(RequestDelegate app)

Start with a `RequestDelegate`:

```
C#

using (var host = WebHost.Start(app => app.Response.WriteAsync("Hello,
World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!" `WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, RequestDelegate app)

Start with a URL and `RequestDelegate`:

```
C#

using (var host = WebHost.Start("http://localhost:8080", app =>
app.Response.WriteAsync("Hello, World!")))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **Start(RequestDelegate app)**, except the app responds on `http://localhost:8080`.

Start(Action<IRouteBuilder> routeBuilder)

Use an instance of `IRouteBuilder` (Microsoft.AspNetCore.Routing) to use routing middleware:

```
C#

using (var host = WebHost.Start(router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]},
{data.Values["name"]}!"))
```

```
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shutdown the host...");
    host.WaitForShutdown();
}
```

Use the following browser requests with the example:

Expand table

Request	Response
`http://localhost:5000/hello/Martin`	Hello, Martin!
`http://localhost:5000/buenosdias/Catrina`	Buenos dias, Catrina!
`http://localhost:5000/throw/ooops!`	Throws an exception with string "ooops!"
`http://localhost:5000/throw`	Throws an exception with string "Uh oh!"
`http://localhost:5000/Sante/Kevin`	Sante, Kevin!
`http://localhost:5000`	Hello World!

`WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

Start(string url, Action<IRouteBuilder> routeBuilder)

Use a URL and an instance of `IRouteBuilder`:

C#

```
using (var host = WebHost.Start("http://localhost:8080", router => router
    .MapGet("hello/{name}", (req, res, data) =>
        res.WriteAsync($"Hello, {data.Values["name"]}!"))
    .MapGet("buenosdias/{name}", (req, res, data) =>
        res.WriteAsync($"Buenos dias, {data.Values["name"]}!"))
    .MapGet("throw/{message?}", (req, res, data) =>
        throw new Exception((string)data.Values["message"] ?? "Uh oh!"))
    .MapGet("{greeting}/{name}", (req, res, data) =>
        res.WriteAsync($"{data.Values["greeting"]},
{data.Values["name"]}!"))
    .MapGet("", (req, res, data) => res.WriteAsync("Hello, World!"))))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **Start(Action<IRouteBuilder> routeBuilder)**, except the app responds at `http://localhost:8080`.

StartWith(Action<IApplicationBuilder> app)

Provide a delegate to configure an `IApplicationBuilder`:

```
C#
```

```
using (var host = WebHost.StartWith(app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        };
    })))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Make a request in the browser to `http://localhost:5000` to receive the response "Hello World!" `WaitForShutdown` blocks until a break (Ctrl-C/SIGINT or SIGTERM) is issued. The app displays the `Console.WriteLine` message and waits for a keypress to exit.

StartWith(string url, Action<IApplicationBuilder> app)

Provide a URL and a delegate to configure an `IApplicationBuilder`:

```
C#
```

```
using (var host = WebHost.StartWith("http://localhost:8080", app =>
    app.Use(next =>
    {
        return async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        };
    })))
{
    Console.WriteLine("Use Ctrl-C to shut down the host...");
    host.WaitForShutdown();
}
```

Produces the same result as **StartWith(Action<IApplicationBuilder> app)**, except the app responds on `http://localhost:8080`.

IWebHostEnvironment interface

The `IWebHostEnvironment` interface provides information about the app's web hosting environment. Use constructor injection to obtain the `IWebHostEnvironment` in order to use its properties and extension methods:

C#

```
public class CustomFileReader
{
    private readonly IWebHostEnvironment _env;

    public CustomFileReader(IWebHostEnvironment env)
    {
        _env = env;
    }

    public string ReadFile(string filePath)
    {
        var fileProvider = _env.WebRootFileProvider;
        // Process the file here
    }
}
```

A convention-based approach can be used to configure the app at startup based on the environment. Alternatively, inject the `IWebHostEnvironment` into the `Startup` constructor for use in `ConfigureServices`:

C#

```
public class Startup
{
    public Startup(IWebHostEnvironment env)
    {
        HostingEnvironment = env;
    }

    public IWebHostEnvironment HostingEnvironment { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        if (HostingEnvironment.IsDevelopment())
        {
            // Development configuration
        }
        else
        {
            // Staging/Production configuration
        }
```

```
        var contentRootPath = HostingEnvironment.ContentRootPath;
    }
}
```

> **Note**
>
> In addition to the `IsDevelopment` extension method, `IWebHostEnvironment` offers `IsStaging`, `IsProduction`, and `IsEnvironment(string environmentName)` methods. For more information, see Use multiple environments in ASP.NET Core.

The `IWebHostEnvironment` service can also be injected directly into the `Configure` method for setting up the processing pipeline:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        // In Development, use the Developer Exception Page
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // In Staging/Production, route exceptions to /error
        app.UseExceptionHandler("/error");
    }

    var contentRootPath = env.ContentRootPath;
}
```

`IWebHostEnvironment` can be injected into the `Invoke` method when creating custom middleware:

C#

```
public async Task Invoke(HttpContext context, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        // Configure middleware for Development
    }
    else
    {
        // Configure middleware for Staging/Production
    }
```

```
    var contentRootPath = env.ContentRootPath;
}
```

IHostApplicationLifetime interface

`IHostApplicationLifetime` allows for post-startup and shutdown activities. Three properties on the interface are cancellation tokens used to register `Action` methods that define startup and shutdown events.

Expand table

Cancellation Token	Triggered when...
`ApplicationStarted`	The host has fully started.
`ApplicationStopped`	The host is completing a graceful shutdown. All requests should be processed. Shutdown blocks until this event completes.
`ApplicationStopping`	The host is performing a graceful shutdown. Requests may still be processing. Shutdown blocks until this event completes.

C#

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IHostApplicationLifetime
appLifetime)
    {
        appLifetime.ApplicationStarted.Register(OnStarted);
        appLifetime.ApplicationStopping.Register(OnStopping);
        appLifetime.ApplicationStopped.Register(OnStopped);

        Console.CancelKeyPress += (sender, eventArgs) =>
        {
            appLifetime.StopApplication();
            // Don't terminate the process immediately, wait for the Main
thread to exit gracefully.
            eventArgs.Cancel = true;
        };
    }

    private void OnStarted()
    {
        // Perform post-startup activities here
    }

    private void OnStopping()
    {
        // Perform on-stopping activities here
```

```
    }

    private void OnStopped()
    {
        // Perform post-stopped activities here
    }
}
```

`StopApplication` requests termination of the app. The following class uses `StopApplication` to gracefully shut down an app when the class's `Shutdown` method is called:

C#

```
public class MyClass
{
    private readonly IHostApplicationLifetime _appLifetime;

    public MyClass(IHostApplicationLifetime appLifetime)
    {
        _appLifetime = appLifetime;
    }

    public void Shutdown()
    {
        _appLifetime.StopApplication();
    }
}
```

Scope validation

CreateDefaultBuilder sets ServiceProviderOptions.ValidateScopes to `true` if the app's environment is Development.

When `ValidateScopes` is set to `true`, the default service provider performs checks to verify that:

- Scoped services aren't directly or indirectly resolved from the root service provider.
- Scoped services aren't directly or indirectly injected into singletons.

The root service provider is created when BuildServiceProvider is called. The root service provider's lifetime corresponds to the app/server's lifetime when the provider starts with the app and is disposed when the app shuts down.

Scoped services are disposed by the container that created them. If a scoped service is created in the root container, the service's lifetime is effectively promoted to singleton

because it's only disposed by the root container when app/server is shut down. Validating service scopes catches these situations when `BuildServiceProvider` is called.

To always validate scopes, including in the Production environment, configure the ServiceProviderOptions with UseDefaultServiceProvider on the host builder:

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseDefaultServiceProvider((context, options) => {
        options.ValidateScopes = true;
    })
```

Additional resources

- Host ASP.NET Core on Windows with IIS
- Host ASP.NET Core on Linux with Nginx
- Host ASP.NET Core in a Windows Service

Configuration in ASP.NET Core

Article • 04/26/2024

By Rick Anderson and Kirk Larkin

Application configuration in ASP.NET Core is performed using one or more configuration providers. Configuration providers read configuration data from key-value pairs using a variety of configuration sources:

- Settings files, such as `appsettings.json`
- Environment variables
- Azure Key Vault
- Azure App Configuration
- Command-line arguments
- Custom providers, installed or created
- Directory files
- In-memory .NET objects

This article provides information on configuration in ASP.NET Core. For information on using configuration in console apps, see .NET Configuration.

For Blazor configuration guidance, which adds to or supersedes the guidance in this node, see ASP.NET Core Blazor configuration.

Application and Host Configuration

ASP.NET Core apps configure and launch a *host*. The host is responsible for app startup and lifetime management. The ASP.NET Core templates create a WebApplicationBuilder which contains the host. While some configuration can be done in both the host and the application configuration providers, generally, only configuration that is necessary for the host should be done in host configuration.

Application configuration is the highest priority and is detailed in the next section. Host configuration follows application configuration, and is described in this article.

Default application configuration sources

ASP.NET Core web apps created with dotnet new or Visual Studio generate the following code:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The initialized `WebApplicationBuilder` (`builder`) provides default configuration for the app in the following order, from highest to lowest priority:

1. Command-line arguments using the Command-line configuration provider.
2. Non-prefixed environment variables using the Non-prefixed environment variables configuration provider.
3. User secrets when the app runs in the `Development` environment.
4. `appsettings.{Environment}.json` using the JSON configuration provider. For example, `appsettings.Production.json` and `appsettings.Development.json`.
5. appsettings.json using the JSON configuration provider.
6. A fallback to the host configuration described in the next section.

Default host configuration sources

The following list contains the default host configuration sources from highest to lowest priority for WebApplicationBuilder:

1. Command-line arguments using the Command-line configuration provider
2. `DOTNET_`-prefixed environment variables using the Environment variables configuration provider.
3. `ASPNETCORE_`-prefixed environment variables using the Environment variables configuration provider.

For the .NET Generic Host and Web Host, the default host configuration sources from highest to lowest priority is:

1. `ASPNETCORE_`-prefixed environment variables using the Environment variables configuration provider.
2. Command-line arguments using the Command-line configuration provider
3. `DOTNET_`-prefixed environment variables using the Environment variables configuration provider.

When a configuration value is set in host and application configuration, the application configuration is used.

Host variables

The following variables are locked in early when initializing the host builders and can't be influenced by application config:

- Application name
- Environment name, for example `Development`, `Production`, and `Staging`
- Content root
- Web root
- Whether to scan for hosting startup assemblies and which assemblies to scan for.
- Variables read by app and library code from HostBuilderContext.Configuration in IHostBuilder.ConfigureAppConfiguration callbacks.

Every other host setting is read from application config instead of host config.

`URLS` is one of the many common host settings that is not a bootstrap setting. Like every other host setting not in the previous list, `URLS` is read later from application config. Host config is a fallback for application config, so host config can be used to set `URLS`, but it will be overridden by any configuration source in application config like `appsettings.json`.

For more information, see Change the content root, app name, and environment and Change the content root, app name, and environment by environment variables or command line

The remaining sections in this article refer to application configuration.

Application configuration providers

The following code displays the enabled configuration providers in the order they were added:

C#

```
public class Index2Model : PageModel
{
    private IConfigurationRoot ConfigRoot;

    public Index2Model(IConfiguration configRoot)
    {
        ConfigRoot = (IConfigurationRoot)configRoot;
    }

    public ContentResult OnGet()
    {
        string str = "";
        foreach (var provider in ConfigRoot.Providers.ToList())
        {
```

```
            str += provider.ToString() + "\n";
        }

        return Content(str);
    }
}
```

The preceding list of highest to lowest priority default configuration sources shows the providers in the opposite order they are added to template generated application. For example, the JSON configuration provider is added before the Command-line configuration provider.

Configuration providers that are added later have higher priority and override previous key settings. For example, if `MyKey` is set in both `appsettings.json` and the environment, the environment value is used. Using the default configuration providers, the Command-line configuration provider overrides all other providers.

For more information on `CreateBuilder`, see Default builder settings.

`appsettings.json`

Consider the following `appsettings.json` file:

JSON

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "MyKey": "My appsettings.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

The following code from the sample download displays several of the preceding configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

The default JsonConfigurationProvider loads configuration in the following order:

1. `appsettings.json`
2. `appsettings.{Environment}.json` : For example, the `appsettings.Production.json` and `appsettings.Development.json` files. The environment version of the file is loaded based on the IHostingEnvironment.EnvironmentName. For more information, see Use multiple environments in ASP.NET Core.

`appsettings.{Environment}.json` values override keys in `appsettings.json`. For example, by default:

- In development, `appsettings.Development.json` configuration overwrites values found in `appsettings.json`.
- In production, `appsettings.Production.json` configuration overwrites values found in `appsettings.json`. For example, when deploying the app to Azure.

If a configuration value must be guaranteed, see GetValue. The preceding example only reads strings and doesn't support a default value.

Using the default configuration, the `appsettings.json` and `appsettings.{Environment}.json` files are enabled with reloadOnChange: true. Changes made to

the `appsettings.json` and `appsettings.{Environment}.json` file ***after*** the app starts are read by the JSON configuration provider.

Comments in appsettings.json

Comments in `appsettings.json` and `appsettings.{Environment}.json` files are supported using JavaScript or C# style comments.

Bind hierarchical configuration data using the options pattern

The preferred way to read related configuration values is using the options pattern. For example, to read the following configuration values:

JSON

```
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  }
```

Create the following `PositionOptions` class:

C#

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; } = String.Empty;
    public string Name { get; set; } = String.Empty;
}
```

An options class:

- Must be non-abstract with a public parameterless constructor.
- All public read-write properties of the type are bound.
- Fields are ***not*** bound. In the preceding code, `Position` is not bound. The `Position` field is used so the string `"Position"` doesn't need to be hard coded in the app when binding the class to a configuration provider.

The following code:

- Calls ConfigurationBinder.Bind to bind the `PositionOptions` class to the `Position` section.
- Displays the `Position` configuration data.

C#

```
public class Test22Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test22Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var positionOptions = new PositionOptions();

Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

ConfigurationBinder.Get<T> binds and returns the specified type. `ConfigurationBinder.Get<T>` may be more convenient than using `ConfigurationBinder.Bind`. The following code shows how to use `ConfigurationBinder.Get<T>` with the `PositionOptions` class:

C#

```
public class Test21Model : PageModel
{
    private readonly IConfiguration Configuration;
    public PositionOptions? positionOptions { get; private set; }

    public Test21Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        positionOptions = Configuration.GetSection(PositionOptions.Position)
                                                    .Get<PositionOptions>
```

```
();

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

An alternative approach when using the ***options pattern*** is to bind the `Position` section and add it to the dependency injection service container. In the following code, `PositionOptions` is added to the service container with Configure and bound to configuration:

C#

```
using ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));

var app = builder.Build();
```

Using the preceding code, the following code reads the position options:

C#

```
public class Test2Model : PageModel
{
    private readonly PositionOptions _options;

    public Test2Model(IOptions<PositionOptions> options)
    {
        _options = options.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Title: {_options.Title} \n" +
                       $"Name: {_options.Name}");
    }
}
```

In the preceding code, changes to the JSON configuration file after the app has started are ***not*** read. To read changes after the app has started, use IOptionsSnapshot.

Using the default configuration, the `appsettings.json` and `appsettings.{Environment}.json` files are enabled with reloadOnChange: true. Changes made to the `appsettings.json` and `appsettings.{Environment}.json` file ***after*** the app starts are read by the JSON configuration provider.

See JSON configuration provider in this document for information on adding additional JSON configuration files.

Combining service collection

Consider the following which registers services and configures options:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));
builder.Services.Configure<ColorOptions>(
    builder.Configuration.GetSection(ColorOptions.Color));

builder.Services.AddScoped<IMyDependency, MyDependency>();
builder.Services.AddScoped<IMyDependency2, MyDependency2>();

var app = builder.Build();
```

Related groups of registrations can be moved to an extension method to register services. For example, the configuration services are added to the following class:

C#

```
using ConfigSample.Options;
using Microsoft.Extensions.Configuration;

namespace Microsoft.Extensions.DependencyInjection
{
    public static class MyConfigServiceCollectionExtensions
    {
        public static IServiceCollection AddConfig(
```

```
             this IServiceCollection services, IConfiguration config)
        {
            services.Configure<PositionOptions>(
                config.GetSection(PositionOptions.Position));
            services.Configure<ColorOptions>(
                config.GetSection(ColorOptions.Color));

            return services;
        }

        public static IServiceCollection AddMyDependencyGroup(
             this IServiceCollection services)
        {
            services.AddScoped<IMyDependency, MyDependency>();
            services.AddScoped<IMyDependency2, MyDependency2>();

            return services;
        }
    }
}
```

The remaining services are registered in a similar class. The following code uses the new extension methods to register the services:

C#

```
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddConfig(builder.Configuration)
    .AddMyDependencyGroup();

builder.Services.AddRazorPages();

var app = builder.Build();
```

Note: Each `services.Add{GROUP_NAME}` extension method adds and potentially configures services. For example, AddControllersWithViews adds the services MVC controllers with views require, and AddRazorPages adds the services Razor Pages requires.

Security and user secrets

Configuration data guidelines:

- Never store passwords or other sensitive data in configuration provider code or in plain text configuration files. The Secret Manager tool can be used to store secrets

in development.
- Don't use production secrets in development or test environments.
- Specify secrets outside of the project so that they can't be accidentally committed to a source code repository.
- Production apps should use the most secure authentication flow available. For more information, see Secure authentication flows.

By default, the user secrets configuration source is registered after the JSON configuration sources. Therefore, user secrets keys take precedence over keys in `appsettings.json` and `appsettings.{Environment}.json`.

For more information on storing passwords or other sensitive data:

- Use multiple environments in ASP.NET Core
- Safe storage of app secrets in development in ASP.NET Core: Includes advice on using environment variables to store sensitive data. The Secret Manager tool uses the File configuration provider to store user secrets in a JSON file on the local system.
- Azure Key Vault safely stores app secrets for ASP.NET Core apps. For more information, see Azure Key Vault configuration provider in ASP.NET Core.

Non-prefixed environment variables

Non-prefixed environment variables are environment variables other than those prefixed by `ASPNETCORE_` or `DOTNET_`. For example, the ASP.NET Core web application templates set `"ASPNETCORE_ENVIRONMENT": "Development"` in `launchSettings.json`. For more information on `ASPNETCORE_` and `DOTNET_` environment variables, see:

- List of highest to lowest priority default configuration sources including non-prefixed, `ASPNETCORE_`-prefixed and `DOTNETCORE_`-prefixed environment variables.
- DOTNET_ environment variables used outside of Microsoft.Extensions.Hosting.

Using the default configuration, the EnvironmentVariablesConfigurationProvider loads configuration from environment variable key-value pairs after reading `appsettings.json`, `appsettings.{Environment}.json`, and user secrets. Therefore, key values read from the environment override values read from `appsettings.json`, `appsettings.{Environment}.json`, and user secrets.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

The following commands:

- Set the environment keys and values of the preceding example on Windows.
- Test the settings when using the sample download. The `dotnet run` command must be run in the project directory.

.NET CLI

```
set MyKey="My key from Environment"
set Position__Title=Environment_Editor
set Position__Name=Environment_Rick
dotnet run
```

The preceding environment settings:

- Are only set in processes launched from the command window they were set in.
- Won't be read by browsers launched with Visual Studio.

The following setx commands can be used to set the environment keys and values on Windows. Unlike `set`, `setx` settings are persisted. `/M` sets the variable in the system environment. If the `/M` switch isn't used, a user environment variable is set.

Console

```
setx MyKey "My key from setx Environment" /M
setx Position__Title Environment_Editor /M
setx Position__Name Environment_Rick /M
```

To test that the preceding commands override `appsettings.json` and `appsettings.{Environment}.json`:

- With Visual Studio: Exit and restart Visual Studio.
- With the CLI: Start a new command window and enter `dotnet run`.

Call AddEnvironmentVariables with a string to specify a prefix for environment variables:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Configuration.AddEnvironmentVariables(prefix: "MyCustomPrefix_");
```

```
var app = builder.Build();
```

In the preceding code:

- `builder.Configuration.AddEnvironmentVariables(prefix: "MyCustomPrefix_")` is added after the default configuration providers. For an example of ordering the configuration providers, see JSON configuration provider.
- Environment variables set with the `MyCustomPrefix_` prefix override the default configuration providers. This includes environment variables without the prefix.

The prefix is stripped off when the configuration key-value pairs are read.

The following commands test the custom prefix:

.NET CLI

```
set MyCustomPrefix_MyKey="My key with MyCustomPrefix_ Environment"
set MyCustomPrefix_Position__Title=Editor_with_customPrefix
set MyCustomPrefix_Position__Name=Environment_Rick_cp
dotnet run
```

The default configuration loads environment variables and command line arguments prefixed with `DOTNET_` and `ASPNETCORE_`. The `DOTNET_` and `ASPNETCORE_` prefixes are used by ASP.NET Core for host and app configuration, but not for user configuration. For more information on host and app configuration, see .NET Generic Host.

On Azure App Service, select **New application setting** on the **Settings** > **Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

For more information, see Azure Apps: Override app configuration using the Azure Portal.

See Connection string prefixes for information on Azure database connection strings.

Naming of environment variables

Environment variable names reflect the structure of an `appsettings.json` file. Each element in the hierarchy is separated by a double underscore (preferable) or a colon. When the element structure includes an array, the array index should be treated as an

additional element name in this path. Consider the following `appsettings.json` file and its equivalent values represented as environment variables.

appsettings.json

JSON

```
{
    "SmtpServer": "smtp.example.com",
    "Logging": [
        {
            "Name": "ToEmail",
            "Level": "Critical",
            "Args": {
                "FromAddress": "MySystem@example.com",
                "ToAddress": "SRE@example.com"
            }
        },
        {
            "Name": "ToConsole",
            "Level": "Information"
        }
    ]
}
```

environment variables

Console

```
setx SmtpServer smtp.example.com
setx Logging__0__Name ToEmail
setx Logging__0__Level Critical
setx Logging__0__Args__FromAddress MySystem@example.com
setx Logging__0__Args__ToAddress SRE@example.com
setx Logging__1__Name ToConsole
setx Logging__1__Level Information
```

Environment variables set in generated launchSettings.json

Environment variables set in `launchSettings.json` override those set in the system environment. For example, the ASP.NET Core web templates generate a `launchSettings.json` file that sets the endpoint configuration to:

JSON

```
"applicationUrl": "https://localhost:5001;http://localhost:5000"
```

Configuring the `applicationUrl` sets the `ASPNETCORE_URLS` environment variable and overrides values set in the environment.

Escape environment variables on Linux

On Linux, the value of URL environment variables must be escaped so `systemd` can parse it. Use the linux tool `systemd-escape` which yields `http:--localhost:5001`

Windows Command Prompt

```
groot@terminus:~$ systemd-escape http://localhost:5001
http:--localhost:5001
```

Display environment variables

The following code displays the environment variables and values on application startup, which can be helpful when debugging environment settings:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

foreach (var c in builder.Configuration.AsEnumerable())
{
    Console.WriteLine(c.Key + " = " + c.Value);
}
```

Command-line

Using the default configuration, the CommandLineConfigurationProvider loads configuration from command-line argument key-value pairs after the following configuration sources:

- `appsettings.json` and `appsettings.{Environment}.json` files.
- App secrets in the Development environment.
- Environment variables.

By default, configuration values set on the command-line override configuration values set with all the other configuration providers.

Command-line arguments

The following command sets keys and values using `=`:

.NET CLI

```
dotnet run MyKey="Using =" Position:Title=Cmd Position:Name=Cmd_Rick
```

The following command sets keys and values using `/`:

.NET CLI

```
dotnet run /MyKey "Using /" /Position:Title=Cmd /Position:Name=Cmd_Rick
```

The following command sets keys and values using `--`:

.NET CLI

```
dotnet run --MyKey "Using --" --Position:Title=Cmd --Position:Name=Cmd_Rick
```

The key value:

- Must follow `=`, or the key must have a prefix of `--` or `/` when the value follows a space.
- Isn't required if `=` is used. For example, `MySetting=`.

Within the same command, don't mix command-line argument key-value pairs that use `=` with key-value pairs that use a space.

Switch mappings

Switch mappings allow **key** name replacement logic. Provide a dictionary of switch replacements to the AddCommandLine method.

When the switch mappings dictionary is used, the dictionary is checked for a key that matches the key provided by a command-line argument. If the command-line key is found in the dictionary, the dictionary value is passed back to set the key-value pair into the app's configuration. A switch mapping is required for any command-line key prefixed with a single dash (`-`).

Switch mappings dictionary key rules:

- Switches must start with `-` or `--`.

- The switch mappings dictionary must not contain duplicate keys.

To use a switch mappings dictionary, pass it into the call to `AddCommandLine`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var switchMappings = new Dictionary<string, string>()
         {
             { "-k1", "key1" },
             { "-k2", "key2" },
             { "--alt3", "key3" },
             { "--alt4", "key4" },
             { "--alt5", "key5" },
             { "--alt6", "key6" },
         };

builder.Configuration.AddCommandLine(args, switchMappings);

var app = builder.Build();
```

Run the following command works to test key replacement:

.NET CLI

```
dotnet run -k1 value1 -k2 value2 --alt3=value2 /alt4=value3 --alt5 value5
/alt6 value6
```

The following code shows the key values for the replaced keys:

C#

```
public class Test3Model : PageModel
{
    private readonly IConfiguration Config;

    public Test3Model(IConfiguration configuration)
    {
        Config = configuration;
    }

    public ContentResult OnGet()
    {
        return Content(
                $"Key1: '{Config["Key1"]}'\n" +
                $"Key2: '{Config["Key2"]}'\n" +
```

```
                $"Key3: '{Config["Key3"]}'\n" +
                $"Key4: '{Config["Key4"]}'\n" +
                $"Key5: '{Config["Key5"]}'\n" +
                $"Key6: '{Config["Key6"]}'");
    }
}
```

For apps that use switch mappings, the call to `CreateDefaultBuilder` shouldn't pass arguments. The `CreateDefaultBuilder` method's `AddCommandLine` call doesn't include mapped switches, and there's no way to pass the switch-mapping dictionary to `CreateDefaultBuilder`. The solution isn't to pass the arguments to `CreateDefaultBuilder` but instead to allow the `ConfigurationBuilder` method's `AddCommandLine` method to process both the arguments and the switch-mapping dictionary.

Set environment and command-line arguments with Visual Studio

Environment and command-line arguments can be set in Visual Studio from the launch profiles dialog:

- In Solution Explorer, right click the project and select **Properties**.
- Select the **Debug** > **General** tab and select **Open debug launch profiles UI**.

Hierarchical configuration data

The Configuration API reads hierarchical configuration data by flattening the hierarchical data with the use of a delimiter in the configuration keys.

The sample download contains the following `appsettings.json` file:

JSON

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "MyKey": "My appsettings.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
```

```
  "AllowedHosts": "*"
}
```

The following code from the sample download displays several of the configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

The preferred way to read hierarchical configuration data is using the options pattern. For more information, see Bind hierarchical configuration data in this document.

GetSection and GetChildren methods are available to isolate sections and children of a section in the configuration data. These methods are described later in GetSection, GetChildren, and Exists.

Configuration keys and values

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC)

is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

Configuration keys:

- Are case-insensitive. For example, `ConnectionString` and `connectionstring` are treated as equivalent keys.
- If a key and value is set in more than one configuration provider, the value from the last provider added is used. For more information, see Default configuration.
- Hierarchical keys
  - Within the Configuration API, a colon separator (`:`) works on all platforms.
  - In environment variables, a colon separator may not work on all platforms. A double underscore, `__`, is supported by all platforms and is automatically converted into a colon `:`.
  - In Azure Key Vault, hierarchical keys use `--` as a separator. The Azure Key Vault configuration provider automatically replaces `--` with a `:` when the secrets are loaded into the app's configuration.
- The ConfigurationBinder supports binding arrays to objects using array indices in configuration keys. Array binding is described in the Bind an array to a class section.

Configuration values:

- Are strings.
- Null values can't be stored in configuration or bound to objects.

Configuration providers

The following table shows the configuration providers available to ASP.NET Core apps.

Expand table

Provider	Provides configuration from
Azure Key Vault configuration provider	Azure Key Vault
Azure App configuration provider	Azure App Configuration
Command-line configuration provider	Command-line parameters
Custom configuration provider	Custom source

Provider	Provides configuration from
Environment Variables configuration provider	Environment variables
File configuration provider	INI, JSON, and XML files
Key-per-file configuration provider	Directory files
Memory configuration provider	In-memory collections
User secrets	File in the user profile directory

Configuration sources are read in the order that their configuration providers are specified. Order configuration providers in code to suit the priorities for the underlying configuration sources that the app requires.

A typical sequence of configuration providers is:

1. `appsettings.json`
2. `appsettings.{Environment}.json`
3. User secrets
4. Environment variables using the Environment Variables configuration provider.
5. Command-line arguments using the Command-line configuration provider.

A common practice is to add the Command-line configuration provider last in a series of providers to allow command-line arguments to override configuration set by the other providers.

The preceding sequence of providers is used in the default configuration.

Connection string prefixes

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

The Configuration API has special processing rules for four connection string environment variables. These connection strings are involved in configuring Azure

connection strings for the app environment. Environment variables with the prefixes shown in the table are loaded into the app with the default configuration or when no prefix is supplied to `AddEnvironmentVariables`.

Expand table

Connection string prefix	Provider
`CUSTOMCONNSTR_`	Custom provider
`MYSQLCONNSTR_`	MySQL
`SQLAZURECONNSTR_`	Azure SQL Database
`SQLCONNSTR_`	SQL Server

When an environment variable is discovered and loaded into configuration with any of the four prefixes shown in the table:

- The configuration key is created by removing the environment variable prefix and adding a configuration key section (`ConnectionStrings`).
- A new configuration key-value pair is created that represents the database connection provider (except for `CUSTOMCONNSTR_`, which has no stated provider).

Expand table

Environment variable key	Converted configuration key	Provider configuration entry
`CUSTOMCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Configuration entry not created.
`MYSQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` Value: `MySql.Data.MySqlClient`
`SQLAZURECONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` Value: `System.Data.SqlClient`
`SQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	Key: `ConnectionStrings:{KEY}_ProviderName` Value: `System.Data.SqlClient`

File configuration provider

FileConfigurationProvider is the base class for loading configuration from the file system. The following configuration providers derive from `FileConfigurationProvider`:

- INI configuration provider
- JSON configuration provider
- XML configuration provider

INI configuration provider

The IniConfigurationProvider loads configuration from INI file key-value pairs at runtime.

The following code adds several configuration providers:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration
    .AddIniFile("MyIniConfig.ini", optional: true, reloadOnChange: true)
    .AddIniFile($"MyIniConfig.{builder.Environment.EnvironmentName}.ini",
                optional: true, reloadOnChange: true);

builder.Configuration.AddEnvironmentVariables();
builder.Configuration.AddCommandLine(args);

builder.Services.AddRazorPages();

var app = builder.Build();
```

In the preceding code, settings in the `MyIniConfig.ini` and `MyIniConfig.{Environment}.ini` files are overridden by settings in the:

- Environment variables configuration provider
- Command-line configuration provider.

The sample download contains the following `MyIniConfig.ini` file:

ini

```
MyKey="MyIniConfig.ini Value"

[Position]
Title="My INI Config title"
Name="My INI Config name"

[Logging:LogLevel]
```

```
Default=Information
Microsoft=Warning
```

The following code from the sample download displays several of the preceding configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

JSON configuration provider

The JsonConfigurationProvider loads configuration from JSON file key-value pairs.

Overloads can specify:

- Whether the file is optional.
- Whether the configuration is reloaded if the file changes.

Consider the following code:

C#

```
using Microsoft.Extensions.DependencyInjection.ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Configuration.AddJsonFile("MyConfig.json",
        optional: true,
        reloadOnChange: true);

builder.Services.AddRazorPages();

var app = builder.Build();
```

The preceding code:

- Configures the JSON configuration provider to load the `MyConfig.json` file with the following options:
  - `optional: true`: The file is optional.
  - `reloadOnChange: true` : The file is reloaded when changes are saved.
- Reads the default configuration providers before the `MyConfig.json` file. Settings in the `MyConfig.json` file override setting in the default configuration providers, including the Environment variables configuration provider and the Command-line configuration provider.

You typically ***don't*** want a custom JSON file overriding values set in the Environment variables configuration provider and the Command-line configuration provider.

XML configuration provider

The XmlConfigurationProvider loads configuration from XML file key-value pairs at runtime.

The following code adds several configuration providers:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration
    .AddXmlFile("MyXMLFile.xml", optional: true, reloadOnChange: true)
    .AddXmlFile($"MyXMLFile.{builder.Environment.EnvironmentName}.xml",
                optional: true, reloadOnChange: true);

builder.Configuration.AddEnvironmentVariables();
builder.Configuration.AddCommandLine(args);

builder.Services.AddRazorPages();

var app = builder.Build();
```

In the preceding code, settings in the `MyXMLFile.xml` and `MyXMLFile.{Environment}.xml` files are overridden by settings in the:

- Environment variables configuration provider
- Command-line configuration provider.

The sample download contains the following `MyXMLFile.xml` file:

XML

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <MyKey>MyXMLFile Value</MyKey>
  <Position>
    <Title>Title from  MyXMLFile</Title>
    <Name>Name from MyXMLFile</Name>
  </Position>
  <Logging>
    <LogLevel>
      <Default>Information</Default>
      <Microsoft>Warning</Microsoft>
    </LogLevel>
  </Logging>
</configuration>
```

The following code from the sample download displays several of the preceding configurations settings:

C#

```csharp
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
```

```
                        $"Default Log Level: {defaultLogLevel}");
    }
}
```

Repeating elements that use the same element name work if the `name` attribute is used to distinguish the elements:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="section0">
    <key name="key0">value 00</key>
    <key name="key1">value 01</key>
  </section>
  <section name="section1">
    <key name="key0">value 10</key>
    <key name="key1">value 11</key>
  </section>
</configuration>
```

The following code reads the previous configuration file and displays the keys and values:

C#

```
public class IndexModel : PageModel
{
    private readonly IConfiguration Configuration;

    public IndexModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var key00 = "section:section0:key:key0";
        var key01 = "section:section0:key:key1";
        var key10 = "section:section1:key:key0";
        var key11 = "section:section1:key:key1";

        var val00 = Configuration[key00];
        var val01 = Configuration[key01];
        var val10 = Configuration[key10];
        var val11 = Configuration[key11];

        return Content($"{key00} value: {val00} \n" +
                       $"{key01} value: {val01} \n" +
                       $"{key10} value: {val10} \n" +
                       $"{key10} value: {val11} \n"
```

```
                        );
    }
}
```

Attributes can be used to supply values:

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
  </section>
</configuration>
```

The previous configuration file loads the following keys with `value`:

- key:attribute
- section:key:attribute

Key-per-file configuration provider

The KeyPerFileConfigurationProvider uses a directory's files as configuration key-value pairs. The key is the file name. The value contains the file's contents. The Key-per-file configuration provider is used in Docker hosting scenarios.

To activate key-per-file configuration, call the AddKeyPerFile extension method on an instance of ConfigurationBuilder. The `directoryPath` to the files must be an absolute path.

Overloads permit specifying:

- An `Action<KeyPerFileConfigurationSource>` delegate that configures the source.
- Whether the directory is optional and the path to the directory.

The double-underscore (`__`) is used as a configuration key delimiter in file names. For example, the file name `Logging__LogLevel__System` produces the configuration key `Logging:LogLevel:System`.

Call `ConfigureAppConfiguration` when building the host to specify the app's configuration:

C#

```
.ConfigureAppConfiguration((hostingContext, config) =>
{
    var path = Path.Combine(
        Directory.GetCurrentDirectory(), "path/to/files");
    config.AddKeyPerFile(directoryPath: path, optional: true);
})
```

Memory configuration provider

The MemoryConfigurationProvider uses an in-memory collection as configuration key-value pairs.

The following code adds a memory collection to the configuration system:

C#

```
var builder = WebApplication.CreateBuilder(args);

var Dict = new Dictionary<string, string>
        {
           {"MyKey", "Dictionary MyKey Value"},
           {"Position:Title", "Dictionary_Title"},
           {"Position:Name", "Dictionary_Name" },
           {"Logging:LogLevel:Default", "Warning"}
        };

builder.Configuration.AddInMemoryCollection(Dict);
builder.Configuration.AddEnvironmentVariables();
builder.Configuration.AddCommandLine(args);

builder.Services.AddRazorPages();

var app = builder.Build();
```

The following code from the sample download displays the preceding configurations settings:

C#

```
public class TestModel : PageModel
{
    // requires using Microsoft.Extensions.Configuration;
    private readonly IConfiguration Configuration;

    public TestModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }
```

```
    public ContentResult OnGet()
    {
        var myKeyValue = Configuration["MyKey"];
        var title = Configuration["Position:Title"];
        var name = Configuration["Position:Name"];
        var defaultLogLevel = Configuration["Logging:LogLevel:Default"];


        return Content($"MyKey value: {myKeyValue} \n" +
                       $"Title: {title} \n" +
                       $"Name: {name} \n" +
                       $"Default Log Level: {defaultLogLevel}");
    }
}
```

In the preceding code, `config.AddInMemoryCollection(Dict)` is added after the default configuration providers. For an example of ordering the configuration providers, see JSON configuration provider.

See Bind an array for another example using `MemoryConfigurationProvider`.

Kestrel endpoint configuration

Kestrel specific endpoint configuration overrides all cross-server endpoint configurations. Cross-server endpoint configurations include:

- UseUrls
- `--urls` on the command line
- The environment variable `ASPNETCORE_URLS`

Consider the following `appsettings.json` file used in an ASP.NET Core web app:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "Https": {
        "Url": "https://localhost:9999"
      }
    }
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
```

```
  },
  "AllowedHosts": "*"
}
```

When the preceding highlighted markup is used in an ASP.NET Core web app ***and*** the app is launched on the command line with the following cross-server endpoint configuration:

```
dotnet run --urls="https://localhost:7777"
```

Kestrel binds to the endpoint configured specifically for Kestrel in the `appsettings.json` file (`https://localhost:9999`) and not `https://localhost:7777`.

Consider the Kestrel specific endpoint configured as an environment variable:

```
set Kestrel__Endpoints__Https__Url=https://localhost:8888
```

In the preceding environment variable, `Https` is the name of the Kestrel specific endpoint. The preceding `appsettings.json` file also defines a Kestrel specific endpoint named `Https`. By default, environment variables using the Environment Variables configuration provider are read after `appsettings.{Environment}.json`, therefore, the preceding environment variable is used for the `Https` endpoint.

GetValue

ConfigurationBinder.GetValue extracts a single value from configuration with a specified key and converts it to the specified type:

C#

```
public class TestNumModel : PageModel
{
    private readonly IConfiguration Configuration;

    public TestNumModel(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var number = Configuration.GetValue<int>("NumberKey", 99);
        return Content($"{number}");
    }
}
```

In the preceding code, if `NumberKey` isn't found in the configuration, the default value of `99` is used.

GetSection, GetChildren, and Exists

For the examples that follow, consider the following `MySubsection.json` file:

JSON

```
{
  "section0": {
    "key0": "value00",
    "key1": "value01"
  },
  "section1": {
    "key0": "value10",
    "key1": "value11"
  },
  "section2": {
    "subsection0": {
      "key0": "value200",
      "key1": "value201"
    },
    "subsection1": {
      "key0": "value210",
      "key1": "value211"
    }
  }
}
```

The following code adds `MySubsection.json` to the configuration providers:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration
    .AddJsonFile("MySubsection.json",
                 optional: true,
                 reloadOnChange: true);

builder.Services.AddRazorPages();

var app = builder.Build();
```

GetSection

IConfiguration.GetSection returns a configuration subsection with the specified subsection key.

The following code returns values for `section1`:

C#

```
public class TestSectionModel : PageModel
{
    private readonly IConfiguration Config;

    public TestSectionModel(IConfiguration configuration)
    {
        Config = configuration.GetSection("section1");
    }

    public ContentResult OnGet()
    {
        return Content(
                $"section1:key0: '{Config["key0"]}'\n" +
                $"section1:key1: '{Config["key1"]}'");
    }
}
```

The following code returns values for `section2:subsection0`:

C#

```
public class TestSection2Model : PageModel
{
    private readonly IConfiguration Config;

    public TestSection2Model(IConfiguration configuration)
    {
        Config = configuration.GetSection("section2:subsection0");
    }

    public ContentResult OnGet()
    {
        return Content(
                $"section2:subsection0:key0 '{Config["key0"]}'\n" +
                $"section2:subsection0:key1:'{Config["key1"]}'");
    }
}
```

`GetSection` never returns `null`. If a matching section isn't found, an empty `IConfigurationSection` is returned.

When `GetSection` returns a matching section, Value isn't populated. A Key and Path are returned when the section exists.

GetChildren and Exists

The following code calls IConfiguration.GetChildren and returns values for `section2:subsection0`:

C#

```
public class TestSection4Model : PageModel
{
    private readonly IConfiguration Config;

    public TestSection4Model(IConfiguration configuration)
    {
        Config = configuration;
    }

    public ContentResult OnGet()
    {
        string s = "";
        var selection = Config.GetSection("section2");
        if (!selection.Exists())
        {
            throw new Exception("section2 does not exist.");
        }
        var children = selection.GetChildren();

        foreach (var subSection in children)
        {
            int i = 0;
            var key1 = subSection.Key + ":key" + i++.ToString();
            var key2 = subSection.Key + ":key" + i.ToString();
            s += key1 + " value: " + selection[key1] + "\n";
            s += key2 + " value: " + selection[key2] + "\n";
        }
        return Content(s);
    }
}
```

The preceding code calls ConfigurationExtensions.Exists to verify the section exists:

Bind an array

The ConfigurationBinder.Bind supports binding arrays to objects using array indices in configuration keys. Any array format that exposes a numeric key segment is capable of array binding to a POCO class array.

Consider `MyArray.json` from the sample download:

JSON

```
{
  "array": {
    "entries": {
      "0": "value00",
      "1": "value10",
      "2": "value20",
      "4": "value40",
      "5": "value50"
    }
  }
}
```

The following code adds `MyArray.json` to the configuration providers:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration
    .AddJsonFile("MyArray.json",
                 optional: true,
                 reloadOnChange: true);

builder.Services.AddRazorPages();

var app = builder.Build();
```

The following code reads the configuration and displays the values:

C#

```
public class ArrayModel : PageModel
{
    private readonly IConfiguration Config;
    public ArrayExample? _array { get; private set; }

    public ArrayModel(IConfiguration config)
    {
        Config = config;
    }

    public ContentResult OnGet()
    {
       _array = Config.GetSection("array").Get<ArrayExample>();
        if (_array == null)
        {
            throw new ArgumentNullException(nameof(_array));
```

```
        }
        string s = String.Empty;

        for (int j = 0; j < _array.Entries.Length; j++)
        {
            s += $"Index: {j}  Value:  {_array.Entries[j]} \n";
        }

        return Content(s);
    }
}
```

C#

```
public class ArrayExample
{
    public string[]? Entries { get; set; }
}
```

The preceding code returns the following output:

text

```
Index: 0  Value: value00
Index: 1  Value: value10
Index: 2  Value: value20
Index: 3  Value: value40
Index: 4  Value: value50
```

In the preceding output, Index 3 has value `value40`, corresponding to `"4": "value40",` in `MyArray.json`. The bound array indices are continuous and not bound to the configuration key index. The configuration binder isn't capable of binding null values or creating null entries in bound objects.

Custom configuration provider

The sample app demonstrates how to create a basic configuration provider that reads configuration key-value pairs from a database using Entity Framework (EF).

The provider has the following characteristics:

- The EF in-memory database is used for demonstration purposes. To use a database that requires a connection string, implement a secondary `ConfigurationBuilder` to supply the connection string from another configuration provider.

- The provider reads a database table into configuration at startup. The provider doesn't query the database on a per-key basis.
- Reload-on-change isn't implemented, so updating the database after the app starts has no effect on the app's configuration.

Define an `EFConfigurationValue` entity for storing configuration values in the database.

`Models/EFConfigurationValue.cs`:

C#

```
public class EFConfigurationValue
{
    public string Id { get; set; } = String.Empty;
    public string Value { get; set; } = String.Empty;
}
```

Add an `EFConfigurationContext` to store and access the configured values.

`EFConfigurationProvider/EFConfigurationContext.cs`:

C#

```
public class EFConfigurationContext : DbContext
{
    public EFConfigurationContext(DbContextOptions<EFConfigurationContext>
options) : base(options)
    {
    }

    public DbSet<EFConfigurationValue> Values => Set<EFConfigurationValue>
();
}
```

Create a class that implements IConfigurationSource.

`EFConfigurationProvider/EFConfigurationSource.cs`:

C#

```
public class EFConfigurationSource : IConfigurationSource
{
    private readonly Action<DbContextOptionsBuilder> _optionsAction;

    public EFConfigurationSource(Action<DbContextOptionsBuilder>
optionsAction) => _optionsAction = optionsAction;

    public IConfigurationProvider Build(IConfigurationBuilder builder) =>
```

```
new EFConfigurationProvider(_optionsAction);
}
```

Create the custom configuration provider by inheriting from ConfigurationProvider. The configuration provider initializes the database when it's empty. Since configuration keys are case-insensitive, the dictionary used to initialize the database is created with the case-insensitive comparer (StringComparer.OrdinalIgnoreCase).

`EFConfigurationProvider/EFConfigurationProvider.cs`:

C#

```
public class EFConfigurationProvider : ConfigurationProvider
{
    public EFConfigurationProvider(Action<DbContextOptionsBuilder>
optionsAction)
    {
        OptionsAction = optionsAction;
    }

    Action<DbContextOptionsBuilder> OptionsAction { get; }

    public override void Load()
    {
        var builder = new DbContextOptionsBuilder<EFConfigurationContext>();

        OptionsAction(builder);

        using (var dbContext = new EFConfigurationContext(builder.Options))
        {
            if (dbContext == null || dbContext.Values == null)
            {
                throw new Exception("Null DB context");
            }
            dbContext.Database.EnsureCreated();

            Data = !dbContext.Values.Any()
                ? CreateAndSaveDefaultValues(dbContext)
                : dbContext.Values.ToDictionary(c => c.Id, c => c.Value);
        }
    }

    private static IDictionary<string, string> CreateAndSaveDefaultValues(
        EFConfigurationContext dbContext)
    {
        // Quotes (c)2005 Universal Pictures: Serenity
        // https://www.uphe.com/movies/serenity-2005
        var configValues =
            new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase)
            {
                    { "quote1", "I aim to misbehave." },
                    { "quote2", "I swallowed a bug." },
```

```
                    { "quote3", "You can't stop the signal, Mal." }
            };

        if (dbContext == null || dbContext.Values == null)
        {
            throw new Exception("Null DB context");
        }

        dbContext.Values.AddRange(configValues
            .Select(kvp => new EFConfigurationValue
            {
                Id = kvp.Key,
                Value = kvp.Value
            })
            .ToArray());

        dbContext.SaveChanges();

        return configValues;
    }
}
```

An `AddEFConfiguration` extension method permits adding the configuration source to a `ConfigurationBuilder`.

`Extensions/EntityFrameworkExtensions.cs`:

C#

```
public static class EntityFrameworkExtensions
{
    public static IConfigurationBuilder AddEFConfiguration(
                this IConfigurationBuilder builder,
                Action<DbContextOptionsBuilder> optionsAction)
    {
        return builder.Add(new EFConfigurationSource(optionsAction));
    }
}
```

The following code shows how to use the custom `EFConfigurationProvider` in `Program.cs`:

C#

```
//using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddEFConfiguration(
    opt => opt.UseInMemoryDatabase("InMemoryDb"));
```

```
var app = builder.Build();

app.Run();
```

Access configuration with Dependency Injection (DI)

Configuration can be injected into services using Dependency Injection (DI) by resolving the IConfiguration service:

C#

```
public class Service
{
    private readonly IConfiguration _config;

    public Service(IConfiguration config) =>
        _config = config;

    public void DoSomething()
    {
        var configSettingValue = _config["ConfigSetting"];

        // ...
    }
}
```

For information on how to access values using `IConfiguration`, see GetValue and GetSection, GetChildren, and Exists in this article.

Access configuration in Razor Pages

The following code displays configuration data in a Razor Page:

CSHTML

```
@page
@model Test5Model
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

Configuration value for 'MyKey': @Configuration["MyKey"]
```

In the following code, `MyOptions` is added to the service container with Configure and bound to configuration:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));

var app = builder.Build();
```

The following markup uses the @inject Razor directive to resolve and display the options values:

CSHTML

```
@page
@model SampleApp.Pages.Test3Model
@using Microsoft.Extensions.Options
@using SampleApp.Models
@inject IOptions<MyOptions> optionsAccessor


<p><b>Option1:</b> @optionsAccessor.Value.Option1</p>
<p><b>Option2:</b> @optionsAccessor.Value.Option2</p>
```

Access configuration in a MVC view file

The following code displays configuration data in a MVC view:

CSHTML

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration

Configuration value for 'MyKey': @Configuration["MyKey"]
```

Access configuration in `Program.cs`

The following code accesses configuration in the `Program.cs` file.

C#

```
var builder = WebApplication.CreateBuilder(args);

var key1 = builder.Configuration.GetValue<string>("KeyOne");

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

var key2 = app.Configuration.GetValue<int>("KeyTwo");
var key3 = app.Configuration.GetValue<bool>("KeyThree");

app.Logger.LogInformation("KeyOne: {KeyOne}", key1);
app.Logger.LogInformation("KeyTwo: {KeyTwo}", key2);
app.Logger.LogInformation("KeyThree: {KeyThree}", key3);

app.Run();
```

In `appsettings.json` for the preceding example:

JSON

```
{
  ...
  "KeyOne": "Key One Value",
  "KeyTwo": 1999,
  "KeyThree": true
}
```

Configure options with a delegate

Options configured in a delegate override values set in the configuration providers.

In the following code, an IConfigureOptions<TOptions> service is added to the service container. It uses a delegate to configure values for `MyOptions`:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(myOptions =>
{
    myOptions.Option1 = "Value configured in delegate";
    myOptions.Option2 = 500;
```

```
});

var app = builder.Build();
```

The following code displays the options values:

C#

```
public class Test2Model : PageModel
{
    private readonly IOptions<MyOptions> _optionsDelegate;

    public Test2Model(IOptions<MyOptions> optionsDelegate )
    {
        _optionsDelegate = optionsDelegate;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_optionsDelegate.Value.Option1} \n" +
                       $"Option2: {_optionsDelegate.Value.Option2}");
    }
}
```

In the preceding example, the values of `Option1` and `Option2` are specified in `appsettings.json` and then overridden by the configured delegate.

Host versus app configuration

Before the app is configured and started, a *host* is configured and launched. The host is responsible for app startup and lifetime management. Both the app and the host are configured using the configuration providers described in this topic. Host configuration key-value pairs are also included in the app's configuration. For more information on how the configuration providers are used when the host is built and how configuration sources affect host configuration, see ASP.NET Core fundamentals overview.

Default host configuration

For details on the default configuration when using the Web Host, see the ASP.NET Core 2.2 version of this topic.

- Host configuration is provided from:
  - Environment variables prefixed with `DOTNET_` (for example, `DOTNET_ENVIRONMENT`) using the Environment Variables configuration provider. The prefix (`DOTNET_`) is

stripped when the configuration key-value pairs are loaded.
    - Command-line arguments using the Command-line configuration provider.
- Web Host default configuration is established (`ConfigureWebHostDefaults`):
    - Kestrel is used as the web server and configured using the app's configuration providers.
    - Add Host Filtering Middleware.
    - Add Forwarded Headers Middleware if the `ASPNETCORE_FORWARDEDHEADERS_ENABLED` environment variable is set to `true`.
    - Enable IIS integration.

Other configuration

This topic only pertains to *app configuration*. Other aspects of running and hosting ASP.NET Core apps are configured using configuration files not covered in this topic:

- `launch.json`/`launchSettings.json` are tooling configuration files for the Development environment, described:
    - In Use multiple environments in ASP.NET Core.
    - Across the documentation set where the files are used to configure ASP.NET Core apps for Development scenarios.
- `web.config` is a server configuration file, described in the following topics:
    - Host ASP.NET Core on Windows with IIS
    - ASP.NET Core Module (ANCM) for IIS

Environment variables set in `launchSettings.json` override those set in the system environment.

For more information on migrating app configuration from earlier versions of ASP.NET, see Update from ASP.NET to ASP.NET Core.

Add configuration from an external assembly

An IHostingStartup implementation allows adding enhancements to an app at startup from an external assembly outside of the app's `Startup` class. For more information, see Use hosting startup assemblies in ASP.NET Core.

Configuration-binding source generator

The Configuration-binding source generator provides AOT and trim-friendly configuration. For more information, see Configuration-binding source generator.

Additional resources

- Configuration source code
- WebApplicationBuilder source code
- View or download sample code (how to download)
- Options pattern in ASP.NET Core
- ASP.NET Core Blazor configuration

Options pattern in ASP.NET Core

Article • 10/26/2023

By Rick Anderson.

The options pattern uses classes to provide strongly typed access to groups of related settings. When configuration settings are isolated by scenario into separate classes, the app adheres to two important software engineering principles:

- Encapsulation:
  - Classes that depend on configuration settings depend only on the configuration settings that they use.
- Separation of Concerns:
  - Settings for different parts of the app aren't dependent or coupled to one another.

Options also provide a mechanism to validate configuration data. For more information, see the Options validation section.

This article provides information on the options pattern in ASP.NET Core. For information on using the options pattern in console apps, see Options pattern in .NET.

Bind hierarchical configuration

The preferred way to read related configuration values is using the options pattern. For example, to read the following configuration values:

JSON

```
"Position": {
  "Title": "Editor",
  "Name": "Joe Smith"
}
```

Create the following `PositionOptions` class:

C#

```
public class PositionOptions
{
    public const string Position = "Position";

    public string Title { get; set; } = String.Empty;
```

```
    public string Name { get; set; } = String.Empty;
}
```

An options class:

- Must be non-abstract.
- Has public read-write properties of the type that have corresponding items in config are bound.
- Has its read-write properties bound to matching entries in configuration.
- Does ***not*** have its fields bound. In the preceding code, `Position` is not bound. The `Position` field is used so the string `"Position"` doesn't need to be hard coded in the app when binding the class to a configuration provider.

The following code:

- Calls ConfigurationBinder.Bind to bind the `PositionOptions` class to the `Position` section.
- Displays the `Position` configuration data.

C#

```
public class Test22Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test22Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var positionOptions = new PositionOptions();

Configuration.GetSection(PositionOptions.Position).Bind(positionOptions);

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

ConfigurationBinder.Get<T> binds and returns the specified type.
`ConfigurationBinder.Get<T>` may be more convenient than using

`ConfigurationBinder.Bind`. The following code shows how to use `ConfigurationBinder.Get<T>` with the `PositionOptions` class:

C#

```
public class Test21Model : PageModel
{
    private readonly IConfiguration Configuration;
    public PositionOptions? positionOptions { get; private set; }

    public Test21Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        positionOptions = Configuration.GetSection(PositionOptions.Position)
                                                        .Get<PositionOptions>
();

        return Content($"Title: {positionOptions.Title} \n" +
                       $"Name: {positionOptions.Name}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

Bind also allows the concretion of an abstract class. Consider the following code which uses the abstract class `SomethingWithAName`:

C#

```
namespace ConfigSample.Options;

public abstract class SomethingWithAName
{
    public abstract string? Name { get; set; }
}

public class NameTitleOptions(int age) : SomethingWithAName
{
    public const string NameTitle = "NameTitle";

    public override string? Name { get; set; }
    public string Title { get; set; } = string.Empty;

    public int Age { get; set; } = age;
}
```

The following code displays the `NameTitleOptions` configuration values:

C#

```
public class Test33Model : PageModel
{
    private readonly IConfiguration Configuration;

    public Test33Model(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public ContentResult OnGet()
    {
        var nameTitleOptions = new NameTitleOptions(22);

Configuration.GetSection(NameTitleOptions.NameTitle).Bind(nameTitleOptions);

        return Content($"Title: {nameTitleOptions.Title} \n" +
                       $"Name: {nameTitleOptions.Name}  \n" +
                       $"Age: {nameTitleOptions.Age}"
                       );
    }
}
```

Calls to `Bind` are less strict than calls to `Get<>`:

- `Bind` allows the concretion of an abstract.
- `Get<>` has to create an instance itself.

The Options Pattern

An alternative approach when using the ***options pattern*** is to bind the `Position` section and add it to the dependency injection service container. In the following code, `PositionOptions` is added to the service container with Configure and bound to configuration:

C#

```
using ConfigSample.Options;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));
```

```
var app = builder.Build();
```

Using the preceding code, the following code reads the position options:

C#

```
public class Test2Model : PageModel
{
    private readonly PositionOptions _options;

    public Test2Model(IOptions<PositionOptions> options)
    {
        _options = options.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Title: {_options.Title} \n" +
                       $"Name: {_options.Name}");
    }
}
```

In the preceding code, changes to the JSON configuration file after the app has started are ***not*** read. To read changes after the app has started, use IOptionsSnapshot.

Options interfaces

IOptions<TOptions>:

- Does ***not*** support:
  - Reading of configuration data after the app has started.
  - Named options
- Is registered as a Singleton and can be injected into any service lifetime.

IOptionsSnapshot<TOptions>:

- Is useful in scenarios where options should be recomputed on every request. For more information, see Use IOptionsSnapshot to read updated data.
- Is registered as Scoped and therefore can't be injected into a Singleton service.
- Supports named options

IOptionsMonitor<TOptions>:

- Is used to retrieve options and manage options notifications for `TOptions` instances.
- Is registered as a Singleton and can be injected into any service lifetime.
- Supports:
  - Change notifications
  - named options
  - Reloadable configuration
  - Selective options invalidation (IOptionsMonitorCache<TOptions>)

Post-configuration scenarios enable setting or changing options after all IConfigureOptions<TOptions> configuration occurs.

IOptionsFactory<TOptions> is responsible for creating new options instances. It has a single Create method. The default implementation takes all registered IConfigureOptions<TOptions> and IPostConfigureOptions<TOptions> and runs all the configurations first, followed by the post-configuration. It distinguishes between IConfigureNamedOptions<TOptions> and IConfigureOptions<TOptions> and only calls the appropriate interface.

IOptionsMonitorCache<TOptions> is used by IOptionsMonitor<TOptions> to cache `TOptions` instances. The IOptionsMonitorCache<TOptions> invalidates options instances in the monitor so that the value is recomputed (TryRemove). Values can be manually introduced with TryAdd. The Clear method is used when all named instances should be recreated on demand.

Use IOptionsSnapshot to read updated data

Using IOptionsSnapshot<TOptions>:

- Options are computed once per request when accessed and cached for the lifetime of the request.
- May incur a significant performance penalty because it's a Scoped service and is recomputed per request. For more information, see this GitHub issue and Improve the performance of configuration binding.
- Changes to the configuration are read after the app starts when using configuration providers that support reading updated configuration values.

The difference between `IOptionsMonitor` and `IOptionsSnapshot` is that:

- `IOptionsMonitor` is a Singleton service that retrieves current option values at any time, which is especially useful in singleton dependencies.

- `IOptionsSnapshot` is a Scoped service and provides a snapshot of the options at the time the `IOptionsSnapshot<T>` object is constructed. Options snapshots are designed for use with transient and scoped dependencies.

The following code uses IOptionsSnapshot<TOptions>.

C#

```
public class TestSnapModel : PageModel
{
    private readonly MyOptions _snapshotOptions;

    public TestSnapModel(IOptionsSnapshot<MyOptions>
snapshotOptionsAccessor)
    {
        _snapshotOptions = snapshotOptionsAccessor.Value;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_snapshotOptions.Option1} \n" +
                       $"Option2: {_snapshotOptions.Option2}");
    }
}
```

The following code registers a configuration instance which `MyOptions` binds against:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));

var app = builder.Build();
```

In the preceding code, changes to the JSON configuration file after the app has started are read.

IOptionsMonitor

The following code registers a configuration instance which `MyOptions` binds against.

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));

var app = builder.Build();
```

The following example uses IOptionsMonitor<TOptions>:

C#

```
public class TestMonitorModel : PageModel
{
    private readonly IOptionsMonitor<MyOptions> _optionsDelegate;

    public TestMonitorModel(IOptionsMonitor<MyOptions> optionsDelegate )
    {
        _optionsDelegate = optionsDelegate;
    }

    public ContentResult OnGet()
    {
        return Content($"Option1: {_optionsDelegate.CurrentValue.Option1}
\n" +
                       $"Option2: {_optionsDelegate.CurrentValue.Option2}");
    }
}
```

In the preceding code, by default, changes to the JSON configuration file after the app has started are read.

Named options support using IConfigureNamedOptions

Named options:

- Are useful when multiple configuration sections bind to the same properties.
- Are case sensitive.

Consider the following `appsettings.json` file:

JSON

```
{
  "TopItem": {
    "Month": {
      "Name": "Green Widget",
      "Model": "GW46"
    },
    "Year": {
      "Name": "Orange Gadget",
      "Model": "OG35"
    }
  }
}
```

Rather than creating two classes to bind `TopItem:Month` and `TopItem:Year`, the following class is used for each section:

C#

```
public class TopItemSettings
{
    public const string Month = "Month";
    public const string Year = "Year";

    public string Name { get; set; } = string.Empty;
    public string Model { get; set; } = string.Empty;
}
```

The following code configures the named options:

C#

```
using SampleApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<TopItemSettings>(TopItemSettings.Month,
    builder.Configuration.GetSection("TopItem:Month"));
builder.Services.Configure<TopItemSettings>(TopItemSettings.Year,
    builder.Configuration.GetSection("TopItem:Year"));

var app = builder.Build();
```

The following code displays the named options:

C#

```
public class TestNOModel : PageModel
{
    private readonly TopItemSettings _monthTopItem;
    private readonly TopItemSettings _yearTopItem;

    public TestNOModel(IOptionsSnapshot<TopItemSettings>
namedOptionsAccessor)
    {
        _monthTopItem = namedOptionsAccessor.Get(TopItemSettings.Month);
        _yearTopItem = namedOptionsAccessor.Get(TopItemSettings.Year);
    }

    public ContentResult OnGet()
    {
        return Content($"Month:Name {_monthTopItem.Name} \n" +
                       $"Month:Model {_monthTopItem.Model} \n\n" +
                       $"Year:Name {_yearTopItem.Name} \n" +
                       $"Year:Model {_yearTopItem.Model} \n"    );
    }
}
```

All options are named instances. IConfigureOptions<TOptions> instances are treated as targeting the `Options.DefaultName` instance, which is `string.Empty`. IConfigureNamedOptions<TOptions> also implements IConfigureOptions<TOptions>. The default implementation of the IOptionsFactory<TOptions> has logic to use each appropriately. The `null` named option is used to target all of the named instances instead of a specific named instance. ConfigureAll and PostConfigureAll use this convention.

OptionsBuilder API

OptionsBuilder<TOptions> is used to configure `TOptions` instances. `OptionsBuilder` streamlines creating named options as it's only a single parameter to the initial `AddOptions<TOptions>(string optionsName)` call instead of appearing in all of the subsequent calls. Options validation and the `ConfigureOptions` overloads that accept service dependencies are only available via `OptionsBuilder`.

`OptionsBuilder` is used in the Options validation section.

See Use AddOptions to configure custom repository for information adding a custom repository.

Use DI services to configure options

Services can be accessed from dependency injection while configuring options in two ways:

- Pass a configuration delegate to Configure on OptionsBuilder<TOptions>. `OptionsBuilder<TOptions>` provides overloads of Configure that allow use of up to five services to configure options:

  C#

  ```
  builder.Services.AddOptions<MyOptions>("optionalName")
      .Configure<Service1, Service2, Service3, Service4, Service5>(
          (o, s, s2, s3, s4, s5) =>
              o.Property = DoSomethingWith(s, s2, s3, s4, s5));
  ```

- Create a type that implements IConfigureOptions<TOptions> or IConfigureNamedOptions<TOptions> and register the type as a service.

We recommend passing a configuration delegate to Configure, since creating a service is more complex. Creating a type is equivalent to what the framework does when calling Configure. Calling Configure registers a transient generic IConfigureNamedOptions<TOptions>, which has a constructor that accepts the generic service types specified.

Options validation

Options validation enables option values to be validated.

Consider the following `appsettings.json` file:

JSON

```
{
  "MyConfig": {
    "Key1": "My Key One",
    "Key2": 10,
    "Key3": 32
  }
}
```

The following class is used to bind to the `"MyConfig"` configuration section and applies a couple of `DataAnnotations` rules:

C#

```
public class MyConfigOptions
{
    public const string MyConfig = "MyConfig";

    [RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$")]
    public string Key1 { get; set; }
    [Range(0, 1000,
        ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public int Key2 { get; set; }
    public int Key3 { get; set; }
}
```

The following code:

- Calls AddOptions to get an OptionsBuilder<TOptions> that binds to the `MyConfigOptions` class.
- Calls ValidateDataAnnotations to enable validation using `DataAnnotations`.

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()

.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig))
            .ValidateDataAnnotations();

var app = builder.Build();
```

The `ValidateDataAnnotations` extension method is defined in the Microsoft.Extensions.Options.DataAnnotations NuGet package. For web apps that use the `Microsoft.NET.Sdk.Web` SDK, this package is referenced implicitly from the shared framework.

The following code displays the configuration values or the validation errors:

C#

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly IOptions<MyConfigOptions> _config;

    public HomeController(IOptions<MyConfigOptions> config,
```

```
                          ILogger<HomeController> logger)
    {
        _config = config;
        _logger = logger;

        try
        {
            var configValue = _config.Value;

        }
        catch (OptionsValidationException ex)
        {
            foreach (var failure in ex.Failures)
            {
                _logger.LogError(failure);
            }
        }
    }

    public ContentResult Index()
    {
        string msg;
        try
        {
            msg = $"Key1: {_config.Value.Key1} \n" +
                  $"Key2: {_config.Value.Key2} \n" +
                  $"Key3: {_config.Value.Key3}";
        }
        catch (OptionsValidationException optValEx)
        {
            return Content(optValEx.Message);
        }
        return Content(msg);
    }
```

The following code applies a more complex validation rule using a delegate:

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()

.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig))
            .ValidateDataAnnotations()
        .Validate(config =>
        {
            if (config.Key2 != 0)
            {
```

```
                return config.Key3 > config.Key2;
            }

            return true;
        }, "Key3 must be > than Key2.");   // Failure message.

var app = builder.Build();
```

`IValidateOptions<TOptions>` and `IValidatableObject`

The following class implements IValidateOptions<TOptions>:

C#

```
public class MyConfigValidation : IValidateOptions<MyConfigOptions>
{
    public MyConfigOptions _config { get; private set; }

    public  MyConfigValidation(IConfiguration config)
    {
        _config = config.GetSection(MyConfigOptions.MyConfig)
            .Get<MyConfigOptions>();
    }

    public ValidateOptionsResult Validate(string name, MyConfigOptions
options)
    {
        string? vor = null;
        var rx = new Regex(@"^[a-zA-Z''-'\s]{1,40}$");
        var match = rx.Match(options.Key1!);

        if (string.IsNullOrEmpty(match.Value))
        {
            vor = $"{options.Key1} doesn't match RegEx \n";
        }

        if ( options.Key2 < 0 || options.Key2 > 1000)
        {
            vor = $"{options.Key2} doesn't match Range 0 - 1000 \n";
        }

        if (_config.Key2 != default)
        {
            if(_config.Key3 <= _config.Key2)
            {
                vor +=  "Key3 must be > than Key2.";
            }
        }

        if (vor != null)
        {
            return ValidateOptionsResult.Fail(vor);
```

```
        }

        return ValidateOptionsResult.Success;
    }
}
```

`IValidateOptions` enables moving the validation code out of `Program.cs` and into a class.

Using the preceding code, validation is enabled in `Program.cs` with the following code:

C#

```
using Microsoft.Extensions.Options;
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.Configure<MyConfigOptions>
(builder.Configuration.GetSection(
                                        MyConfigOptions.MyConfig));

builder.Services.AddSingleton<IValidateOptions
                              <MyConfigOptions>, MyConfigValidation>();

var app = builder.Build();
```

Options validation also supports IValidatableObject. To perform class-level validation of a class within the class itself:

- Implement the `IValidatableObject` interface and its Validate method within the class.
- Call ValidateDataAnnotations in `Program.cs`.

`ValidateOnStart`

Options validation runs the first time a `TOption` instance is created. That means, for instance, when first access to IOptionsSnapshot<TOptions>.Value occurs in a request pipeline or when IOptionsMonitor<TOptions>.Get(string) is called on settings present. After settings are reloaded, validation runs again. The ASP.NET Core runtime uses OptionsCache<TOptions> to cache the options instance once it is created.

To run options validation eagerly, when the app starts, call ValidateOnStart<TOptions>(OptionsBuilder<TOptions>)in `Program.cs`:

C#

```
builder.Services.AddOptions<MyConfigOptions>()
    .Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig))
    .ValidateDataAnnotations()
    .ValidateOnStart();
```

Options post-configuration

Set post-configuration with IPostConfigureOptions<TOptions>. Post-configuration runs after all IConfigureOptions<TOptions> configuration occurs:

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()

.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig));

builder.Services.PostConfigure<MyConfigOptions>(myOptions =>
{
    myOptions.Key1 = "post_configured_key1_value";
});
```

PostConfigure is available to post-configure named options:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.Configure<TopItemSettings>(TopItemSettings.Month,
    builder.Configuration.GetSection("TopItem:Month"));
builder.Services.Configure<TopItemSettings>(TopItemSettings.Year,
    builder.Configuration.GetSection("TopItem:Year"));

builder.Services.PostConfigure<TopItemSettings>("Month", myOptions =>
{
    myOptions.Name = "post_configured_name_value";
    myOptions.Model = "post_configured_model_value";
});
```

```
var app = builder.Build();
```

Use PostConfigureAll to post-configure all configuration instances:

C#

```
using OptionsValidationSample.Configuration;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

builder.Services.AddOptions<MyConfigOptions>()

.Bind(builder.Configuration.GetSection(MyConfigOptions.MyConfig));

builder.Services.PostConfigureAll<MyConfigOptions>(myOptions =>
{
    myOptions.Key1 = "post_configured_key1_value";
});
```

Access options in `Program.cs`

To access IOptions<TOptions> or IOptionsMonitor<TOptions> in `Program.cs`, call GetRequiredService on WebApplication.Services:

C#

```
var app = builder.Build();

var option1 = app.Services.GetRequiredService<IOptionsMonitor<MyOptions>>()
    .CurrentValue.Option1;
```

Additional resources

- View or download sample code (how to download)

Use multiple environments in ASP.NET Core

Article • 04/26/2024

By Rick Anderson and Kirk Larkin

ASP.NET Core configures app behavior based on the runtime environment using an environment variable.

For Blazor environments guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor environments.

Environments

To determine the runtime environment, ASP.NET Core reads from the following environment variables:

1. DOTNET_ENVIRONMENT
2. `ASPNETCORE_ENVIRONMENT` when the WebApplication.CreateBuilder method is called. The default ASP.NET Core web app templates call `WebApplication.CreateBuilder`. The `DOTNET_ENVIRONMENT` value overrides `ASPNETCORE_ENVIRONMENT` when `WebApplicationBuilder` is used. For other hosts, such as `ConfigureWebHostDefaults` and `WebHost.CreateDefaultBuilder`, `ASPNETCORE_ENVIRONMENT` has higher precedence.

`IHostEnvironment.EnvironmentName` can be set to any value, but the following values are provided by the framework:

- Development: The launchSettings.json file sets `ASPNETCORE_ENVIRONMENT` to `Development` on the local machine.
- Staging
- Production: The default if `DOTNET_ENVIRONMENT` and `ASPNETCORE_ENVIRONMENT` have not been set.

The following code:

- Is similar to the code generated by the ASP.NET Core templates.
- Enables the Developer Exception Page when `ASPNETCORE_ENVIRONMENT` is set to `Development`. This is done automatically by the WebApplication.CreateBuilder method.

- Calls UseExceptionHandler when the value of `ASPNETCORE_ENVIRONMENT` is anything other than `Development`.
- Provides an IWebHostEnvironment instance in the Environment property of `WebApplication`.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The Environment Tag Helper uses the value of IHostEnvironment.EnvironmentName to include or exclude markup in the element:

CSHTML

```
<environment include="Development">
    <div>Environment is Development</div>
</environment>
<environment exclude="Development">
    <div>Environment is NOT Development</div>
</environment>
<environment include="Staging,Development,Staging_2">
    <div>Environment is: Staging, Development or Staging_2</div>
</environment>
```

The About page from the sample code includes the preceding markup and displays the value of `IWebHostEnvironment.EnvironmentName`.

On Windows and macOS, environment variables and values aren't case-sensitive. Linux environment variables and values are case-sensitive by default.

Create EnvironmentsSample

The sample code used in this article is based on a Razor Pages project named *EnvironmentsSample*.

The following .NET CLI commands create and run a web app named *EnvironmentsSample*:

Bash

```
dotnet new webapp -o EnvironmentsSample
cd EnvironmentsSample
dotnet run --verbosity normal
```

When the app runs, it displays output similar to the following:

Bash

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7152
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5105
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Path\To\EnvironmentsSample
```

Set environment on the command line

Use the `--environment` flag to set the environment. For example:

.NET CLI

```
dotnet run --environment Production
```

The preceding command sets the environment to `Production` and displays output similar to the following in the command window:

Bash

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7262
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5005
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Path\To\EnvironmentsSample
```

Development and launchSettings.json

The development environment can enable features that shouldn't be exposed in production. For example, the ASP.NET Core project templates enable the Developer Exception Page in the development environment. Because of the performance cost, scope validation and dependency validation only happens in development.

The environment for local machine development can be set in the *Properties\launchSettings.json* file of the project. Environment values set in `launchSettings.json` override values set in the system environment.

The `launchSettings.json` file:

- Is only used on the local development machine.
- Is not deployed.
- Contains profile settings.

The following JSON shows the `launchSettings.json` file for an ASP.NET Core web project named *EnvironmentsSample* created with Visual Studio or `dotnet new`:

JSON

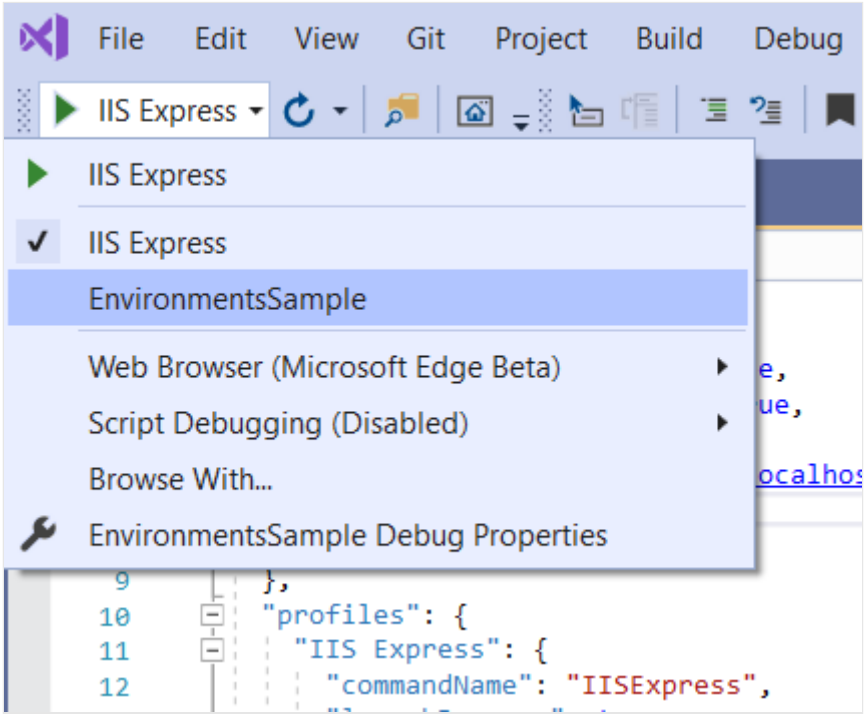
```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:59481",
      "sslPort": 44308
    }
  },
  "profiles": {
    "EnvironmentsSample": {
      "commandName": "Project",
```

```
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The preceding JSON contains two profiles:

- `EnvironmentsSample`: The profile name is the project name. As the first profile listed, this profile is used by default. The `"commandName"` key has the value `"Project"`, therefore, the Kestrel web server is launched.
- `IIS Express`: The `"commandName"` key has the value `"IISExpress"`, therefore, IISExpress is the web server.

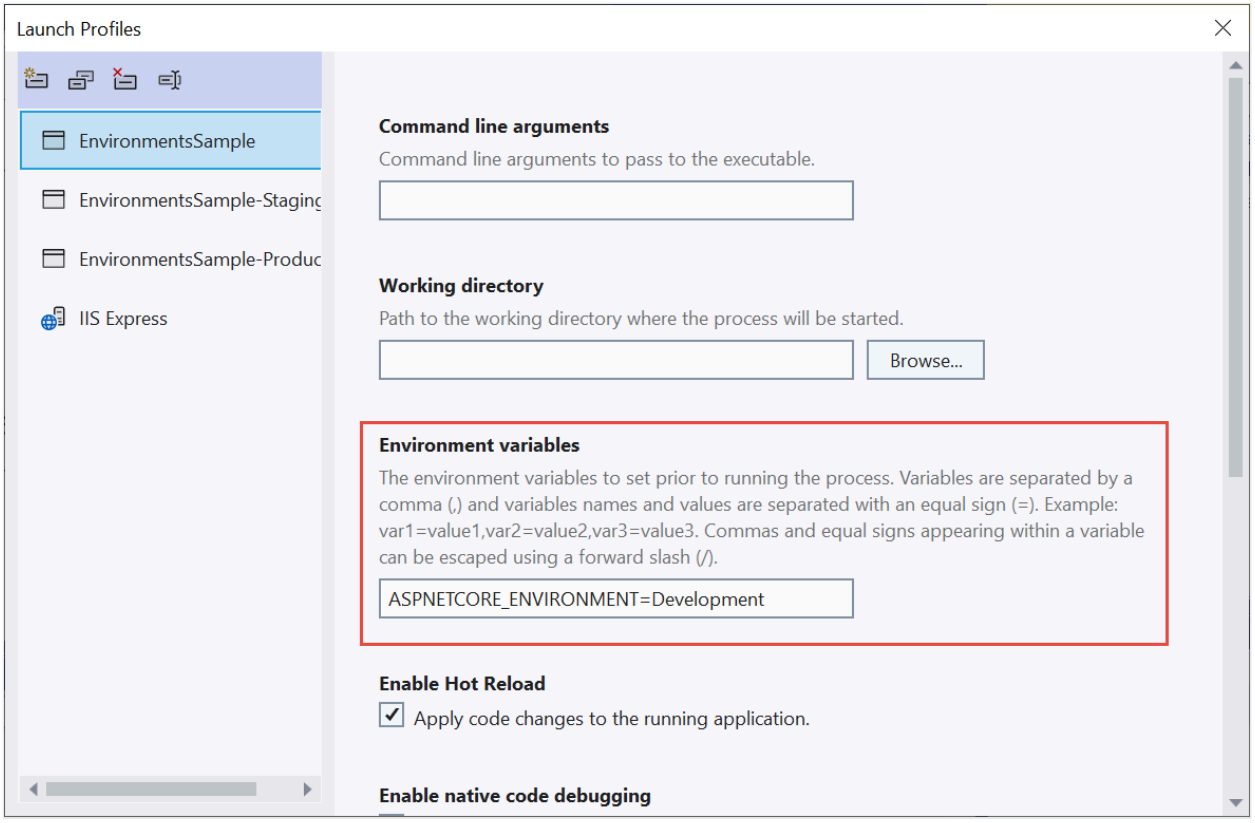
You can set the launch profile to the project or any other profile included in `launchSettings.json`. For example, in the image below, selecting the project name launches the Kestrel web server.



The value of `commandName` can specify the web server to launch. `commandName` can be any one of the following:

- `IISExpress` : Launches IIS Express.
- `IIS` : No web server launched. IIS is expected to be available.
- `Project` : Launches Kestrel.

The Visual Studio 2022 project properties **Debug / General** tab provides an **Open debug launch profiles UI** link. This link opens a **Launch Profiles** dialog that lets you edit the environment variable settings in the `launchSettings.json` file. You can also open the **Launch Profiles** dialog from the **Debug** menu by selecting **<project name> Debug Properties**. Changes made to project profiles may not take effect until the web server is restarted. Kestrel must be restarted before it can detect changes made to its environment.



The following `launchSettings.json` file contains multiple profiles:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:59481",
      "sslPort": 44308
    }
```

```
  },
  "profiles": {
    "EnvironmentsSample": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "EnvironmentsSample-Staging": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Staging",
        "ASPNETCORE_DETAILEDERRORS": "1",
        "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
      }
    },
    "EnvironmentsSample-Production": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7152;http://localhost:5105",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Profiles can be selected:

- From the Visual Studio UI.
- Using the dotnet run CLI command with the `--launch-profile` option set to the profile's name. *This approach only supports Kestrel profiles.*

  .NET CLI

  ```
  dotnet run --launch-profile "EnvironmentsSample"
  ```

> ⚠ **Warning**
>
> `launchSettings.json` shouldn't store secrets. The Secret Manager tool can be used to store secrets for local development.

When using Visual Studio Code, environment variables can be set in the `.vscode/launch.json` file. The following example sets several environment variables for Host configuration values:

JSON

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            // Configuration ommitted for brevity.
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development",
                "ASPNETCORE_URLS": "https://localhost:5001",
                "ASPNETCORE_DETAILEDERRORS": "1",
                "ASPNETCORE_SHUTDOWNTIMEOUTSECONDS": "3"
            },
            // Configuration ommitted for brevity.
```

The `.vscode/launch.json` file is used only by Visual Studio Code.

Production

The production environment should be configured to maximize security, performance, and application robustness. Some common settings that differ from development include:

- Caching.
- Client-side resources are bundled, minified, and potentially served from a CDN.
- Diagnostic error pages disabled.
- Friendly error pages enabled.
- Production logging and monitoring enabled. For example, using Application Insights.

Set the environment by setting an environment variable

It's often useful to set a specific environment for testing with an environment variable or platform setting. If the environment isn't set, it defaults to `Production`, which disables most debugging features. The method for setting the environment depends on the operating system.

When the host is built, the last environment setting read by the app determines the app's environment. The app's environment can't be changed while the app is running.

The About page from the sample code displays the value of `IWebHostEnvironment.EnvironmentName`.

Azure App Service

Production is the default value if `DOTNET_ENVIRONMENT` and `ASPNETCORE_ENVIRONMENT` have not been set. Apps deployed to Azure are `Production` by default.

To set the environment in an Azure App Service app by using the portal:

1. Select the app from the **App Services** page.
2. In the **Settings** group, select **Environment variables**.
3. In the **App settings** tab, select **+ Add**.
4. In the **Add/Edit application setting** window, provide `ASPNETCORE_ENVIRONMENT` for the **Name**. For **Value**, provide the environment (for example, `Staging`).
5. Select the **Deployment slot setting** checkbox if you wish the environment setting to remain with the current slot when deployment slots are swapped. For more information, see Set up staging environments in Azure App Service in the Azure documentation.
6. Select **OK** to close the **Add/Edit application setting** dialog.
7. Select **Save** at the top of the **Configuration** page.

Azure App Service automatically restarts the app after an app setting is added, changed, or deleted in the Azure portal.

Windows - Set environment variable for a process

Environment values in `launchSettings.json` override values set in the system environment.

To set the `ASPNETCORE_ENVIRONMENT` for the current session when the app is started using dotnet run, use the following commands at a command prompt or in PowerShell:

```
Console
```

```
set ASPNETCORE_ENVIRONMENT=Staging
dotnet run --no-launch-profile
```

PowerShell

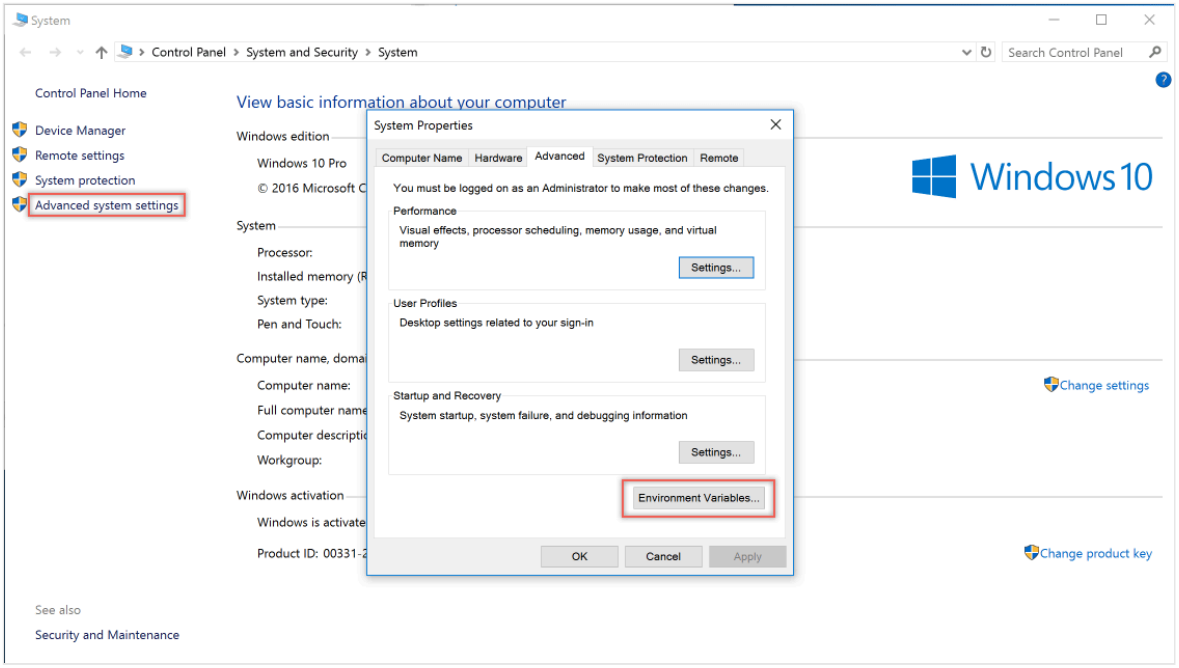
```
$Env:ASPNETCORE_ENVIRONMENT = "Staging"
dotnet run --no-launch-profile
```

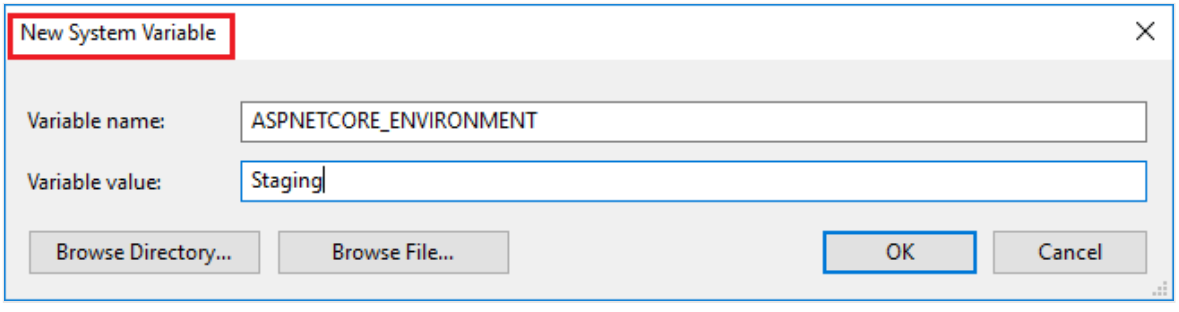
Windows - Set environment variable globally

The preceding commands set `ASPNETCORE_ENVIRONMENT` only for processes launched from that command window.

To set the value globally in Windows, use either of the following approaches:

- Open the **Control Panel** > **System** > **Advanced system settings** and add or edit the `ASPNETCORE_ENVIRONMENT` value:





- Open an administrative command prompt and use the `setx` command or open an administrative PowerShell command prompt and use

`[Environment]::SetEnvironmentVariable`:

- Console

  ```
  setx ASPNETCORE_ENVIRONMENT Staging /M
  ```

  The `/M` switch sets the environment variable at the system level. If the `/M` switch isn't used, the environment variable is set for the user account.

- PowerShell

  ```
  [Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT",
  "Staging", "Machine")
  ```

  The `Machine` option sets the environment variable at the system level. If the option value is changed to `User`, the environment variable is set for the user account.

When the `ASPNETCORE_ENVIRONMENT` environment variable is set globally, it takes effect for `dotnet run` in any command window opened after the value is set. Environment values in `launchSettings.json` override values set in the system environment.

Windows - Use web.config

To set the `ASPNETCORE_ENVIRONMENT` environment variable with `web.config`, see the *Set environment variables* section of web.config file.

Windows - IIS deployments

Include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in *web.config* when the project is published:

XML

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

To set the `ASPNETCORE_ENVIRONMENT` environment variable for an app running in an isolated Application Pool (supported on IIS 10.0 or later), see the *AppCmd.exe command* section of Environment Variables <environmentVariables>. When the

`ASPNETCORE_ENVIRONMENT` environment variable is set for an app pool, its value overrides a setting at the system level.

When hosting an app in IIS and adding or changing the `ASPNETCORE_ENVIRONMENT` environment variable, use one of the following approaches to have the new value picked up by apps:

- Execute `net stop was /y` followed by `net start w3svc` from a command prompt.
- Restart the server.

macOS

Setting the current environment for macOS can be performed in-line when running the app:

```
Bash

ASPNETCORE_ENVIRONMENT=Staging dotnet run
```

Alternatively, set the environment with `export` prior to running the app:

```
Bash

export ASPNETCORE_ENVIRONMENT=Staging
```

Machine-level environment variables are set in the *.bashrc* or *.bash_profile* file. Edit the file using any text editor. Add the following statement:

```
Bash

export ASPNETCORE_ENVIRONMENT=Staging
```

Linux

For Linux distributions, use the `export` command at a command prompt for session-based variable settings and the *bash_profile* file for machine-level environment settings.

Set the environment in code

To set the environment in code, use WebApplicationOptions.EnvironmentName when creating WebApplicationBuilder, as shown in the following example:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    EnvironmentName = Environments.Staging
});

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Configuration by environment

To load configuration by environment, see Configuration in ASP.NET Core.

Configure services and middleware by environment

Use WebApplicationBuilder.Environment or WebApplication.Environment to conditionally add services or middleware depending on the current environment. The project template includes an example of code that adds middleware only when the current environment isn't Development:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The highlighted code checks the current environment while building the request pipeline. To check the current environment while configuring services, use `builder.Environment` instead of `app.Environment`.

Additional resources

- View or download sample code (how to download)
- App startup in ASP.NET Core
- Configuration in ASP.NET Core
- ASP.NET Core Blazor environments

Logging in .NET Core and ASP.NET Core

Article • 11/28/2023

By Kirk Larkin, Juergen Gutsch, and Rick Anderson

This article describes logging in .NET as it applies to ASP.NET Core apps. For detailed information on logging in .NET, see Logging in .NET.

For Blazor logging guidance, which adds to or supersedes the guidance in this node, see ASP.NET Core Blazor logging.

Logging providers

Logging providers store logs, except for the `Console` provider which displays logs. For example, the Azure Application Insights provider stores logs in Azure Application Insights. Multiple providers can be enabled.

The default ASP.NET Core web app templates call WebApplication.CreateBuilder, which adds the following logging providers:

- Console
- Debug
- EventSource
- EventLog: Windows only

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();
```

```
app.MapRazorPages();

app.Run();
```

The preceding code shows the `Program.cs` file created with the ASP.NET Core web app templates. The next several sections provide samples based on the ASP.NET Core web app templates.

The following code overrides the default set of logging providers added by `WebApplication.CreateBuilder`:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Logging.ClearProviders();
builder.Logging.AddConsole();

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Alternatively, the preceding logging code can be written as follows:

C#

```
var builder = WebApplication.CreateBuilder();
builder.Host.ConfigureLogging(logging =>
{
    logging.ClearProviders();
    logging.AddConsole();
});
```

For additional providers, see:

- Built-in logging providers
- Third-party logging providers.

Create logs

To create logs, use an ILogger<TCategoryName> object from dependency injection (DI).

The following example:

- Creates a logger, `ILogger<AboutModel>`, which uses a log *category* of the fully qualified name of the type `AboutModel`. The log category is a string that is associated with each log.
- Calls LogInformation to log at the Information level. The Log *level* indicates the severity of the logged event.

C#

```
public class AboutModel : PageModel
{
    private readonly ILogger _logger;

    public AboutModel(ILogger<AboutModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        _logger.LogInformation("About page visited at {DT}",
            DateTime.UtcNow.ToLongTimeString());
    }
}
```

Levels and categories are explained in more detail later in this document.

For information on Blazor, see ASP.NET Core Blazor logging.

Configure logging

Logging configuration is commonly provided by the `Logging` section of `appsettings.{ENVIRONMENT}.json` files, where the `{ENVIRONMENT}` placeholder is the environment. The following `appsettings.Development.json` file is generated by the ASP.NET Core web app templates:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

In the preceding JSON:

- The `"Default"` and `"Microsoft.AspNetCore"` categories are specified.
- The `"Microsoft.AspNetCore"` category applies to all categories that start with `"Microsoft.AspNetCore"`. For example, this setting applies to the `"Microsoft.AspNetCore.Routing.EndpointMiddleware"` category.
- The `"Microsoft.AspNetCore"` category logs at log level `Warning` and higher.
- A specific log provider is not specified, so `LogLevel` applies to all the enabled logging providers except for the Windows EventLog.

The `Logging` property can have LogLevel and log provider properties. The `LogLevel` specifies the minimum level to log for selected categories. In the preceding JSON, `Information` and `Warning` log levels are specified. `LogLevel` indicates the severity of the log and ranges from 0 to 6:

`Trace` = 0, `Debug` = 1, `Information` = 2, `Warning` = 3, `Error` = 4, `Critical` = 5, and `None` = 6.

When a `LogLevel` is specified, logging is enabled for messages at the specified level and higher. In the preceding JSON, the `Default` category is logged for `Information` and higher. For example, `Information`, `Warning`, `Error`, and `Critical` messages are logged. If no `LogLevel` is specified, logging defaults to the `Information` level. For more information, see Log levels.

A provider property can specify a `LogLevel` property. `LogLevel` under a provider specifies levels to log for that provider, and overrides the non-provider log settings. Consider the following `appsettings.json` file:

JSON

```
{
  "Logging": {
    "LogLevel": { // All providers, LogLevel applies to all the enabled
providers.
```

```
      "Default": "Error", // Default logging, Error and higher.
      "Microsoft": "Warning" // All Microsoft* categories, Warning and
higher.
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information", // Overrides preceding LogLevel:Default
setting.
        "Microsoft.Hosting": "Trace" // Debug:Microsoft.Hosting category.
      }
    },
    "EventSource": { // EventSource provider
      "LogLevel": {
        "Default": "Warning" // All categories of EventSource provider.
      }
    }
  }
}
```

Settings in `Logging.{PROVIDER NAME}.LogLevel` override settings in `Logging.LogLevel`, where the `{PROVIDER NAME}` placeholder is the provider name. In the preceding JSON, the `Debug` provider's default log level is set to `Information`:

`Logging:Debug:LogLevel:Default:Information`

The preceding setting specifies the `Information` log level for every `Logging:Debug:` category except `Microsoft.Hosting`. When a specific category is listed, the specific category overrides the default category. In the preceding JSON, the `Logging:Debug:LogLevel` categories `"Microsoft.Hosting"` and `"Default"` override the settings in `Logging:LogLevel`.

The minimum log level can be specified for any of:

- Specific providers: For example, `Logging:EventSource:LogLevel:Default:Information`
- Specific categories: For example, `Logging:LogLevel:Microsoft:Warning`
- All providers and all categories: `Logging:LogLevel:Default:Warning`

Any logs below the minimum level are ***not***:

- Passed to the provider.
- Logged or displayed.

To suppress all logs, specify LogLevel.None. `LogLevel.None` has a value of 6, which is higher than `LogLevel.Critical` (5).

If a provider supports log scopes, `IncludeScopes` indicates whether they're enabled. For more information, see log scopes.

The following `appsettings.json` file contains all the providers enabled by default:

JSON

```
{
  "Logging": {
    "LogLevel": { // No provider, LogLevel applies to all the enabled
providers.
      "Default": "Error",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Warning"
    },
    "Debug": { // Debug provider.
      "LogLevel": {
        "Default": "Information" // Overrides preceding LogLevel:Default
setting.
      }
    },
    "Console": {
      "IncludeScopes": true,
      "LogLevel": {
        "Microsoft.AspNetCore.Mvc.Razor.Internal": "Warning",
        "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
        "Microsoft.AspNetCore.Mvc.Razor": "Error",
        "Default": "Information"
      }
    },
    "EventSource": {
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "EventLog": {
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "AzureAppServicesFile": {
      "IncludeScopes": true,
      "LogLevel": {
        "Default": "Warning"
      }
    },
    "AzureAppServicesBlob": {
      "IncludeScopes": true,
      "LogLevel": {
        "Microsoft": "Information"
      }
    },
    "ApplicationInsights": {
```

```
      "LogLevel": {
        "Default": "Information"
      }
    }
  }
}
```

In the preceding sample:

- The categories and levels aren't suggested values. The sample is provided to show all of the default providers.
- Settings in `Logging.{PROVIDER NAME}.LogLevel` override settings in `Logging.LogLevel`, where the `{PROVIDER NAME}` placeholder is the provider name. For example, the level in `Debug.LogLevel.Default` overrides the level in `LogLevel.Default`.
- Each default provider *alias* is used. Each provider defines an *alias* that can be used in configuration in place of the fully qualified type name. The built-in providers aliases are:
  - `Console`
  - `Debug`
  - `EventSource`
  - `EventLog`
  - `AzureAppServicesFile`
  - `AzureAppServicesBlob`
  - `ApplicationInsights`

Log in `Program.cs`

The following example calls Builder.WebApplication.Logger in `Program.cs` and logs informational messages:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.Logger.LogInformation("Adding Routes");
app.MapGet("/", () => "Hello World!");
app.Logger.LogInformation("Starting the app");
app.Run();
```

The following example calls AddConsole in `Program.cs` and logs the `/Test` endpoint:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Logging.AddConsole();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapGet("/Test", async (ILogger<Program> logger, HttpResponse response)
=>
{
    logger.LogInformation("Testing logging in Program.cs");
    await response.WriteAsync("Testing");
});

app.Run();
```

The following example calls AddSimpleConsole in `Program.cs`, disables color output, and logs the `/Test` endpoint:

C#

```
using Microsoft.Extensions.Logging.Console;

var builder = WebApplication.CreateBuilder(args);

builder.Logging.AddSimpleConsole(i => i.ColorBehavior =
LoggerColorBehavior.Disabled);

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapGet("/Test", async (ILogger<Program> logger, HttpResponse response)
=>
{
    logger.LogInformation("Testing logging in Program.cs");
    await response.WriteAsync("Testing");
});

app.Run();
```

Set log level by command line, environment variables, and other configuration

Log level can be set by any of the configuration providers.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

The following commands:

- Set the environment key `Logging:LogLevel:Microsoft` to a value of `Information` on Windows.
- Test the settings when using an app created with the ASP.NET Core web application templates. The `dotnet run` command must be run in the project directory after using `set`.

.NET CLI

```
set Logging__LogLevel__Microsoft=Information
dotnet run
```

The preceding environment setting:

- Is only set in processes launched from the command window they were set in.
- Isn't read by browsers launched with Visual Studio.

The following setx command also sets the environment key and value on Windows. Unlike `set`, `setx` settings are persisted. The `/M` switch sets the variable in the system environment. If `/M` isn't used, a user environment variable is set.

Console

```
setx Logging__LogLevel__Microsoft Information /M
```

Consider the following `appsettings.json` file:

JSON

```
"Logging": {
  "Console": {
    "LogLevel": {
      "Microsoft.Hosting.Lifetime": "Trace"
    }
  }
}
```

The following command sets the preceding configuration in the environment:

```
Console
setx Logging__Console__LogLevel__Microsoft.Hosting.Lifetime Trace /M
```

> **Note**
>
> When configuring environment variables with names that contain `.` (periods) in macOS and Linux, consider the "Exporting a variable with a dot (.) in it" question on **Stack Exchange** and its corresponding accepted answer.

On Azure App Service, select **New application setting** on the **Settings** > **Configuration** page. Azure App Service application settings are:

- Encrypted at rest and transmitted over an encrypted channel.
- Exposed as environment variables.

For more information, see Azure Apps: Override app configuration using the Azure portal.

For more information on setting ASP.NET Core configuration values using environment variables, see environment variables. For information on using other configuration sources, including the command line, Azure Key Vault, Azure App Configuration, other file formats, and more, see Configuration in ASP.NET Core.

How filtering rules are applied

When an ILogger<TCategoryName> object is created, the ILoggerFactory object selects a single rule per provider to apply to that logger. All messages written by an `ILogger` instance are filtered based on the selected rules. The most specific rule for each provider and category pair is selected from the available rules.

The following algorithm is used for each provider when an `ILogger` is created for a given category:

- Select all rules that match the provider or its alias. If no match is found, select all rules with an empty provider.
- From the result of the preceding step, select rules with longest matching category prefix. If no match is found, select all rules that don't specify a category.
- If multiple rules are selected, take the **last** one.
- If no rules are selected, use `MinimumLevel`.

Logging output from dotnet run and Visual Studio

Logs created with the default logging providers are displayed:

- In Visual Studio
  - In the Debug output window when debugging.
  - In the ASP.NET Core Web Server window.
- In the console window when the app is run with `dotnet run`.

Logs that begin with "Microsoft" categories are from .NET. .NET and application code use the same logging API and providers.

Log category

When an `ILogger` object is created, a *category* is specified. That category is included with each log message created by that instance of `ILogger`. The category string is arbitrary, but the convention is to use the fully qualified class name. For example, in a controller the name might be `"TodoApi.Controllers.TodoController"`. The ASP.NET Core web apps use `ILogger<T>` to automatically get an `ILogger` instance that uses the fully qualified type name of `T` as the category:

C#

```
public class PrivacyModel : PageModel
{
    private readonly ILogger<PrivacyModel> _logger;

    public PrivacyModel(ILogger<PrivacyModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.PrivacyModel called.");
    }
}
```

If further categorization is desired, the convention is to use a hierarchical name by appending a subcategory to the fully qualified class name, and explicitly specify the category using `ILoggerFactory.CreateLogger`:

C#

```
public class ContactModel : PageModel
{
    private readonly ILogger _logger;

    public ContactModel(ILoggerFactory logger)
    {
        _logger =
logger.CreateLogger("TodoApi.Pages.ContactModel.MyCategory");
    }

    public void OnGet()
    {
        _logger.LogInformation("GET Pages.ContactModel called.");
    }
```

Calling `CreateLogger` with a fixed name can be useful when used in multiple methods so the events can be organized by category.

`ILogger<T>` is equivalent to calling `CreateLogger` with the fully qualified type name of `T`.

Log level

The following table lists the LogLevel values, the convenience `Log{LogLevel}` extension method, and the suggested usage:

Expand table

LogLevel	Value	Method	Description
Trace	0	LogTrace	Contain the most detailed messages. These messages may contain sensitive app data. These messages are disabled by default and should ***not*** be enabled in production.
Debug	1	LogDebug	For debugging and development. Use with caution in production due to the high volume.
Information	2	LogInformation	Tracks the general flow of the app. May have long-term value.
Warning	3	LogWarning	For abnormal or unexpected events. Typically includes errors or conditions that don't cause the app to fail.
Error	4	LogError	For errors and exceptions that cannot be handled. These messages indicate a failure in the current operation or request, not an app-wide failure.

LogLevel	Value	Method	Description
Critical	5	LogCritical	For failures that require immediate attention. Examples: data loss scenarios, out of disk space.
None	6		Specifies that a logging category shouldn't write messages.

In the previous table, the `LogLevel` is listed from lowest to highest severity.

The Log method's first parameter, LogLevel, indicates the severity of the log. Rather than calling `Log(LogLevel, ...)`, most developers call the Log{LOG LEVEL} extension methods, where the `{LOG LEVEL}` placeholder is the log level. For example, the following two logging calls are functionally equivalent and produce the same log:

C#

```
[HttpGet]
public IActionResult Test1(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);

    _logger.Log(LogLevel.Information, MyLogEvents.TestItem, routeInfo);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    return ControllerContext.MyDisplayRouteInfo();
}
```

`MyLogEvents.TestItem` is the event ID. `MyLogEvents` is part of the sample app and is displayed in the Log event ID section.

MyDisplayRouteInfo and ToCtxString are provided by the Rick.Docs.Samples.RouteInfo NuGet package. The methods display `Controller` and `Razor Page` route information.

The following code creates `Information` and `Warning` logs:

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
```

```
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT
FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

In the preceding code, the first `Log{LOG LEVEL}` parameter, `MyLogEvents.GetItem`, is the Log event ID. The second parameter is a message template with placeholders for argument values provided by the remaining method parameters. The method parameters are explained in the message template section later in this document.

Call the appropriate `Log{LOG LEVEL}` method to control how much log output is written to a particular storage medium. For example:

- In production:
  - Logging at the `Trace`, `Debug`, or `Information` levels produces a high-volume of detailed log messages. To control costs and not exceed data storage limits, log `Trace`, `Debug`, or `Information` level messages to a high-volume, low-cost data store. Consider limiting `Trace`, `Debug`, or `Information` to specific categories.
  - Logging at `Warning` through `Critical` levels should produce few log messages.
    - Costs and storage limits usually aren't a concern.
    - Few logs allow more flexibility in data store choices.
- In development:
  - Set to `Warning`.
  - Add `Trace`, `Debug`, or `Information` messages when troubleshooting. To limit output, set `Trace`, `Debug`, or `Information` only for the categories under investigation.

ASP.NET Core writes logs for framework events. For example, consider the log output for:

- A Razor Pages app created with the ASP.NET Core templates.
- Logging set to `Logging:Console:LogLevel:Microsoft:Information`.
- Navigation to the Privacy page:

Console

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 GET https://localhost:5001/Privacy
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint '/Privacy'
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
```

```
      Route matched with {page = "/Privacy"}. Executing page /Privacy
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
      Executing handler method DefaultRP.Pages.PrivacyModel.OnGet -
ModelState is Valid
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
      Executed handler method OnGet, returned result .
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[103]
      Executing an implicit handler method - ModelState is Valid
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[104]
      Executed an implicit handler method, returned result
Microsoft.AspNetCore.Mvc.RazorPages.PageResult.
info:
Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
      Executed page /Privacy in 74.5188ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint '/Privacy'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 149.3023ms 200 text/html; charset=utf-8
```

The following JSON sets `Logging:Console:LogLevel:Microsoft:Information`:

JSON

```
{
  "Logging": {      // Default, all providers.
    "LogLevel": {
      "Microsoft": "Warning"
    },
    "Console": { // Console provider.
      "LogLevel": {
        "Microsoft": "Information"
      }
    }
  }
}
```

Log event ID

Each log can specify an *event ID*. The sample app uses the `MyLogEvents` class to define event IDs:

C#

```
public class MyLogEvents
{
```

```
    public const int GenerateItems = 1000;
    public const int ListItems     = 1001;
    public const int GetItem       = 1002;
    public const int InsertItem    = 1003;
    public const int UpdateItem    = 1004;
    public const int DeleteItem    = 1005;

    public const int TestItem      = 3000;

    public const int GetItemNotFound    = 4000;
    public const int UpdateItemNotFound = 4001;
}
```

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT
FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

An event ID associates a set of events. For example, all logs related to displaying a list of items on a page might be 1001.

The logging provider may store the event ID in an ID field, in the logging message, or not at all. The Debug provider doesn't show event IDs. The console provider shows event IDs in brackets after the category:

Console

```
info: TodoApi.Controllers.TodoItemsController[1002]
      Getting item 1
warn: TodoApi.Controllers.TodoItemsController[4000]
      Get(1) NOT FOUND
```

Some logging providers store the event ID in a field, which allows for filtering on the ID.

Log message template

Each log API uses a message template. The message template can contain placeholders for which arguments are provided. Use names for the placeholders, not numbers.

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}", id);

    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, "Get({Id}) NOT
FOUND", id);
        return NotFound();
    }

    return ItemToDTO(todoItem);
}
```

The *order of the parameters*, not their placeholder names, determines which parameters are used to provide placeholder values in log messages. In the following code, the parameter names are out of sequence in the placeholders of the message template:

C#

```
var apples = 1;
var pears = 2;
var bananas = 3;

_logger.LogInformation("Parameters: {Pears}, {Bananas}, {Apples}", apples,
pears, bananas);
```

However, the parameters are assigned to the placeholders in the order: `apples`, `pears`, `bananas`. The log message reflects the *order of the parameters*:

text

```
Parameters: 1, 2, 3
```

This approach allows logging providers to implement semantic or structured logging. The arguments themselves are passed to the logging system, not just the formatted

message template. This enables logging providers to store the parameter values as fields. For example, consider the following logger method:

```
C#

_logger.LogInformation("Getting item {Id} at {RequestTime}", id,
DateTime.Now);
```

For example, when logging to Azure Table Storage:

- Each Azure Table entity can have `ID` and `RequestTime` properties.
- Tables with properties simplify queries on logged data. For example, a query can find all logs within a particular `RequestTime` range without having to parse the time out of the text message.

Log exceptions

The logger methods have overloads that take an exception parameter:

```
C#

[HttpGet("{id}")]
public IActionResult TestExp(int id)
{
    var routeInfo = ControllerContext.ToCtxString(id);
    _logger.LogInformation(MyLogEvents.TestItem, routeInfo);

    try
    {
        if (id == 3)
        {
            throw new Exception("Test exception");
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(MyLogEvents.GetItemNotFound, ex, "TestExp({Id})",
id);
        return NotFound();
    }

    return ControllerContext.MyDisplayRouteInfo();
}
```

MyDisplayRouteInfo and ToCtxString are provided by the Rick.Docs.Samples.RouteInfo NuGet package. The methods display `Controller` and `Razor Page` route information.

Exception logging is provider-specific.

Default log level

If the default log level is not set, the default log level value is `Information`.

For example, consider the following web app:

- Created with the ASP.NET web app templates.
- `appsettings.json` and `appsettings.Development.json` deleted or renamed.

With the preceding setup, navigating to the privacy or home page produces many `Trace`, `Debug`, and `Information` messages with `Microsoft` in the category name.

The following code sets the default log level when the default log level is not set in configuration:

C#

```
var builder = WebApplication.CreateBuilder();
builder.Logging.SetMinimumLevel(LogLevel.Warning);
```

Generally, log levels should be specified in configuration and not code.

Filter function

A filter function is invoked for all providers and categories that don't have rules assigned to them by configuration or code:

C#

```
var builder = WebApplication.CreateBuilder();
builder.Logging.AddFilter((provider, category, logLevel) =>
{
    if (provider.Contains("ConsoleLoggerProvider")
        && category.Contains("Controller")
        && logLevel >= LogLevel.Information)
    {
        return true;
    }
    else if (provider.Contains("ConsoleLoggerProvider")
        && category.Contains("Microsoft")
        && logLevel >= LogLevel.Information)
    {
        return true;
    }
    else
```

```
    {
        return false;
    }
});
```

The preceding code displays console logs when the category contains `Controller` or `Microsoft` and the log level is `Information` or higher.

Generally, log levels should be specified in configuration and not code.

ASP.NET Core categories

The following table contains some categories used by ASP.NET Core.

Expand table

Category	Notes
`Microsoft.AspNetCore`	General ASP.NET Core diagnostics.
`Microsoft.AspNetCore.DataProtection`	Which keys were considered, found, and used.
`Microsoft.AspNetCore.HostFiltering`	Hosts allowed.
`Microsoft.AspNetCore.Hosting`	How long HTTP requests took to complete and what time they started. Which hosting startup assemblies were loaded.
`Microsoft.AspNetCore.Mvc`	MVC and Razor diagnostics. Model binding, filter execution, view compilation, action selection.
`Microsoft.AspNetCore.Routing`	Route matching information.
`Microsoft.AspNetCore.Server`	Connection start, stop, and keep alive responses. HTTPS certificate information.
`Microsoft.AspNetCore.StaticFiles`	Files served.

To view more categories in the console window, set **`appsettings.Development.json`** to the following:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Trace",
```

```
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

For a list of Entity Framework categories, see EF Message categories.

Log scopes

A *scope* can group a set of logical operations. This grouping can be used to attach the same data to each log that's created as part of a set. For example, every log created as part of processing a transaction can include the transaction ID.

A scope:

- Is an IDisposable type that's returned by the BeginScope method.
- Lasts until it's disposed.

The following providers support scopes:

- `Console`
- AzureAppServicesFile and AzureAppServicesBlob

Use a scope by wrapping logger calls in a `using` block:

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
{
    TodoItem todoItem;
    var transactionId = Guid.NewGuid().ToString();
    using (_logger.BeginScope(new List<KeyValuePair<string, object>>
        {
            new KeyValuePair<string, object>("TransactionId",
transactionId),
        }))
    {
        _logger.LogInformation(MyLogEvents.GetItem, "Getting item {Id}",
id);

        todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            _logger.LogWarning(MyLogEvents.GetItemNotFound,
                "Get({Id}) NOT FOUND", id);
            return NotFound();
        }
```

```
    }

    return ItemToDTO(todoItem);
}
```

Built-in logging providers

ASP.NET Core includes the following logging providers as part of the shared framework:

- Console
- Debug
- EventSource
- EventLog

The following logging providers are shipped by Microsoft, but not as part of the shared framework. They must be installed as additional nuget.

- AzureAppServicesFile and AzureAppServicesBlob
- ApplicationInsights

ASP.NET Core doesn't include a logging provider for writing logs to files. To write logs to files from an ASP.NET Core app, consider using a third-party logging provider.

For information on `stdout` and debug logging with the ASP.NET Core Module, see Troubleshoot ASP.NET Core on Azure App Service and IIS and ASP.NET Core Module (ANCM) for IIS.

Console

The `Console` provider logs output to the console. For more information on viewing `Console` logs in development, see Logging output from dotnet run and Visual Studio.

Debug

The `Debug` provider writes log output by using the System.Diagnostics.Debug class. Calls to `System.Diagnostics.Debug.WriteLine` write to the `Debug` provider.

On Linux, the `Debug` provider log location is distribution-dependent and may be one of the following:

- `/var/log/message`
- `/var/log/syslog`

Event Source

The `EventSource` provider writes to a cross-platform event source with the name `Microsoft-Extensions-Logging`. On Windows, the provider uses ETW.

dotnet-trace tooling

The dotnet-trace tool is a cross-platform CLI global tool that enables the collection of .NET Core traces of a running process. The tool collects Microsoft.Extensions.Logging.EventSource provider data using a LoggingEventSource.

For installation instructions, see dotnet-trace.

Use the `dotnet-trace` tooling to collect a trace from an app:

1. Run the app with the `dotnet run` command.
2. Determine the process identifier (PID) of the .NET Core app:

   .NET CLI

   ```
   dotnet-trace ps
   ```

   Find the PID for the process that has the same name as the app's assembly.
3. Execute the `dotnet-trace` command.

   General command syntax:

   .NET CLI

   ```
   dotnet-trace collect -p {PID}
       --providers Microsoft-Extensions-Logging:{Keyword}:{Provider Level}
           :FilterSpecs=\"
               {Logger Category 1}:{Category Level 1};
               {Logger Category 2}:{Category Level 2};
               ...
               {Logger Category N}:{Category Level N}\"
   ```

   When using a PowerShell command shell, enclose the `--providers` value in single quotes (`'`):

   .NET CLI

   ```
   dotnet-trace collect -p {PID}
       --providers 'Microsoft-Extensions-Logging:{Keyword}:{Provider
   ```

```
Level}
        :FilterSpecs=\"
            {Logger Category 1}:{Category Level 1};
            {Logger Category 2}:{Category Level 2};
            ...
            {Logger Category N}:{Category Level N}\"'
```

On non-Windows platforms, add the `-f speedscope` option to change the format of the output trace file to `speedscope`.

The following table defines the Keyword:

Expand table

Keyword	Description
1	Log meta events about the `LoggingEventSource`. Doesn't log events from `ILogger`.
2	Turns on the `Message` event when `ILogger.Log()` is called. Provides information in a programmatic (not formatted) way.
4	Turns on the `FormatMessage` event when `ILogger.Log()` is called. Provides the formatted string version of the information.
8	Turns on the `MessageJson` event when `ILogger.Log()` is called. Provides a JSON representation of the arguments.

The following table lists the provider levels:

Expand table

Provider Level	Description
0	`LogAlways`
1	`Critical`
2	`Error`
3	`Warning`
4	`Informational`
5	`Verbose`

The parsing for a category level can be either a string or a number:

Expand table

Category named value	Numeric value
`Trace`	0
`Debug`	1
`Information`	2
`Warning`	3
`Error`	4
`Critical`	5

The provider level and category level:

- Are in reverse order.
- The string constants aren't all identical.

If no `FilterSpecs` are specified then the `EventSourceLogger` implementation attempts to convert the provider level to a category level and applies it to all categories.

Expand table

Provider Level	Category Level
`Verbose` (5)	`Debug` (1)
`Informational` (4)	`Information` (2)
`Warning` (3)	`Warning` (3)
`Error` (2)	`Error` (4)
`Critical` (1)	`Critical` (5)

If `FilterSpecs` are provided, any category that is included in the list uses the category level encoded there, all other categories are filtered out.

The following examples assume:

- An app is running and calling `logger.LogDebug("12345")`.
- The process ID (PID) has been set via `set PID=12345`, where `12345` is the actual PID.

Consider the following command:

.NET CLI

```
dotnet-trace collect -p %PID% --providers Microsoft-Extensions-
Logging:4:5
```

The preceding command:

- Captures debug messages.
- Doesn't apply a `FilterSpecs`.
- Specifies level 5 which maps category Debug.

Consider the following command:

.NET CLI

```
dotnet-trace collect -p %PID%  --providers Microsoft-Extensions-
Logging:4:5:\"FilterSpecs=*:5\"
```

The preceding command:

- Doesn't capture debug messages because the category level 5 is `Critical`.
- Provides a `FilterSpecs`.

The following command captures debug messages because category level 1 specifies `Debug`.

.NET CLI

```
dotnet-trace collect -p %PID%  --providers Microsoft-Extensions-
Logging:4:5:\"FilterSpecs=*:1\"
```

The following command captures debug messages because category specifies `Debug`.

.NET CLI

```
dotnet-trace collect -p %PID%  --providers Microsoft-Extensions-
Logging:4:5:\"FilterSpecs=*:Debug\"
```

`FilterSpecs` entries for `{Logger Category}` and `{Category Level}` represent additional log filtering conditions. Separate `FilterSpecs` entries with the `;` semicolon character.

Example using a Windows command shell:

.NET CLI

```
dotnet-trace collect -p %PID% --providers Microsoft-Extensions-
Logging:4:2:FilterSpecs=\"Microsoft.AspNetCore.Hosting*:4\"
```

The preceding command activates:

- The Event Source logger to produce formatted strings (`4`) for errors (`2`).
- `Microsoft.AspNetCore.Hosting` logging at the `Informational` logging level (`4`).

4. Stop the `dotnet-trace` tooling by pressing the Enter key or `Ctrl`+`C`.

   The trace is saved with the name `trace.nettrace` in the folder where the `dotnet-trace` command is executed.

5. Open the trace with Perfview. Open the `trace.nettrace` file and explore the trace events.

If the app doesn't build the host with WebApplication.CreateBuilder, add the Event Source provider to the app's logging configuration.

For more information, see:

- Trace for performance analysis utility (dotnet-trace) (.NET Core documentation)
- Trace for performance analysis utility (dotnet-trace) (dotnet/diagnostics GitHub repository documentation)
- LoggingEventSource
- EventLevel
- Perfview: Useful for viewing Event Source traces.

Perfview

Use the PerfView utility to collect and view logs. There are other tools for viewing ETW logs, but PerfView provides the best experience for working with the ETW events emitted by ASP.NET Core.

To configure PerfView for collecting events logged by this provider, add the string `*Microsoft-Extensions-Logging` to the **Additional Providers** list. Don't miss the `*` at the start of the string.

Windows EventLog

The `EventLog` provider sends log output to the Windows Event Log. Unlike the other providers, the `EventLog` provider does ***not*** inherit the default non-provider settings. If `EventLog` log settings aren't specified, they default to LogLevel.Warning.

To log events lower than LogLevel.Warning, explicitly set the log level. The following example sets the Event Log default log level to LogLevel.Information:

JSON

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

AddEventLog overloads can pass in EventLogSettings. If `null` or not specified, the following default settings are used:

- `LogName`: "Application"
- `SourceName`: ".NET Runtime"
- `MachineName`: The local machine name is used.

The following code changes the `SourceName` from the default value of `".NET Runtime"` to `MyLogs`:

C#

```
var builder = WebApplication.CreateBuilder();
builder.Logging.AddEventLog(eventLogSettings =>
{
    eventLogSettings.SourceName = "MyLogs";
});
```

Azure App Service

The Microsoft.Extensions.Logging.AzureAppServices provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account.

The provider package isn't included in the shared framework. To use the provider, add the provider package to the project.

To configure provider settings, use AzureFileLoggerOptions and AzureBlobLoggerOptions, as shown in the following example:

C#

```
using Microsoft.Extensions.Logging.AzureAppServices;

var builder = WebApplication.CreateBuilder();
builder.Logging.AddAzureWebAppDiagnostics();
builder.Services.Configure<AzureFileLoggerOptions>(options =>
{
    options.FileName = "azure-diagnostics-";
    options.FileSizeLimit = 50 * 1024;
    options.RetainedFileCountLimit = 5;
});
builder.Services.Configure<AzureBlobLoggerOptions>(options =>
{
    options.BlobName = "log.txt";
});
```

When deployed to Azure App Service, the app uses the settings in the App Service logs section of the **App Service** page of the Azure portal. When the following settings are updated, the changes take effect immediately without requiring a restart or redeployment of the app.

- **Application Logging (Filesystem)**
- **Application Logging (Blob)**

The default location for log files is in the `D:\\home\\LogFiles\\Application` folder, and the default file name is `diagnostics-yyyymmdd.txt`. The default file size limit is 10 MB, and the default maximum number of files retained is 2. The default blob name is `{app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt`.

This provider only logs when the project runs in the Azure environment.

Azure log streaming

Azure log streaming supports viewing log activity in real time from:

- The app server
- The web server
- Failed request tracing

To configure Azure log streaming:

- Navigate to the **App Service logs** page from the app's portal page.
- Set **Application Logging (Filesystem)** to **On**.
- Choose the log **Level**. This setting only applies to Azure log streaming.

Navigate to the **Log Stream** page to view logs. The logged messages are logged with the `ILogger` interface.

Azure Application Insights

The Microsoft.Extensions.Logging.ApplicationInsights provider package writes logs to Azure Application Insights. Application Insights is a service that monitors a web app and provides tools for querying and analyzing the telemetry data. If you use this provider, you can query and analyze your logs by using the Application Insights tools.

The logging provider is included as a dependency of Microsoft.ApplicationInsights.AspNetCore, which is the package that provides all available telemetry for ASP.NET Core. If you use this package, you don't have to install the provider package.

The Microsoft.ApplicationInsights.Web package is for ASP.NET 4.x, not ASP.NET Core.

For more information, see the following resources:

- Application Insights overview
- Application Insights for ASP.NET Core applications: Start here if you want to implement the full range of Application Insights telemetry along with logging.
- ApplicationInsightsLoggerProvider for .NET Core ILogger logs: Start here if you want to implement the logging provider without the rest of Application Insights telemetry.
- Application Insights logging adapters
- Install, configure, and initialize the Application Insights SDK interactive tutorial.

Third-party logging providers

Third-party logging frameworks that work with ASP.NET Core:

- elmah.io (GitHub repo)
- Gelf (GitHub repo)
- JSNLog (GitHub repo)
- KissLog.net (GitHub repo)
- Log4Net (GitHub repo)

- NLog (GitHub repo)
- PLogger (GitHub repo)
- Sentry (GitHub repo)
- Serilog (GitHub repo)
- Stackdriver (Github repo)

Some third-party frameworks can perform semantic logging, also known as structured logging.

Using a third-party framework is similar to using one of the built-in providers:

1. Add a NuGet package to your project.
2. Call an `ILoggerFactory` extension method provided by the logging framework.

For more information, see each provider's documentation. Third-party logging providers aren't supported by Microsoft.

No asynchronous logger methods

Logging should be so fast that it isn't worth the performance cost of asynchronous code. If a logging data store is slow, don't write to it directly. Consider writing the log messages to a fast store initially, then moving them to the slow store later. For example, when logging to SQL Server, don't do so directly in a `Log` method, since the `Log` methods are synchronous. Instead, synchronously add log messages to an in-memory queue and have a background worker pull the messages out of the queue to do the asynchronous work of pushing data to SQL Server. For more information, see Guidance on how to log to a message queue for slow data stores (dotnet/AspNetCore.Docs #11801).

Change log levels in a running app

The Logging API doesn't include a scenario to change log levels while an app is running. However, some configuration providers are capable of reloading configuration, which takes immediate effect on logging configuration. For example, the File Configuration Provider, reloads logging configuration by default. If configuration is changed in code while an app is running, the app can call IConfigurationRoot.Reload to update the app's logging configuration.

ILogger and ILoggerFactory

The ILogger<TCategoryName> and ILoggerFactory interfaces and implementations are included in the .NET Core SDK. They are also available in the following NuGet packages:

- The interfaces are in Microsoft.Extensions.Logging.Abstractions.
- The default implementations are in Microsoft.Extensions.Logging.

Apply log filter rules in code

The preferred approach for setting log filter rules is by using Configuration.

The following example shows how to register filter rules in code:

C#

```
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Logging.Debug;

var builder = WebApplication.CreateBuilder();
builder.Logging.AddFilter("System", LogLevel.Debug);
builder.Logging.AddFilter<DebugLoggerProvider>("Microsoft",
LogLevel.Information);
builder.Logging.AddFilter<ConsoleLoggerProvider>("Microsoft",
LogLevel.Trace);
```

`logging.AddFilter("System", LogLevel.Debug)` specifies the `System` category and log level `Debug`. The filter is applied to all providers because a specific provider was not configured.

`AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)` specifies:

- The `Debug` logging provider.
- Log level `Information` and higher.
- All categories starting with `"Microsoft"`.

Automatically log scope with `SpanId`, `TraceId`, `ParentId`, `Baggage`, and `Tags`.

The logging libraries implicitly create a scope object with `SpanId`, `TraceId`, `ParentId`, `Baggage`, and `Tags`. This behavior is configured via ActivityTrackingOptions.

C#

```
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Logging.AddSimpleConsole(options =>
{
    options.IncludeScopes = true;
});

builder.Logging.Configure(options =>
{
    options.ActivityTrackingOptions = ActivityTrackingOptions.SpanId
                                        | ActivityTrackingOptions.TraceId
                                        | ActivityTrackingOptions.ParentId
                                        | ActivityTrackingOptions.Baggage
                                        | ActivityTrackingOptions.Tags;
});
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

If the `traceparent` http request header is set, the `ParentId` in the log scope shows the W3C `parent-id` from in-bound `traceparent` header and the `SpanId` in the log scope shows the updated `parent-id` for the next out-bound step/span. For more information, see Mutating the traceparent Field.

Create a custom logger

To create a custom logger, see Implement a custom logging provider in .NET.

Additional resources

- Improving logging performance with source generators
- Behind [LogProperties] and the new telemetry logging source generator
- Microsoft.Extensions.Logging source on GitHub
- View or download sample code (how to download).
- High performance logging
- Logging bugs should be created in the dotnet/runtime GitHub repository.
- ASP.NET Core Blazor logging

HTTP logging in ASP.NET Core

Article • 06/17/2024

HTTP logging is a middleware that logs information about incoming HTTP requests and HTTP responses. HTTP logging provides logs of:

- HTTP request information
- Common properties
- Headers
- Body
- HTTP response information

HTTP logging can:

- Log all requests and responses or only requests and responses that meet certain criteria.
- Select which parts of the request and response are logged.
- Allow you to redact sensitive information from the logs.

HTTP logging ***can reduce the performance of an app***, especially when logging the request and response bodies. Consider the performance impact when selecting fields to log. Test the performance impact of the selected logging properties.

> ⚠ **Warning**
>
> HTTP logging can potentially log personally identifiable information (PII). Consider the risk and avoid logging sensitive information.

Enable HTTP logging

HTTP logging is enabled by calling AddHttpLogging and UseHttpLogging, as shown in the following example:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(o => { });

var app = builder.Build();

app.UseHttpLogging();
```

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
}
app.UseStaticFiles();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The empty lambda in the preceding example of calling `AddHttpLogging` adds the middleware with the default configuration. By default, HTTP logging logs common properties such as path, status-code, and headers for requests and responses.

Add the following line to the `appsettings.Development.json` file at the `"LogLevel": {` level so the HTTP logs are displayed:

JSON

```
 "Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware": "Information"
```

With the default configuration, a request and response is logged as a pair of messages similar to the following example:

Output

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Protocol: HTTP/2
      Method: GET
      Scheme: https
      PathBase:
      Path: /
      Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,
*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
      Host: localhost:52941
      User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.0.0 Safari/537.36
Edg/118.0.2088.61
      Accept-Encoding: gzip, deflate, br
      Accept-Language: en-US,en;q=0.9
      Upgrade-Insecure-Requests: [Redacted]
      sec-ch-ua: [Redacted]
      sec-ch-ua-mobile: [Redacted]
      sec-ch-ua-platform: [Redacted]
      sec-fetch-site: [Redacted]
      sec-fetch-mode: [Redacted]
      sec-fetch-user: [Redacted]
      sec-fetch-dest: [Redacted]
```

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: text/plain; charset=utf-8
      Date: Tue, 24 Oct 2023 02:03:53 GMT
      Server: Kestrel
```

HTTP logging options

To configure global options for the HTTP logging middleware, call AddHttpLogging in `Program.cs`, using the lambda to configure HttpLoggingOptions.

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
}

app.UseStaticFiles();

app.UseHttpLogging();

app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

app.MapGet("/", () => "Hello World!");
```

```
app.Run();
```

> ⓘ Note
>
> In the preceding sample and following samples, `UseHttpLogging` is called after `UseStaticFiles`, so HTTP logging is not enabled for static files. To enable static file HTTP logging, call `UseHttpLogging` before `UseStaticFiles`.

`LoggingFields`

HttpLoggingOptions.LoggingFields is an enum flag that configures specific parts of the request and response to log. `HttpLoggingOptions.LoggingFields` defaults to RequestPropertiesAndHeaders | ResponsePropertiesAndHeaders.

`RequestHeaders` and `ResponseHeaders`

RequestHeaders and ResponseHeaders are sets of HTTP headers that are logged. Header values are only logged for header names that are in these collections. The following code adds `sec-ch-ua` to the RequestHeaders, so the value of the `sec-ch-ua` header is logged. And it adds `MyResponseHeader` to the ResponseHeaders, so the value of the `MyResponseHeader` header is logged. If these lines are removed, the values of these headers are `[Redacted]`.

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
```

```
{
    app.UseExceptionHandler("/Error");
}

app.UseStaticFiles();

app.UseHttpLogging();

app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

app.MapGet("/", () => "Hello World!");

app.Run();
```

`MediaTypeOptions`

MediaTypeOptions provides configuration for selecting which encoding to use for a specific media type.

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
}

app.UseStaticFiles();

app.UseHttpLogging();
```

```
app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

app.MapGet("/", () => "Hello World!");

app.Run();
```

This approach can also be used to enable logging for data that isn't logged by default (for example, form data, which might have a media type such as `application/x-www-form-urlencoded` or `multipart/form-data`).

`MediaTypeOptions` methods

- AddText
- AddBinary
- Clear

`RequestBodyLogLimit` and `ResponseBodyLogLimit`

- RequestBodyLogLimit
- ResponseBodyLogLimit

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
```

```
{
    app.UseExceptionHandler("/Error");
}

app.UseStaticFiles();

app.UseHttpLogging();

app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

app.MapGet("/", () => "Hello World!");

app.Run();
```

`CombineLogs`

Setting CombineLogs to `true` configures the middleware to consolidate all of its enabled logs for a request and response into one log at the end. This includes the request, request body, response, response body, and duration.

C#

```
using Microsoft.AspNetCore.HttpLogging;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.All;
    logging.RequestHeaders.Add("sec-ch-ua");
    logging.ResponseHeaders.Add("MyResponseHeader");
    logging.MediaTypeOptions.AddText("application/javascript");
    logging.RequestBodyLogLimit = 4096;
    logging.ResponseBodyLogLimit = 4096;
    logging.CombineLogs = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
}

app.UseStaticFiles();
```

```
app.UseHttpLogging();

app.Use(async (context, next) =>
{
    context.Response.Headers["MyResponseHeader"] =
        new string[] { "My Response Header Value" };

    await next();
});

app.MapGet("/", () => "Hello World!");

app.Run();
```

Endpoint-specific configuration

For endpoint-specific configuration in minimal API apps, a WithHttpLogging extension method is available. The following example shows how to configure HTTP logging for one endpoint:

C#

```
app.MapGet("/response", () => "Hello World! (logging response)")
    .WithHttpLogging(HttpLoggingFields.ResponsePropertiesAndHeaders);
```

For endpoint-specific configuration in apps that use controllers, the [HttpLogging] attribute is available. The attribute can also be used in minimal API apps, as shown in the following example:

C#

```
app.MapGet("/duration", [HttpLogging(loggingFields:
HttpLoggingFields.Duration)]
    () => "Hello World! (logging duration)");
```

`IHttpLoggingInterceptor`

IHttpLoggingInterceptor is the interface for a service that can be implemented to handle per-request and per-response callbacks for customizing what details get logged. Any endpoint-specific log settings are applied first and can then be overridden in these callbacks. An implementation can:

- Inspect a request or response.

- Enable or disable any HttpLoggingFields.
- Adjust how much of the request or response body is logged.
- Add custom fields to the logs.

Register an `IHttpLoggingInterceptor` implementation by calling AddHttpLoggingInterceptor<T> in `Program.cs`. If multiple `IHttpLoggingInterceptor` instances are registered, they're run in the order registered.

The following example shows how to register an `IHttpLoggingInterceptor` implementation:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpLogging(logging =>
{
    logging.LoggingFields = HttpLoggingFields.Duration;
});
builder.Services.AddHttpLoggingInterceptor<SampleHttpLoggingInterceptor>();
```

The following example is an `IHttpLoggingInterceptor` implementation that:

- Inspects the request method and disables logging for POST requests.
- For non-POST requests:
  - Redacts request path, request headers, and response headers.
  - Adds custom fields and field values to the request and response logs.

C#

```
using Microsoft.AspNetCore.HttpLogging;

namespace HttpLoggingSample;

internal sealed class SampleHttpLoggingInterceptor : IHttpLoggingInterceptor
{
    public ValueTask OnRequestAsync(HttpLoggingInterceptorContext
logContext)
    {
        if (logContext.HttpContext.Request.Method == "POST")
        {
            // Don't log anything if the request is a POST.
            logContext.LoggingFields = HttpLoggingFields.None;
        }

        // Don't enrich if we're not going to log any part of the request.
        if (!logContext.IsAnyEnabled(HttpLoggingFields.Request))
        {
            return default;
```

```
        }

        if (logContext.TryDisable(HttpLoggingFields.RequestPath))
        {
            RedactPath(logContext);
        }

        if (logContext.TryDisable(HttpLoggingFields.RequestHeaders))
        {
            RedactRequestHeaders(logContext);
        }

        EnrichRequest(logContext);

        return default;
    }

    public ValueTask OnResponseAsync(HttpLoggingInterceptorContext
logContext)
    {
        // Don't enrich if we're not going to log any part of the response
        if (!logContext.IsAnyEnabled(HttpLoggingFields.Response))
        {
            return default;
        }

        if (logContext.TryDisable(HttpLoggingFields.ResponseHeaders))
        {
            RedactResponseHeaders(logContext);
        }

        EnrichResponse(logContext);

        return default;
    }

    private void RedactPath(HttpLoggingInterceptorContext logContext)
    {
        logContext.AddParameter(nameof(logContext.HttpContext.Request.Path),
"RedactedPath");
    }

    private void RedactRequestHeaders(HttpLoggingInterceptorContext
logContext)
    {
        foreach (var header in logContext.HttpContext.Request.Headers)
        {
            logContext.AddParameter(header.Key, "RedactedHeader");
        }
    }

    private void EnrichRequest(HttpLoggingInterceptorContext logContext)
    {
        logContext.AddParameter("RequestEnrichment", "Stuff");
    }
```

```
    private void RedactResponseHeaders(HttpLoggingInterceptorContext
logContext)
    {
        foreach (var header in logContext.HttpContext.Response.Headers)
        {
            logContext.AddParameter(header.Key, "RedactedHeader");
        }
    }

    private void EnrichResponse(HttpLoggingInterceptorContext logContext)
    {
        logContext.AddParameter("ResponseEnrichment", "Stuff");
    }
}
```

With this interceptor, a POST request doesn't generate any logs even if HTTP logging is configured to log `HttpLoggingFields.All`. A GET request generates logs similar to the following example:

Output

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[1]
      Request:
      Path: RedactedPath
      Accept: RedactedHeader
      Host: RedactedHeader
      User-Agent: RedactedHeader
      Accept-Encoding: RedactedHeader
      Accept-Language: RedactedHeader
      Upgrade-Insecure-Requests: RedactedHeader
      sec-ch-ua: RedactedHeader
      sec-ch-ua-mobile: RedactedHeader
      sec-ch-ua-platform: RedactedHeader
      sec-fetch-site: RedactedHeader
      sec-fetch-mode: RedactedHeader
      sec-fetch-user: RedactedHeader
      sec-fetch-dest: RedactedHeader
      RequestEnrichment: Stuff
      Protocol: HTTP/2
      Method: GET
      Scheme: https
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      Content-Type: RedactedHeader
      MyResponseHeader: RedactedHeader
      ResponseEnrichment: Stuff
      StatusCode: 200
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[4]
      ResponseBody: Hello World!
```

```
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[8]
      Duration: 2.2778ms
```

Logging configuration order of precedence

The following list shows the order of precedence for logging configuration:

1. Global configuration from HttpLoggingOptions, set by calling AddHttpLogging.
2. Endpoint-specific configuration from the [HttpLogging] attribute or the WithHttpLogging extension method overrides global configuration.
3. IHttpLoggingInterceptor is called with the results and can further modify the configuration per request.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

W3CLogger in ASP.NET Core

Article • 07/26/2024

W3CLogger is a middleware that writes log files in the W3C standard format. The logs contain information about HTTP requests and HTTP responses. W3CLogger provides logs of:

- HTTP request information
- Common properties
- Headers
- HTTP response information
- Metadata about the request/response pair (date/time started, time taken)

W3CLogger is valuable in several scenarios to:

- Record information about incoming requests and responses.
- Filter which parts of the request and response are logged.
- Filter which headers to log.

W3CLogger ***can reduce the performance of an app***. Consider the performance impact when selecting fields to log - the performance reduction will increase as you log more properties. Test the performance impact of the selected logging properties.

> ⚠ **Warning**
>
> W3CLogger can potentially log personally identifiable information (PII). Consider the risk and avoid logging sensitive information. By default, fields that could contain PII aren't logged.

Enable W3CLogger

Enable W3CLogger with UseW3CLogging, which adds the W3CLogger middleware:

C#

```
var app = builder.Build();

app.UseW3CLogging();

app.UseRouting();
```

By default, W3CLogger logs common properties such as path, status-code, date, time, and protocol. All information about a single request/response pair is written to the same line.

```
#Version: 1.0
#Start-Date: 2021-09-29 22:18:28
#Fields: date time c-ip s-computername s-ip s-port cs-method cs-uri-stem cs-uri-query sc-status time-taken cs-version cs-host cs(User-Agent) cs(Referer)
2021-09-29 22:18:28 ::1 DESKTOP-LH3TLTA ::1 5000 GET / - 200 59.9171 HTTP/1.1 localhost:5000 Mozilla/5.0+(Windows+NT+10.0;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/93.0.4577.82+Safari/537.36 -
2021-09-29 22:18:28 ::1 DESKTOP-LH3TLTA ::1 5000 GET / - 200 0.1802 HTTP/1.1 localhost:5000 Mozilla/5.0+(Windows+NT+10.0;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/93.0.4577.82+Safari/537.36 -
2021-09-29 22:18:30 ::1 DESKTOP-LH3TLTA ::1 5000 GET / - 200 0.0966 HTTP/1.1 localhost:5000 Mozilla/5.0+(Windows+NT+10.0;+WOW64)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Chrome/93.0.4577.82+Safari/537.36 -
```

W3CLogger options

To configure the W3CLogger middleware, call AddW3CLogging in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddW3CLogging(logging =>
{
    // Log all W3C fields
    logging.LoggingFields = W3CLoggingFields.All;

    logging.AdditionalRequestHeaders.Add("x-forwarded-for");
    logging.AdditionalRequestHeaders.Add("x-client-ssl-protocol");
    logging.FileSizeLimit = 5 * 1024 * 1024;
    logging.RetainedFileCountLimit = 2;
    logging.FileName = "MyLogFile";
    logging.LogDirectory = @"C:\logs";
    logging.FlushInterval = TimeSpan.FromSeconds(2);
});
```

`LoggingFields`

W3CLoggerOptions.LoggingFields is a bit flag enumeration that configures specific parts of the request and response to log, and other information about the connection.

`LoggingFields` defaults to include all possible fields except `UserName` and `Cookie`. For a complete list of available fields, see W3CLoggingFields.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Health checks in ASP.NET Core

Article • 01/11/2024

By Glenn Condron and Juergen Gutsch

ASP.NET Core offers Health Checks Middleware and libraries for reporting the health of app infrastructure components.

Health checks are exposed by an app as HTTP endpoints. Health check endpoints can be configured for various real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, a container orchestrator may respond to a failing health check by halting a rolling deployment or restarting a container. A load balancer might react to an unhealthy app by routing traffic away from the failing instance to a healthy instance.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

Health checks are typically used with an external monitoring service or container orchestrator to check the status of an app. Before adding health checks to an app, decide on which monitoring system to use. The monitoring system dictates what types of health checks to create and how to configure their endpoints.

Basic health probe

For many apps, a basic health probe configuration that reports the app's availability to process requests (*liveness*) is sufficient to discover the status of the app.

The basic configuration registers health check services and calls the Health Checks Middleware to respond at a URL endpoint with a health response. By default, no specific health checks are registered to test any particular dependency or subsystem. The app is considered healthy if it can respond at the health endpoint URL. The default response writer writes HealthStatus as a plaintext response to the client. The `HealthStatus` is HealthStatus.Healthy, HealthStatus.Degraded, or HealthStatus.Unhealthy.

Register health check services with AddHealthChecks in `Program.cs`. Create a health check endpoint by calling MapHealthChecks.

The following example creates a health check endpoint at `/healthz`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHealthChecks();

var app = builder.Build();

app.MapHealthChecks("/healthz");

app.Run();
```

Docker `HEALTHCHECK`

Docker offers a built-in `HEALTHCHECK` directive that can be used to check the status of an app that uses the basic health check configuration:

Dockerfile

```
HEALTHCHECK CMD curl --fail http://localhost:5000/healthz || exit
```

The preceding example uses `curl` to make an HTTP request to the health check endpoint at `/healthz`. `curl` isn't included in the .NET Linux container images, but it can be added by installing the required package in the Dockerfile. Containers that use images based on Alpine Linux can use the included `wget` in place of `curl`.

Create health checks

Health checks are created by implementing the IHealthCheck interface. The CheckHealthAsync method returns a HealthCheckResult that indicates the health as `Healthy`, `Degraded`, or `Unhealthy`. The result is written as a plaintext response with a configurable status code. Configuration is described in the Health check options section. HealthCheckResult can also return optional key-value pairs.

The following example demonstrates the layout of a health check:

C#

```
public class SampleHealthCheck : IHealthCheck
{
    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken =
default)
    {
```

```
        var isHealthy = true;

        // ...

        if (isHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("A healthy result."));
        }

        return Task.FromResult(
            new HealthCheckResult(
                context.Registration.FailureStatus, "An unhealthy
result."));
    }
}
```

The health check's logic is placed in the CheckHealthAsync method. The preceding example sets a dummy variable, `isHealthy`, to `true`. If the value of `isHealthy` is set to `false`, the HealthCheckRegistration.FailureStatus status is returned.

If CheckHealthAsync throws an exception during the check, a new HealthReportEntry is returned with its HealthReportEntry.Status set to the FailureStatus. This status is defined by AddCheck (see the Register health check services section) and includes the inner exception that caused the check failure. The Description is set to the exception's message.

Register health check services

To register a health check service, call AddCheck in `Program.cs`:

C#

```
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>("Sample");
```

The AddCheck overload shown in the following example sets the failure status (HealthStatus) to report when the health check reports a failure. If the failure status is set to `null` (default), HealthStatus.Unhealthy is reported. This overload is a useful scenario for library authors, where the failure status indicated by the library is enforced by the app when a health check failure occurs if the health check implementation honors the setting.

Tags can be used to filter health checks. Tags are described in the Filter health checks section.

C#

```
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheck>(
        "Sample",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "sample" });
```

AddCheck can also execute a lambda function. In the following example, the health check always returns a healthy result:

C#

```
builder.Services.AddHealthChecks()
    .AddCheck("Sample", () => HealthCheckResult.Healthy("A healthy
result."));
```

Call AddTypeActivatedCheck to pass arguments to a health check implementation. In the following example, a type-activated health check accepts an integer and a string in its constructor:

C#

```
public class SampleHealthCheckWithArgs : IHealthCheck
{
    private readonly int _arg1;
    private readonly string _arg2;

    public SampleHealthCheckWithArgs(int arg1, string arg2)
        => (_arg1, _arg2) = (arg1, arg2);

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken =
default)
    {
        // ...

        return Task.FromResult(HealthCheckResult.Healthy("A healthy
result."));
    }
}
```

To register the preceding health check, call `AddTypeActivatedCheck` with the integer and string passed as arguments:

C#

```
builder.Services.AddHealthChecks()
    .AddTypeActivatedCheck<SampleHealthCheckWithArgs>(
        "Sample",
        failureStatus: HealthStatus.Degraded,
        tags: new[] { "sample" },
        args: new object[] { 1, "Arg" });
```

Use Health Checks Routing

In `Program.cs`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

C#

```
app.MapHealthChecks("/healthz");
```

Require host

Call `RequireHost` to specify one or more permitted hosts for the health check endpoint. Hosts should be Unicode rather than punycode and may include a port. If a collection isn't supplied, any host is accepted:

C#

```
app.MapHealthChecks("/healthz")
    .RequireHost("www.contoso.com:5001");
```

To restrict the health check endpoint to respond only on a specific port, specify a port in the call to `RequireHost`. This approach is typically used in a container environment to expose a port for monitoring services:

C#

```
app.MapHealthChecks("/healthz")
    .RequireHost("*:5001");
```

> ⚠ **Warning**
>
> API that relies on the Host header, such as HttpRequest.Host and RequireHost, are subject to potential spoofing by clients.

To prevent host and port spoofing, use one of the following approaches:

- Use HttpContext.Connection (ConnectionInfo.LocalPort) where the ports are checked.
- Employ Host filtering.

To prevent unauthorized clients from spoofing the port, call RequireAuthorization:

```
C#

app.MapHealthChecks("/healthz")
    .RequireHost("*:5001")
    .RequireAuthorization();
```

For more information, see Host matching in routes with RequireHost.

Require authorization

Call RequireAuthorization to run Authorization Middleware on the health check request endpoint. A `RequireAuthorization` overload accepts one or more authorization policies. If a policy isn't provided, the default authorization policy is used:

```
C#

app.MapHealthChecks("/healthz")
    .RequireAuthorization();
```

Enable Cross-Origin Requests (CORS)

Although running health checks manually from a browser isn't a common scenario, CORS Middleware can be enabled by calling RequireCors on the health checks endpoints. The RequireCors overload accepts a CORS policy builder delegate (`CorsPolicyBuilder`) or a policy name. For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core.

Health check options

HealthCheckOptions provide an opportunity to customize health check behavior:

- Filter health checks
- Customize the HTTP status code

- Suppress cache headers
- Customize output

Filter health checks

By default, the Health Checks Middleware runs all registered health checks. To run a subset of health checks, provide a function that returns a boolean to the Predicate option.

The following example filters the health checks so that only those tagged with `sample` run:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("sample")
});
```

Customize the HTTP status code

Use ResultStatusCodes to customize the mapping of health status to HTTP status codes. The following StatusCodes assignments are the default values used by the middleware. Change the status code values to meet your requirements:

C#

```
app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    ResultStatusCodes =
    {
        [HealthStatus.Healthy] = StatusCodes.Status200OK,
        [HealthStatus.Degraded] = StatusCodes.Status200OK,
        [HealthStatus.Unhealthy] = StatusCodes.Status503ServiceUnavailable
    }
});
```

Suppress cache headers

AllowCachingResponses controls whether the Health Checks Middleware adds HTTP headers to a probe response to prevent response caching. If the value is `false` (default), the middleware sets or overrides the `Cache-Control`, `Expires`, and `Pragma` headers to

prevent response caching. If the value is `true`, the middleware doesn't modify the cache headers of the response:

```
C#

app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    AllowCachingResponses = true
});
```

Customize output

To customize the output of a health checks report, set the HealthCheckOptions.ResponseWriter property to a delegate that writes the response:

```
C#

app.MapHealthChecks("/healthz", new HealthCheckOptions
{
    ResponseWriter = WriteResponse
});
```

The default delegate writes a minimal plaintext response with the string value of HealthReport.Status. The following custom delegate outputs a custom JSON response using System.Text.Json:

```
C#

private static Task WriteResponse(HttpContext context, HealthReport
healthReport)
{
    context.Response.ContentType = "application/json; charset=utf-8";

    var options = new JsonWriterOptions { Indented = true };

    using var memoryStream = new MemoryStream();
    using (var jsonWriter = new Utf8JsonWriter(memoryStream, options))
    {
        jsonWriter.WriteStartObject();
        jsonWriter.WriteString("status", healthReport.Status.ToString());
        jsonWriter.WriteStartObject("results");

        foreach (var healthReportEntry in healthReport.Entries)
        {
            jsonWriter.WriteStartObject(healthReportEntry.Key);
            jsonWriter.WriteString("status",
                healthReportEntry.Value.Status.ToString());
            jsonWriter.WriteString("description",
```

```
                healthReportEntry.Value.Description);
            jsonWriter.WriteStartObject("data");

            foreach (var item in healthReportEntry.Value.Data)
            {
                jsonWriter.WritePropertyName(item.Key);

                JsonSerializer.Serialize(jsonWriter, item.Value,
                    item.Value?.GetType() ?? typeof(object));
            }

            jsonWriter.WriteEndObject();
            jsonWriter.WriteEndObject();
        }

        jsonWriter.WriteEndObject();
        jsonWriter.WriteEndObject();
    }

    return context.Response.WriteAsync(
        Encoding.UTF8.GetString(memoryStream.ToArray()));
}
```

The health checks API doesn't provide built-in support for complex JSON return formats because the format is specific to your choice of monitoring system. Customize the response in the preceding examples as needed. For more information on JSON serialization with `System.Text.Json`, see How to serialize and deserialize JSON in .NET.

Database probe

A health check can specify a database query to run as a boolean test to indicate if the database is responding normally.

AspNetCore.Diagnostics.HealthChecks, a health check library for ASP.NET Core apps, includes a health check that runs against a SQL Server database. `AspNetCore.Diagnostics.HealthChecks` executes a `SELECT 1` query against the database to confirm the connection to the database is healthy.

> **⚠ Warning**
>
> When checking a database connection with a query, choose a query that returns quickly. The query approach runs the risk of overloading the database and degrading its performance. In most cases, running a test query isn't necessary. Merely making a successful connection to the database is sufficient. If you find it necessary to run a query, choose a simple SELECT query, such as `SELECT 1`.

To use this SQL Server health check, include a package reference to the AspNetCore.HealthChecks.SqlServer NuGet package. The following example registers the SQL Server health check:

```
C#

var conStr = builder.Configuration.GetConnectionString("DefaultConnection");
if (string.IsNullOrEmpty(conStr))
{
    throw new InvalidOperationException(
                        "Could not find a connection string named
'DefaultConnection'.");
}
builder.Services.AddHealthChecks()
    .AddSqlServer(conStr);
```

> **Note**
>
> AspNetCore.Diagnostics.HealthChecks isn't maintained or supported by Microsoft.

Entity Framework Core DbContext probe

The `DbContext` check confirms that the app can communicate with the database configured for an EF Core `DbContext`. The `DbContext` check is supported in apps that:

- Use Entity Framework (EF) Core.
- Include a package reference to the Microsoft.Extensions.Diagnostics.HealthChecks.EntityFrameworkCore NuGet package.

AddDbContextCheck registers a health check for a DbContext. The `DbContext` is supplied to the method as the `TContext`. An overload is available to configure the failure status, tags, and a custom test query.

By default:

- The `DbContextHealthCheck` calls EF Core's `CanConnectAsync` method. You can customize what operation is run when checking health using `AddDbContextCheck` method overloads.
- The name of the health check is the name of the `TContext` type.

The following example registers a `DbContext` and an associated `DbContextHealthCheck`:

C#

```
builder.Services.AddDbContext<SampleDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddHealthChecks()
    .AddDbContextCheck<SampleDbContext>();
```

Separate readiness and liveness probes

In some hosting scenarios, a pair of health checks is used to distinguish two app states:

- *Readiness* indicates if the app is running normally but isn't ready to receive requests.
- *Liveness* indicates if an app has crashed and must be restarted.

Consider the following example: An app must download a large configuration file before it's ready to process requests. We don't want the app to be restarted if the initial download fails because the app can retry downloading the file several times. We use a *liveness probe* to describe the liveness of the process, no other checks are run. We also want to prevent requests from being sent to the app before the configuration file download has succeeded. We use a *readiness probe* to indicate a "not ready" state until the download succeeds and the app is ready to receive requests.

The following background task simulates a startup process that takes roughly 15 seconds. Once it completes, the task sets the `StartupHealthCheck.StartupCompleted` property to true:

C#

```
public class StartupBackgroundService : BackgroundService
{
    private readonly StartupHealthCheck _healthCheck;

    public StartupBackgroundService(StartupHealthCheck healthCheck)
        => _healthCheck = healthCheck;

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        // Simulate the effect of a long-running task.
        await Task.Delay(TimeSpan.FromSeconds(15), stoppingToken);

        _healthCheck.StartupCompleted = true;
```

```
    }
}
```

The `StartupHealthCheck` reports the completion of the long-running startup task and exposes the `StartupCompleted` property that gets set by the background service:

C#

```
public class StartupHealthCheck : IHealthCheck
{
    private volatile bool _isReady;

    public bool StartupCompleted
    {
        get => _isReady;
        set => _isReady = value;
    }

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken =
default)
    {
        if (StartupCompleted)
        {
            return Task.FromResult(HealthCheckResult.Healthy("The startup
task has completed."));
        }

        return Task.FromResult(HealthCheckResult.Unhealthy("That startup
task is still running."));
    }
}
```

The health check is registered with AddCheck in `Program.cs` along with the hosted service. Because the hosted service must set the property on the health check, the health check is also registered in the service container as a singleton:

C#

```
builder.Services.AddHostedService<StartupBackgroundService>();
builder.Services.AddSingleton<StartupHealthCheck>();

builder.Services.AddHealthChecks()
    .AddCheck<StartupHealthCheck>(
        "Startup",
        tags: new[] { "ready" });
```

To create two different health check endpoints, call `MapHealthChecks` twice:

C#

```
app.MapHealthChecks("/healthz/ready", new HealthCheckOptions
{
    Predicate = healthCheck => healthCheck.Tags.Contains("ready")
});

app.MapHealthChecks("/healthz/live", new HealthCheckOptions
{
    Predicate = _ => false
});
```

The preceding example creates the following health check endpoints:

- `/healthz/ready` for the readiness check. The readiness check filters health checks to those tagged with `ready`.
- `/healthz/live` for the liveness check. The liveness check filters out all health checks by returning `false` in the HealthCheckOptions.Predicate delegate. For more information on filtering health checks, see Filter health checks in this article.

Before the startup task completes, the `/healthz/ready` endpoint reports an `Unhealthy` status. Once the startup task completes, this endpoint reports a `Healthy` status. The `/healthz/live` endpoint excludes all checks and reports a `Healthy` status for all calls.

Kubernetes example

Using separate readiness and liveness checks is useful in an environment such as Kubernetes. In Kubernetes, an app might be required to run time-consuming startup work before accepting requests, such as a test of the underlying database availability. Using separate checks allows the orchestrator to distinguish whether the app is functioning but not yet ready or if the app has failed to start. For more information on readiness and liveness probes in Kubernetes, see Configure Liveness and Readiness Probes in the Kubernetes documentation.

The following example demonstrates a Kubernetes readiness probe configuration:

YAML

```
spec:
  template:
  spec:
    readinessProbe:
      # an http probe
      httpGet:
        path: /healthz/ready
        port: 80
```

```
    # length of time to wait for a pod to initialize
    # after pod startup, before applying health checking
    initialDelaySeconds: 30
    timeoutSeconds: 1
  ports:
    - containerPort: 80
```

Distribute a health check library

To distribute a health check as a library:

1. Write a health check that implements the IHealthCheck interface as a standalone class. The class can rely on dependency injection (DI), type activation, and named options to access configuration data.

2. Write an extension method with parameters that the consuming app calls in its `Program.cs` method. Consider the following example health check, which accepts `arg1` and `arg2` as constructor parameters:

   C#

   ```
   public SampleHealthCheckWithArgs(int arg1, string arg2)
       => (_arg1, _arg2) = (arg1, arg2);
   ```

   The preceding signature indicates that the health check requires custom data to process the health check probe logic. The data is provided to the delegate used to create the health check instance when the health check is registered with an extension method. In the following example, the caller specifies:

   - `arg1`: An integer data point for the health check.
   - `arg2`: A string argument for the health check.
   - `name`: An optional health check name. If `null`, a default value is used.
   - `failureStatus`: An optional HealthStatus, which is reported for a failure status. If `null`, HealthStatus.Unhealthy is used.
   - `tags`: An optional `IEnumerable<string>` collection of tags.

   C#

   ```
   public static class SampleHealthCheckBuilderExtensions
   {
       private const string DefaultName = "Sample";

       public static IHealthChecksBuilder AddSampleHealthCheck(
           this IHealthChecksBuilder healthChecksBuilder,
   ```

```
            int arg1,
            string arg2,
            string? name = null,
            HealthStatus? failureStatus = null,
            IEnumerable<string>? tags = default)
        {
            return healthChecksBuilder.Add(
                new HealthCheckRegistration(
                    name ?? DefaultName,
                    _ => new SampleHealthCheckWithArgs(arg1, arg2),
                    failureStatus,
                    tags));
        }
    }
```

Health Check Publisher

When an IHealthCheckPublisher is added to the service container, the health check system periodically executes your health checks and calls PublishAsync with the result. This process is useful in a push-based health monitoring system scenario that expects each process to call the monitoring system periodically to determine health.

HealthCheckPublisherOptions allow you to set the:

- Delay: The initial delay applied after the app starts before executing IHealthCheckPublisher instances. The delay is applied once at startup and doesn't apply to later iterations. The default value is five seconds.
- Period: The period of IHealthCheckPublisher execution. The default value is 30 seconds.
- Predicate: If Predicate is `null` (default), the health check publisher service runs all registered health checks. To run a subset of health checks, provide a function that filters the set of checks. The predicate is evaluated each period.
- Timeout: The timeout for executing the health checks for all IHealthCheckPublisher instances. Use InfiniteTimeSpan to execute without a timeout. The default value is 30 seconds.

The following example demonstrates the layout of a health publisher:

C#

```
public class SampleHealthCheckPublisher : IHealthCheckPublisher
{
    public Task PublishAsync(HealthReport report, CancellationToken
cancellationToken)
    {
        if (report.Status == HealthStatus.Healthy)
```

```
        {
            // ...
        }
        else
        {
            // ...
        }

        return Task.CompletedTask;
    }
}
```

The HealthCheckPublisherOptions class provides properties for configuring the behavior of the health check publisher.

The following example registers a health check publisher as a singleton and configures HealthCheckPublisherOptions:

C#

```
builder.Services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.FromSeconds(2);
    options.Predicate = healthCheck => healthCheck.Tags.Contains("sample");
});

builder.Services.AddSingleton<IHealthCheckPublisher,
SampleHealthCheckPublisher>();
```

AspNetCore.Diagnostics.HealthChecks :

- Includes publishers for several systems, including Application Insights.
- Is ***not*** maintained or supported by Microsoft.

Individual Healthchecks

Delay and Period can be set on each HealthCheckRegistration individually. This is useful when you want to run some health checks at a different rate than the period set in HealthCheckPublisherOptions.

The following code sets the `Delay` and `Period` for the `SampleHealthCheck1`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHealthChecks()
   .Add(new HealthCheckRegistration(
```

```
        name: "SampleHealthCheck1",
        instance: new SampleHealthCheck(),
        failureStatus: null,
        tags: null,
        timeout: default)
    {
        Delay = TimeSpan.FromSeconds(40),
        Period = TimeSpan.FromSeconds(30)
    });

var app = builder.Build();

app.MapHealthChecks("/healthz");

app.Run();
```

Dependency Injection and Health Checks

It's possible to use dependency injection to consume an instance of a specific `Type` inside a Health Check class. Dependency injection can be useful to inject options or a global configuration to a Health Check. Using dependency injection is ***not*** a common scenario to configure Health Checks. Usually, each Health Check is quite specific to the actual test and is configured using `IHealthChecksBuilder` extension methods.

The following example shows a sample Health Check that retrieves a configuration object via dependency injection:

C#

```
public class SampleHealthCheckWithDI : IHealthCheck
{
    private readonly SampleHealthCheckWithDiConfig _config;

    public SampleHealthCheckWithDI(SampleHealthCheckWithDiConfig config)
        => _config = config;

    public Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context, CancellationToken cancellationToken =
default)
    {
        var isHealthy = true;

        // use _config ...

        if (isHealthy)
        {
            return Task.FromResult(
                HealthCheckResult.Healthy("A healthy result."));
        }
```

```
        return Task.FromResult(
            new HealthCheckResult(
                context.Registration.FailureStatus, "An unhealthy
result."));
    }
}
```

The `SampleHealthCheckWithDiConfig` and the Health check needs to be added to the service container :

C#

```
builder.Services.AddSingleton<SampleHealthCheckWithDiConfig>(new
SampleHealthCheckWithDiConfig
{
    BaseUriToCheck = new Uri("https://sample.contoso.com/api/")
});
builder.Services.AddHealthChecks()
    .AddCheck<SampleHealthCheckWithDI>(
        "With Dependency Injection",
        tags: new[] { "inject" });
```

UseHealthChecks vs. MapHealthChecks

There are two ways to make health checks accessible to callers:

- UseHealthChecks registers middleware for handling health checks requests in the middleware pipeline.
- MapHealthChecks registers a health checks endpoint. The endpoint is matched and executed along with other endpoints in the app.

The advantage of using `MapHealthChecks` over `UseHealthChecks` is the ability to use endpoint aware middleware, such as authorization, and to have greater fine-grained control over the matching policy. The primary advantage of using `UseHealthChecks` over `MapHealthChecks` is controlling exactly where health checks runs in the middleware pipeline.

UseHealthChecks:

- Terminates the pipeline when a request matches the health check endpoint. Short-circuiting is often desirable because it avoids unnecessary work, such as logging and other middleware.
- Is primarily used for configuring the health check middleware in the pipeline.

- Can match any path on a port with a `null` or empty `PathString`. Allows performing a health check on any request made to the specified port.
- Source code

MapHealthChecks allows:

- Terminating the pipeline when a request matches the health check endpoint, by calling ShortCircuit. For example, `app.MapHealthChecks("/healthz").ShortCircuit();`. For more information, see Short-circuit middleware after routing.
- Mapping specific routes or endpoints for health checks.
- Customization of the URL or path where the health check endpoint is accessible.
- Mapping multiple health check endpoints with different routes or configurations. Multiple endpoint support:
  - Enables separate endpoints for different types of health checks or components.
  - Is used to differentiate between different aspects of the app's health or apply specific configurations to subsets of health checks.
- Source code

Additional resources

- View or download sample code (how to download)

> **Note**
>
> This article was partially created with the help of artificial intelligence. Before publishing, an author reviewed and revised the content as needed. See Our principles for using AI-generated content in Microsoft Learn.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core metrics

Article • 11/14/2024

Metrics are numerical measurements reported over time. They're typically used to monitor the health of an app and generate alerts. For example, a web service might track how many:

- Requests it received per second.
- Milliseconds it took to respond.
- Responses sent an error.

These metrics can be reported to a monitoring system at regular intervals. Dashboards can be setup to view metrics and alerts created to notify people of problems. If the web service is intended to respond to requests within 400 ms and starts responding in 600 ms, the monitoring system can notify the operations staff that the app response is slower than normal.

> **Tip**
>
> See **ASP.NET Core metrics** for a comprehensive list of all instruments together with their attributes.

Using metrics

There are two parts to using metrics in a .NET app:

- **Instrumentation:** Code in .NET libraries takes measurements and associates these measurements with a metric name. .NET and ASP.NET Core include many built-in metrics.
- **Collection:** A .NET app configures named metrics to be transmitted from the app for external storage and analysis. Some tools may perform configuration outside the app using configuration files or a UI tool.

Instrumented code can record numeric measurements, but the measurements need to be aggregated, transmitted, and stored to create useful metrics for monitoring. The process of aggregating, transmitting, and storing data is called collection. This tutorial shows several examples of collecting metrics:

- Populating metrics in Grafana with OpenTelemetry and Prometheus.
- Viewing metrics in real time with dotnet-counters

Measurements can also be associated with key-value pairs called tags that allow data to be categorized for analysis. For more information, see Multi-dimensional metrics.

Create the starter app

Create a new ASP.NET Core app with the following command:

.NET CLI

```
dotnet new web -o WebMetric
cd WebMetric
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore --prerelease
dotnet add package OpenTelemetry.Extensions.Hosting
```

Replace the contents of `Program.cs` with the following code:

C#

```
using OpenTelemetry.Metrics;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddOpenTelemetry()
    .WithMetrics(builder =>
    {
        builder.AddPrometheusExporter();

        builder.AddMeter("Microsoft.AspNetCore.Hosting",
                         "Microsoft.AspNetCore.Server.Kestrel");
        builder.AddView("http.server.request.duration",
            new ExplicitBucketHistogramConfiguration
            {
                Boundaries = new double[] { 0, 0.005, 0.01, 0.025, 0.05,
                       0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10 }
            });
    });
var app = builder.Build();

app.MapPrometheusScrapingEndpoint();

app.MapGet("/", () => "Hello OpenTelemetry! ticks:"
                    + DateTime.Now.Ticks.ToString()[^3..]);

app.Run();
```

View metrics with dotnet-counters

dotnet-counters is a command-line tool that can view live metrics for .NET Core apps on demand. It doesn't require setup, making it useful for ad-hoc investigations or verifying that metric instrumentation is working. It works with both System.Diagnostics.Metrics based APIs and EventCounters.

If the dotnet-counters tool isn't installed, run the following command:

.NET CLI

```
dotnet tool update -g dotnet-counters
```

While the test app is running, launch dotnet-counters. The following command shows an example of `dotnet-counters` monitoring all metrics from the Microsoft.AspNetCore.Hosting meter.

.NET CLI

```
dotnet-counters monitor -n WebMetric --counters Microsoft.AspNetCore.Hosting
```

Output similar to the following is displayed:

.NET CLI

```
Press p to pause, r to resume, q to quit.
    Status: Running

[Microsoft.AspNetCore.Hosting]
    http-server-current-requests
        host=localhost,method=GET,port=5045,scheme=http                    0
    http-server-request-duration (s)
        host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro
0.001
        host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro
0.001
        host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro
0.001
        host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro           0
        host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro           0
        host=localhost,method=GET,port=5045,protocol=HTTP/1.1,ro           0
```

For more information, see dotnet-counters.

Enrich the ASP.NET Core request metric

ASP.NET Core has many built-in metrics. The `http.server.request.duration` metric:

- Records the duration of HTTP requests on the server.
- Captures request information in tags, such as the matched route and response status code.

The `http.server.request.duration` metric supports tag enrichment using IHttpMetricsTagsFeature. Enrichment is when a library or app adds its own tags to a metric. This is useful if an app wants to add a custom categorization to dashboards or alerts built with metrics.

C#

```
using Microsoft.AspNetCore.Http.Features;

var builder = WebApplication.CreateBuilder();
var app = builder.Build();

app.Use(async (context, next) =>
{
    var tagsFeature = context.Features.Get<IHttpMetricsTagsFeature>();
    if (tagsFeature != null)
    {
        var source = context.Request.Query["utm_medium"].ToString() switch
        {
            "" => "none",
            "social" => "social",
            "email" => "email",
            "organic" => "organic",
            _ => "other"
        };
        tagsFeature.Tags.Add(new KeyValuePair<string, object?>("mkt_medium",
source));
    }

    await next.Invoke();
});

app.MapGet("/", () => "Hello World!");

app.Run();
```

The proceeding example:

- Adds middleware to enrich the ASP.NET Core request metric.
- Gets the IHttpMetricsTagsFeature from the `HttpContext`. The feature is only present on the context if someone is listening to the metric. Verify `IHttpMetricsTagsFeature` is not `null` before using it.
- Adds a custom tag containing the request's marketing source to the http.server.request.duration metric.

- The tag has the name `mkt_medium` and a value based on the utm_medium query string value. The `utm_medium` value is resolved to a known range of values.
- The tag allows requests to be categorized by marketing medium type, which could be useful when analyzing web app traffic.

> ⓘ Note
>
> Follow the multi-dimensional metrics best practices when enriching with custom tags. Tags that are too numerous or have an unbound range create many tag combinations, resulting in high dimensions. Collection tools have limits on supported dimensions for a counter and may filter results to prevent excessive memory use.

Create custom metrics

Metrics are created using APIs in the System.Diagnostics.Metrics namespace. See Create custom metrics for information on creating custom metrics.

Creating metrics in ASP.NET Core apps with `IMeterFactory`

We recommended creating Meter instances in ASP.NET Core apps with IMeterFactory.

ASP.NET Core registers IMeterFactory in dependency injection (DI) by default. The meter factory integrates metrics with DI, making isolating and collecting metrics easy. `IMeterFactory` is especially useful for testing. It allows for multiple tests to run side-by-side and only collecting metrics values that are recorded in a test.

To use `IMeterFactory` in an app, create a type that uses `IMeterFactory` to create the app's custom metrics:

C#

```
public class ContosoMetrics
{
    private readonly Counter<int> _productSoldCounter;

    public ContosoMetrics(IMeterFactory meterFactory)
    {
        var meter = meterFactory.Create("Contoso.Web");
        _productSoldCounter = meter.CreateCounter<int>
("contoso.product.sold");
    }
```

```
    public void ProductSold(string productName, int quantity)
    {
        _productSoldCounter.Add(quantity,
            new KeyValuePair<string, object?>("contoso.product.name",
productName));
    }
}
```

Register the metrics type with DI in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<ContosoMetrics>();
```

Inject the metrics type and record values where needed. Because the metrics type is registered in DI it can be use with MVC controllers, minimal APIs, or any other type that is created by DI:

C#

```
app.MapPost("/complete-sale", (SaleModel model, ContosoMetrics metrics) =>
{
    // ... business logic such as saving the sale to a database ...

    metrics.ProductSold(model.ProductName, model.QuantitySold);
});
```

To monitor the "Contoso.Web" meter, use the following dotnet-counters command.

.NET CLI

```
dotnet-counters monitor -n WebMetric --counters Contoso.Web
```

Output similar to the following is displayed:

.NET CLI

```
Press p to pause, r to resume, q to quit.
    Status: Running

[Contoso.Web]
    contoso.product.sold (Count / 1 sec)
        contoso.product.name=Eggs            12
        contoso.product.name=Milk            0
```

View metrics in Grafana with OpenTelemetry and Prometheus

Overview

OpenTelemetry:

- Is a vendor-neutral open-source project supported by the Cloud Native Computing Foundation.
- Standardizes generating and collecting telemetry for cloud-native software.
- Works with .NET using the .NET metric APIs.
- Is endorsed by Azure Monitor and many APM vendors.

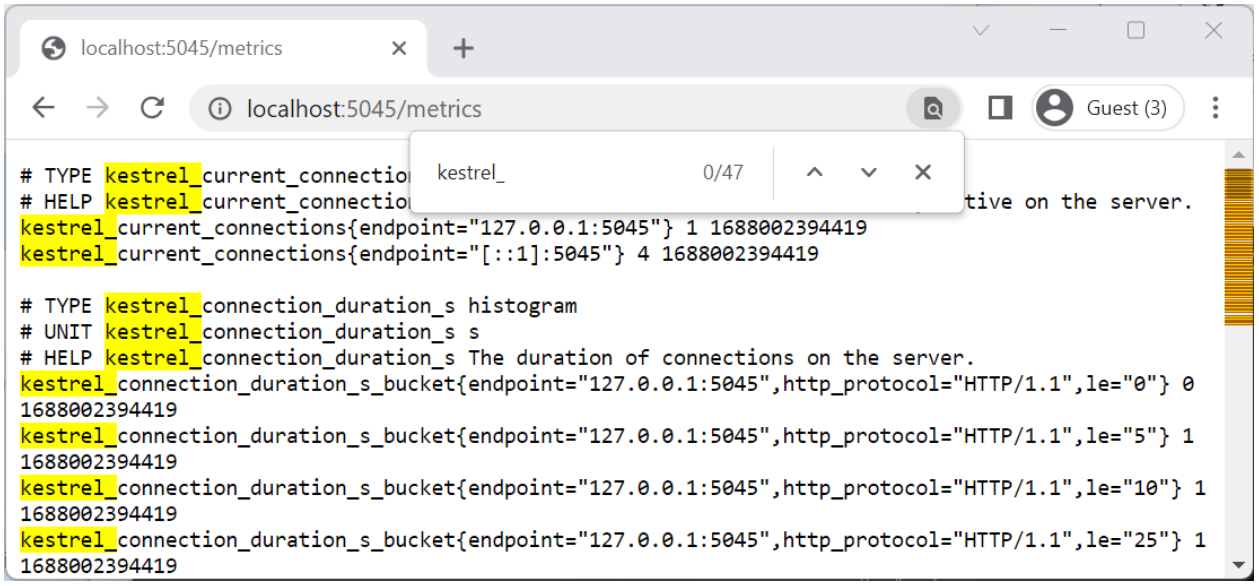
This tutorial shows one of the integrations available for OpenTelemetry metrics using the OSS Prometheus and Grafana projects. The metrics data flow:

1. The ASP.NET Core metric APIs record measurements from the example app.
2. The OpenTelemetry .NET library running in the app aggregates the measurements.
3. The Prometheus exporter library makes the aggregated data available via an HTTP metrics endpoint. 'Exporter' is what OpenTelemetry calls the libraries that transmit telemetry to vendor-specific backends.
4. A Prometheus server:
    - Polls the metrics endpoint
    - Reads the data
    - Stores the data in a database for long-term persistence. Prometheus refers to reading and storing data as *scraping* an endpoint.
    - Can run on a different machine
5. The Grafana server:
    - Queries the data stored in Prometheus and displays it on a web-based monitoring dashboard.
    - Can run on a different machine.

View metrics from sample app

Navigate to the sample app. The browser displays `Hello OpenTelemetry! ticks: <3digits>` where `3digits` are the last 3 digits of the current DateTime.Ticks.

Append `/metrics` to the URL to view the metrics endpoint. The browser displays the metrics being collected:



Set up and configure Prometheus

Follow the Prometheus first steps to set up a Prometheus server and confirm it's working.

Modify the *prometheus.yml* configuration file so that Prometheus scrapes the metrics endpoint that the example app is exposing. Add the following highlighted text in the `scrape_configs` section:

YAML

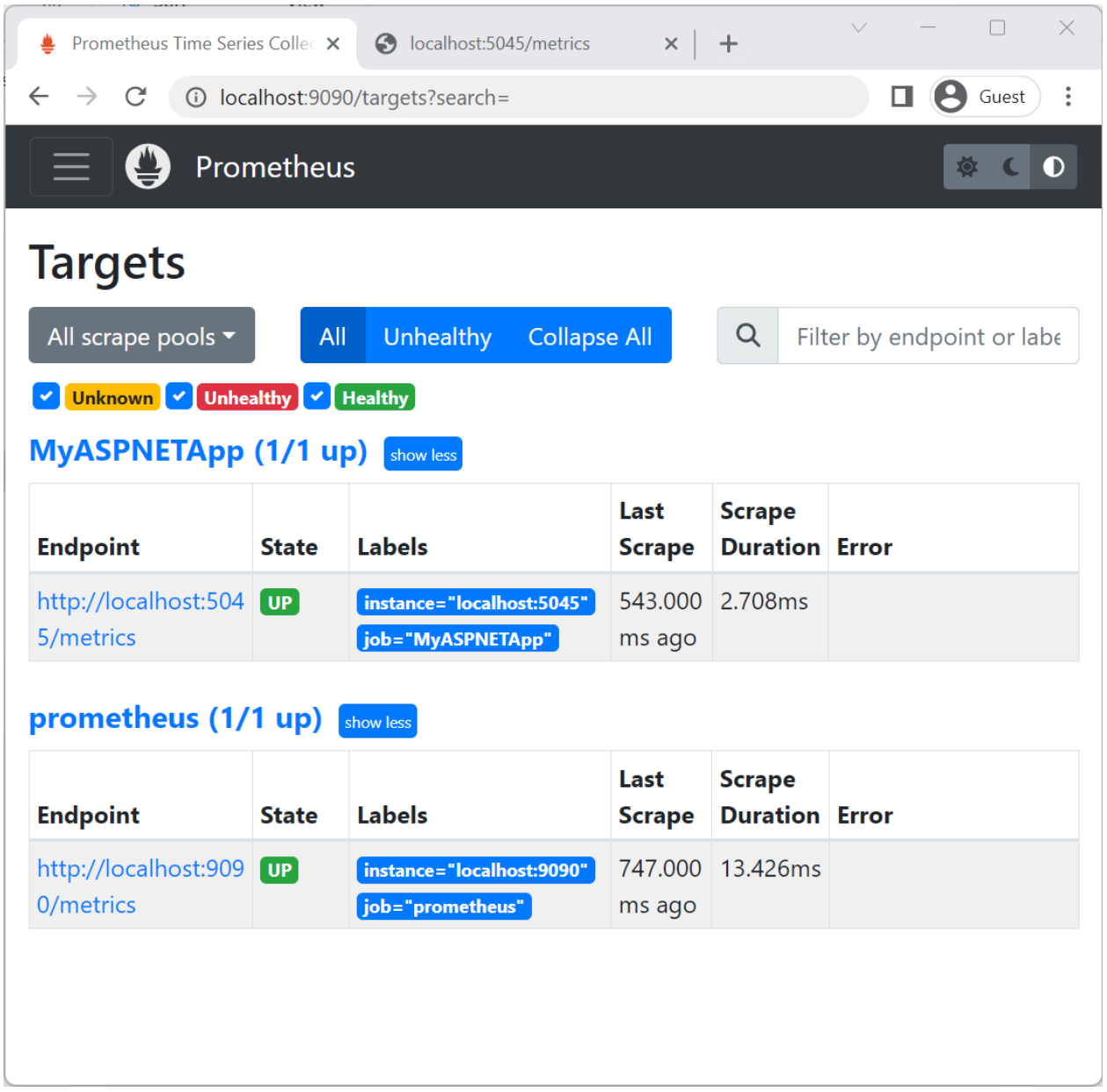
```yaml
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds.
Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is
every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
        - targets:
          # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global
'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"
```

```
# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries
scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]

  - job_name: 'MyASPNETApp'
    scrape_interval: 5s # Poll every 5 seconds for a more responsive demo.
    static_configs:
      - targets: ["localhost:5045"]  ## Enter the HTTP port number of the
demo app.
```

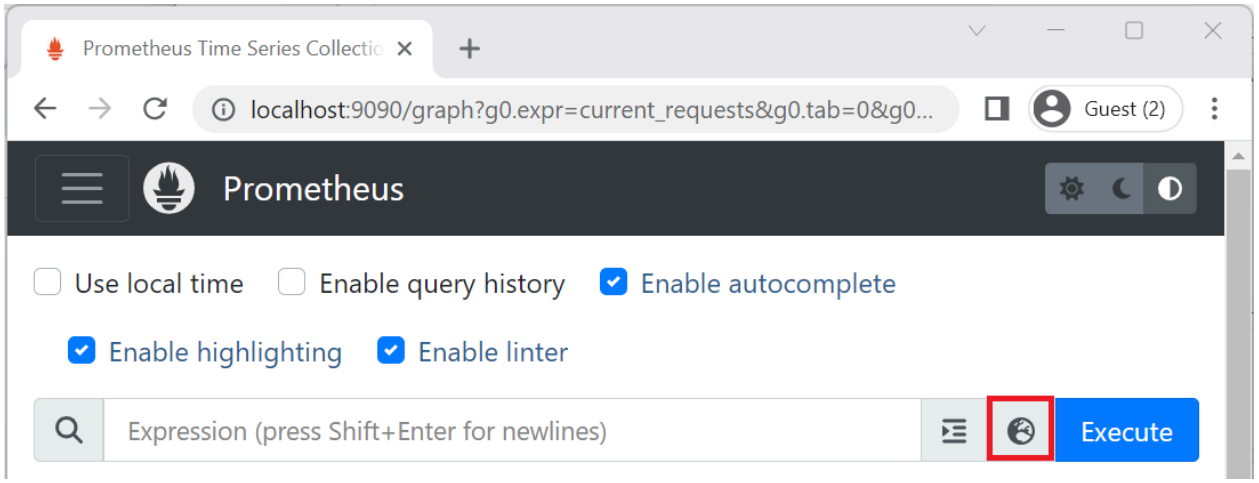
In the preceding highlighted YAML, replace `5045` with the port number that the example app is running on.

Start Prometheus

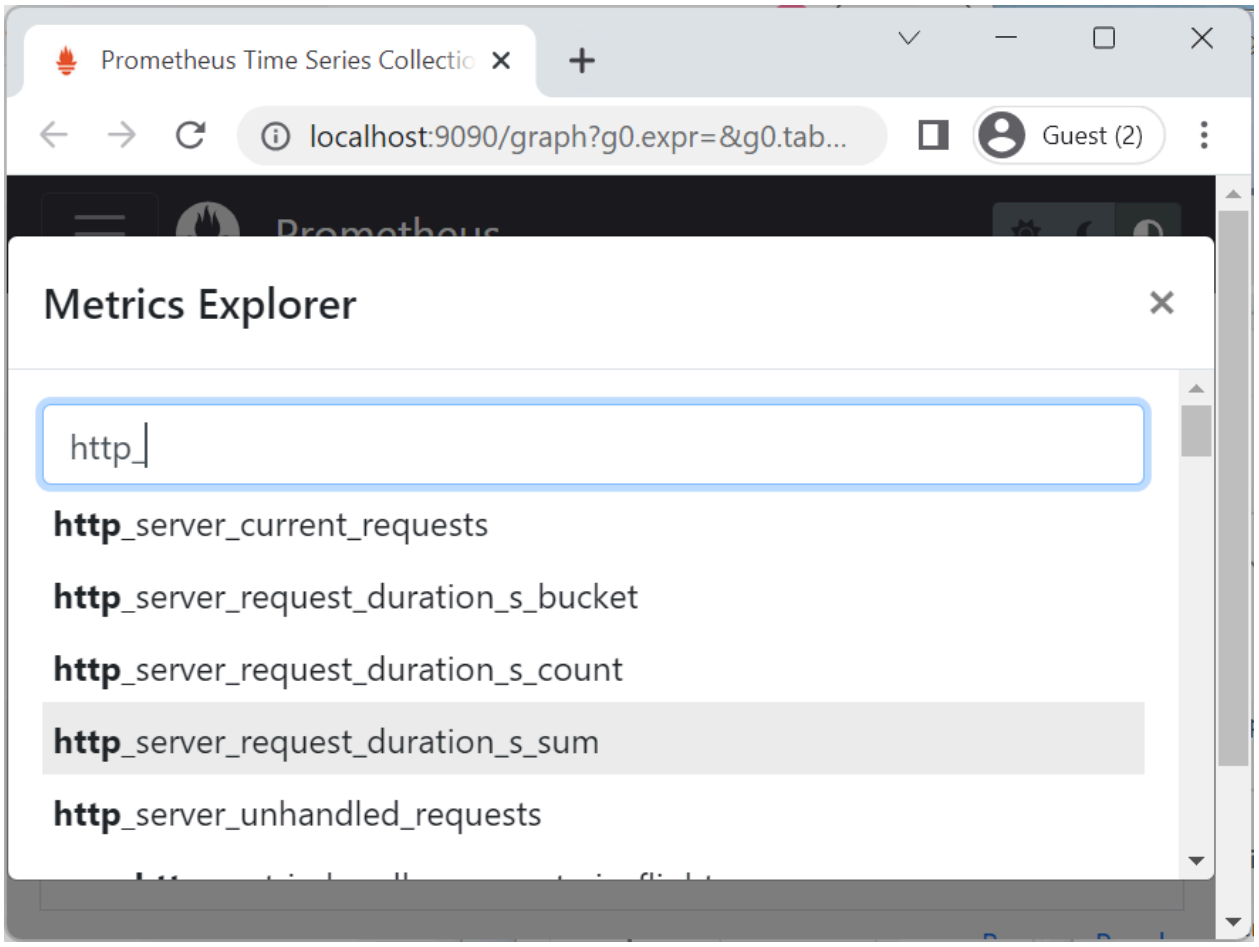
1. Reload the configuration or restart the Prometheus server.
2. Confirm that OpenTelemetryTest is in the UP state in the **Status** > **Targets** page of the Prometheus web portal.



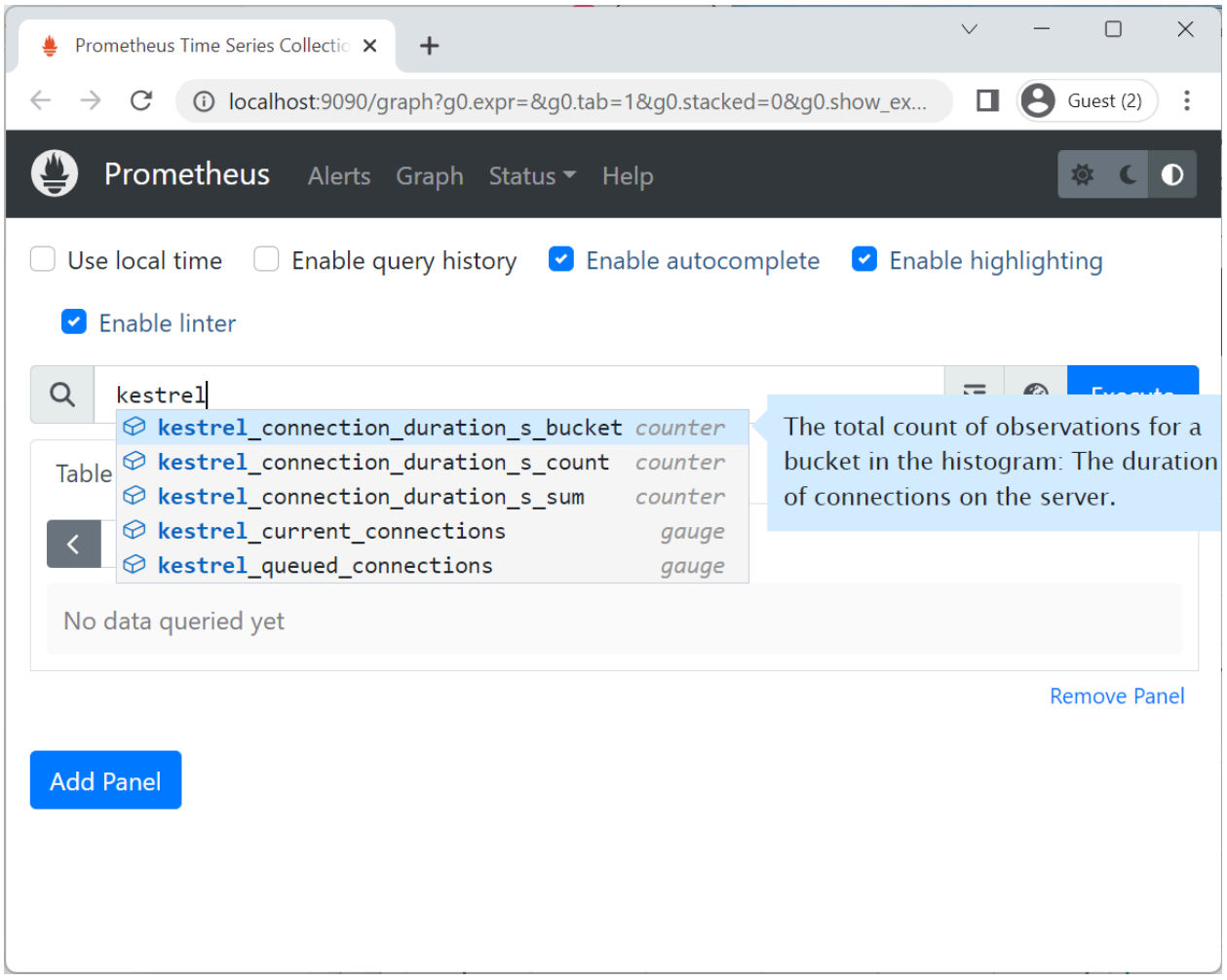
Select the **Open metric explorer** icon to see available metrics:



Enter counter category such as `http_` in the **Expression** input box to see the available metrics:

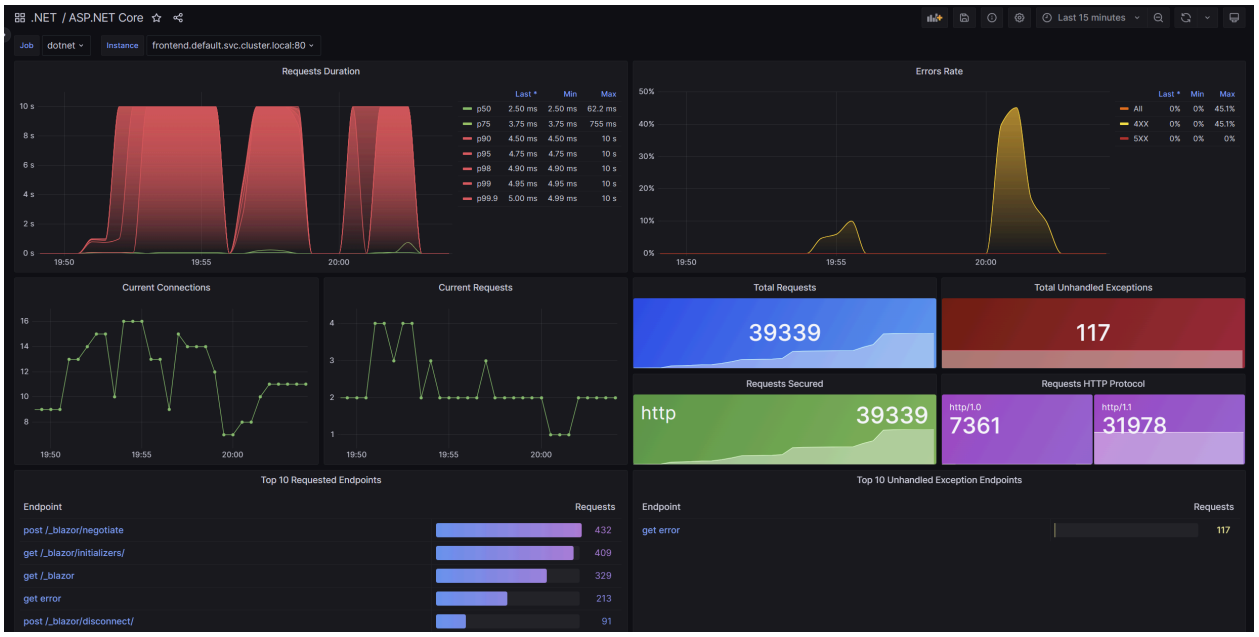


Alternatively, enter counter category such as `kestrel` in the **Expression** input box to see the available metrics:



Show metrics on a Grafana dashboard

- Follow the installation instructions to install Grafana and connect it to a Prometheus data source.
- Follow Creating a Prometheus graph. Alternatively, pre-built dashboards for .NET metrics are available to download at .NET team dashboards @ grafana.com. Downloaded dashboard JSON can be imported into Grafana.



Test metrics in ASP.NET Core apps

It's possible to test metrics in ASP.NET Core apps. One way to do that is collect and assert metrics values in ASP.NET Core integration tests using MetricCollector<T>.

C#

```
public class BasicTests : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly WebApplicationFactory<Program> _factory;
    public BasicTests(WebApplicationFactory<Program> factory) => _factory =
factory;

    [Fact]
    public async Task Get_RequestCounterIncreased()
    {
        // Arrange
        var client = _factory.CreateClient();
        var meterFactory =
_factory.Services.GetRequiredService<IMeterFactory>();
        var collector = new MetricCollector<double>(meterFactory,
            "Microsoft.AspNetCore.Hosting", "http.server.request.duration");

        // Act
        var response = await client.GetAsync("/");

        // Assert
        Assert.Contains("Hello OpenTelemetry!", await
response.Content.ReadAsStringAsync());

        await collector.WaitForMeasurementsAsync(minCount:
1).WaitAsync(TimeSpan.FromSeconds(5));
        Assert.Collection(collector.GetMeasurementSnapshot(),
            measurement =>
```

```
            {
                Assert.Equal("http", measurement.Tags["url.scheme"]);
                Assert.Equal("GET",
measurement.Tags["http.request.method"]);
                Assert.Equal("/", measurement.Tags["http.route"]);
            });
    }
}
```

The proceeding test:

- Bootstraps a web app in memory with WebApplicationFactory<TEntryPoint>. `Program` in the factory's generic argument specifies the web app.
- Collects metrics values with MetricCollector<T>
  - Requires a package reference to `Microsoft.Extensions.Diagnostics.Testing`
  - The `MetricCollector<T>` is created using the web app's IMeterFactory. This allows the collector to only report metrics values recorded by test.
  - Includes the meter name, `Microsoft.AspNetCore.Hosting`, and counter name, `http.server.request.duration` to collect.
- Makes an HTTP request to the web app.
- Asserts the test using results from the metrics collector.

ASP.NET Core meters and counters

See ASP.NET Core metrics for a list of ASP.NET Core meters and counters.

ASP.NET Core metrics

Article • 02/05/2024

This article describes the metrics built-in for ASP.NET Core produced using the System.Diagnostics.Metrics API. For a listing of metrics based on the older EventCounters API, see here.

> **Tip**
>
> For more information about how to collect, report, enrich, and test ASP.NET Core metrics, see Using ASP.NET Core metrics.

`Microsoft.AspNetCore.Hosting`

The `Microsoft.AspNetCore.Hosting` metrics report high-level information about HTTP requests received by ASP.NET Core:

- http.server.request.duration
- http.server.active_requests

Metric: `http.server.request.duration`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
http.server.request.duration	Histogram	`s`	Measures the duration of inbound HTTP requests.

Expand table

Attribute	Type	Description	Examples	Presence
`http.route`	string	The matched route.	`{controller}/{action}/{id?}`	If it's available.
`error.type`	string	Describes a class of error the operation ended with.	`timeout`; `name_resolution_error`; `500`	If request has ended with an error.
`http.request.method`	string	HTTP request method.	`GET`; `POST`; `HEAD`	Always
`http.response.status_code`	int	HTTP response status code.	`200`	If one was sent.
`network.protocol.version`	string	Version of the protocol specified in `network.protocol.name`.	`3.1.1`	Always

Attribute	Type	Description	Examples	Presence
`url.scheme`	string	The URI scheme component identifying the used protocol.	`http`; `https`	Always
`aspnetcore.request.is_unhandled`	Boolean	True when the request wasn't handled by the application pipeline.	`true`	If the request was unhandled.

The time used to handle an inbound HTTP request as measured at the hosting layer of ASP.NET Core. The time measurement starts once the underlying web host has:

- Sufficiently parsed the HTTP request headers on the inbound network stream to identify the new request.
- Initialized the context data structures such as the HttpContext.

The time ends when:

- The ASP.NET Core handler pipeline is finished executing.
- All response data has been sent.
- The context data structures for the request are being disposed.

When using OpenTelemetry, the default buckets for this metric are set to [0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10].

Available starting in: .NET 8.0.

Metric: `http.server.active_requests`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
http.server.active_requests	UpDownCounter	`{request}`	Measures the number of concurrent HTTP requests that are currently in-flight.

Expand table

Attribute	Type	Description	Examples	Presence
`http.request.method`	string	HTTP request method. [1]	`GET`; `POST`; `HEAD`	Always
`url.scheme`	string	The URI scheme component identifying the used protocol.	`http`; `https`	Always

Available starting in: .NET 8.0.

`Microsoft.AspNetCore.Routing`

The `Microsoft.AspNetCore.Routing` metrics report information about routing HTTP requests to ASP.NET Core endpoints:

- aspnetcore.routing.match_attempts

Metric: `aspnetcore.routing.match_attempts`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
aspnetcore.routing.match_attempts	Counter	`{match_attempt}`	Number of requests that were attempted to be matched to an endpoint.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.routing.match_status`	string	Match result	`success`; `failure`	Always
`aspnetcore.routing.is_fallback_route`	boolean	A value that indicates whether the matched route is a fallback route.	`True`	If a route was successfully matched.
`http.route`	string	The matched route	`{controller}/{action}/{id?}`	If a route was successfully matched.

Available starting in: .NET 8.0.

`Microsoft.AspNetCore.Diagnostics`

The `Microsoft.AspNetCore.Diagnostics` metrics report diagnostics information from ASP.NET Core error handling middleware:

- aspnetcore.diagnostics.exceptions

Metric: `aspnetcore.diagnostics.exceptions`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
aspnetcore.diagnostics.exceptions	Counter	`{exception}`	Number of exceptions caught by exception handling middleware.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.diagnostics.exception.result`	string	ASP.NET Core exception middleware handling result	`handled`; `unhandled`	Always
`aspnetcore.diagnostics.handler.type`	string	Full type name of the IExceptionHandler implementation that handled the exception.	`Contoso.MyHandler`	If the exception was handled by this handler.
`exception.type`	string	The full name of exception type.	`System.OperationCanceledException`; `Contoso.MyException`	Always

Available starting in: .NET 8.0.

`Microsoft.AspNetCore.RateLimiting`

The `Microsoft.AspNetCore.RateLimiting` metrics report rate limiting information from ASP.NET Core rate-limiting middleware:

- aspnetcore.rate_limiting.active_request_leases
- aspnetcore.rate_limiting.request_lease.duration
- aspnetcore.rate_limiting.queued_requests
- aspnetcore.rate_limiting.request.time_in_queue
- aspnetcore.rate_limiting.requests

Metric: `aspnetcore.rate_limiting.active_request_leases`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
aspnetcore.rate_limiting.active_request_leases	UpDownCounter	`{request}`	Number of requests that are currently active on the server that hold a rate limiting lease.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.rate_limiting.policy`	string	Rate limiting policy name.	`fixed`; `sliding`; `token`	If the matched endpoint for the request had a rate-limiting policy.

Available starting in: .NET 8.0.

Metric: `aspnetcore.rate_limiting.request_lease.duration`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
aspnetcore.rate_limiting.request_lease.duration	Histogram	`s`	The duration of the rate limiting lease held by requests on the server.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.rate_limiting.policy`	string	Rate limiting policy name.	`fixed`; `sliding`; `token`	If the matched endpoint for the request had a rate-limiting policy.

Available starting in: .NET 8.0.

Metric: `aspnetcore.rate_limiting.queued_requests`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
aspnetcore.rate_limiting.queued_requests	UpDownCounter	`{request}`	Number of requests that are currently queued waiting to acquire a rate limiting lease.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.rate_limiting.policy`	string	Rate limiting policy name.	`fixed`; `sliding`; `token`	If the matched endpoint for the request had a rate-limiting policy.

Available starting in: .NET 8.0.

Metric: `aspnetcore.rate_limiting.request.time_in_queue`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
aspnetcore.rate_limiting.request.time_in_queue	Histogram	`s`	The time a request spent in a queue waiting to acquire a rate limiting lease.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.rate_limiting.policy`	string	Rate limiting policy name.	`fixed`; `sliding`; `token`	If the matched endpoint for the request had a rate-limiting policy.
`aspnetcore.rate_limiting.result`	string	The rate limiting result shows whether lease was acquired or contains a rejection reason.	`acquired`; `request_canceled`	Always

Available starting in: .NET 8.0.

Metric: `aspnetcore.rate_limiting.requests`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
aspnetcore.rate_limiting.requests	Counter	`{request}`	Number of requests that tried to acquire a rate limiting lease.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.rate_limiting.policy`	string	Rate limiting policy name.	`fixed`; `sliding`; `token`	If the matched endpoint for the request had a rate-limiting policy.
`aspnetcore.rate_limiting.result`	string	The rate limiting result shows whether lease was acquired or contains a rejection reason.	`acquired`; `request_canceled`	Always

Available starting in: .NET 8.0.

`Microsoft.AspNetCore.HeaderParsing`

The `Microsoft.AspNetCore.HeaderParsing` metrics report information about ASP.NET Core header parsing:

- aspnetcore.header_parsing.parse_errors
- aspnetcore.header_parsing.cache_accesses

Metric: `aspnetcore.header_parsing.parse_errors`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
`aspnetcore.header_parsing.parse_errors`	Counter	`{parse_error}`	Number of errors that occurred when parsing HTTP request headers.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.header_parsing.header.name`	string	The header name.	`Content-Type`	Always
`error.type`	string	The error message.	`Unable to parse media type value.`	Always

Available starting in: .NET 8.0.

Metric: `aspnetcore.header_parsing.cache_accesses`

The metric is emitted only for HTTP request header parsers that support caching.

Expand table

Name	Instrument Type	Unit (UCUM)	Description
`aspnetcore.header_parsing.cache_accesses`	Counter	`{cache_access}`	Number of times a cache storing parsed header values was accessed.

Expand table

Attribute	Type	Description	Examples	Presence
`aspnetcore.header_parsing.header.name`	string	The header name.	`Content-Type`	Always

Attribute	Type	Description	Examples	Presence
`aspnetcore.header_parsing.cache_access.type`	string	A value indicating whether the header's value was found in the cache or not.	`Hit`; `Miss`	Always

Available starting in: .NET 8.0.

`Microsoft.AspNetCore.Server.Kestrel`

The `Microsoft.AspNetCore.Server.Kestrel` metrics report HTTP connection information from ASP.NET Core Kestrel web server:

- kestrel.active_connections
- kestrel.connection.duration
- kestrel.rejected_connections
- kestrel.queued_connections
- kestrel.queued_requests
- kestrel.upgraded_connections
- kestrel.tls_handshake.duration
- kestrel.active_tls_handshakes

Metric: `kestrel.active_connections`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.active_connections	UpDownCounter	`{connection}`	Number of connections that are currently active on the server.

Expand table

Attribute	Type	Description	Examples	Presence
`network.transport`	string	OSI transport layer or inter-process communication method.	`tcp`; `unix`	Always
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.

Available starting in: .NET 8.0.

Metric: `kestrel.connection.duration`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.connection.duration	Histogram	`s`	The duration of connections on the server.

Expand table

Attribute	Type	Description	Examples	Presence
`error.type`	string	Describes a type of error the connection ended with, or the unhandled exception type thrown during the connection pipeline. Known connection errors can be found at Semantic Conventions for Kestrel web server metrics.	`connection_reset`; `invalid_request_headers`; `System.OperationCanceledException`	If the connection ended with a known error or an exception was thrown.
`network.protocol.name`	string	OSI application layer or non-OSI equivalent.	`http`; `web_sockets`	Always
`network.protocol.version`	string	Version of the protocol specified in `network.protocol.name`.	`1.1`; `2`	Always
`network.transport`	string	OSI transport layer or inter-process communication method.	`tcp`; `unix`	Always
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.
`tls.protocol.version`	string	TLS protocol version.	`1.2`; `1.3`	If the connection is secured with TLS.

As this metric is tracking the connection duration, and ideally http connections are used for multiple requests, the buckets should be longer than those used for request durations. For example, using [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 30, 60, 120, 300] provides an upper bucket of 5 mins.

Starting in .NET 9, when a connection ends with a known error, the `error.type` attribute value is set to the known error type. Known connection errors can be found at Semantic Conventions for Kestrel web server metrics.

Available starting in: .NET 8.

Metric: `kestrel.rejected_connections`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.rejected_connections	Counter	`{connection}`	Number of connections rejected by the server.

Expand table

Attribute	Type	Description	Examples	Presence
`network.transport`	string	OSI transport layer or inter-process communication method.	`tcp`; `unix`	Always
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.

Connections are rejected when the currently active count exceeds the value configured with `MaxConcurrentConnections`.

Available starting in: .NET 8.0.

Metric: `kestrel.queued_connections`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.queued_connections	UpDownCounter	`{connection}`	Number of connections that are currently queued and are waiting to start.

Expand table

Attribute	Type	Description	Examples	Presence
`network.transport`	string	OSI transport layer or inter-process communication method.	`tcp`; `unix`	Always
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.

Available starting in: .NET 8.0.

Metric: `kestrel.queued_requests`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.queued_requests	UpDownCounter	`{request}`	Number of HTTP requests on multiplexed connections (HTTP/2 and HTTP/3) that are currently queued and are waiting to start.

Expand table

Attribute	Type	Description	Examples	Presence
`network.protocol.name`	string	OSI application layer or non-OSI equivalent.	`http`; `web_sockets`	Always
`network.protocol.version`	string	Version of the protocol specified in `network.protocol.name`.	`1.1`; `2`	Always
`network.transport`	string	OSI transport layer or inter-process communication method.	`tcp`; `unix`	Always
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.

Available starting in: .NET 8.0.

Metric: `kestrel.upgraded_connections`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.upgraded_connections	UpDownCounter	`{connection}`	Number of connections that are currently upgraded (WebSockets).

Expand table

Attribute	Type	Description	Examples	Presence
`network.transport`	string	OSI transport layer or inter-process communication method.	`tcp`; `unix`	Always
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.

The counter only tracks HTTP/1.1 connections.

Available starting in: .NET 8.0.

Metric: `kestrel.tls_handshake.duration`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.tls_handshake.duration	Histogram	`s`	The duration of TLS handshakes on the server.

Expand table

Attribute	Type	Description	Examples	Presence
`error.type`	string	The full name of exception type.	`System.OperationCanceledException`; `Contoso.MyException`	If an exception was thrown.
`network.transport`	string	OSI transport layer or inter-process	`tcp`; `unix`	Always

Attribute	Type	Description	Examples	Presence
		communication method.		
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.
`tls.protocol.version`	string	TLS protocol version.	`1.2`; `1.3`	If the connection is secured with TLS.

When using OpenTelemetry, the default buckets for this metic are set to [0.005, 0.01, 0.025, 0.05, 0.075, 0.1, 0.25, 0.5, 0.75, 1, 2.5, 5, 7.5, 10].

Available starting in: .NET 8.0.

Metric: `kestrel.active_tls_handshakes`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
kestrel.active_tls_handshakes	UpDownCounter	`{handshake}`	Number of TLS handshakes that are currently in progress on the server.

Expand table

Attribute	Type	Description	Examples	Presence
`network.transport`	string	OSI transport layer or inter-process communication method.	`tcp`; `unix`	Always
`network.type`	string	OSI network layer or non-OSI equivalent.	`ipv4`; `ipv6`	If the transport is `tcp` or `udp`.
`server.address`	string	Server address domain name if available without reverse DNS lookup; otherwise, IP address or Unix domain socket name.	`example.com`	Always

Attribute	Type	Description	Examples	Presence
`server.port`	int	Server port number	`80`; `8080`; `443`	If the transport is `tcp` or `udp`.

Available starting in: .NET 8.0.

`Microsoft.AspNetCore.Http.Connections`

The `Microsoft.AspNetCore.Http.Connections` metrics report connection information from ASP.NET Core SignalR:

- signalr.server.connection.duration
- signalr.server.active_connections

Metric: `signalr.server.connection.duration`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
signalr.server.connection.duration	Histogram	`s`	The duration of connections on the server.

Expand table

Attribute	Type	Description	Examples	Presence
`signalr.connection.status`	string	SignalR HTTP connection closure status.	`app_shutdown`; `timeout`	Always
`signalr.transport`	string	SignalR transport type	`web_sockets`; `long_polling`	Always

Available starting in: .NET 8.0.

Expand table

Value	Description
`normal_closure`	The connection was closed normally.
`timeout`	The connection was closed due to a timeout.
`app_shutdown`	The connection was closed because the app is shutting down.

`signalr.transport` is one of the following:

Expand table

Value	Protocol
`server_sent_events`	server-sent events
`long_polling`	Long Polling
`web_sockets`	WebSocket

As this metric is tracking the connection duration, and ideally SignalR connections are durable, the buckets should be longer than those used for request durations. For example, using [0, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 30, 60, 120, 300] provides an upper bucket of 5 mins.

Available starting in: .NET 8.0.

Metric: `signalr.server.active_connections`

Expand table

Name	Instrument Type	Unit (UCUM)	Description
signalr.server.active_connections	UpDownCounter	`{connection}`	Number of connections that are currently active on the server.

Expand table

Attribute	Type	Description	Examples	Presence
`signalr.connection.status`	string	SignalR HTTP connection closure status.	`app_shutdown`; `timeout`	Always
`signalr.transport`	string	SignalR transport type	`web_sockets`; `long_polling`	Always

Available starting in: .NET 8.0.

Use HttpContext in ASP.NET Core

Article • 12/13/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

HttpContext encapsulates all information about an individual HTTP request and response. An `HttpContext` instance is initialized when an HTTP request is received. The `HttpContext` instance is accessible by middleware and app frameworks such as Blazor Web Apps, Web API controllers, Razor Pages, SignalR, gRPC, and more.

`HttpRequest`

HttpContext.Request provides access to HttpRequest. `HttpRequest` has information about the incoming HTTP request, and it's initialized when an HTTP request is received by the server. `HttpRequest` isn't read-only, and middleware can change request values in the middleware pipeline.

Commonly used members on `HttpRequest` include:

Expand table

Property	Description	Example
HttpRequest.Path	The request path.	`/en/article/getstarted`
HttpRequest.Method	The request method.	`GET`
HttpRequest.Headers	A collection of request headers.	`user-agent=Edge` `x-custom-header=MyValue`
HttpRequest.RouteValues	A collection of route values. The collection is set when the request is matched to a route.	`language=en` `article=getstarted`
HttpRequest.Query	A collection of query values parsed from QueryString.	`filter=hello` `page=1`
HttpRequest.ReadFormAsync()	A method that reads the request body as a form and returns a form values	`email=user@contoso.com`

Property	Description	Example
	collection. For information about why `ReadFormAsync` should be used to access form data, see Prefer ReadFormAsync over Request.Form.	
HttpRequest.Body	A Stream for reading the request body.	UTF-8 JSON payload

Get request headers

HttpRequest.Headers provides access to the request headers sent with the HTTP request. There are two ways to access headers using this collection:

- Provide the header name to the indexer on the header collection. The header name isn't case-sensitive. The indexer can access any header value.
- The header collection also has properties for getting and setting commonly used HTTP headers. The properties provide a fast, IntelliSense driven way to access headers.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", (HttpRequest request) =>
{
    var userAgent = request.Headers.UserAgent;
    var customHeader = request.Headers["x-custom-header"];

    return Results.Ok(new { userAgent = userAgent, customHeader =
customHeader });
});

app.Run();
```

For information on efficiently handling headers that appear more than once, see A brief look at StringValues.

Read request body

An HTTP request can include a request body. The request body is data associated with the request, such as the content of an HTML form, UTF-8 JSON payload, or a file.

HttpRequest.Body allows the request body to be read with Stream:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapPost("/uploadstream", async (IConfiguration config, HttpContext
context) =>
{
    var filePath = Path.Combine(config["StoredFilesPath"],
Path.GetRandomFileName());

    await using var writeStream = File.Create(filePath);
    await context.Request.Body.CopyToAsync(writeStream);
});

app.Run();
```

`HttpRequest.Body` can be read directly or used with other APIs that accept stream.

> ⓘ Note
>
> Minimal APIs supports binding HttpRequest.Body directly to a Stream parameter.

Enable request body buffering

The request body can only be read once, from beginning to end. Forward-only reading of the request body avoids the overhead of buffering the entire request body and reduces memory usage. However, in some scenarios, there's a need to read the request body multiple times. For example, middleware might need to read the request body and then rewind it so it's available for the endpoint.

The EnableBuffering extension method enables buffering of the HTTP request body and is the recommended way to enable multiple reads. Because a request can be any size, `EnableBuffering` supports options for buffering large request bodies to disk, or rejecting them entirely.

The middleware in the following example:

- Enables multiple reads with `EnableBuffering`. It must be called before reading the request body.
- Reads the request body.
- Rewinds the request body to the start so other middleware or the endpoint can read it.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) =>
{
    context.Request.EnableBuffering();
    await ReadRequestBody(context.Request.Body);
    context.Request.Body.Position = 0;

    await next.Invoke();
});

app.Run();
```

BodyReader

An alternative way to read the request body is to use the HttpRequest.BodyReader property. The `BodyReader` property exposes the request body as a PipeReader. This API is from I/O pipelines, an advanced, high-performance way to read the request body.

The reader directly accesses the request body and manages memory on the caller's behalf. Unlike `HttpRequest.Body`, the reader doesn't copy request data into a buffer. However, a reader is more complicated to use than a stream and should be used with caution.

For information on how to read content from `BodyReader`, see I/O pipelines PipeReader.

`HttpResponse`

HttpContext.Response provides access to HttpResponse. `HttpResponse` is used to set information on the HTTP response sent back to the client.

Commonly used members on `HttpResponse` include:

Expand table

Property	Description	Example
HttpResponse.StatusCode	The response code. Must be set before writing to the response body.	`200`
HttpResponse.ContentType	The response `content-type` header. Must be set before writing to the response body.	`application/json`

Property	Description	Example
HttpResponse.Headers	A collection of response headers. Must be set before writing to the response body.	`server=Kestrel` `x-custom-header=MyValue`
HttpResponse.Body	A Stream for writing the response body.	Generated web page

Set response headers

HttpResponse.Headers provides access to the response headers sent with the HTTP response. There are two ways to access headers using this collection:

- Provide the header name to the indexer on the header collection. The header name isn't case-sensitive. The indexer can access any header value.
- Use the header collection properties for getting and setting commonly used HTTP headers. The properties provide a fast, IntelliSense driven way to access headers.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", (HttpResponse response) =>
{
    response.Headers.CacheControl = "no-cache";
    response.Headers["x-custom-header"] = "Custom value";

    return Results.File(File.OpenRead("helloworld.txt"));
});

app.Run();
```

An app can't modify headers after the response has started. Once the response starts, the headers are sent to the client. A response is started by flushing the response body or calling HttpResponse.StartAsync(CancellationToken). The HttpResponse.HasStarted property indicates whether the response has started. An error is thrown when attempting to modify headers after the response has started:

> System.InvalidOperationException: Headers are read-only, response has already started.

Note

Unless response buffering is enabled, all write operations (for example, WriteAsync) flush the response body internally and mark the response as started. Response buffering is disabled by default.

Write response body

An HTTP response can include a response body. The response body is data associated with the response, such as generated web page content, UTF-8 JSON payload, or a file.

HttpResponse.Body allows the response body to be written with Stream:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapPost("/downloadfile", async (IConfiguration config, HttpContext
context) =>
{
    var filePath = Path.Combine(config["StoredFilesPath"],
"helloworld.txt");

    await using var fileStream = File.OpenRead(filePath);
    await fileStream.CopyToAsync(context.Response.Body);
});

app.Run();
```

`HttpResponse.Body` can be written directly or used with other APIs that write to a stream.

BodyWriter

An alternative way to write the response body is to use the HttpResponse.BodyWriter property. The `BodyWriter` property exposes the response body as a PipeWriter. This API is from I/O pipelines, and it's an advanced, high-performance way to write the response.

The writer provides direct access to the response body and manages memory on the caller's behalf. Unlike `HttpResponse.Body`, the write doesn't copy request data into a buffer. However, a writer is more complicated to use than a stream and writer code should be thoroughly tested.

For information on how to write content to `BodyWriter`, see I/O pipelines PipeWriter.

Set response trailers

HTTP/2 and HTTP/3 support response trailers. Trailers are headers sent with the response after the response body is complete. Because trailers are sent after the response body, trailers can be added to the response at any time.

The following code sets trailers using AppendTrailer:

```C#
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", (HttpResponse response) =>
{
    // Write body
    response.WriteAsync("Hello world");

    if (response.SupportsTrailers())
    {
        response.AppendTrailer("trailername", "TrailerValue");
    }
});

app.Run();
```

RequestAborted

The HttpContext.RequestAborted cancellation token can be used to notify that the HTTP request has been aborted by the client or server. The cancellation token should be passed to long-running tasks so they can be canceled if the request is aborted. For example, aborting a database query or HTTP request to get data to return in the response.

```C#
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

var httpClient = new HttpClient();
app.MapPost("/books/{bookId}", async (int bookId, HttpContext context) =>
{
    var stream = await httpClient.GetStreamAsync(
        $"http://contoso/books/{bookId}.json", context.RequestAborted);

    // Proxy the response as JSON
    return Results.Stream(stream, "application/json");
});

app.Run();
```

The `RequestAborted` cancellation token doesn't need to be used for request body read operations because reads always throw immediately when the request is aborted. The `RequestAborted` token is also usually unnecessary when writing response bodies, because writes immediately no-op when the request is aborted.

In some cases, passing the `RequestAborted` token to write operations can be a convenient way to force a write loop to exit early with an OperationCanceledException. However, it's typically better to pass the `RequestAborted` token into any asynchronous operations responsible for retrieving the response body content instead.

> ⓘ Note
>
> Minimal APIs supports binding HttpContext.RequestAborted directly to a CancellationToken parameter.

`Abort()`

The HttpContext.Abort() method can be used to abort an HTTP request from the server. Aborting the HTTP request immediately triggers the HttpContext.RequestAborted cancellation token and sends a notification to the client that the server has aborted the request.

The middleware in the following example:

- Adds a custom check for malicious requests.
- Aborts the HTTP request if the request is malicious.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Use(async (context, next) =>
{
    if (RequestAppearsMalicious(context.Request))
    {
        // Malicious requests don't even deserve an error response (e.g.
400).
        context.Abort();
        return;
    }

    await next.Invoke();
});
```

```
app.Run();
```

User

The HttpContext.User property is used to get or set the user, represented by ClaimsPrincipal, for the request. The ClaimsPrincipal is typically set by ASP.NET Core authentication.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/user/current", [Authorize] async (HttpContext context) =>
{
    var user = await GetUserAsync(context.User.Identity.Name);
    return Results.Ok(user);
});

app.Run();
```

> ⓘ Note
>
> Minimal APIs supports binding HttpContext.User directly to a ClaimsPrincipal parameter.

Features

The HttpContext.Features property provides access to the collection of feature interfaces for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features. Some advanced features are only available by accessing the associated interface through the feature collection.

The following example:

- Gets IHttpMinRequestBodyDataRateFeature from the features collection.
- Sets MinDataRate to null. This removes the minimum data rate that the request body must be sent by the client for this HTTP request.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/long-running-stream", async (HttpContext context) =>
{
    var feature = context.Features.Get<IHttpMinRequestBodyDataRateFeature>
();
    if (feature != null)
    {
        feature.MinDataRate = null;
    }

    // await and read long-running stream from request body.
    await Task.Yield();
});

app.Run();
```

For more information about using request features and `HttpContext`, see Request Features in ASP.NET Core.

HttpContext isn't thread safe

This article primarily discusses using `HttpContext` in request and response flow from Blazor Web App components, Razor Pages, controllers, middleware, and so forth. Consider the following when using `HttpContext` outside the request and response flow:

- The `HttpContext` is **NOT** thread safe. Accessing it from multiple threads can result in unpredictable results, such as exceptions and data corruption.
- The IHttpContextAccessor interface should be used with caution. As always, the `HttpContext` must ***not*** be captured outside of the request flow.

  `IHttpContextAccessor`:
  - Relies on AsyncLocal<T>, which can have a negative performance impact on asynchronous calls.
  - Creates a dependency on "ambient state" which can make testing more difficult.
- IHttpContextAccessor.HttpContext might be `null` if accessed outside of the request flow.
- To access information from `HttpContext` outside the request flow, copy the information inside the request flow. Be careful to copy the actual data and not just references. For example, rather than copying a reference to an `IHeaderDictionary`, copy the relevant header values or copy the entire dictionary key by key before leaving the request flow.
- Don't capture `IHttpContextAccessor.HttpContext` in a constructor.

The following sample logs GitHub branches when requested from the `/branch` endpoint:

C#

```
using System.Text.Json;
using HttpContextInBackgroundThread;
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpContextAccessor();
builder.Services.AddHostedService<PeriodicBranchesLoggerService>();

builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // The GitHub API requires two headers. The Use-Agent header is added
    // dynamically through UserAgentHeaderHandler
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.Accept, "application/vnd.github.v3+json");
}).AddHttpMessageHandler<UserAgentHeaderHandler>();

builder.Services.AddTransient<UserAgentHeaderHandler>();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapGet("/branches", async (IHttpClientFactory httpClientFactory,
                          HttpContext context, Logger<Program> logger) =>
{
    var httpClient = httpClientFactory.CreateClient("GitHub");
    var httpResponseMessage = await httpClient.GetAsync(
        "repos/dotnet/AspNetCore.Docs/branches");

    if (!httpResponseMessage.IsSuccessStatusCode)
        return Results.BadRequest();

    await using var contentStream =
        await httpResponseMessage.Content.ReadAsStreamAsync();

    var response = await JsonSerializer.DeserializeAsync
        <IEnumerable<GitHubBranch>>(contentStream);

    app.Logger.LogInformation($"/branches request: " +
                              $"{JsonSerializer.Serialize(response)}");

    return Results.Ok(response);
});

app.Run();
```

The GitHub API requires two headers. The `User-Agent` header is added dynamically by the `UserAgentHeaderHandler`:

C#

```
using System.Text.Json;
using HttpContextInBackgroundThread;
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpContextAccessor();
builder.Services.AddHostedService<PeriodicBranchesLoggerService>();

builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // The GitHub API requires two headers. The Use-Agent header is added
    // dynamically through UserAgentHeaderHandler
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.Accept, "application/vnd.github.v3+json");
}).AddHttpMessageHandler<UserAgentHeaderHandler>();

builder.Services.AddTransient<UserAgentHeaderHandler>();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapGet("/branches", async (IHttpClientFactory httpClientFactory,
                          HttpContext context, Logger<Program> logger) =>
{
    var httpClient = httpClientFactory.CreateClient("GitHub");
    var httpResponseMessage = await httpClient.GetAsync(
        "repos/dotnet/AspNetCore.Docs/branches");

    if (!httpResponseMessage.IsSuccessStatusCode)
        return Results.BadRequest();

    await using var contentStream =
        await httpResponseMessage.Content.ReadAsStreamAsync();

    var response = await JsonSerializer.DeserializeAsync
        <IEnumerable<GitHubBranch>>(contentStream);

    app.Logger.LogInformation($"/branches request: " +
                              $"{JsonSerializer.Serialize(response)}");

    return Results.Ok(response);
});

app.Run();
```

The `UserAgentHeaderHandler`:

C#

```
using Microsoft.Net.Http.Headers;

namespace HttpContextInBackgroundThread;

public class UserAgentHeaderHandler : DelegatingHandler
{
    private readonly IHttpContextAccessor _httpContextAccessor;
    private readonly ILogger _logger;

    public UserAgentHeaderHandler(IHttpContextAccessor httpContextAccessor,
                                  ILogger<UserAgentHeaderHandler> logger)
    {
        _httpContextAccessor = httpContextAccessor;
        _logger = logger;
    }

    protected override async Task<HttpResponseMessage>
                                    SendAsync(HttpRequestMessage request,
                                    CancellationToken cancellationToken)
    {
        var contextRequest = _httpContextAccessor.HttpContext?.Request;
        string? userAgentString = contextRequest?.Headers["user-
agent"].ToString();

        if (string.IsNullOrEmpty(userAgentString))
        {
            userAgentString = "Unknown";
        }

        request.Headers.Add(HeaderNames.UserAgent, userAgentString);
        _logger.LogInformation($"User-Agent: {userAgentString}");

        return await base.SendAsync(request, cancellationToken);
    }
}
```

In the preceding code, when the `HttpContext` is `null`, the `userAgent` string is set to `"Unknown"`. If possible, `HttpContext` should be explicitly passed to the service. Explicitly passing in `HttpContext` data:

- Makes the service API more useable outside the request flow.
- Is better for performance.
- Makes the code easier to understand and reason about than relying on ambient state.

When the service must access `HttpContext`, it should account for the possibility of `HttpContext` being `null` when not called from a request thread.

The application also includes `PeriodicBranchesLoggerService`, which logs the open GitHub branches of the specified repository every 30 seconds:

C#

```
using System.Text.Json;

namespace HttpContextInBackgroundThread;

public class PeriodicBranchesLoggerService : BackgroundService
{
    private readonly IHttpClientFactory _httpClientFactory;
    private readonly ILogger _logger;
    private readonly PeriodicTimer _timer;

    public PeriodicBranchesLoggerService(IHttpClientFactory
httpClientFactory,

ILogger<PeriodicBranchesLoggerService> logger)
    {
        _httpClientFactory = httpClientFactory;
        _logger = logger;
        _timer = new PeriodicTimer(TimeSpan.FromSeconds(30));
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (await _timer.WaitForNextTickAsync(stoppingToken))
        {
            try
            {
                // Cancel sending the request to sync branches if it takes
too long
                // rather than miss sending the next request scheduled 30
seconds from now.
                // Having a single loop prevents this service from sending
an unbounded
                // number of requests simultaneously.
                using var syncTokenSource =
CancellationTokenSource.CreateLinkedTokenSource(stoppingToken);
                syncTokenSource.CancelAfter(TimeSpan.FromSeconds(30));

                var httpClient = _httpClientFactory.CreateClient("GitHub");
                var httpResponseMessage = await
httpClient.GetAsync("repos/dotnet/AspNetCore.Docs/branches",

stoppingToken);

                if (httpResponseMessage.IsSuccessStatusCode)
```

```
                {
                    await using var contentStream =
                        await
httpResponseMessage.Content.ReadAsStreamAsync(stoppingToken);

                    // Sync the response with preferred datastore.
                    var response = await JsonSerializer.DeserializeAsync<
                        IEnumerable<GitHubBranch>>(contentStream,
cancellationToken: stoppingToken);

                    _logger.LogInformation(
                        $"Branch sync successful! Response:
{JsonSerializer.Serialize(response)}");
                }
                else
                {
                    _logger.LogError(1, $"Branch sync failed! HTTP status
code: {httpResponseMessage.StatusCode}");
                }
            }
            catch (Exception ex)
            {
                _logger.LogError(1, ex, "Branch sync failed!");
            }
        }
    }

    public override Task StopAsync(CancellationToken stoppingToken)
    {
        // This will cause any active call to WaitForNextTickAsync() to
return false immediately.
        _timer.Dispose();
        // This will cancel the stoppingToken and await
ExecuteAsync(stoppingToken).
        return base.StopAsync(stoppingToken);
    }
}
```

`PeriodicBranchesLoggerService` is a hosted service, which runs outside the request and response flow. Logging from the `PeriodicBranchesLoggerService` has a null `HttpContext`. The `PeriodicBranchesLoggerService` was written to not depend on the `HttpContext`.

C#

```
using System.Text.Json;
using HttpContextInBackgroundThread;
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpContextAccessor();
builder.Services.AddHostedService<PeriodicBranchesLoggerService>();
```

```
builder.Services.AddHttpClient("GitHub", httpClient =>
{
```

Additional resources

For more information about accessing `HttpContext`, see Access HttpContext in ASP.NET Core.

Routing in ASP.NET Core

Article • 09/20/2023

By Ryan Nowak, Kirk Larkin, and Rick Anderson

Routing is responsible for matching incoming HTTP requests and dispatching those requests to the app's executable endpoints. Endpoints are the app's units of executable request-handling code. Endpoints are defined in the app and configured when the app starts. The endpoint matching process can extract values from the request's URL and provide those values for request processing. Using endpoint information from the app, routing is also able to generate URLs that map to endpoints.

Apps can configure routing using:

- Controllers
- Razor Pages
- SignalR
- gRPC Services
- Endpoint-enabled middleware such as Health Checks.
- Delegates and lambdas registered with routing.

This article covers low-level details of ASP.NET Core routing. For information on configuring routing:

- For controllers, see Routing to controller actions in ASP.NET Core.
- For Razor Pages conventions, see Razor Pages route and app conventions in ASP.NET Core.
- For Blazor routing guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor routing and navigation.

Routing basics

The following code shows a basic example of routing:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding example includes a single endpoint using the MapGet method:

- When an HTTP `GET` request is sent to the root URL `/`:
  - The request delegate executes.
  - `Hello World!` is written to the HTTP response.
- If the request method is not `GET` or the root URL is not `/`, no route matches and an HTTP 404 is returned.

Routing uses a pair of middleware, registered by UseRouting and UseEndpoints:

- `UseRouting` adds route matching to the middleware pipeline. This middleware looks at the set of endpoints defined in the app, and selects the best match based on the request.
- `UseEndpoints` adds endpoint execution to the middleware pipeline. It runs the delegate associated with the selected endpoint.

Apps typically don't need to call `UseRouting` or `UseEndpoints`. WebApplicationBuilder configures a middleware pipeline that wraps middleware added in `Program.cs` with `UseRouting` and `UseEndpoints`. However, apps can change the order in which `UseRouting` and `UseEndpoints` run by calling these methods explicitly. For example, the following code makes an explicit call to `UseRouting`:

C#

```
app.Use(async (context, next) =>
{
    // ...
    await next(context);
});

app.UseRouting();

app.MapGet("/", () => "Hello World!");
```

In the preceding code:

- The call to `app.Use` registers a custom middleware that runs at the start of the pipeline.
- The call to `UseRouting` configures the route matching middleware to run *after* the custom middleware.
- The endpoint registered with `MapGet` runs at the end of the pipeline.

If the preceding example didn't include a call to `UseRouting`, the custom middleware would run *after* the route matching middleware.

Note: Routes added directly to the WebApplication execute at the ***end*** of the pipeline.

Endpoints

The `MapGet` method is used to define an **endpoint**. An endpoint is something that can be:

- Selected, by matching the URL and HTTP method.
- Executed, by running the delegate.

Endpoints that can be matched and executed by the app are configured in `UseEndpoints`. For example, MapGet, MapPost, and similar methods connect request delegates to the routing system. Additional methods can be used to connect ASP.NET Core framework features to the routing system:

- MapRazorPages for Razor Pages
- MapControllers for controllers
- MapHub<THub> for SignalR
- MapGrpcService<TService> for gRPC

The following example shows routing with a more sophisticated route template:

C#

```
app.MapGet("/hello/{name:alpha}", (string name) => $"Hello {name}!");
```

The string `/hello/{name:alpha}` is a **route template**. A route template is used to configure how the endpoint is matched. In this case, the template matches:

- A URL like `/hello/Docs`
- Any URL path that begins with `/hello/` followed by a sequence of alphabetic characters. `:alpha` applies a route constraint that matches only alphabetic characters. Route constraints are explained later in this article.

The second segment of the URL path, `{name:alpha}`:

- Is bound to the `name` parameter.
- Is captured and stored in HttpRequest.RouteValues.

The following example shows routing with health checks and authorization:

C#

```
app.UseAuthentication();
app.UseAuthorization();

app.MapHealthChecks("/healthz").RequireAuthorization();
app.MapGet("/", () => "Hello World!");
```

The preceding example demonstrates how:

- The authorization middleware can be used with routing.
- Endpoints can be used to configure authorization behavior.

The MapHealthChecks call adds a health check endpoint. Chaining RequireAuthorization on to this call attaches an authorization policy to the endpoint.

Calling UseAuthentication and UseAuthorization adds the authentication and authorization middleware. These middleware are placed between UseRouting and `UseEndpoints` so that they can:

- See which endpoint was selected by `UseRouting`.
- Apply an authorization policy before UseEndpoints dispatches to the endpoint.

Endpoint metadata

In the preceding example, there are two endpoints, but only the health check endpoint has an authorization policy attached. If the request matches the health check endpoint, `/healthz`, an authorization check is performed. This demonstrates that endpoints can have extra data attached to them. This extra data is called endpoint **metadata**:

- The metadata can be processed by routing-aware middleware.
- The metadata can be of any .NET type.

Routing concepts

The routing system builds on top of the middleware pipeline by adding the powerful **endpoint** concept. Endpoints represent units of the app's functionality that are distinct from each other in terms of routing, authorization, and any number of ASP.NET Core's systems.

ASP.NET Core endpoint definition

An ASP.NET Core endpoint is:

- Executable: Has a RequestDelegate.
- Extensible: Has a Metadata collection.
- Selectable: Optionally, has routing information.
- Enumerable: The collection of endpoints can be listed by retrieving the EndpointDataSource from DI.

The following code shows how to retrieve and inspect the endpoint matching the current request:

C#

```
app.Use(async (context, next) =>
{
    var currentEndpoint = context.GetEndpoint();

    if (currentEndpoint is null)
    {
        await next(context);
        return;
    }

    Console.WriteLine($"Endpoint: {currentEndpoint.DisplayName}");

    if (currentEndpoint is RouteEndpoint routeEndpoint)
    {
        Console.WriteLine($"  - Route Pattern:
{routeEndpoint.RoutePattern}");
    }

    foreach (var endpointMetadata in currentEndpoint.Metadata)
    {
        Console.WriteLine($"  - Metadata: {endpointMetadata}");
    }

    await next(context);
});

app.MapGet("/", () => "Inspect Endpoint.");
```

The endpoint, if selected, can be retrieved from the `HttpContext`. Its properties can be inspected. Endpoint objects are immutable and cannot be modified after creation. The most common type of endpoint is a RouteEndpoint. `RouteEndpoint` includes information that allows it to be selected by the routing system.

In the preceding code, app.Use configures an inline middleware.

The following code shows that, depending on where `app.Use` is called in the pipeline, there may not be an endpoint:

C#

```
// Location 1: before routing runs, endpoint is always null here.
app.Use(async (context, next) =>
{
    Console.WriteLine($"1. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    await next(context);
});

app.UseRouting();

// Location 2: after routing runs, endpoint will be non-null if routing
found a match.
app.Use(async (context, next) =>
{
    Console.WriteLine($"2. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    await next(context);
});

// Location 3: runs when this endpoint matches
app.MapGet("/", (HttpContext context) =>
{
    Console.WriteLine($"3. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    return "Hello World!";
}).WithDisplayName("Hello");

app.UseEndpoints(_ => { });

// Location 4: runs after UseEndpoints - will only run if there was no
match.
app.Use(async (context, next) =>
{
    Console.WriteLine($"4. Endpoint: {context.GetEndpoint()?.DisplayName ??
"(null)"}");
    await next(context);
});
```

The preceding sample adds `Console.WriteLine` statements that display whether or not an endpoint has been selected. For clarity, the sample assigns a display name to the provided `/` endpoint.

The preceding sample also includes calls to `UseRouting` and `UseEndpoints` to control exactly when these middleware run within the pipeline.

Running this code with a URL of `/` displays:

txt

```
1. Endpoint: (null)
2. Endpoint: Hello
3. Endpoint: Hello
```

Running this code with any other URL displays:

txt

```
1. Endpoint: (null)
2. Endpoint: (null)
4. Endpoint: (null)
```

This output demonstrates that:

- The endpoint is always null before `UseRouting` is called.
- If a match is found, the endpoint is non-null between `UseRouting` and UseEndpoints.
- The `UseEndpoints` middleware is **terminal** when a match is found. Terminal middleware is defined later in this article.
- The middleware after `UseEndpoints` execute only when no match is found.

The `UseRouting` middleware uses the SetEndpoint method to attach the endpoint to the current context. It's possible to replace the `UseRouting` middleware with custom logic and still get the benefits of using endpoints. Endpoints are a low-level primitive like middleware, and aren't coupled to the routing implementation. Most apps don't need to replace `UseRouting` with custom logic.

The `UseEndpoints` middleware is designed to be used in tandem with the `UseRouting` middleware. The core logic to execute an endpoint isn't complicated. Use GetEndpoint to retrieve the endpoint, and then invoke its RequestDelegate property.

The following code demonstrates how middleware can influence or react to routing:

C#

```
app.UseHttpMethodOverride();
app.UseRouting();

app.Use(async (context, next) =>
{
    if (context.GetEndpoint()?.Metadata.GetMetadata<RequiresAuditAttribute>
() is not null)
    {
        Console.WriteLine($"ACCESS TO SENSITIVE DATA AT:
{DateTime.UtcNow}");
```

```
    }

    await next(context);
});

app.MapGet("/", () => "Audit isn't required.");
app.MapGet("/sensitive", () => "Audit required for sensitive data.")
    .WithMetadata(new RequiresAuditAttribute());
```

C#

```
public class RequiresAuditAttribute : Attribute { }
```

The preceding example demonstrates two important concepts:

- Middleware can run before `UseRouting` to modify the data that routing operates upon.
  - Usually middleware that appears before routing modifies some property of the request, such as UseRewriter, UseHttpMethodOverride, or UsePathBase.
- Middleware can run between `UseRouting` and UseEndpoints to process the results of routing before the endpoint is executed.
  - Middleware that runs between `UseRouting` and `UseEndpoints`:
    - Usually inspects metadata to understand the endpoints.
    - Often makes security decisions, as done by `UseAuthorization` and `UseCors`.
  - The combination of middleware and metadata allows configuring policies per-endpoint.

The preceding code shows an example of a custom middleware that supports per-endpoint policies. The middleware writes an *audit log* of access to sensitive data to the console. The middleware can be configured to *audit* an endpoint with the `RequiresAuditAttribute` metadata. This sample demonstrates an *opt-in* pattern where only endpoints that are marked as sensitive are audited. It's possible to define this logic in reverse, auditing everything that isn't marked as safe, for example. The endpoint metadata system is flexible. This logic could be designed in whatever way suits the use case.

The preceding sample code is intended to demonstrate the basic concepts of endpoints. **The sample is not intended for production use**. A more complete version of an *audit log* middleware would:

- Log to a file or database.
- Include details such as the user, IP address, name of the sensitive endpoint, and more.

The audit policy metadata `RequiresAuditAttribute` is defined as an `Attribute` for easier use with class-based frameworks such as controllers and SignalR. When using *route to code*:

- Metadata is attached with a builder API.
- Class-based frameworks include all attributes on the corresponding method and class when creating endpoints.

The best practices for metadata types are to define them either as interfaces or attributes. Interfaces and attributes allow code reuse. The metadata system is flexible and doesn't impose any limitations.

Compare terminal middleware with routing

The following example demonstrates both terminal middleware and routing:

C#

```
// Approach 1: Terminal Middleware.
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/")
    {
        await context.Response.WriteAsync("Terminal Middleware.");
        return;
    }

    await next(context);
});

app.UseRouting();

// Approach 2: Routing.
app.MapGet("/Routing", () => "Routing.");
```

The style of middleware shown with `Approach 1:` is **terminal middleware**. It's called terminal middleware because it does a matching operation:

- The matching operation in the preceding sample is `Path == "/"` for the middleware and `Path == "/Routing"` for routing.
- When a match is successful, it executes some functionality and returns, rather than invoking the `next` middleware.

It's called terminal middleware because it terminates the search, executes some functionality, and then returns.

The following list compares terminal middleware with routing:

- Both approaches allow terminating the processing pipeline:
  - Middleware terminates the pipeline by returning rather than invoking `next`.
  - Endpoints are always terminal.
- Terminal middleware allows positioning the middleware at an arbitrary place in the pipeline:
  - Endpoints execute at the position of UseEndpoints.
- Terminal middleware allows arbitrary code to determine when the middleware matches:
  - Custom route matching code can be verbose and difficult to write correctly.
  - Routing provides straightforward solutions for typical apps. Most apps don't require custom route matching code.
- Endpoints interface with middleware such as `UseAuthorization` and `UseCors`.
  - Using a terminal middleware with `UseAuthorization` or `UseCors` requires manual interfacing with the authorization system.

An endpoint defines both:

- A delegate to process requests.
- A collection of arbitrary metadata. The metadata is used to implement cross-cutting concerns based on policies and configuration attached to each endpoint.

Terminal middleware can be an effective tool, but can require:

- A significant amount of coding and testing.
- Manual integration with other systems to achieve the desired level of flexibility.

Consider integrating with routing before writing a terminal middleware.

Existing terminal middleware that integrates with Map or MapWhen can usually be turned into a routing aware endpoint. MapHealthChecks demonstrates the pattern for router-ware:

- Write an extension method on IEndpointRouteBuilder.
- Create a nested middleware pipeline using CreateApplicationBuilder.
- Attach the middleware to the new pipeline. In this case, UseHealthChecks.
- Build the middleware pipeline into a RequestDelegate.
- Call `Map` and provide the new middleware pipeline.
- Return the builder object provided by `Map` from the extension method.

The following code shows use of MapHealthChecks:

```
C#
```

```
app.UseAuthentication();
app.UseAuthorization();

app.MapHealthChecks("/healthz").RequireAuthorization();
```

The preceding sample shows why returning the builder object is important. Returning the builder object allows the app developer to configure policies such as authorization for the endpoint. In this example, the health checks middleware has no direct integration with the authorization system.

The metadata system was created in response to the problems encountered by extensibility authors using terminal middleware. It's problematic for each middleware to implement its own integration with the authorization system.

URL matching

- Is the process by which routing matches an incoming request to an endpoint.
- Is based on data in the URL path and headers.
- Can be extended to consider any data in the request.

When a routing middleware executes, it sets an `Endpoint` and route values to a request feature on the HttpContext from the current request:

- Calling HttpContext.GetEndpoint gets the endpoint.
- `HttpRequest.RouteValues` gets the collection of route values.

Middleware that runs after the routing middleware can inspect the endpoint and take action. For example, an authorization middleware can interrogate the endpoint's metadata collection for an authorization policy. After all of the middleware in the request processing pipeline is executed, the selected endpoint's delegate is invoked.

The routing system in endpoint routing is responsible for all dispatching decisions. Because the middleware applies policies based on the selected endpoint, it's important that:

- Any decision that can affect dispatching or the application of security policies is made inside the routing system.

> **⚠ Warning**
>
> For backward-compatibility, when a Controller or Razor Pages endpoint delegate is executed, the properties of RouteContext.RouteData are set to appropriate values based on the request processing performed thus far.

The `RouteContext` type will be marked obsolete in a future release:

- Migrate `RouteData.Values` to `HttpRequest.RouteValues`.
- Migrate `RouteData.DataTokens` to retrieve IDataTokensMetadata from the endpoint metadata.

URL matching operates in a configurable set of phases. In each phase, the output is a set of matches. The set of matches can be narrowed down further by the next phase. The routing implementation does not guarantee a processing order for matching endpoints. **All** possible matches are processed at once. The URL matching phases occur in the following order. ASP.NET Core:

1. Processes the URL path against the set of endpoints and their route templates, collecting **all** of the matches.
2. Takes the preceding list and removes matches that fail with route constraints applied.
3. Takes the preceding list and removes matches that fail the set of MatcherPolicy instances.
4. Uses the EndpointSelector to make a final decision from the preceding list.

The list of endpoints is prioritized according to:

- The RouteEndpoint.Order
- The route template precedence

All matching endpoints are processed in each phase until the EndpointSelector is reached. The `EndpointSelector` is the final phase. It chooses the highest priority endpoint from the matches as the best match. If there are other matches with the same priority as the best match, an ambiguous match exception is thrown.

The route precedence is computed based on a **more specific** route template being given a higher priority. For example, consider the templates `/hello` and `/{message}`:

- Both match the URL path `/hello`.
- `/hello` is more specific and therefore higher priority.

In general, route precedence does a good job of choosing the best match for the kinds of URL schemes used in practice. Use Order only when necessary to avoid an ambiguity.

Due to the kinds of extensibility provided by routing, it isn't possible for the routing system to compute ahead of time the ambiguous routes. Consider an example such as the route templates `/{message:alpha}` and `/{message:int}`:

- The `alpha` constraint matches only alphabetic characters.
- The `int` constraint matches only numbers.
- These templates have the same route precedence, but there's no single URL they both match.
- If the routing system reported an ambiguity error at startup, it would block this valid use case.

> ⚠ **Warning**
>
> The order of operations inside **UseEndpoints** doesn't influence the behavior of routing, with one exception. **MapControllerRoute** and **MapAreaRoute** automatically assign an order value to their endpoints based on the order they are invoked. This simulates long-time behavior of controllers without the routing system providing the same guarantees as older routing implementations.
>
> Endpoint routing in ASP.NET Core:
>
> - Doesn't have the concept of routes.
> - Doesn't provide ordering guarantees. All endpoints are processed at once.

Route template precedence and endpoint selection order

Route template precedence is a system that assigns each route template a value based on how specific it is. Route template precedence:

- Avoids the need to adjust the order of endpoints in common cases.
- Attempts to match the common-sense expectations of routing behavior.

For example, consider templates `/Products/List` and `/Products/{id}`. It would be reasonable to assume that `/Products/List` is a better match than `/Products/{id}` for the URL path `/Products/List`. This works because the literal segment `/List` is considered to have better precedence than the parameter segment `/{id}`.

The details of how precedence works are coupled to how route templates are defined:

- Templates with more segments are considered more specific.
- A segment with literal text is considered more specific than a parameter segment.
- A parameter segment with a constraint is considered more specific than one without.
- A complex segment is considered as specific as a parameter segment with a constraint.

- Catch-all parameters are the least specific. See **catch-all** in the Route templates section for important information on catch-all routes.

URL generation concepts

URL generation:

- Is the process by which routing can create a URL path based on a set of route values.
- Allows for a logical separation between endpoints and the URLs that access them.

Endpoint routing includes the LinkGenerator API. `LinkGenerator` is a singleton service available from DI. The `LinkGenerator` API can be used outside of the context of an executing request. Mvc.IUrlHelper and scenarios that rely on IUrlHelper, such as Tag Helpers, HTML Helpers, and Action Results, use the `LinkGenerator` API internally to provide link generating capabilities.

The link generator is backed by the concept of an **address** and **address schemes**. An address scheme is a way of determining the endpoints that should be considered for link generation. For example, the route name and route values scenarios many users are familiar with from controllers and Razor Pages are implemented as an address scheme.

The link generator can link to controllers and Razor Pages via the following extension methods:

- GetPathByAction
- GetUriByAction
- GetPathByPage
- GetUriByPage

Overloads of these methods accept arguments that include the `HttpContext`. These methods are functionally equivalent to Url.Action and Url.Page, but offer additional flexibility and options.

The `GetPath*` methods are most similar to `Url.Action` and `Url.Page`, in that they generate a URI containing an absolute path. The `GetUri*` methods always generate an absolute URI containing a scheme and host. The methods that accept an `HttpContext` generate a URI in the context of the executing request. The ambient route values, URL base path, scheme, and host from the executing request are used unless overridden.

LinkGenerator is called with an address. Generating a URI occurs in two steps:

1. An address is bound to a list of endpoints that match the address.

2. Each endpoint's RoutePattern is evaluated until a route pattern that matches the supplied values is found. The resulting output is combined with the other URI parts supplied to the link generator and returned.

The methods provided by LinkGenerator support standard link generation capabilities for any type of address. The most convenient way to use the link generator is through extension methods that perform operations for a specific address type:

Expand table

Extension Method	Description
GetPathByAddress	Generates a URI with an absolute path based on the provided values.
GetUriByAddress	Generates an absolute URI based on the provided values.

> ⚠ **Warning**
>
> Pay attention to the following implications of calling LinkGenerator methods:
>
> - Use `GetUri*` extension methods with caution in an app configuration that doesn't validate the `Host` header of incoming requests. If the `Host` header of incoming requests isn't validated, untrusted request input can be sent back to the client in URIs in a view or page. We recommend that all production apps configure their server to validate the `Host` header against known valid values.
> - Use LinkGenerator with caution in middleware in combination with `Map` or `MapWhen`. `Map*` changes the base path of the executing request, which affects the output of link generation. All of the LinkGenerator APIs allow specifying a base path. Specify an empty base path to undo the `Map*` affect on link generation.

Middleware example

In the following example, a middleware uses the LinkGenerator API to create a link to an action method that lists store products. Using the link generator by injecting it into a class and calling `GenerateLink` is available to any class in an app:

C#

```
public class ProductsMiddleware
{
```

```
    private readonly LinkGenerator _linkGenerator;

    public ProductsMiddleware(RequestDelegate next, LinkGenerator
linkGenerator) =>
        _linkGenerator = linkGenerator;

    public async Task InvokeAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Plain;

        var productsPath = _linkGenerator.GetPathByAction("Products",
"Store");

        await httpContext.Response.WriteAsync(
            $"Go to {productsPath} to see our products.");
    }
}
```

Route templates

Tokens within `{}` define route parameters that are bound if the route is matched. More than one route parameter can be defined in a route segment, but route parameters must be separated by a literal value. For example:

`{controller=Home}{action=Index}`

isn't a valid route, because there's no literal value between `{controller}` and `{action}`. Route parameters must have a name and may have additional attributes specified.

Literal text other than route parameters (for example, `{id}`) and the path separator `/` must match the text in the URL. Text matching is case-insensitive and based on the decoded representation of the URL's path. To match a literal route parameter delimiter `{` or `}`, escape the delimiter by repeating the character. For example `{{` or `}}`.

Asterisk `*` or double asterisk `**`:

- Can be used as a prefix to a route parameter to bind to the rest of the URI.
- Are called a **catch-all** parameters. For example, `blog/{**slug}`:
  - Matches any URI that starts with `blog/` and has any value following it.
  - The value following `blog/` is assigned to the slug route value.

Catch-all parameters can also match the empty string.

The catch-all parameter escapes the appropriate characters when the route is used to generate a URL, including path separator `/` characters. For example, the route `foo/{*path}` with route values `{ path = "my/path" }` generates `foo/my%2Fpath`. Note the

escaped forward slash. To round-trip path separator characters, use the `**` route parameter prefix. The route `foo/{**path}` with `{ path = "my/path" }` generates `foo/my/path`.

URL patterns that attempt to capture a file name with an optional file extension have additional considerations. For example, consider the template `files/{filename}.{ext?}`. When values for both `filename` and `ext` exist, both values are populated. If only a value for `filename` exists in the URL, the route matches because the trailing `.` is optional. The following URLs match this route:

- `/files/myFile.txt`
- `/files/myFile`

Route parameters may have **default values** designated by specifying the default value after the parameter name separated by an equals sign (`=`). For example, `{controller=Home}` defines `Home` as the default value for `controller`. The default value is used if no value is present in the URL for the parameter. Route parameters are made optional by appending a question mark (`?`) to the end of the parameter name. For example, `id?`. The difference between optional values and default route parameters is:

- A route parameter with a default value always produces a value.
- An optional parameter has a value only when a value is provided by the request URL.

Route parameters may have constraints that must match the route value bound from the URL. Adding `:` and constraint name after the route parameter name specifies an inline constraint on a route parameter. If the constraint requires arguments, they're enclosed in parentheses `(...)` after the constraint name. Multiple *inline constraints* can be specified by appending another `:` and constraint name.

The constraint name and arguments are passed to the IInlineConstraintResolver service to create an instance of IRouteConstraint to use in URL processing. For example, the route template `blog/{article:minlength(10)}` specifies a `minlength` constraint with the argument `10`. For more information on route constraints and a list of the constraints provided by the framework, see the Route constraints section.

Route parameters may also have parameter transformers. Parameter transformers transform a parameter's value when generating links and matching actions and pages to URLs. Like constraints, parameter transformers can be added inline to a route parameter by adding a `:` and transformer name after the route parameter name. For example, the route template `blog/{article:slugify}` specifies a `slugify` transformer. For more information on parameter transformers, see the Parameter transformers section.

The following table demonstrates example route templates and their behavior:

Expand table

Route Template	Example Matching URI	The request URI...
`hello`	`/hello`	Only matches the single path `/hello`.
`{Page=Home}`	`/`	Matches and sets `Page` to `Home`.
`{Page=Home}`	`/Contact`	Matches and sets `Page` to `Contact`.
`{controller}/{action}/{id?}`	`/Products/List`	Maps to the `Products` controller and `List` action.
`{controller}/{action}/{id?}`	`/Products/Details/123`	Maps to the `Products` controller and `Details` action with `id` set to 123.
`{controller=Home}/{action=Index}/{id?}`	`/`	Maps to the `Home` controller and `Index` method. `id` is ignored.
`{controller=Home}/{action=Index}/{id?}`	`/Products`	Maps to the `Products` controller and `Index` method. `id` is ignored.

Using a template is generally the simplest approach to routing. Constraints and defaults can also be specified outside the route template.

Complex segments

Complex segments are processed by matching up literal delimiters from right to left in a non-greedy way. For example, `[Route("/a{b}c{d}")]` is a complex segment. Complex segments work in a particular way that must be understood to use them successfully. The example in this section demonstrates why complex segments only really work well when the delimiter text doesn't appear inside the parameter values. Using a regex and then manually extracting the values is needed for more complex cases.

> ⚠ **Warning**

When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

This is a summary of the steps that routing performs with the template `/a{b}c{d}` and the URL path `/abcd`. The `|` is used to help visualize how the algorithm works:

- The first literal, right to left, is `c`. So `/abcd` is searched from right and finds `/ab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/ab|c|d` is searched starting where we left off, then `a` is found `/|a|b|c|d`.
- The value to the right (`b`) is now matched to the route parameter `{b}`.
- There is no remaining text and no remaining route template, so this is a match.

Here's an example of a negative case using the same template `/a{b}c{d}` and the URL path `/aabcd`. The `|` is used to help visualize how the algorithm works. This case isn't a match, which is explained by the same algorithm:

- The first literal, right to left, is `c`. So `/aabcd` is searched from right and finds `/aab|c|d`.
- Everything to the right (`d`) is now matched to the route parameter `{d}`.
- The next literal, right to left, is `a`. So `/aab|c|d` is searched starting where we left off, then `a` is found `/a|a|b|c|d`.
- The value to the right (`b`) is now matched to the route parameter `{b}`.
- At this point there is remaining text `a`, but the algorithm has run out of route template to parse, so this is not a match.

Since the matching algorithm is non-greedy:

- It matches the smallest amount of text possible in each step.
- Any case where the delimiter value appears inside the parameter values results in not matching.

Regular expressions provide much more control over their matching behavior.

Greedy matching, also known as *maximal matching* attempts to find the longest possible match in the input text that satisfies the regex pattern. Non-greedy matching, also known as *lazy matching*, seeks the shortest possible match in the input text that satisfies the regex pattern.

Routing with special characters

Routing with special characters can lead to unexpected results. For example, consider a controller with the following action method:

C#

```
[HttpGet("{id?}/name")]
public async Task<ActionResult<string>> GetName(string id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null || todoItem.Name == null)
    {
        return NotFound();
    }

    return todoItem.Name;
}
```

When `string id` contains the following encoded values, unexpected results might occur:

Expand table

ASCII	Encoded
`/`	`%2F`
` `	`+`

Route parameters are not always URL decoded. This problem may be addressed in the future. For more information, see this GitHub issue ;

Route constraints

Route constraints execute when a match has occurred to the incoming URL and the URL path is tokenized into route values. Route constraints generally inspect the route value associated via the route template and make a true or false decision about whether the value is acceptable. Some route constraints use data outside the route value to consider whether the request can be routed. For example, the HttpMethodRouteConstraint can accept or reject a request based on its HTTP verb. Constraints are used in routing requests and link generation.

> ⚠ **Warning**
>
> Don't use constraints for input validation. If constraints are used for input validation, invalid input results in a `404` Not Found response. Invalid input should produce a `400` Bad Request with an appropriate error message. Route constraints are used to disambiguate similar routes, not to validate the inputs for a particular route.

The following table demonstrates example route constraints and their expected behavior:

Expand table

constraint	Example	Example Matches	Notes
`int`	`{id:int}`	`123456789`, `-123456789`	Matches any integer
`bool`	`{active:bool}`	`true`, `FALSE`	Matches `true` or `false`. Case-insensitive
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Matches a valid `DateTime` value in the invariant culture. See preceding warning.
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Matches a valid `decimal` value in the invariant culture. See preceding warning.
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Matches a valid `double` value in the invariant culture. See preceding warning.
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Matches a valid `float` value in the invariant culture. See preceding warning.
`guid`	`{id:guid}`	`CD2C1638-1638-72D5-1638-DEADBEEF1638`	Matches a valid `Guid` value

constraint	Example	Example Matches	Notes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Matches a valid `long` value
`minlength(value)`	`{username:minlength(4)}`	`Rick`	String must be at least 4 characters
`maxlength(value)`	`{filename:maxlength(8)}`	`MyFile`	String must be no more than 8 characters
`length(length)`	`{filename:length(12)}`	`somefile.txt`	String must be exactly 12 characters long
`length(min,max)`	`{filename:length(8,16)}`	`somefile.txt`	String must be at least 8 and no more than 16 characters long
`min(value)`	`{age:min(18)}`	`19`	Integer value must be at least 18
`max(value)`	`{age:max(120)}`	`91`	Integer value must be no more than 120
`range(min,max)`	`{age:range(18,120)}`	`91`	Integer value must be at least 18 but no more than 120
`alpha`	`{name:alpha}`	`Rick`	String must consist of one or more alphabetical characters, `a`-`z` and case-insensitive.
`regex(expression)`	`{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}`	`123-45-6789`	String must match the regular expression. See tips about defining a regular expression.
`required`	`{name:required}`	`Rick`	Used to enforce that a non-parameter value is present during URL generation

> **⚠ Warning**
>
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a

Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Multiple, colon delimited constraints can be applied to a single parameter. For example, the following constraint restricts a parameter to an integer value of 1 or greater:

C#

```
[Route("users/{id:int:min(1)}")]
public User GetUserById(int id) { }
```

> ⚠ **Warning**
>
> Route constraints that verify the URL and are converted to a CLR type always use the invariant culture. For example, conversion to the CLR type `int` or `DateTime`. These constraints assume that the URL is not localizable. The framework-provided route constraints don't modify the values stored in route values. All route values parsed from the URL are stored as strings. For example, the `float` constraint attempts to convert the route value to a float, but the converted value is used only to verify it can be converted to a float.

Regular expressions in constraints

> ⚠ **Warning**
>
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Regular expressions can be specified as inline constraints using the `regex(...)` route constraint. Methods in the MapControllerRoute family also accept an object literal of constraints. If that form is used, string values are interpreted as regular expressions.

The following code uses an inline regex constraint:

C#

```
app.MapGet("{message:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}$)}",
```

```
    () => "Inline Regex Constraint Matched");
```

The following code uses an object literal to specify a regex constraint:

```
C#

app.MapControllerRoute(
    name: "people",
    pattern: "people/{ssn}",
    constraints: new { ssn = "^\\d{3}-\\d{2}-\\d{4}$", },
    defaults: new { controller = "People", action = "List" });
```

The ASP.NET Core framework adds `RegexOptions.IgnoreCase | RegexOptions.Compiled | RegexOptions.CultureInvariant` to the regular expression constructor. See RegexOptions for a description of these members.

Regular expressions use delimiters and tokens similar to those used by routing and the C# language. Regular expression tokens must be escaped. To use the regular expression `^\d{3}-\d{2}-\d{4}$` in an inline constraint, use one of the following:

- Replace `\` characters provided in the string as `\\` characters in the C# source file in order to escape the `\` string escape character.
- Verbatim string literals.

To escape routing parameter delimiter characters `{`, `}`, `[`, `]`, double the characters in the expression, for example, `{{`, `}}`, `[[`, `]]`. The following table shows a regular expression and its escaped version:

Expand table

Regular expression	Escaped regular expression
`^\d{3}-\d{2}-\d{4}$`	`^\\d{{3}}-\\d{{2}}-\\d{{4}}$`
`^[a-z]{2}$`	`^[[a-z]]{{2}}$`

Regular expressions used in routing often start with the `^` character and match the starting position of the string. The expressions often end with the `$` character and match the end of the string. The `^` and `$` characters ensure that the regular expression matches the entire route parameter value. Without the `^` and `$` characters, the regular expression matches any substring within the string, which is often undesirable. The following table provides examples and explains why they match or fail to match:

Expand table

Expression	String	Match	Comment
`[a-z]{2}`	hello	Yes	Substring matches
`[a-z]{2}`	123abc456	Yes	Substring matches
`[a-z]{2}`	mz	Yes	Matches expression
`[a-z]{2}`	MZ	Yes	Not case sensitive
`^[a-z]{2}$`	hello	No	See `^` and `$` above
`^[a-z]{2}$`	123abc456	No	See `^` and `$` above

For more information on regular expression syntax, see .NET Framework Regular Expressions.

To constrain a parameter to a known set of possible values, use a regular expression. For example, `{action:regex(^(list|get|create)$)}` only matches the `action` route value to `list`, `get`, or `create`. If passed into the constraints dictionary, the string `^(list|get|create)$` is equivalent. Constraints that are passed in the constraints dictionary that don't match one of the known constraints are also treated as regular expressions. Constraints that are passed within a template that don't match one of the known constraints are not treated as regular expressions.

Custom route constraints

Custom route constraints can be created by implementing the IRouteConstraint interface. The `IRouteConstraint` interface contains Match, which returns `true` if the constraint is satisfied and `false` otherwise.

Custom route constraints are rarely needed. Before implementing a custom route constraint, consider alternatives, such as model binding.

The ASP.NET Core Constraints folder provides good examples of creating constraints. For example, GuidRouteConstraint.

To use a custom `IRouteConstraint`, the route constraint type must be registered with the app's ConstraintMap in the service container. A `ConstraintMap` is a dictionary that maps route constraint keys to `IRouteConstraint` implementations that validate those constraints. An app's `ConstraintMap` can be updated in `Program.cs` either as part of an

AddRouting call or by configuring RouteOptions directly with `builder.Services.Configure<RouteOptions>`. For example:

C#

```
builder.Services.AddRouting(options =>
    options.ConstraintMap.Add("noZeroes", typeof(NoZeroesRouteConstraint)));
```

The preceding constraint is applied in the following code:

C#

```
[ApiController]
[Route("api/[controller]")]
public class NoZeroesController : ControllerBase
{
    [HttpGet("{id:noZeroes}")]
    public IActionResult Get(string id) =>
        Content(id);
}
```

The implementation of `NoZeroesRouteConstraint` prevents `0` being used in a route parameter:

C#

```
public class NoZeroesRouteConstraint : IRouteConstraint
{
    private static readonly Regex _regex = new(
        @"^[1-9]*$",
        RegexOptions.CultureInvariant | RegexOptions.IgnoreCase,
        TimeSpan.FromMilliseconds(100));

    public bool Match(
        HttpContext? httpContext, IRouter? route, string routeKey,
        RouteValueDictionary values, RouteDirection routeDirection)
    {
        if (!values.TryGetValue(routeKey, out var routeValue))
        {
            return false;
        }

        var routeValueString = Convert.ToString(routeValue,
CultureInfo.InvariantCulture);

        if (routeValueString is null)
        {
            return false;
        }
```

```
            return _regex.IsMatch(routeValueString);
        }
    }
```

> ⚠ **Warning**
>
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

The preceding code:

- Prevents `0` in the `{id}` segment of the route.
- Is shown to provide a basic example of implementing a custom constraint. It should not be used in a production app.

The following code is a better approach to preventing an `id` containing a `0` from being processed:

C#

```
[HttpGet("{id}")]
public IActionResult Get(string id)
{
    if (id.Contains('0'))
    {
        return StatusCode(StatusCodes.Status406NotAcceptable);
    }

    return Content(id);
}
```

The preceding code has the following advantages over the `NoZeroesRouteConstraint` approach:

- It doesn't require a custom constraint.
- It returns a more descriptive error when the route parameter includes `0`.

Parameter transformers

Parameter transformers:

- Execute when generating a link using LinkGenerator.

- Implement Microsoft.AspNetCore.Routing.IOutboundParameterTransformer.
- Are configured using ConstraintMap.
- Take the parameter's route value and transform it to a new string value.
- Result in using the transformed value in the generated link.

For example, a custom `slugify` parameter transformer in route pattern `blog\{article:slugify}` with `Url.Action(new { article = "MyTestArticle" })` generates `blog\my-test-article`.

Consider the following `IOutboundParameterTransformer` implementation:

C#

```
public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string? TransformOutbound(object? value)
    {
        if (value is null)
        {
            return null;
        }

        return Regex.Replace(
            value.ToString()!,
                "([a-z])([A-Z])",
            "$1-$2",
            RegexOptions.CultureInvariant,
            TimeSpan.FromMilliseconds(100))
            .ToLowerInvariant();
    }
}
```

To use a parameter transformer in a route pattern, configure it using ConstraintMap in `Program.cs`:

C#

```
builder.Services.AddRouting(options =>
    options.ConstraintMap["slugify"] = typeof(SlugifyParameterTransformer));
```

The ASP.NET Core framework uses parameter transformers to transform the URI where an endpoint resolves. For example, parameter transformers transform the route values used to match an `area`, `controller`, `action`, and `page`:

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller:slugify=Home}/{action:slugify=Index}/{id?}");
```

With the preceding route template, the action `SubscriptionManagementController.GetAll` is matched with the URI `/subscription-management/get-all`. A parameter transformer doesn't change the route values used to generate a link. For example, `Url.Action("GetAll", "SubscriptionManagement")` outputs `/subscription-management/get-all`.

ASP.NET Core provides API conventions for using parameter transformers with generated routes:

- The Microsoft.AspNetCore.Mvc.ApplicationModels.RouteTokenTransformerConvention MVC convention applies a specified parameter transformer to all attribute routes in the app. The parameter transformer transforms attribute route tokens as they are replaced. For more information, see Use a parameter transformer to customize token replacement.
- Razor Pages uses the PageRouteTransformerConvention API convention. This convention applies a specified parameter transformer to all automatically discovered Razor Pages. The parameter transformer transforms the folder and file name segments of Razor Pages routes. For more information, see Use a parameter transformer to customize page routes.

URL generation reference

This section contains a reference for the algorithm implemented by URL generation. In practice, most complex examples of URL generation use controllers or Razor Pages. See routing in controllers for additional information.

The URL generation process begins with a call to LinkGenerator.GetPathByAddress or a similar method. The method is provided with an address, a set of route values, and optionally information about the current request from `HttpContext`.

The first step is to use the address to resolve a set of candidate endpoints using an IEndpointAddressScheme<TAddress> that matches the address's type.

Once the set of candidates is found by the address scheme, the endpoints are ordered and processed iteratively until a URL generation operation succeeds. URL generation does **not** check for ambiguities, the first result returned is the final result.

Troubleshooting URL generation with logging

The first step in troubleshooting URL generation is setting the logging level of `Microsoft.AspNetCore.Routing` to `TRACE`. `LinkGenerator` logs many details about its processing which can be useful to troubleshoot problems.

See URL generation reference for details on URL generation.

Addresses

Addresses are the concept in URL generation used to bind a call into the link generator to a set of candidate endpoints.

Addresses are an extensible concept that come with two implementations by default:

- Using *endpoint name* (`string`) as the address:
  - Provides similar functionality to MVC's route name.
  - Uses the IEndpointNameMetadata metadata type.
  - Resolves the provided string against the metadata of all registered endpoints.
  - Throws an exception on startup if multiple endpoints use the same name.
  - Recommended for general-purpose use outside of controllers and Razor Pages.
- Using *route values* (RouteValuesAddress) as the address:
  - Provides similar functionality to controllers and Razor Pages legacy URL generation.
  - Very complex to extend and debug.
  - Provides the implementation used by `IUrlHelper`, Tag Helpers, HTML Helpers, Action Results, etc.

The role of the address scheme is to make the association between the address and matching endpoints by arbitrary criteria:

- The endpoint name scheme performs a basic dictionary lookup.
- The route values scheme has a complex best subset of set algorithm.

Ambient values and explicit values

From the current request, routing accesses the route values of the current request `HttpContext.Request.RouteValues`. The values associated with the current request are referred to as the **ambient values**. For the purpose of clarity, the documentation refers to the route values passed in to methods as **explicit values**.

The following example shows ambient values and explicit values. It provides ambient values from the current request and explicit values:

C#

```
public class WidgetController : ControllerBase
{
    private readonly LinkGenerator _linkGenerator;

    public WidgetController(LinkGenerator linkGenerator) =>
        _linkGenerator = linkGenerator;

    public IActionResult Index()
    {
        var indexPath = _linkGenerator.GetPathByAction(
            HttpContext, values: new { id = 17 })!;

        return Content(indexPath);
    }

    // ...
```

The preceding code:

- Returns `/Widget/Index/17`
- Gets LinkGenerator via DI.

The following code provides only explicit values and no ambient values:

C#

```
var subscribePath = _linkGenerator.GetPathByAction(
    "Subscribe", "Home", new { id = 17 })!;
```

The preceding method returns `/Home/Subscribe/17`

The following code in the `WidgetController` returns `/Widget/Subscribe/17`:

C#

```
var subscribePath = _linkGenerator.GetPathByAction(
    HttpContext, "Subscribe", null, new { id = 17 });
```

The following code provides the controller from ambient values in the current request and explicit values:

C#

```
public class GadgetController : ControllerBase
{
    public IActionResult Index() =>
        Content(Url.Action("Edit", new { id = 17 })!);
}
```

In the preceding code:

- `/Gadget/Edit/17` is returned.
- Url gets the IUrlHelper.
- Action generates a URL with an absolute path for an action method. The URL contains the specified `action` name and `route` values.

The following code provides ambient values from the current request and explicit values:

C#

```
public class IndexModel : PageModel
{
    public void OnGet()
    {
        var editUrl = Url.Page("./Edit", new { id = 17 });

        // ...
    }
}
```

The preceding code sets `url` to `/Edit/17` when the Edit Razor Page contains the following page directive:

`@page "{id:int}"`

If the Edit page doesn't contain the `"{id:int}"` route template, `url` is `/Edit?id=17`.

The behavior of MVC's IUrlHelper adds a layer of complexity in addition to the rules described here:

- `IUrlHelper` always provides the route values from the current request as ambient values.
- IUrlHelper.Action always copies the current `action` and `controller` route values as explicit values unless overridden by the developer.
- IUrlHelper.Page always copies the current `page` route value as an explicit value unless overridden.

- `IUrlHelper.Page` always overrides the current `handler` route value with `null` as an explicit values unless overridden.

Users are often surprised by the behavioral details of ambient values, because MVC doesn't seem to follow its own rules. For historical and compatibility reasons, certain route values such as `action`, `controller`, `page`, and `handler` have their own special-case behavior.

The equivalent functionality provided by `LinkGenerator.GetPathByAction` and `LinkGenerator.GetPathByPage` duplicates these anomalies of `IUrlHelper` for compatibility.

URL generation process

Once the set of candidate endpoints are found, the URL generation algorithm:

- Processes the endpoints iteratively.
- Returns the first successful result.

The first step in this process is called **route value invalidation**. Route value invalidation is the process by which routing decides which route values from the ambient values should be used and which should be ignored. Each ambient value is considered and either combined with the explicit values, or ignored.

The best way to think about the role of ambient values is that they attempt to save application developers typing, in some common cases. Traditionally, the scenarios where ambient values are helpful are related to MVC:

- When linking to another action in the same controller, the controller name doesn't need to be specified.
- When linking to another controller in the same area, the area name doesn't need to be specified.
- When linking to the same action method, route values don't need to be specified.
- When linking to another part of the app, you don't want to carry over route values that have no meaning in that part of the app.

Calls to `LinkGenerator` or `IUrlHelper` that return `null` are usually caused by not understanding route value invalidation. Troubleshoot route value invalidation by explicitly specifying more of the route values to see if that solves the problem.

Route value invalidation works on the assumption that the app's URL scheme is hierarchical, with a hierarchy formed from left-to-right. Consider the basic controller route template `{controller}/{action}/{id?}` to get an intuitive sense of how this works

in practice. A **change** to a value **invalidates** all of the route values that appear to the right. This reflects the assumption about hierarchy. If the app has an ambient value for `id`, and the operation specifies a different value for the `controller`:

- `id` won't be reused because `{controller}` is to the left of `{id?}`.

Some examples demonstrating this principle:

- If the explicit values contain a value for `id`, the ambient value for `id` is ignored. The ambient values for `controller` and `action` can be used.
- If the explicit values contain a value for `action`, any ambient value for `action` is ignored. The ambient values for `controller` can be used. If the explicit value for `action` is different from the ambient value for `action`, the `id` value won't be used. If the explicit value for `action` is the same as the ambient value for `action`, the `id` value can be used.
- If the explicit values contain a value for `controller`, any ambient value for `controller` is ignored. If the explicit value for `controller` is different from the ambient value for `controller`, the `action` and `id` values won't be used. If the explicit value for `controller` is the same as the ambient value for `controller`, the `action` and `id` values can be used.

This process is further complicated by the existence of attribute routes and dedicated conventional routes. Controller conventional routes such as `{controller}/{action}/{id?}` specify a hierarchy using route parameters. For dedicated conventional routes and attribute routes to controllers and Razor Pages:

- There is a hierarchy of route values.
- They don't appear in the template.

For these cases, URL generation defines the **required values** concept. Endpoints created by controllers and Razor Pages have required values specified that allow route value invalidation to work.

The route value invalidation algorithm in detail:

- The required value names are combined with the route parameters, then processed from left-to-right.
- For each parameter, the ambient value and explicit value are compared:
  - If the ambient value and explicit value are the same, the process continues.
  - If the ambient value is present and the explicit value isn't, the ambient value is used when generating the URL.

    - If the ambient value isn't present and the explicit value is, reject the ambient value and all subsequent ambient values.
    - If the ambient value and the explicit value are present, and the two values are different, reject the ambient value and all subsequent ambient values.

At this point, the URL generation operation is ready to evaluate route constraints. The set of accepted values is combined with the parameter default values, which is provided to constraints. If the constraints all pass, the operation continues.

Next, the **accepted values** can be used to expand the route template. The route template is processed:

- From left-to-right.
- Each parameter has its accepted value substituted.
- With the following special cases:
    - If the accepted values is missing a value and the parameter has a default value, the default value is used.
    - If the accepted values is missing a value and the parameter is optional, processing continues.
    - If any route parameter to the right of a missing optional parameter has a value, the operation fails.
    - Contiguous default-valued parameters and optional parameters are collapsed where possible.

Values explicitly provided that don't match a segment of the route are added to the query string. The following table shows the result when using the route template `{controller}/{action}/{id?}`.

Expand table

Ambient Values	Explicit Values	Result
controller = "Home"	action = "About"	`/Home/About`
controller = "Home"	controller = "Order", action = "About"	`/Order/About`
controller = "Home", color = "Red"	action = "About"	`/Home/About`
controller = "Home"	action = "About", color = "Red"	`/Home/About?color=Red`

Optional route parameter order

Optional route parameters must come after all required route parameters and literals. In the following code, the `id` and `name` parameters must come after the `color` parameter:

C#

```
using Microsoft.AspNetCore.Mvc;

namespace WebApplication1.Controllers;

[Route("api/[controller]")]
public class MyController : ControllerBase
{
    // GET /api/my/red/2/joe
    // GET /api/my/red/2
    // GET /api/my
    [HttpGet("{color}/{id:int?}/{name?}")]
    public IActionResult GetByIdAndOptionalName(string color, int id = 1,
string? name = null)
    {
        return Ok($"{color} {id} {name ?? ""}");
    }
}
```

Problems with route value invalidation

The following code shows an example of a URL generation scheme that's not supported by routing:

C#

```
app.MapControllerRoute(
    "default",
    "{culture}/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    "blog",
    "{culture}/{**slug}",
    new { controller = "Blog", action = "ReadPost" });
```

In the preceding code, the `culture` route parameter is used for localization. The desire is to have the `culture` parameter always accepted as an ambient value. However, the `culture` parameter is not accepted as an ambient value because of the way required values work:

- In the `"default"` route template, the `culture` route parameter is to the left of `controller`, so changes to `controller` won't invalidate `culture`.

- In the `"blog"` route template, the `culture` route parameter is considered to be to the right of `controller`, which appears in the required values.

Parse URL paths with `LinkParser`

The LinkParser class adds support for parsing a URL path into a set of route values. The ParsePathByEndpointName method takes an endpoint name and a URL path, and returns a set of route values extracted from the URL path.

In the following example controller, the `GetProduct` action uses a route template of `api/Products/{id}` and has a Name of `GetProduct`:

C#

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}", Name = nameof(GetProduct))]
    public IActionResult GetProduct(string id)
    {
        // ...
```

In the same controller class, the `AddRelatedProduct` action expects a URL path, `pathToRelatedProduct`, which can be provided as a query-string parameter:

C#

```
[HttpPost("{id}/Related")]
public IActionResult AddRelatedProduct(
    string id, string pathToRelatedProduct, [FromServices] LinkParser
linkParser)
{
    var routeValues = linkParser.ParsePathByEndpointName(
        nameof(GetProduct), pathToRelatedProduct);
    var relatedProductId = routeValues?["id"];

    // ...
```

In the preceding example, the `AddRelatedProduct` action extracts the `id` route value from the URL path. For example, with a URL path of `/api/Products/1`, the `relatedProductId` value is set to `1`. This approach allows the API's clients to use URL paths when referring to resources, without requiring knowledge of how such a URL is structured.

Configure endpoint metadata

The following links provide information on how to configure endpoint metadata:

- Enable Cors with endpoint routing
- IAuthorizationPolicyProvider sample using a custom `[MinimumAgeAuthorize]` attribute
- Test authentication with the [Authorize] attribute
- RequireAuthorization
- Selecting the scheme with the [Authorize] attribute
- Apply policies using the [Authorize] attribute
- Role-based authorization in ASP.NET Core

Host matching in routes with RequireHost

RequireHost applies a constraint to the route which requires the specified host. The `RequireHost` or [Host] parameter can be a:

- Host: `www.domain.com`, matches `www.domain.com` with any port.
- Host with wildcard: `*.domain.com`, matches `www.domain.com`, `subdomain.domain.com`, or `www.subdomain.domain.com` on any port.
- Port: `*:5000`, matches port 5000 with any host.
- Host and port: `www.domain.com:5000` or `*.domain.com:5000`, matches host and port.

Multiple parameters can be specified using `RequireHost` or `[Host]`. The constraint matches hosts valid for any of the parameters. For example, `[Host("domain.com", "*.domain.com")]` matches `domain.com`, `www.domain.com`, and `subdomain.domain.com`.

The following code uses `RequireHost` to require the specified host on the route:

C#

```
app.MapGet("/", () => "Contoso").RequireHost("contoso.com");
app.MapGet("/", () => "AdventureWorks").RequireHost("adventure-works.com");

app.MapHealthChecks("/healthz").RequireHost("*:8080");
```

The following code uses the `[Host]` attribute on the controller to require any of the specified hosts:

C#

```
[Host("contoso.com", "adventure-works.com")]
public class HostsController : Controller
{
    public IActionResult Index() =>
        View();

    [Host("example.com")]
    public IActionResult Example() =>
        View();
}
```

When the `[Host]` attribute is applied to both the controller and action method:

- The attribute on the action is used.
- The controller attribute is ignored.

> ⚠ **Warning**
>
> API that relies on the Host header, such as HttpRequest.Host and RequireHost, are subject to potential spoofing by clients.
>
> To prevent host and port spoofing, use one of the following approaches:
>
> - Use HttpContext.Connection (ConnectionInfo.LocalPort) where the ports are checked.
> - Employ Host filtering.

Route groups

The MapGroup extension method helps organize groups of endpoints with a common prefix. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata which add endpoint metadata.

For example, the following code creates two similar groups of endpoints:

C#

```
app.MapGroup("/public/todos")
    .MapTodosApi()
    .WithTags("Public");

app.MapGroup("/private/todos")
    .MapTodosApi()
```

```
    .WithTags("Private")
    .AddEndpointFilterFactory(QueryPrivateTodos)
    .RequireAuthorization();


EndpointFilterDelegate QueryPrivateTodos(EndpointFilterFactoryContext
factoryContext, EndpointFilterDelegate next)
{
    var dbContextIndex = -1;

    foreach (var argument in factoryContext.MethodInfo.GetParameters())
    {
        if (argument.ParameterType == typeof(TodoDb))
        {
            dbContextIndex = argument.Position;
            break;
        }
    }

    // Skip filter if the method doesn't have a TodoDb parameter.
    if (dbContextIndex < 0)
    {
        return next;
    }

    return async invocationContext =>
    {
        var dbContext = invocationContext.GetArgument<TodoDb>
(dbContextIndex);
        dbContext.IsPrivate = true;

        try
        {
            return await next(invocationContext);
        }
        finally
        {
            // This should only be relevant if you're pooling or otherwise
reusing the DbContext instance.
            dbContext.IsPrivate = false;
        }
    };
}
```

C#

```
public static RouteGroupBuilder MapTodosApi(this RouteGroupBuilder group)
{
    group.MapGet("/", GetAllTodos);
    group.MapGet("/{id}", GetTodo);
    group.MapPost("/", CreateTodo);
    group.MapPut("/{id}", UpdateTodo);
    group.MapDelete("/{id}", DeleteTodo);
```

```
    return group;
}
```

In this scenario, you can use a relative address for the `Location` header in the `201 Created` result:

C#

```
public static async Task<Created<Todo>> CreateTodo(Todo todo, TodoDb
database)
{
    await database.AddAsync(todo);
    await database.SaveChangesAsync();

    return TypedResults.Created($"{todo.Id}", todo);
}
```

The first group of endpoints will only match requests prefixed with `/public/todos` and are accessible without any authentication. The second group of endpoints will only match requests prefixed with `/private/todos` and require authentication.

The `QueryPrivateTodos` endpoint filter factory is a local function that modifies the route handler's `TodoDb` parameters to allow to access and store private todo data.

Route groups also support nested groups and complex prefix patterns with route parameters and constraints. In the following example, and route handler mapped to the `user` group can capture the `{org}` and `{group}` route parameters defined in the outer group prefixes.

The prefix can also be empty. This can be useful for adding endpoint metadata or filters to a group of endpoints without changing the route pattern.

C#

```
var all = app.MapGroup("").WithOpenApi();
var org = all.MapGroup("{org}");
var user = org.MapGroup("{user}");
user.MapGet("", (string org, string user) => $"{org}/{user}");
```

Adding filters or metadata to a group behaves the same way as adding them individually to each endpoint before adding any extra filters or metadata that may have been added to an inner group or specific endpoint.

C#

```
var outer = app.MapGroup("/outer");
var inner = outer.MapGroup("/inner");

inner.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/inner group filter");
    return next(context);
});

outer.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/outer group filter");
    return next(context);
});

inner.MapGet("/", () => "Hi!").AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("MapGet filter");
    return next(context);
});
```

In the above example, the outer filter will log the incoming request before the inner filter even though it was added second. Because the filters were applied to different groups, the order they were added relative to each other does not matter. The order filters are added does matter if applied to the same group or specific endpoint.

A request to `/outer/inner/` will log the following:

.NET CLI

```
/outer group filter
/inner group filter
MapGet filter
```

Performance guidance for routing

When an app has performance problems, routing is often suspected as the problem. The reason routing is suspected is that frameworks like controllers and Razor Pages report the amount of time spent inside the framework in their logging messages. When there's a significant difference between the time reported by controllers and the total time of the request:

- Developers eliminate their app code as the source of the problem.
- It's common to assume routing is the cause.

Routing is performance tested using thousands of endpoints. It's unlikely that a typical app will encounter a performance problem just by being too large. The most common root cause of slow routing performance is usually a badly-behaving custom middleware.

This following code sample demonstrates a basic technique for narrowing down the source of delay:

C#

```
var logger = app.Services.GetRequiredService<ILogger<Program>>();

app.Use(async (context, next) =>
{
    var stopwatch = Stopwatch.StartNew();
    await next(context);
    stopwatch.Stop();

    logger.LogInformation("Time 1: {ElapsedMilliseconds}ms",
stopwatch.ElapsedMilliseconds);
});

app.UseRouting();

app.Use(async (context, next) =>
{
    var stopwatch = Stopwatch.StartNew();
    await next(context);
    stopwatch.Stop();

    logger.LogInformation("Time 2: {ElapsedMilliseconds}ms",
stopwatch.ElapsedMilliseconds);
});

app.UseAuthorization();

app.Use(async (context, next) =>
{
    var stopwatch = Stopwatch.StartNew();
    await next(context);
    stopwatch.Stop();

    logger.LogInformation("Time 3: {ElapsedMilliseconds}ms",
stopwatch.ElapsedMilliseconds);
});

app.MapGet("/", () => "Timing Test.");
```

To time routing:

- Interleave each middleware with a copy of the timing middleware shown in the preceding code.

- Add a unique identifier to correlate the timing data with the code.

This is a basic way to narrow down the delay when it's significant, for example, more than `10ms`. Subtracting `Time 2` from `Time 1` reports the time spent inside the `UseRouting` middleware.

The following code uses a more compact approach to the preceding timing code:

C#

```
public sealed class AutoStopwatch : IDisposable
{
    private readonly ILogger _logger;
    private readonly string _message;
    private readonly Stopwatch _stopwatch;
    private bool _disposed;

    public AutoStopwatch(ILogger logger, string message) =>
        (_logger, _message, _stopwatch) = (logger, message,
Stopwatch.StartNew());

    public void Dispose()
    {
        if (_disposed)
        {
            return;
        }

        _logger.LogInformation("{Message}: {ElapsedMilliseconds}ms",
            _message, _stopwatch.ElapsedMilliseconds);

        _disposed = true;
    }
}
```

C#

```
var logger = app.Services.GetRequiredService<ILogger<Program>>();
var timerCount = 0;

app.Use(async (context, next) =>
{
    using (new AutoStopwatch(logger, $"Time {++timerCount}"))
    {
        await next(context);
    }
});

app.UseRouting();

app.Use(async (context, next) =>
```

```
{
    using (new AutoStopwatch(logger, $"Time {++timerCount}"))
    {
        await next(context);
    }
});

app.UseAuthorization();

app.Use(async (context, next) =>
{
    using (new AutoStopwatch(logger, $"Time {++timerCount}"))
    {
        await next(context);
    }
});

app.MapGet("/", () => "Timing Test.");
```

Potentially expensive routing features

The following list provides some insight into routing features that are relatively expensive compared with basic route templates:

- Regular expressions: It's possible to write regular expressions that are complex, or have long running time with a small amount of input.
- Complex segments (`{x}-{y}-{z}`):
  - Are significantly more expensive than parsing a regular URL path segment.
  - Result in many more substrings being allocated.
- Synchronous data access: Many complex apps have database access as part of their routing. Use extensibility points such as MatcherPolicy and EndpointSelectorContext, which are asynchronous.

Guidance for large route tables

By default ASP.NET Core uses a routing algorithm that trades memory for CPU time. This has the nice effect that route matching time is dependent only on the length of the path to match and not the number of routes. However, this approach can be potentially problematic in some cases, when the app has a large number of routes (in the thousands) and there is a high amount of variable prefixes in the routes. For example, if the routes have parameters in early segments of the route, like `{parameter}/some/literal`.

It is unlikely for an app to run into a situation where this is a problem unless:

- There are a high number of routes in the app using this pattern.
- There is a large number of routes in the app.

How to determine if an app is running into the large route table problem

- There are two symptoms to look for:
  - The app is slow to start on the first request.
    - Note that this is required but not sufficient. There are many other non-route problems than can cause slow app startup. Check for the condition below to accurately determine the app is running into this situation.
  - The app consumes a lot of memory during startup and a memory dump shows a large number of `Microsoft.AspNetCore.Routing.Matching.DfaNode` instances.

How to address this issue

There are several techniques and optimizations that can be applied to routes that largely improve this scenario:

- Apply route constraints to your parameters, for example `{parameter:int}`, `{parameter:guid}`, `{parameter:regex(\\d+)}`, etc. where possible.
  - This allows the routing algorithm to internally optimize the structures used for matching and drastically reduce the memory used.
  - In the vast majority of cases this will suffice to get back to an acceptable behavior.
- Change the routes to move parameters to later segments in the template.
  - This reduces the number of possible "paths" to match an endpoint given a path.
- Use a dynamic route and perform the mapping to a controller/page dynamically.
  - This can be achieved using `MapDynamicControllerRoute` and `MapDynamicPageRoute`.

Short-circuit middleware after routing

When routing matches an endpoint, it typically lets the rest of the middleware pipeline run before invoking the endpoint logic. Services can reduce resource usage by filtering out known requests early in the pipeline. Use the ShortCircuit extension method to cause routing to invoke the endpoint logic immediately and then end the request. For example, a given route might not need to go through authentication or CORS middleware. The following example short-circuits requests that match the `/short-circuit` route:

C#

```
app.MapGet("/short-circuit", () => "Short circuiting!").ShortCircuit();
```

The ShortCircuit(IEndpointConventionBuilder, Nullable<Int32>) method can optionally take a status code.

Use the MapShortCircuit method to set up short-circuiting for multiple routes at once, by passing to it a params array of URL prefixes. For example, browsers and bots often probe servers for well known paths like `robots.txt` and `favicon.ico`. If the app doesn't have those files, one line of code can configure both routes:

C#

```
app.MapShortCircuit(404, "robots.txt", "favicon.ico");
```

`MapShortCircuit` returns IEndpointConventionBuilder so that additional route constraints like host filtering can be added to it.

The `ShortCircuit` and `MapShortCircuit` methods do not affect middleware placed before `UseRouting`. Trying to use these methods with endpoints that also have `[Authorize]` or `[RequireCors]` metadata will cause requests to fail with an `InvalidOperationException`. This metadata is applied by [Authorize] or [EnableCors] attributes or by RequireCors or RequireAuthorization methods.

To see the effect of short-circuiting middleware, set the "Microsoft" logging category to "Information" in `appsettings.Development.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Run the following code:

C#

```
var app = WebApplication.Create();

app.UseHttpLogging();

app.MapGet("/", () => "No short-circuiting!");
app.MapGet("/short-circuit", () => "Short circuiting!").ShortCircuit();
app.MapShortCircuit(404, "robots.txt", "favicon.ico");

app.Run();
```

The following example is from the console logs produced by running the `/` endpoint. It includes output from the logging middleware:

```
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'HTTP: GET /'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'HTTP: GET /'
info: Microsoft.AspNetCore.HttpLogging.HttpLoggingMiddleware[2]
      Response:
      StatusCode: 200
      Content-Type: text/plain; charset=utf-8
      Date: Wed, 03 May 2023 21:05:59 GMT
      Server: Kestrel
      Alt-Svc: h3=":5182"; ma=86400
      Transfer-Encoding: chunked
```

The following example is from running the `/short-circuit` endpoint. It doesn't have anything from the logging middleware because the middleware was short-circuited:

```
info: Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware[4]
      The endpoint 'HTTP: GET /short-circuit' is being executed without
running additional middleware.
info: Microsoft.AspNetCore.Routing.EndpointRoutingMiddleware[5]
      The endpoint 'HTTP: GET /short-circuit' has been executed without
running additional middleware.
```

Guidance for library authors

This section contains guidance for library authors building on top of routing. These details are intended to ensure that app developers have a good experience using libraries and frameworks that extend routing.

Define endpoints

To create a framework that uses routing for URL matching, start by defining a user experience that builds on top of UseEndpoints.

DO build on top of IEndpointRouteBuilder. This allows users to compose your framework with other ASP.NET Core features without confusion. Every ASP.NET Core template includes routing. Assume routing is present and familiar for users.

```
C#

// Your framework
app.MapMyFramework(...);

app.MapHealthChecks("/healthz");
```

DO return a sealed concrete type from a call to `MapMyFramework(...)` that implements IEndpointConventionBuilder. Most framework `Map...` methods follow this pattern. The `IEndpointConventionBuilder` interface:

- Allows for metadata to be composed.
- Is targeted by a variety of extension methods.

Declaring your own type allows you to add your own framework-specific functionality to the builder. It's ok to wrap a framework-declared builder and forward calls to it.

```
C#

// Your framework
app.MapMyFramework(...)
    .RequireAuthorization()
    .WithMyFrameworkFeature(awesome: true);

app.MapHealthChecks("/healthz");
```

CONSIDER writing your own EndpointDataSource. `EndpointDataSource` is the low-level primitive for declaring and updating a collection of endpoints. `EndpointDataSource` is a powerful API used by controllers and Razor Pages. For more information, see Dynamic endpoint routing.

The routing tests have a basic example of a non-updating data source.

CONSIDER implementing GetGroupedEndpoints. This gives complete control over running group conventions and the final metadata on the grouped endpoints. For

example, this allows custom `EndpointDataSource` implementations to run endpoint filters added to groups.

DO NOT attempt to register an `EndpointDataSource` by default. Require users to register your framework in UseEndpoints. The philosophy of routing is that nothing is included by default, and that `UseEndpoints` is the place to register endpoints.

Creating routing-integrated middleware

CONSIDER defining metadata types as an interface.

DO make it possible to use metadata types as an attribute on classes and methods.

C#

```
public interface ICoolMetadata
{
    bool IsCool { get; }
}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class CoolMetadataAttribute : Attribute, ICoolMetadata
{
    public bool IsCool => true;
}
```

Frameworks like controllers and Razor Pages support applying metadata attributes to types and methods. If you declare metadata types:

- Make them accessible as attributes.
- Most users are familiar with applying attributes.

Declaring a metadata type as an interface adds another layer of flexibility:

- Interfaces are composable.
- Developers can declare their own types that combine multiple policies.

DO make it possible to override metadata, as shown in the following example:

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class SuppressCoolMetadataAttribute : Attribute, ICoolMetadata
{
    public bool IsCool => false;
}
```

```
[CoolMetadata]
public class MyController : Controller
{
    public void MyCool() { }

    [SuppressCoolMetadata]
    public void Uncool() { }
}
```

The best way to follow these guidelines is to avoid defining **marker metadata**:

- Don't just look for the presence of a metadata type.
- Define a property on the metadata and check the property.

The metadata collection is ordered and supports overriding by priority. In the case of controllers, metadata on the action method is most specific.

DO make middleware useful with and without routing:

C#

```
app.UseAuthorization(new AuthorizationPolicy() { ... });

// Your framework
app.MapMyFramework(...).RequireAuthorization();
```

As an example of this guideline, consider the `UseAuthorization` middleware. The authorization middleware allows you to pass in a fallback policy. The fallback policy, if specified, applies to both:

- Endpoints without a specified policy.
- Requests that don't match an endpoint.

This makes the authorization middleware useful outside of the context of routing. The authorization middleware can be used for traditional middleware programming.

Debug diagnostics

For detailed routing diagnostic output, set `Logging:LogLevel:Microsoft` to `Debug`. In the development environment, set the log level in `appsettings.Development.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
```

```
      "Default": "Information",
      "Microsoft": "Debug",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Additional resources

- View or download sample code (how to download)

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Handle errors in ASP.NET Core

Article • 09/21/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Tom Dykstra

This article covers common approaches to handling errors in ASP.NET Core web apps. See also Handle errors in ASP.NET Core controller-based web APIs and Handle errors in minimal APIs.

Developer exception page

The *Developer Exception Page* displays detailed information about unhandled request exceptions. ASP.NET Core apps enable the developer exception page by default when both:

- Running in the Development environment.
- App created with the current templates, that is, using WebApplication.CreateBuilder. Apps created using the WebHost.CreateDefaultBuilder must enable the developer exception page by calling `app.UseDeveloperExceptionPage` in `Configure`.

The developer exception page runs early in the middleware pipeline, so that it can catch unhandled exceptions thrown in middleware that follows.

Detailed exception information shouldn't be displayed publicly when the app runs in the Production environment. For more information on configuring environments, see Use multiple environments in ASP.NET Core.

The Developer Exception Page can include the following information about the exception and the request:

- Stack trace
- Query string parameters, if any
- Cookies, if any
- Headers

The Developer Exception Page isn't guaranteed to provide any information. Use Logging for complete error information.

Exception handler page

To configure a custom error handling page for the Production environment, call UseExceptionHandler. This exception handling middleware:

- Catches and logs unhandled exceptions.
- Re-executes the request in an alternate pipeline using the path indicated. The request isn't re-executed if the response has started. The template-generated code re-executes the request using the `/Error` path.

> ⚠ **Warning**
>
> If the alternate pipeline throws an exception of its own, Exception Handling Middleware rethrows the original exception.

Since this middleware can re-execute the request pipeline:

- Middlewares need to handle reentrancy with the same request. This normally means either cleaning up their state after calling `_next` or caching their processing on the `HttpContext` to avoid redoing it. When dealing with the request body, this either means buffering or caching the results like the Form reader.
- For the UseExceptionHandler(IApplicationBuilder, String) overload that is used in templates, only the request path is modified, and the route data is cleared. Request data such as headers, method, and items are all reused as-is.
- Scoped services remain the same.

In the following example, UseExceptionHandler adds the exception handling middleware in non-Development environments:

C#

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

The Razor Pages app template provides an Error page (`.cshtml`) and PageModel class (`ErrorModel`) in the *Pages* folder. For an MVC app, the project template includes an `Error` action method and an Error view for the Home controller.

The exception handling middleware re-executes the request using the *original* HTTP method. If an error handler endpoint is restricted to a specific set of HTTP methods, it runs only for those HTTP methods. For example, an MVC controller action that uses the `[HttpGet]` attribute runs only for GET requests. To ensure that *all* requests reach the custom error handling page, don't restrict them to a specific set of HTTP methods.

To handle exceptions differently based on the original HTTP method:

- For Razor Pages, create multiple handler methods. For example, use `OnGet` to handle GET exceptions and use `OnPost` to handle POST exceptions.
- For MVC, apply HTTP verb attributes to multiple actions. For example, use `[HttpGet]` to handle GET exceptions and use `[HttpPost]` to handle POST exceptions.

To allow unauthenticated users to view the custom error handling page, ensure that it supports anonymous access.

Access the exception

Use IExceptionHandlerPathFeature to access the exception and the original request path in an error handler. The following example uses `IExceptionHandlerPathFeature` to get more information about the exception that was thrown:

C#

```
[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore
= true)]
[IgnoreAntiforgeryToken]
public class ErrorModel : PageModel
{
    public string? RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public string? ExceptionMessage { get; set; }

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
```

```
        var exceptionHandlerPathFeature =
            HttpContext.Features.Get<IExceptionHandlerPathFeature>();

        if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
        {
            ExceptionMessage = "The file was not found.";
        }

        if (exceptionHandlerPathFeature?.Path == "/")
        {
            ExceptionMessage ??= string.Empty;
            ExceptionMessage += " Page: Home.";
        }
    }
}
```

> ⚠ **Warning**
>
> Do **not** serve sensitive error information to clients. Serving errors is a security risk.

Exception handler lambda

An alternative to a custom exception handler page is to provide a lambda to UseExceptionHandler. Using a lambda allows access to the error before returning the response.

The following code uses a lambda for exception handling:

C#

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
```

```
    app.UseExceptionHandler(exceptionHandlerApp =>
    {
        exceptionHandlerApp.Run(async context =>
        {
            context.Response.StatusCode =
StatusCodes.Status500InternalServerError;

            // using static System.Net.Mime.MediaTypeNames;
            context.Response.ContentType = Text.Plain;

            await context.Response.WriteAsync("An exception was thrown.");

            var exceptionHandlerPathFeature =
                context.Features.Get<IExceptionHandlerPathFeature>();

            if (exceptionHandlerPathFeature?.Error is FileNotFoundException)
            {
                await context.Response.WriteAsync(" The file was not
found.");
            }

            if (exceptionHandlerPathFeature?.Path == "/")
            {
                await context.Response.WriteAsync(" Page: Home.");
            }
        });
    });

    app.UseHsts();
}
```

> ⚠ **Warning**
>
> Do **not** serve sensitive error information to clients. Serving errors is a security risk.

IExceptionHandler

IExceptionHandler is an interface that gives the developer a callback for handling known exceptions in a central location.

`IExceptionHandler` implementations are registered by calling IServiceCollection.AddExceptionHandler<T> [`IServiceCollection.AddExceptionHandler<T>`]. The lifetime of an `IExceptionHandler` instance is singleton. Multiple implementations can be added, and they're called in the order registered.

If an exception handler handles a request, it can return `true` to stop processing. If an exception isn't handled by any exception handler, then control falls back to the default behavior and options from the middleware. Different metrics and logs are emitted for handled versus unhandled exceptions.

The following example shows an `IExceptionHandler` implementation:

C#

```
using Microsoft.AspNetCore.Diagnostics;

namespace ErrorHandlingSample
{
    public class CustomExceptionHandler : IExceptionHandler
    {
        private readonly ILogger<CustomExceptionHandler> logger;
        public CustomExceptionHandler(ILogger<CustomExceptionHandler>
logger)
        {
            this.logger = logger;
        }
        public ValueTask<bool> TryHandleAsync(
            HttpContext httpContext,
            Exception exception,
            CancellationToken cancellationToken)
        {
            var exceptionMessage = exception.Message;
            logger.LogError(
                "Error Message: {exceptionMessage}, Time of occurrence
{time}",
                exceptionMessage, DateTime.UtcNow);
            // Return false to continue with the default behavior
            // - or - return true to signal that this exception is handled
            return ValueTask.FromResult(false);
        }
    }
}
```

The following example shows how to register an `IExceptionHandler` implementation for dependency injection:

C#

```
using ErrorHandlingSample;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddRazorPages();
builder.Services.AddExceptionHandler<CustomExceptionHandler>();
```

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

// Remaining Program.cs code omitted for brevity
```

When the preceding code runs in the Development environment:

- The `CustomExceptionHandler` is called first to handle an exception.
- After logging the exception, the `TryHandleException` method returns `false`, so the developer exception page is shown.

In other environments:

- The `CustomExceptionHandler` is called first to handle an exception.
- After logging the exception, the `TryHandleException` method returns `false`, so the /Error page is shown.

UseStatusCodePages

By default, an ASP.NET Core app doesn't provide a status code page for HTTP error status codes, such as *404 - Not Found*. When the app sets an HTTP 400-599 error status code that doesn't have a body, it returns the status code and an empty response body. To enable default text-only handlers for common error status codes, call UseStatusCodePages in `Program.cs`:

C#

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseStatusCodePages();
```

Call `UseStatusCodePages` before request handling middleware. For example, call `UseStatusCodePages` before the Static File Middleware and the Endpoints Middleware.

When `UseStatusCodePages` isn't used, navigating to a URL without an endpoint returns a browser-dependent error message indicating the endpoint can't be found. When `UseStatusCodePages` is called, the browser returns the following response:

```
Console

Status Code: 404; Not Found
```

`UseStatusCodePages` isn't typically used in production because it returns a message that isn't useful to users.

> **Note**
>
> The status code pages middleware does **not** catch exceptions. To provide a custom error handling page, use the exception handler page.

UseStatusCodePages with format string

To customize the response content type and text, use the overload of UseStatusCodePages that takes a content type and format string:

```
C#

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

// using static System.Net.Mime.MediaTypeNames;
app.UseStatusCodePages(Text.Plain, "Status Code Page: {0}");
```

In the preceding code, `{0}` is a placeholder for the error code.

`UseStatusCodePages` with a format string isn't typically used in production because it returns a message that isn't useful to users.

UseStatusCodePages with lambda

To specify custom error-handling and response-writing code, use the overload of UseStatusCodePages that takes a lambda expression:

C#

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseStatusCodePages(async statusCodeContext =>
{
    // using static System.Net.Mime.MediaTypeNames;
    statusCodeContext.HttpContext.Response.ContentType = Text.Plain;

    await statusCodeContext.HttpContext.Response.WriteAsync(
        $"Status Code Page:
{statusCodeContext.HttpContext.Response.StatusCode}");
});
```

`UseStatusCodePages` with a lambda isn't typically used in production because it returns a message that isn't useful to users.

UseStatusCodePagesWithRedirects

The UseStatusCodePagesWithRedirects extension method:

- Sends a 302 - Found status code to the client.
- Redirects the client to the error handling endpoint provided in the URL template. The error handling endpoint typically displays error information and returns HTTP 200.

C#

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseStatusCodePagesWithRedirects("/StatusCode/{0}");
```

The URL template can include a `{0}` placeholder for the status code, as shown in the preceding code. If the URL template starts with `~` (tilde), the `~` is replaced by the app's

`PathBase`. When specifying an endpoint in the app, create an MVC view or Razor page for the endpoint.

This method is commonly used when the app:

- Should redirect the client to a different endpoint, usually in cases where a different app processes the error. For web apps, the client's browser address bar reflects the redirected endpoint.
- Shouldn't preserve and return the original status code with the initial redirect response.

UseStatusCodePagesWithReExecute

The UseStatusCodePagesWithReExecute extension method:

- Generates the response body by re-executing the request pipeline using an alternate path.
- Does not alter the status code before or after re-executing the pipeline.

The new pipeline execution may alter the response's status code, as the new pipeline has full control of the status code. If the new pipeline does not alter the status code, the original status code will be sent to the client.

C#

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseStatusCodePagesWithReExecute("/StatusCode/{0}");
```

If an endpoint within the app is specified, create an MVC view or Razor page for the endpoint.

This method is commonly used when the app should:

- Process the request without redirecting to a different endpoint. For web apps, the client's browser address bar reflects the originally requested endpoint.
- Preserve and return the original status code with the response.

The URL template must start with `/` and may include a placeholder `{0}` for the status code. To pass the status code as a query-string parameter, pass a second argument into `UseStatusCodePagesWithReExecute`. For example:

C#

```
var app = builder.Build();
app.UseStatusCodePagesWithReExecute("/StatusCode", "?statusCode={0}");
```

The endpoint that processes the error can get the original URL that generated the error, as shown in the following example:

C#

```
[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore
= true)]
public class StatusCodeModel : PageModel
{
    public int OriginalStatusCode { get; set; }

    public string? OriginalPathAndQuery { get; set; }

    public void OnGet(int statusCode)
    {
        OriginalStatusCode = statusCode;

        var statusCodeReExecuteFeature =
            HttpContext.Features.Get<IStatusCodeReExecuteFeature>();

        if (statusCodeReExecuteFeature is not null)
        {
            OriginalPathAndQuery = $"
{statusCodeReExecuteFeature.OriginalPathBase}"
                                   + $"
{statusCodeReExecuteFeature.OriginalPath}"
                                   + $"
{statusCodeReExecuteFeature.OriginalQueryString}";

        }
    }
}
```

Since this middleware can re-execute the request pipeline:

- Middlewares need to handle reentrancy with the same request. This normally means either cleaning up their state after calling `_next` or caching their processing on the `HttpContext` to avoid redoing it. When dealing with the request body, this either means buffering or caching the results like the Form reader.

- Scoped services remain the same.

Disable status code pages

To disable status code pages for an MVC controller or action method, use the [SkipStatusCodePages] attribute.

To disable status code pages for specific requests in a Razor Pages handler method or in an MVC controller, use IStatusCodePagesFeature:

```
C#

public void OnGet()
{
    var statusCodePagesFeature =
        HttpContext.Features.Get<IStatusCodePagesFeature>();

    if (statusCodePagesFeature is not null)
    {
        statusCodePagesFeature.Enabled = false;
    }
}
```

Exception-handling code

Code in exception handling pages can also throw exceptions. Production error pages should be tested thoroughly and take extra care to avoid throwing exceptions of their own.

Response headers

Once the headers for a response are sent:

- The app can't change the response's status code.
- Any exception pages or handlers can't run. The response must be completed or the connection aborted.

Server exception handling

In addition to the exception handling logic in an app, the HTTP server implementation can handle some exceptions. If the server catches an exception before response headers are sent, the server sends a `500 - Internal Server Error` response without a response

body. If the server catches an exception after response headers are sent, the server closes the connection. Requests that aren't handled by the app are handled by the server. Any exception that occurs when the server is handling the request is handled by the server's exception handling. The app's custom error pages, exception handling middleware, and filters don't affect this behavior.

Startup exception handling

Only the hosting layer can handle exceptions that take place during app startup. The host can be configured to capture startup errors and capture detailed errors.

The hosting layer can show an error page for a captured startup error only if the error occurs after host address/port binding. If binding fails:

- The hosting layer logs a critical exception.
- The dotnet process crashes.
- No error page is displayed when the HTTP server is Kestrel.

When running on IIS (or Azure App Service) or IIS Express, a *502.5 - Process Failure* is returned by the ASP.NET Core Module if the process can't start. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Database error page

The Database developer page exception filter AddDatabaseDeveloperPageExceptionFilter captures database-related exceptions that can be resolved by using Entity Framework Core migrations. When these exceptions occur, an HTML response is generated with details of possible actions to resolve the issue. This page is enabled only in the Development environment. The following code adds the Database developer page exception filter:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddRazorPages();
```

Exception filters

In MVC apps, exception filters can be configured globally or on a per-controller or per-action basis. In Razor Pages apps, they can be configured globally or per page model. These filters handle any unhandled exceptions that occur during the execution of a controller action or another filter. For more information, see Filters in ASP.NET Core.

Exception filters are useful for trapping exceptions that occur within MVC actions, but they're not as flexible as the built-in exception handling middleware, UseExceptionHandler. We recommend using `UseExceptionHandler`, unless you need to perform error handling differently based on which MVC action is chosen.

Model state errors

For information about how to handle model state errors, see Model binding and Model validation.

Problem details

Problem Details are not the only response format to describe an HTTP API error, however, they are commonly used to report errors for HTTP APIs.

The problem details service implements the IProblemDetailsService interface, which supports creating problem details in ASP.NET Core. The AddProblemDetails(IServiceCollection) extension method on IServiceCollection registers the default `IProblemDetailsService` implementation.

In ASP.NET Core apps, the following middleware generates problem details HTTP responses when `AddProblemDetails` is called, except when the Accept request HTTP header doesn't include one of the content types supported by the registered IProblemDetailsWriter (default: `application/json`):

- ExceptionHandlerMiddleware: Generates a problem details response when a custom handler is not defined.
- StatusCodePagesMiddleware: Generates a problem details response by default.
- DeveloperExceptionPageMiddleware: Generates a problem details response in development when the `Accept` request HTTP header doesn't include `text/html`.

The following code configures the app to generate a problem details response for all HTTP client and server error responses that ***don't have a body content yet***:

```
C#
```

```
builder.Services.AddProblemDetails();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler();
    app.UseHsts();
}

app.UseStatusCodePages();
```

The next section shows how to customize the problem details response body.

Customize problem details

The automatic creation of a `ProblemDetails` can be customized using any of the following options:

1. Use ProblemDetailsOptions.CustomizeProblemDetails
2. Use a custom IProblemDetailsWriter
3. Call the IProblemDetailsService in a middleware

`CustomizeProblemDetails` operation

The generated problem details can be customized using CustomizeProblemDetails, and the customizations are applied to all auto-generated problem details.

The following code uses ProblemDetailsOptions to set CustomizeProblemDetails:

C#

```
builder.Services.AddProblemDetails(options =>
    options.CustomizeProblemDetails = ctx =>
            ctx.ProblemDetails.Extensions.Add("nodeId",
Environment.MachineName));

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler();
    app.UseHsts();
}

app.UseStatusCodePages();
```

For example, an HTTP Status 400 Bad Request endpoint result produces the following problem details response body:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "Bad Request",
  "status": 400,
  "nodeId": "my-machine-name"
}
```

Custom `IProblemDetailsWriter`

An IProblemDetailsWriter implementation can be created for advanced customizations.

C#

```
public class SampleProblemDetailsWriter : IProblemDetailsWriter
{
    // Indicates that only responses with StatusCode == 400
    // are handled by this writer. All others are
    // handled by different registered writers if available.
    public bool CanWrite(ProblemDetailsContext context)
        => context.HttpContext.Response.StatusCode == 400;

    public ValueTask WriteAsync(ProblemDetailsContext context)
    {
        // Additional customizations.

        // Write to the response.
        var response = context.HttpContext.Response;
        return new
ValueTask(response.WriteAsJsonAsync(context.ProblemDetails));
    }
}
```

Note: When using a custom `IProblemDetailsWriter`, the custom `IProblemDetailsWriter` must be registered before calling AddRazorPages, AddControllers, AddControllersWithViews, or AddMvc:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddTransient<IProblemDetailsWriter,
SampleProblemDetailsWriter>();
```

```
var app = builder.Build();

// Middleware to handle writing problem details to the response.
app.Use(async (context, next) =>
{
    await next(context);
    var mathErrorFeature = context.Features.Get<MathErrorFeature>();
    if (mathErrorFeature is not null)
    {
        if (context.RequestServices.GetService<IProblemDetailsWriter>() is
            { } problemDetailsService)
        {

            if (problemDetailsService.CanWrite(new ProblemDetailsContext() {
HttpContext = context }))
            {
                (string Detail, string Type) details =
mathErrorFeature.MathError switch
                {
                    MathErrorType.DivisionByZeroError => ("Divison by zero
is not defined.",
                        "https://en.wikipedia.org/wiki/Division_by_zero"),
                    _ => ("Negative or complex numbers are not valid
input.",
                        "https://en.wikipedia.org/wiki/Square_root")
                };

                await problemDetailsService.WriteAsync(new
ProblemDetailsContext
                {
                    HttpContext = context,
                    ProblemDetails =
                    {
                        Title = "Bad Input",
                        Detail = details.Detail,
                        Type = details.Type
                    }
                });
            }
        }
    }
});

// /divide?numerator=2&denominator=4
app.MapGet("/divide", (HttpContext context, double numerator, double
denominator) =>
{
    if (denominator == 0)
    {
        var errorType = new MathErrorFeature
        {
            MathError = MathErrorType.DivisionByZeroError
        };
        context.Features.Set(errorType);
        return Results.BadRequest();
```

```
    }

    return Results.Ok(numerator / denominator);
});

// /squareroot?radicand=16
app.MapGet("/squareroot", (HttpContext context, double radicand) =>
{
    if (radicand < 0)
    {
        var errorType = new MathErrorFeature
        {
            MathError = MathErrorType.NegativeRadicandError
        };
        context.Features.Set(errorType);
        return Results.BadRequest();
    }

    return Results.Ok(Math.Sqrt(radicand));
});

app.Run();
```

Problem details from Middleware

An alternative approach to using ProblemDetailsOptions with CustomizeProblemDetails is to set the ProblemDetails in middleware. A problem details response can be written by calling IProblemDetailsService.WriteAsync:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStatusCodePages();

// Middleware to handle writing problem details to the response.
app.Use(async (context, next) =>
{
    await next(context);
    var mathErrorFeature = context.Features.Get<MathErrorFeature>();
    if (mathErrorFeature is not null)
    {
```

```
        if (context.RequestServices.GetService<IProblemDetailsService>() is
                                                                { }
problemDetailsService)
        {
            (string Detail, string Type) details =
mathErrorFeature.MathError switch
            {
                MathErrorType.DivisionByZeroError => ("Divison by zero is
not defined.",
                "https://en.wikipedia.org/wiki/Division_by_zero"),
                _ => ("Negative or complex numbers are not valid input.",
                "https://en.wikipedia.org/wiki/Square_root")
            };

            await problemDetailsService.WriteAsync(new ProblemDetailsContext
            {
                HttpContext = context,
                ProblemDetails =
                {
                    Title = "Bad Input",
                    Detail = details.Detail,
                    Type = details.Type
                }
            });
        }
    }
});

// /divide?numerator=2&denominator=4
app.MapGet("/divide", (HttpContext context, double numerator, double
denominator) =>
{
    if (denominator == 0)
    {
        var errorType = new MathErrorFeature { MathError =

MathErrorType.DivisionByZeroError };
        context.Features.Set(errorType);
        return Results.BadRequest();
    }

    return Results.Ok(numerator / denominator);
});

// /squareroot?radicand=16
app.MapGet("/squareroot", (HttpContext context, double radicand) =>
{
    if (radicand < 0)
    {
        var errorType = new MathErrorFeature { MathError =

MathErrorType.NegativeRadicandError };
        context.Features.Set(errorType);
        return Results.BadRequest();
    }
```

```
    return Results.Ok(Math.Sqrt(radicand));
});

app.MapControllers();

app.Run();
```

In the preceding code, the minimal API endpoints `/divide` and `/squareroot` return the expected custom problem response on error input.

The API controller endpoints return the default problem response on error input, not the custom problem response. The default problem response is returned because the API controller has written to the response stream, Problem details for error status codes, before IProblemDetailsService.WriteAsync is called and the response is **not** written again.

The following `ValuesController` returns BadRequestResult, which writes to the response stream and therefore prevents the custom problem response from being returned.

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // /api/values/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.DivisionByZeroError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values/squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
```

```
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.NegativeRadicandError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }
        return Ok(Math.Sqrt(radicand));
    }

}
```

The following `Values3Controller` returns ControllerBase.Problem so the expected custom problem result is returned:

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class Values3Controller : ControllerBase
{
    // /api/values3/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.DivisionByZeroError
            };
            HttpContext.Features.Set(errorType);
            return Problem(
                title: "Bad Input",
                detail: "Divison by zero is not defined.",
                type: "https://en.wikipedia.org/wiki/Division_by_zero",
                statusCode: StatusCodes.Status400BadRequest
                );
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values3/squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.NegativeRadicandError
```

```
            };
            HttpContext.Features.Set(errorType);
            return Problem(
                title: "Bad Input",
                detail: "Negative or complex numbers are not valid input.",
                type: "https://en.wikipedia.org/wiki/Square_root",
                statusCode: StatusCodes.Status400BadRequest
                );
        }

        return Ok(Math.Sqrt(radicand));
    }

}
```

Produce a ProblemDetails payload for exceptions

Consider the following app:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.MapControllers();
app.Run();
```

In non-development environments, when an exception occurs, the following is a standard ProblemDetails response that is returned to the client:

JSON

```
{
"type":"https://tools.ietf.org/html/rfc7231#section-6.6.1",
"title":"An error occurred while processing your request.",
```

```
"status":500,"traceId":"00-b644<snip>-00"
}
```

For most apps, the preceding code is all that's needed for exceptions. However, the following section shows how to get more detailed problem responses.

An alternative to a custom exception handler page is to provide a lambda to UseExceptionHandler. Using a lambda allows access to the error and writing a problem details response with IProblemDetailsService.WriteAsync:

C#

```
using Microsoft.AspNetCore.Diagnostics;
using static System.Net.Mime.MediaTypeNames;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler(exceptionHandlerApp =>
    {
        exceptionHandlerApp.Run(async context =>
        {
            context.Response.StatusCode =
StatusCodes.Status500InternalServerError;
            context.Response.ContentType = Text.Plain;

            var title = "Bad Input";
            var detail = "Invalid input";
            var type = "https://errors.example.com/badInput";

            if (context.RequestServices.GetService<IProblemDetailsService>()
is
                { } problemDetailsService)
            {
                var exceptionHandlerFeature =
               context.Features.Get<IExceptionHandlerFeature>();

                var exceptionType = exceptionHandlerFeature?.Error;
                if (exceptionType != null &&
```

```
                    exceptionType.Message.Contains("infinity"))
                {
                    title = "Argument exception";
                    detail = "Invalid input";
                    type = "https://errors.example.com/argumentException";
                }

                await problemDetailsService.WriteAsync(new
ProblemDetailsContext
                {
                    HttpContext = context,
                    ProblemDetails =
                {
                    Title = title,
                    Detail = detail,
                    Type = type
                }
                });
            }
        });
    });
}

app.MapControllers();
app.Run();
```

> ⚠ **Warning**
>
> Do **not** serve sensitive error information to clients. Serving errors is a security risk.

An alternative approach to generate problem details is to use the third-party NuGet package Hellang.Middleware.ProblemDetails that can be used to map exceptions and client errors to problem details.

Additional resources

- View or download sample code (how to download)
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core controller-based web APIs
- Handle errors in minimal APIs.

Make HTTP requests using IHttpClientFactory in ASP.NET Core

Article • 04/11/2023

By Kirk Larkin, Steve Gordon, Glenn Condron, and Ryan Nowak.

An IHttpClientFactory can be registered and used to configure and create HttpClient instances in an app. `IHttpClientFactory` offers the following benefits:

- Provides a central location for naming and configuring logical `HttpClient` instances. For example, a client named *github* could be registered and configured to access GitHub. A default client can be registered for general access.
- Codifies the concept of outgoing middleware via delegating handlers in `HttpClient`. Provides extensions for Polly-based middleware to take advantage of delegating handlers in `HttpClient`.
- Manages the pooling and lifetime of underlying `HttpClientMessageHandler` instances. Automatic management avoids common DNS (Domain Name System) problems that occur when manually managing `HttpClient` lifetimes.
- Adds a configurable logging experience (via `ILogger`) for all requests sent through clients created by the factory.

The sample code in this topic version uses System.Text.Json to deserialize JSON content returned in HTTP responses. For samples that use `Json.NET` and `ReadAsAsync<T>`, use the version selector to select a 2.x version of this topic.

Consumption patterns

There are several ways `IHttpClientFactory` can be used in an app:

- Basic usage
- Named clients
- Typed clients
- Generated clients

The best approach depends upon the app's requirements.

Basic usage

Register `IHttpClientFactory` by calling `AddHttpClient` in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddHttpClient();
```

An `IHttpClientFactory` can be requested using dependency injection (DI). The following code uses `IHttpClientFactory` to create an `HttpClient` instance:

C#

```
public class BasicModel : PageModel
{
    private readonly IHttpClientFactory _httpClientFactory;

    public BasicModel(IHttpClientFactory httpClientFactory) =>
        _httpClientFactory = httpClientFactory;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        var httpRequestMessage = new HttpRequestMessage(
            HttpMethod.Get,
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches")
        {
            Headers =
            {
                { HeaderNames.Accept, "application/vnd.github.v3+json" },
                { HeaderNames.UserAgent, "HttpRequestsSample" }
            }
        };

        var httpClient = _httpClientFactory.CreateClient();
        var httpResponseMessage = await
httpClient.SendAsync(httpRequestMessage);

        if (httpResponseMessage.IsSuccessStatusCode)
        {
            using var contentStream =
                await httpResponseMessage.Content.ReadAsStreamAsync();

            GitHubBranches = await JsonSerializer.DeserializeAsync
                <IEnumerable<GitHubBranch>>(contentStream);
        }
    }
}
```

Using `IHttpClientFactory` like in the preceding example is a good way to refactor an existing app. It has no impact on how `HttpClient` is used. In places where `HttpClient`

instances are created in an existing app, replace those occurrences with calls to CreateClient.

Named clients

Named clients are a good choice when:

- The app requires many distinct uses of `HttpClient`.
- Many `HttpClient`s have different configuration.

Specify configuration for a named `HttpClient` during its registration in `Program.cs`:

C#

```
builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // using Microsoft.Net.Http.Headers;
    // The GitHub API requires two headers.
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.Accept, "application/vnd.github.v3+json");
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.UserAgent, "HttpRequestsSample");
});
```

In the preceding code the client is configured with:

- The base address `https://api.github.com/`.
- Two headers required to work with the GitHub API.

CreateClient

Each time CreateClient is called:

- A new instance of `HttpClient` is created.
- The configuration action is called.

To create a named client, pass its name into `CreateClient`:

C#

```
public class NamedClientModel : PageModel
{
    private readonly IHttpClientFactory _httpClientFactory;

    public NamedClientModel(IHttpClientFactory httpClientFactory) =>
```

```
        _httpClientFactory = httpClientFactory;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        var httpClient = _httpClientFactory.CreateClient("GitHub");
        var httpResponseMessage = await httpClient.GetAsync(
            "repos/dotnet/AspNetCore.Docs/branches");

        if (httpResponseMessage.IsSuccessStatusCode)
        {
            using var contentStream =
                await httpResponseMessage.Content.ReadAsStreamAsync();

            GitHubBranches = await JsonSerializer.DeserializeAsync
                <IEnumerable<GitHubBranch>>(contentStream);
        }
    }
}
```

In the preceding code, the request doesn't need to specify a hostname. The code can pass just the path, since the base address configured for the client is used.

Typed clients

Typed clients:

- Provide the same capabilities as named clients without the need to use strings as keys.
- Provides IntelliSense and compiler help when consuming clients.
- Provide a single location to configure and interact with a particular `HttpClient`. For example, a single typed client might be used:
  - For a single backend endpoint.
  - To encapsulate all logic dealing with the endpoint.
- Work with DI and can be injected where required in the app.

A typed client accepts an `HttpClient` parameter in its constructor:

C#

```
public class GitHubService
{
    private readonly HttpClient _httpClient;

    public GitHubService(HttpClient httpClient)
    {
        _httpClient = httpClient;
```

```
        _httpClient.BaseAddress = new Uri("https://api.github.com/");

        // using Microsoft.Net.Http.Headers;
        // The GitHub API requires two headers.
        _httpClient.DefaultRequestHeaders.Add(
            HeaderNames.Accept, "application/vnd.github.v3+json");
        _httpClient.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "HttpRequestsSample");
    }

    public async Task<IEnumerable<GitHubBranch>?>
GetAspNetCoreDocsBranchesAsync() =>
        await _httpClient.GetFromJsonAsync<IEnumerable<GitHubBranch>>(
            "repos/dotnet/AspNetCore.Docs/branches");
}
```

In the preceding code:

- The configuration is moved into the typed client.
- The provided `HttpClient` instance is stored as a private field.

API-specific methods can be created that expose `HttpClient` functionality. For example, the `GetAspNetCoreDocsBranches` method encapsulates code to retrieve docs GitHub branches.

The following code calls AddHttpClient in `Program.cs` to register the `GitHubService` typed client class:

C#

```
builder.Services.AddHttpClient<GitHubService>();
```

The typed client is registered as transient with DI. In the preceding code, `AddHttpClient` registers `GitHubService` as a transient service. This registration uses a factory method to:

1. Create an instance of `HttpClient`.
2. Create an instance of `GitHubService`, passing in the instance of `HttpClient` to its constructor.

The typed client can be injected and consumed directly:

C#

```
public class TypedClientModel : PageModel
{
    private readonly GitHubService _gitHubService;
```

```
    public TypedClientModel(GitHubService gitHubService) =>
        _gitHubService = gitHubService;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        try
        {
            GitHubBranches = await
_gitHubService.GetAspNetCoreDocsBranchesAsync();
        }
        catch (HttpRequestException)
        {
            // ...
        }
    }
}
```

The configuration for a typed client can also be specified during its registration in `Program.cs`, rather than in the typed client's constructor:

C#

```
builder.Services.AddHttpClient<GitHubService>(httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // ...
});
```

Generated clients

`IHttpClientFactory` can be used in combination with third-party libraries such as Refit. Refit is a REST library for .NET. It converts REST APIs into live interfaces. Call `AddRefitClient` to generate a dynamic implementation of an interface, which uses `HttpClient` to make the external HTTP calls.

A custom interface represents the external API:

C#

```
public interface IGitHubClient
{
    [Get("/repos/dotnet/AspNetCore.Docs/branches")]
```

```
    Task<IEnumerable<GitHubBranch>> GetAspNetCoreDocsBranchesAsync();
}
```

Call `AddRefitClient` to generate the dynamic implementation and then call `ConfigureHttpClient` to configure the underlying `HttpClient`:

C#

```
builder.Services.AddRefitClient<IGitHubClient>()
    .ConfigureHttpClient(httpClient =>
    {
        httpClient.BaseAddress = new Uri("https://api.github.com/");

        // using Microsoft.Net.Http.Headers;
        // The GitHub API requires two headers.
        httpClient.DefaultRequestHeaders.Add(
            HeaderNames.Accept, "application/vnd.github.v3+json");
        httpClient.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "HttpRequestsSample");
    });
```

Use DI to access the dynamic implementation of `IGitHubClient`:

C#

```
public class RefitModel : PageModel
{
    private readonly IGitHubClient _gitHubClient;

    public RefitModel(IGitHubClient gitHubClient) =>
        _gitHubClient = gitHubClient;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        try
        {
            GitHubBranches = await
_gitHubClient.GetAspNetCoreDocsBranchesAsync();
        }
        catch (ApiException)
        {
            // ...
        }
    }
}
```

Make POST, PUT, and DELETE requests

In the preceding examples, all HTTP requests use the GET HTTP verb. `HttpClient` also supports other HTTP verbs, including:

- POST
- PUT
- DELETE
- PATCH

For a complete list of supported HTTP verbs, see HttpMethod.

The following example shows how to make an HTTP POST request:

C#

```
public async Task CreateItemAsync(TodoItem todoItem)
{
    var todoItemJson = new StringContent(
        JsonSerializer.Serialize(todoItem),
        Encoding.UTF8,
        Application.Json); // using static System.Net.Mime.MediaTypeNames;

    using var httpResponseMessage =
        await _httpClient.PostAsync("/api/TodoItems", todoItemJson);

    httpResponseMessage.EnsureSuccessStatusCode();
}
```

In the preceding code, the `CreateItemAsync` method:

- Serializes the `TodoItem` parameter to JSON using `System.Text.Json`.
- Creates an instance of StringContent to package the serialized JSON for sending in the HTTP request's body.
- Calls PostAsync to send the JSON content to the specified URL. This is a relative URL that gets added to the HttpClient.BaseAddress.
- Calls EnsureSuccessStatusCode to throw an exception if the response status code doesn't indicate success.

`HttpClient` also supports other types of content. For example, MultipartContent and StreamContent. For a complete list of supported content, see HttpContent.

The following example shows an HTTP PUT request:

C#

```
public async Task SaveItemAsync(TodoItem todoItem)
{
    var todoItemJson = new StringContent(
        JsonSerializer.Serialize(todoItem),
        Encoding.UTF8,
        Application.Json);

    using var httpResponseMessage =
        await _httpClient.PutAsync($"/api/TodoItems/{todoItem.Id}",
todoItemJson);

    httpResponseMessage.EnsureSuccessStatusCode();
}
```

The preceding code is similar to the POST example. The `SaveItemAsync` method calls PutAsync instead of `PostAsync`.

The following example shows an HTTP DELETE request:

C#

```
public async Task DeleteItemAsync(long itemId)
{
    using var httpResponseMessage =
        await _httpClient.DeleteAsync($"/api/TodoItems/{itemId}");

    httpResponseMessage.EnsureSuccessStatusCode();
}
```

In the preceding code, the `DeleteItemAsync` method calls DeleteAsync. Because HTTP DELETE requests typically contain no body, the `DeleteAsync` method doesn't provide an overload that accepts an instance of `HttpContent`.

To learn more about using different HTTP verbs with `HttpClient`, see HttpClient.

Outgoing request middleware

`HttpClient` has the concept of delegating handlers that can be linked together for outgoing HTTP requests. `IHttpClientFactory`:

- Simplifies defining the handlers to apply for each named client.
- Supports registration and chaining of multiple handlers to build an outgoing request middleware pipeline. Each of these handlers is able to perform work before and after the outgoing request. This pattern:
  - Is similar to the inbound middleware pipeline in ASP.NET Core.

- Provides a mechanism to manage cross-cutting concerns around HTTP requests, such as:
  - caching
  - error handling
  - serialization
  - logging

To create a delegating handler:

- Derive from DelegatingHandler.
- Override SendAsync. Execute code before passing the request to the next handler in the pipeline:

```
C#
```

```
public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest)
            {
                Content = new StringContent(
                    "The API key header X-API-KEY is required.")
            };
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

The preceding code checks if the `X-API-KEY` header is in the request. If `X-API-KEY` is missing, BadRequest is returned.

More than one handler can be added to the configuration for an `HttpClient` with Microsoft.Extensions.DependencyInjection.HttpClientBuilderExtensions.AddHttpMessageHandler:

```
C#
```

```
builder.Services.AddTransient<ValidateHeaderHandler>();

builder.Services.AddHttpClient("HttpMessageHandler")
    .AddHttpMessageHandler<ValidateHeaderHandler>();
```

In the preceding code, the `ValidateHeaderHandler` is registered with DI. Once registered, AddHttpMessageHandler can be called, passing in the type for the handler.

Multiple handlers can be registered in the order that they should execute. Each handler wraps the next handler until the final `HttpClientHandler` executes the request:

C#

```
builder.Services.AddTransient<SampleHandler1>();
builder.Services.AddTransient<SampleHandler2>();

builder.Services.AddHttpClient("MultipleHttpMessageHandlers")
    .AddHttpMessageHandler<SampleHandler1>()
    .AddHttpMessageHandler<SampleHandler2>();
```

In the preceding code, `SampleHandler1` runs first, before `SampleHandler2`.

Use DI in outgoing request middleware

When `IHttpClientFactory` creates a new delegating handler, it uses DI to fulfill the handler's constructor parameters. `IHttpClientFactory` creates a **separate** DI scope for each handler, which can lead to surprising behavior when a handler consumes a *scoped* service.

For example, consider the following interface and its implementation, which represents a task as an operation with an identifier, `OperationId`:

C#

```
public interface IOperationScoped
{
    string OperationId { get; }
}

public class OperationScoped : IOperationScoped
{
    public string OperationId { get; } = Guid.NewGuid().ToString()[^4..];
}
```

As its name suggests, `IOperationScoped` is registered with DI using a *scoped* lifetime:

C#

```
builder.Services.AddScoped<IOperationScoped, OperationScoped>();
```

The following delegating handler consumes and uses `IOperationScoped` to set the `X-OPERATION-ID` header for the outgoing request:

```
C#

public class OperationHandler : DelegatingHandler
{
    private readonly IOperationScoped _operationScoped;

    public OperationHandler(IOperationScoped operationScoped) =>
        _operationScoped = operationScoped;

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.Headers.Add("X-OPERATION-ID", _operationScoped.OperationId);

        return await base.SendAsync(request, cancellationToken);
    }
}
```

In the HttpRequestsSample download, navigate to `/Operation` and refresh the page. The request scope value changes for each request, but the handler scope value only changes every 5 seconds.

Handlers can depend upon services of any scope. Services that handlers depend upon are disposed when the handler is disposed.

Use one of the following approaches to share per-request state with message handlers:

- Pass data into the handler using HttpRequestMessage.Options.
- Use IHttpContextAccessor to access the current request.
- Create a custom AsyncLocal<T> storage object to pass the data.

Use Polly-based handlers

`IHttpClientFactory` integrates with the third-party library Polly. Polly is a comprehensive resilience and transient fault-handling library for .NET. It allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Extension methods are provided to enable the use of Polly policies with configured `HttpClient` instances. The Polly extensions support adding Polly-based handlers to clients. Polly requires the Microsoft.Extensions.Http.Polly NuGet package.

Handle transient faults

Faults typically occur when external HTTP calls are transient. AddTransientHttpErrorPolicy allows a policy to be defined to handle transient errors. Policies configured with `AddTransientHttpErrorPolicy` handle the following responses:

- HttpRequestException
- HTTP 5xx
- HTTP 408

`AddTransientHttpErrorPolicy` provides access to a `PolicyBuilder` object configured to handle errors representing a possible transient fault:

C#

```
builder.Services.AddHttpClient("PollyWaitAndRetry")
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.WaitAndRetryAsync(
            3, retryNumber => TimeSpan.FromMilliseconds(600)));
```

In the preceding code, a `WaitAndRetryAsync` policy is defined. Failed requests are retried up to three times with a delay of 600 ms between attempts.

Dynamically select policies

Extension methods are provided to add Polly-based handlers, for example, AddPolicyHandler. The following `AddPolicyHandler` overload inspects the request to decide which policy to apply:

C#

```
var timeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

builder.Services.AddHttpClient("PollyDynamic")
    .AddPolicyHandler(httpRequestMessage =>
        httpRequestMessage.Method == HttpMethod.Get ? timeoutPolicy :
longTimeoutPolicy);
```

In the preceding code, if the outgoing request is an HTTP GET, a 10-second timeout is applied. For any other HTTP method, a 30-second timeout is used.

Add multiple Polly handlers

It's common to nest Polly policies:

```
C#

builder.Services.AddHttpClient("PollyMultiple")
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.RetryAsync(3))
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```

In the preceding example:

- Two handlers are added.
- The first handler uses AddTransientHttpErrorPolicy to add a retry policy. Failed requests are retried up to three times.
- The second `AddTransientHttpErrorPolicy` call adds a circuit breaker policy. Further external requests are blocked for 30 seconds if 5 failed attempts occur sequentially. Circuit breaker policies are stateful. All calls through this client share the same circuit state.

Add policies from the Polly registry

An approach to managing regularly used policies is to define them once and register them with a `PolicyRegistry`. For example:

```
C#

var timeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

var policyRegistry = builder.Services.AddPolicyRegistry();

policyRegistry.Add("Regular", timeoutPolicy);
policyRegistry.Add("Long", longTimeoutPolicy);

builder.Services.AddHttpClient("PollyRegistryRegular")
    .AddPolicyHandlerFromRegistry("Regular");

builder.Services.AddHttpClient("PollyRegistryLong")
    .AddPolicyHandlerFromRegistry("Long");
```

In the preceding code:

- Two policies, `Regular` and `Long`, are added to the Polly registry.
- AddPolicyHandlerFromRegistry configures individual named clients to use these policies from the Polly registry.

For more information on `IHttpClientFactory` and Polly integrations, see the Polly wiki.

HttpClient and lifetime management

A new `HttpClient` instance is returned each time `CreateClient` is called on the `IHttpClientFactory`. An HttpMessageHandler is created per named client. The factory manages the lifetimes of the `HttpMessageHandler` instances.

`IHttpClientFactory` pools the `HttpMessageHandler` instances created by the factory to reduce resource consumption. An `HttpMessageHandler` instance may be reused from the pool when creating a new `HttpClient` instance if its lifetime hasn't expired.

Pooling of handlers is desirable as each handler typically manages its own underlying HTTP connections. Creating more handlers than necessary can result in connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS (Domain Name System) changes.

The default handler lifetime is two minutes. The default value can be overridden on a per named client basis:

C#

```
builder.Services.AddHttpClient("HandlerLifetime")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

`HttpClient` instances can generally be treated as .NET objects **not** requiring disposal. Disposal cancels outgoing requests and guarantees the given `HttpClient` instance can't be used after calling Dispose. `IHttpClientFactory` tracks and disposes resources used by `HttpClient` instances.

Keeping a single `HttpClient` instance alive for a long duration is a common pattern used before the inception of `IHttpClientFactory`. This pattern becomes unnecessary after migrating to `IHttpClientFactory`.

Alternatives to IHttpClientFactory

Using `IHttpClientFactory` in a DI-enabled app avoids:

- Resource exhaustion problems by pooling `HttpMessageHandler` instances.
- Stale DNS problems by cycling `HttpMessageHandler` instances at regular intervals.

There are alternative ways to solve the preceding problems using a long-lived SocketsHttpHandler instance.

- Create an instance of `SocketsHttpHandler` when the app starts and use it for the life of the app.
- Configure PooledConnectionLifetime to an appropriate value based on DNS refresh times.
- Create `HttpClient` instances using `new HttpClient(handler, disposeHandler: false)` as needed.

The preceding approaches solve the resource management problems that `IHttpClientFactory` solves in a similar way.

- The `SocketsHttpHandler` shares connections across `HttpClient` instances. This sharing prevents socket exhaustion.
- The `SocketsHttpHandler` cycles connections according to `PooledConnectionLifetime` to avoid stale DNS problems.

Logging

Clients created via `IHttpClientFactory` record log messages for all requests. Enable the appropriate information level in the logging configuration to see the default log messages. Additional logging, such as the logging of request headers, is only included at trace level.

The log category used for each client includes the name of the client. A client named *MyNamedClient*, for example, logs messages with a category of "System.Net.Http.HttpClient.**MyNamedClient**.LogicalHandler". Messages suffixed with *LogicalHandler* occur outside the request handler pipeline. On the request, messages are logged before any other handlers in the pipeline have processed it. On the response, messages are logged after any other pipeline handlers have received the response.

Logging also occurs inside the request handler pipeline. In the *MyNamedClient* example, those messages are logged with the log category "System.Net.Http.HttpClient.**MyNamedClient**.ClientHandler". For the request, this occurs after all other handlers have run and immediately before the request is sent. On the response, this logging includes the state of the response before it passes back through the handler pipeline.

Enabling logging outside and inside the pipeline enables inspection of the changes made by the other pipeline handlers. This may include changes to request headers or to the response status code.

Including the name of the client in the log category enables log filtering for specific named clients.

Configure the HttpMessageHandler

It may be necessary to control the configuration of the inner `HttpMessageHandler` used by a client.

An `IHttpClientBuilder` is returned when adding named or typed clients. The ConfigurePrimaryHttpMessageHandler extension method can be used to define a delegate. The delegate is used to create and configure the primary `HttpMessageHandler` used by that client:

C#

```
builder.Services.AddHttpClient("ConfiguredHttpMessageHandler")
    .ConfigurePrimaryHttpMessageHandler(() =>
        new HttpClientHandler
        {
            AllowAutoRedirect = true,
            UseDefaultCredentials = true
        });
```

Cookies

The pooled `HttpMessageHandler` instances results in `CookieContainer` objects being shared. Unanticipated `CookieContainer` object sharing often results in incorrect code. For apps that require cookies, consider either:

- Disabling automatic cookie handling
- Avoiding `IHttpClientFactory`

Call ConfigurePrimaryHttpMessageHandler to disable automatic cookie handling:

C#

```
builder.Services.AddHttpClient("NoAutomaticCookies")
    .ConfigurePrimaryHttpMessageHandler(() =>
        new HttpClientHandler
        {
```

```
            UseCookies = false
        });
```

Use IHttpClientFactory in a console app

In a console app, add the following package references to the project:

- Microsoft.Extensions.Hosting
- Microsoft.Extensions.Http

In the following example:

- IHttpClientFactory and `GitHubService` are registered in the Generic Host's service container.
- `GitHubService` is requested from DI, which in-turn requests an instance of `IHttpClientFactory`.
- `GitHubService` uses `IHttpClientFactory` to create an instance of `HttpClient`, which it uses to retrieve docs GitHub branches.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

var host = new HostBuilder()
    .ConfigureServices(services =>
    {
        services.AddHttpClient();
        services.AddTransient<GitHubService>();
    })
    .Build();

try
{
    var gitHubService = host.Services.GetRequiredService<GitHubService>();
    var gitHubBranches = await
gitHubService.GetAspNetCoreDocsBranchesAsync();

    Console.WriteLine($"{gitHubBranches?.Count() ?? 0} GitHub Branches");

    if (gitHubBranches is not null)
    {
        foreach (var gitHubBranch in gitHubBranches)
        {
            Console.WriteLine($"- {gitHubBranch.Name}");
        }
```

```
    }
}
catch (Exception ex)
{
    host.Services.GetRequiredService<ILogger<Program>>()
        .LogError(ex, "Unable to load branches from GitHub.");
}

public class GitHubService
{
    private readonly IHttpClientFactory _httpClientFactory;

    public GitHubService(IHttpClientFactory httpClientFactory) =>
        _httpClientFactory = httpClientFactory;

    public async Task<IEnumerable<GitHubBranch>?>
GetAspNetCoreDocsBranchesAsync()
    {
        var httpRequestMessage = new HttpRequestMessage(
            HttpMethod.Get,
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches")
        {
            Headers =
            {
                { "Accept", "application/vnd.github.v3+json" },
                { "User-Agent", "HttpRequestsConsoleSample" }
            }
        };

        var httpClient = _httpClientFactory.CreateClient();
        var httpResponseMessage = await
httpClient.SendAsync(httpRequestMessage);

        httpResponseMessage.EnsureSuccessStatusCode();

        using var contentStream =
            await httpResponseMessage.Content.ReadAsStreamAsync();

        return await JsonSerializer.DeserializeAsync
            <IEnumerable<GitHubBranch>>(contentStream);
    }
}

public record GitHubBranch(
    [property: JsonPropertyName("name")] string Name);
```

Header propagation middleware

Header propagation is an ASP.NET Core middleware to propagate HTTP headers from the incoming request to the outgoing `HttpClient` requests. To use header propagation:

- Install the Microsoft.AspNetCore.HeaderPropagation package.
- Configure the `HttpClient` and middleware pipeline in `Program.cs`:

```C#
// Add services to the container.
builder.Services.AddControllers();

builder.Services.AddHttpClient("PropagateHeaders")
    .AddHeaderPropagation();

builder.Services.AddHeaderPropagation(options =>
{
    options.Headers.Add("X-TraceId");
});

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseHttpsRedirection();

app.UseHeaderPropagation();

app.MapControllers();
```

- Make outbound requests using the configured `HttpClient` instance, which includes the added headers.

Additional resources

- View or download sample code (how to download)
- Use HttpClientFactory to implement resilient HTTP requests
- Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies
- Implement the Circuit Breaker pattern
- How to serialize and deserialize JSON in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Static files in ASP.NET Core

Article • 11/05/2024

By Rick Anderson

Static files, such as HTML, CSS, images, and JavaScript, are assets an ASP.NET Core app serves directly to clients by default.

Serve static files

Static files are stored within the project's web root directory. The default directory is `{content root}/wwwroot`, but it can be changed with the UseWebRoot method. For more information, see Content root and Web root.

The CreateBuilder method sets the content root to the current directory:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

Static files are accessible via a path relative to the web root. For example, the **Web Application** project templates contain several folders within the `wwwroot` folder:

- `wwwroot`
  - `css`

- `js`
- `lib`

Consider creating the *wwwroot/images* folder and adding the `wwwroot/images/MyImage.jpg` file. The URI format to access a file in the `images` folder is `https://<hostname>/images/<image_file_name>`. For example, `https://localhost:5001/images/MyImage.jpg`

Serve files in web root

The default web app templates call the UseStaticFiles method in `Program.cs`, which enables static files to be served:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

The parameterless `UseStaticFiles` method overload marks the files in web root as servable. The following markup references `wwwroot/images/MyImage.jpg`:

HTML

```
<img src="~/images/MyImage.jpg" class="img" alt="My image" />
```

In the preceding markup, the tilde character `~` points to the web root.

Serve files outside of web root

Consider a directory hierarchy in which the static files to be served reside outside of the web root:

- `wwwroot`
  - `css`
  - `images`
  - `js`
- `MyStaticFiles`
  - `images`
    - `red-rose.jpg`

A request can access the `red-rose.jpg` file by configuring the Static File Middleware as follows:

C#

```
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(
           Path.Combine(builder.Environment.ContentRootPath,
"MyStaticFiles")),
    RequestPath = "/StaticFiles"
});

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

In the preceding code, the *MyStaticFiles* directory hierarchy is exposed publicly via the *StaticFiles* URI segment. A request to `https://<hostname>/StaticFiles/images/red-rose.jpg` serves the `red-rose.jpg` file.

The following markup references `MyStaticFiles/images/red-rose.jpg`:

HTML

```
<img src="~/StaticFiles/images/red-rose.jpg" class="img" alt="A red rose" />
```

To serve files from multiple locations, see Serve files from multiple locations.

Set HTTP response headers

A StaticFileOptions object can be used to set HTTP response headers. In addition to configuring static file serving from the web root, the following code sets the Cache-Control header:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

var cacheMaxAgeOneWeek = (60 * 60 * 24 * 7).ToString();
app.UseStaticFiles(new StaticFileOptions
{
    OnPrepareResponse = ctx =>
    {
        ctx.Context.Response.Headers.Append(
             "Cache-Control", $"public, max-age={cacheMaxAgeOneWeek}");
    }
});

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();
```

```
app.Run();
```

The preceding code makes static files publicly available in the local cache for one week (604800 seconds).

Static file authorization

The ASP.NET Core templates call UseStaticFiles before calling UseAuthorization. Most apps follow this pattern. When the Static File Middleware is called before the authorization middleware:

- No authorization checks are performed on the static files.
- Static files served by the Static File Middleware, such as those under `wwwroot`, are publicly accessible.

To serve static files based on authorization:

- Store them outside of `wwwroot`.
- Call `UseStaticFiles`, specifying a path, after calling `UseAuthorization`.
- Set the fallback authorization policy.

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.FileProviders;
using StaticFileAuth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
```

```
builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(
           Path.Combine(builder.Environment.ContentRootPath,
"MyStaticFiles")),
    RequestPath = "/StaticFiles"
});

app.MapRazorPages();

app.Run();
```

In the preceding code, the fallback authorization policy requires ***all*** users to be authenticated. Endpoints such as controllers, Razor Pages, etc that specify their own authorization requirements don't use the fallback authorization policy. For example, Razor Pages, controllers, or action methods with `[AllowAnonymous]` or `[Authorize(PolicyName="MyPolicy")]` use the applied authorization attribute rather than the fallback authorization policy.

RequireAuthenticatedUser adds DenyAnonymousAuthorizationRequirement to the current instance, which enforces that the current user is authenticated.

Static assets under `wwwroot` are publicly accessible because the default Static File Middleware (`app.UseStaticFiles();`) is called before `UseAuthentication`. Static assets in the ***MyStaticFiles*** folder require authentication. The sample code demonstrates this.

An alternative approach to serve files based on authorization is to:

- Store them outside of `wwwroot` and any directory accessible to the Static File Middleware.
- Serve them via an action method to which authorization is applied and return a FileResult object:

```
C#

[Authorize]
public class BannerImageModel : PageModel
{
    private readonly IWebHostEnvironment _env;

    public BannerImageModel(IWebHostEnvironment env) =>
        _env = env;

    public PhysicalFileResult OnGet()
    {
        var filePath = Path.Combine(
                _env.ContentRootPath, "MyStaticFiles", "images", "red-
rose.jpg");

        return PhysicalFile(filePath, "image/jpeg");
    }
}
```

The preceding approach requires a page or endpoint per file. The following code returns files or uploads files for authenticated users:

```
C#

app.MapGet("/files/{fileName}",  IResult (string fileName) =>
    {
        var filePath = GetOrCreateFilePath(fileName);

        if (File.Exists(filePath))
        {
            return TypedResults.PhysicalFile(filePath, fileDownloadName: $"
{fileName}");
        }

        return TypedResults.NotFound("No file found with the supplied file
name");
    })
```

```
    .WithName("GetFileByName")
    .RequireAuthorization("AuthenticatedUsers");

// IFormFile uses memory buffer for uploading. For handling large file use
streaming instead.
// https://learn.microsoft.com/aspnet/core/mvc/models/file-uploads#upload-
large-files-with-streaming
app.MapPost("/files", async (IFormFile file, LinkGenerator linker,
HttpContext context) =>
    {
        // Don't rely on the file.FileName as it is only metadata that can
be manipulated by the end-user
        // Take a look at the `Utilities.IsFileValid` method that takes an
IFormFile and validates its signature within the AllowedFileSignatures

        var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
        await SaveFileWithCustomFileName(file, fileSaveName);

        context.Response.Headers.Append("Location",
linker.GetPathByName(context, "GetFileByName", new { fileName =
fileSaveName}));
        return TypedResults.Ok("File Uploaded Successfully!");
    })
    .RequireAuthorization("AdminsOnly");

app.Run();
```

See the StaticFileAuth GitHub folder for the complete sample.

Directory browsing

Directory browsing allows directory listing within specified directories.

Directory browsing is disabled by default for security reasons. For more information, see Security considerations for static files.

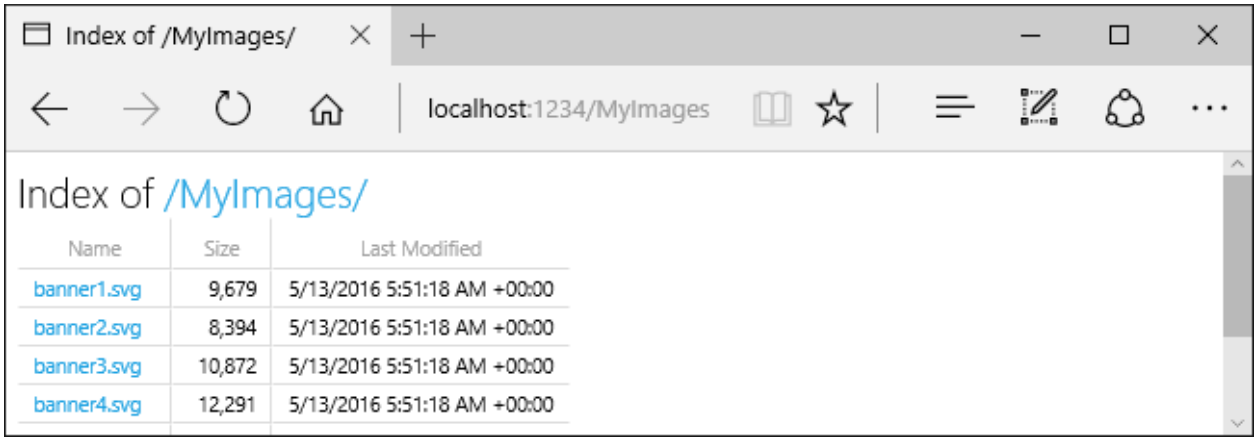
Enable directory browsing with AddDirectoryBrowser and UseDirectoryBrowser:

C#

```
using Microsoft.AspNetCore.StaticFiles;
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDirectoryBrowser();
```

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();

var fileProvider = new
PhysicalFileProvider(Path.Combine(builder.Environment.WebRootPath,
"images"));
var requestPath = "/MyImages";

// Enable displaying browser links.
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = fileProvider,
    RequestPath = requestPath
});

app.UseDirectoryBrowser(new DirectoryBrowserOptions
{
    FileProvider = fileProvider,
    RequestPath = requestPath
});

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

The preceding code allows directory browsing of the *wwwroot/images* folder using the URL `https://<hostname>/MyImages`, with links to each file and folder:



`AddDirectoryBrowser` adds services required by the directory browsing middleware, including HtmlEncoder. These services may be added by other calls, such as AddRazorPages, but we recommend calling `AddDirectoryBrowser` to ensure the services are added in all apps.

Serve default documents

Setting a default page provides visitors a starting point on a site. To serve a default file from `wwwroot` without requiring the request URL to include the file's name, call the UseDefaultFiles method:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseDefaultFiles();

app.UseStaticFiles();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

`UseDefaultFiles` must be called before `UseStaticFiles` to serve the default file. `UseDefaultFiles` is a URL rewriter that doesn't serve the file.

With `UseDefaultFiles`, requests to a folder in `wwwroot` search for:

- `default.htm`
- `default.html`
- `index.htm`

- `index.html`

The first file found from the list is served as though the request included the file's name. The browser URL continues to reflect the URI requested.

The following code changes the default file name to `mydefault.html`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

var options = new DefaultFilesOptions();
options.DefaultFileNames.Clear();
options.DefaultFileNames.Add("mydefault.html");
app.UseDefaultFiles(options);

app.UseStaticFiles();

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

UseFileServer for default documents

UseFileServer combines the functionality of `UseStaticFiles`, `UseDefaultFiles`, and optionally `UseDirectoryBrowser`.

Call `app.UseFileServer` to enable the serving of static files and the default file. Directory browsing isn't enabled:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseFileServer();

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

The following code enables the serving of static files, the default file, and directory browsing:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDirectoryBrowser();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseFileServer(enableDirectoryBrowsing: true);

app.UseRouting();

app.UseAuthorization();
```

```
app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

Consider the following directory hierarchy:

- `wwwroot`
  - `css`
  - `images`
  - `js`
- `MyStaticFiles`
  - `images`
    - `MyImage.jpg`
  - `default.html`

The following code enables the serving of static files, the default file, and directory browsing of `MyStaticFiles`:

C#

```
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDirectoryBrowser();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();
```

```
app.UseFileServer(new FileServerOptions
{
    FileProvider = new PhysicalFileProvider(
           Path.Combine(builder.Environment.ContentRootPath,
"MyStaticFiles")),
    RequestPath = "/StaticFiles",
    EnableDirectoryBrowsing = true
});

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

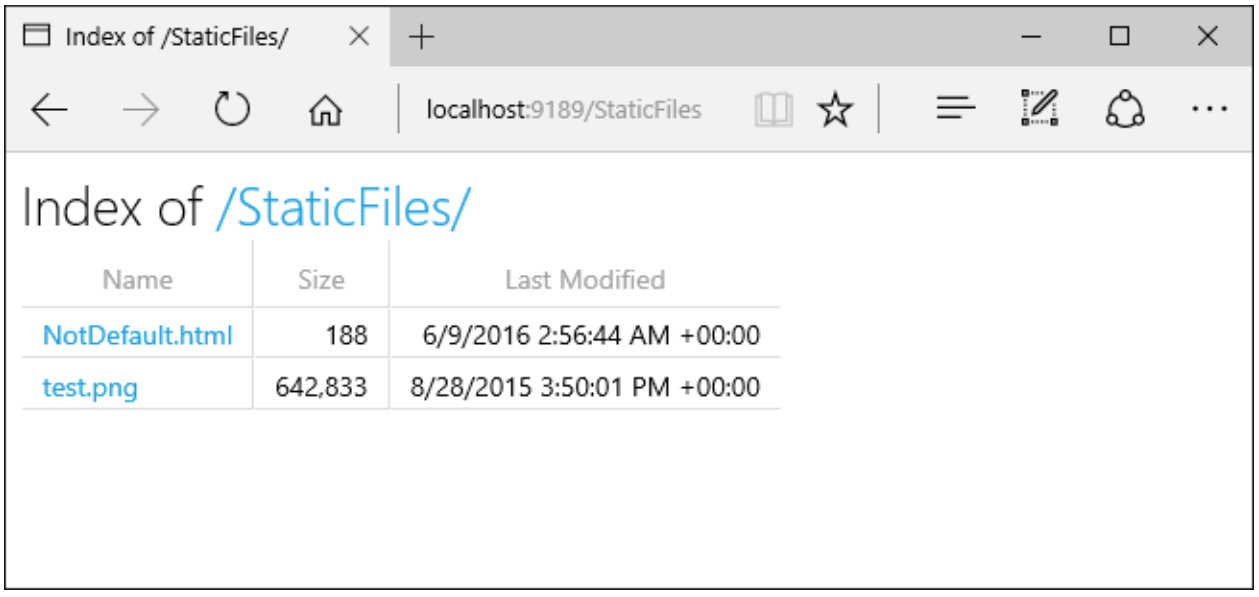
AddDirectoryBrowser must be called when the `EnableDirectoryBrowsing` property value is `true`.

Using the preceding file hierarchy and code, URLs resolve as follows:

Expand table

URI	Response
`https://<hostname>/StaticFiles/images/MyImage.jpg`	`MyStaticFiles/images/MyImage.jpg`
`https://<hostname>/StaticFiles`	`MyStaticFiles/default.html`

If no default-named file exists in the *MyStaticFiles* directory, `https://<hostname>/StaticFiles` returns the directory listing with clickable links:



UseDefaultFiles and UseDirectoryBrowser perform a client-side redirect from the target URI without a trailing `/` to the target URI with a trailing `/`. For example, from

`https://<hostname>/StaticFiles` to `https://<hostname>/StaticFiles/`. Relative URLs within the *StaticFiles* directory are invalid without a trailing slash (`/`) unless the RedirectToAppendTrailingSlash option of DefaultFilesOptions is used.

FileExtensionContentTypeProvider

The FileExtensionContentTypeProvider class contains a `Mappings` property that serves as a mapping of file extensions to MIME content types. In the following sample, several file extensions are mapped to known MIME types. The *.rtf* extension is replaced, and *.mp4* is removed:

C#

```
using Microsoft.AspNetCore.StaticFiles;
using Microsoft.Extensions.FileProviders;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

// Set up custom content types - associating file extension to MIME type
var provider = new FileExtensionContentTypeProvider();
// Add new mappings
provider.Mappings[".myapp"] = "application/x-msdownload";
provider.Mappings[".htm3"] = "text/html";
provider.Mappings[".image"] = "image/png";
// Replace an existing mapping
provider.Mappings[".rtf"] = "application/x-msdownload";
// Remove MP4 videos.
provider.Mappings.Remove(".mp4");

app.UseStaticFiles(new StaticFileOptions
{
    ContentTypeProvider = provider
});

app.UseAuthorization();

app.MapDefaultControllerRoute();
```

```
app.MapRazorPages();

app.Run();
```

See MIME content types.

Non-standard content types

The Static File Middleware understands almost 400 known file content types. If the user requests a file with an unknown file type, the Static File Middleware passes the request to the next middleware in the pipeline. If no middleware handles the request, a *404 Not Found* response is returned. If directory browsing is enabled, a link to the file is displayed in a directory listing.

The following code enables serving unknown types and renders the unknown file as an image:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles(new StaticFileOptions
{
    ServeUnknownFileTypes = true,
    DefaultContentType = "image/png"
});

app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

With the preceding code, a request for a file with an unknown content type is returned as an image.

> ⚠ **Warning**
>
> Enabling ServeUnknownFileTypes is a security risk. It's disabled by default, and its use is discouraged. FileExtensionContentTypeProvider provides a safer alternative to serving files with non-standard extensions.

Serve files from multiple locations

Consider the following Razor page which displays the `/MyStaticFiles/image3.png` file:

CSHTML

```
@page

<p> Test /MyStaticFiles/image3.png</p>

<img src="~/image3.png" class="img" asp-append-version="true" alt="Test">
```

`UseStaticFiles` and `UseFileServer` default to the file provider pointing at `wwwroot`. Additional instances of `UseStaticFiles` and `UseFileServer` can be provided with other file providers to serve files from other locations. The following example calls `UseStaticFiles` twice to serve files from both `wwwroot` and `MyStaticFiles`:

C#

```
app.UseStaticFiles(); // Serve files from wwwroot
app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(
        Path.Combine(builder.Environment.ContentRootPath, "MyStaticFiles"))
});
```

Using the preceding code:

- The `/MyStaticFiles/image3.png` file is displayed.
- The Image Tag Helpers AppendVersion is not applied because the Tag Helpers depend on WebRootFileProvider. `WebRootFileProvider` has not been updated to include the `MyStaticFiles` folder.

The following code updates the `WebRootFileProvider`, which enables the Image Tag Helper to provide a version:

C#

```
var webRootProvider = new
PhysicalFileProvider(builder.Environment.WebRootPath);
var newPathProvider = new PhysicalFileProvider(
  Path.Combine(builder.Environment.ContentRootPath, "MyStaticFiles"));

var compositeProvider = new CompositeFileProvider(webRootProvider,
                                                  newPathProvider);

// Update the default provider.
app.Environment.WebRootFileProvider = compositeProvider;

app.UseStaticFiles();
```

> ⓘ **Note**
>
> The preceding approach applies to Razor Pages and MVC apps. For guidance that applies to Blazor Web Apps, see ASP.NET Core Blazor static files.

Security considerations for static files

> ⚠ **Warning**
>
> `UseDirectoryBrowser` and `UseStaticFiles` can leak secrets. Disabling directory browsing in production is highly recommended. Carefully review which directories are enabled via `UseStaticFiles` or `UseDirectoryBrowser`. The entire directory and its sub-directories become publicly accessible. Store files suitable for serving to the public in a dedicated directory, such as `<content_root>/wwwroot`. Separate these files from MVC views, Razor Pages, configuration files, etc.

- The URLs for content exposed with `UseDirectoryBrowser` and `UseStaticFiles` are subject to the case sensitivity and character restrictions of the underlying file system. For example, Windows is case insensitive, but macOS and Linux aren't.
- ASP.NET Core apps hosted in IIS use the ASP.NET Core Module to forward all requests to the app, including static file requests. The IIS static file handler isn't used and has no chance to handle requests.

- Complete the following steps in IIS Manager to remove the IIS static file handler at the server or website level:

    1. Navigate to the **Modules** feature.
    2. Select **StaticFileModule** in the list.
    3. Click **Remove** in the **Actions** sidebar.

> ⚠ **Warning**
>
> If the IIS static file handler is enabled **and** the ASP.NET Core Module is configured incorrectly, static files are served. This happens, for example, if the *web.config* file isn't deployed.

- Place code files, including `.cs` and `.cshtml`, outside of the app project's web root. A logical separation is therefore created between the app's client-side content and server-based code. This prevents server-side code from being leaked.

Serve files outside wwwroot by updating IWebHostEnvironment.WebRootPath

When IWebHostEnvironment.WebRootPath is set to a folder other than `wwwroot`:

- In the development environment, static assets found in both `wwwroot` and the updated `IWebHostEnvironment.WebRootPath` are served from `wwwroot`.
- In any environment other than development, duplicate static assets are served from the updated `IWebHostEnvironment.WebRootPath` folder.

Consider a web app created with the empty web template:

- Containing an `Index.html` file in `wwwroot` and `wwwroot-custom`.
- With the following updated `Program.cs` file that sets `WebRootPath = "wwwroot-custom"`:

```
C#
```

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in "wwwroot-custom"
    WebRootPath = "wwwroot-custom"
});
```

```
var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();

app.Run();
```

In the preceding code, requests to `/`:

- In the development environment return `wwwroot/Index.html`
- In any environment other than development return `wwwroot-custom/Index.html`

To ensure assets from `wwwroot-custom` are returned, use one of the following approaches:

- Delete duplicate named assets in `wwwroot`.
- Set `"ASPNETCORE_ENVIRONMENT"` in `Properties/launchSettings.json` to any value other than `"Development"`.
- Completely disable static web assets by setting `<StaticWebAssetsEnabled>false</StaticWebAssetsEnabled>` in the project file. ***WARNING, disabling static web assets disables Razor Class Libraries.***
- Add the following JSON to the project file:

XML

```
<ItemGroup>
    <Content Remove="wwwroot\**" />
</ItemGroup>
```

The following code updates `IWebHostEnvironment.WebRootPath` to a non development value, guaranteeing duplicate content is returned from `wwwroot-custom` rather than `wwwroot`:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Examine Hosting environment: logging value
    EnvironmentName = Environments.Staging,
    WebRootPath = "wwwroot-custom"
});

var app = builder.Build();
```

```
app.Logger.LogInformation("ASPNETCORE_ENVIRONMENT: {env}",
      Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT"));

app.Logger.LogInformation("app.Environment.IsDevelopment(): {env}",
      app.Environment.IsDevelopment().ToString());

app.UseDefaultFiles();
app.UseStaticFiles();

app.Run();
```

Additional resources

- View or download sample code (how to download)
- Middleware
- Introduction to ASP.NET Core

dotnet-scaffold telemetry

Article • 11/11/2024

The `dotnet-scaffold` command includes a telemetry feature that collects usage data. This feature helps the `dotnet-scaffold` team understand how the tool is used so it can be improved.

How to opt out

The `dotnet-scaffold` telemetry feature is enabled by default. To opt out of the telemetry feature, set the `DOTNET_SCAFFOLD_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

Disclosure

When you run the `dotnet-scaffold` tool the first time, it displays output similar to the following example. The text may vary slightly depending on the version of the tool you're running. This "first run" experience is how Microsoft notifies you about data collection.

Console

```
dotnet-scaffold collects usage data in order to help us improve your
experience. The data is collected by Microsoft and shared with the
community. You can opt-out of telemetry by setting the
DOTNET_SCAFFOLD_TELEMETRY_OPTOUT environment variable to '1' or 'true' using
your favorite shell.

Read more about dotnet-scaffold telemetry:
https://aka.ms/dotnet-scaffold/telemetry
Read more about .NET CLI Tools telemetry:
https://aka.ms/dotnet-cli-telemetry
```

To suppress the "first run" experience text, set the `DOTNET_SCAFFOLD_SKIP_FIRST_TIME_EXPERIENCE` environment variable to `1` or `true`.

Data points

The telemetry feature doesn't collect:

- Personal data, such as usernames, email addresses, or URLs.
- Any project data.

The data is sent securely to Microsoft servers and held under restricted access.

Protecting your privacy is important to us. If you suspect the telemetry feature is collecting sensitive data or the data is being insecurely or inappropriately handled, take one of the following actions:

- File an issue in the dotnet/scaffolding repository.
- Send an email to dotnet@microsoft.com for investigation.

The telemetry feature collects the following data.

Expand table

.NET SDK versions	Data
>=8.0	Timestamp of invocation.
>=8.0	Three-octet IP address used to determine the geographical location.
>=8.0	Operating system and version.
>=8.0	Runtime ID (RID) the tool is running on.
>=8.0	Whether the tool is running in a container.
>=8.0	Hashed Media Access Control (MAC) address: a cryptographically (SHA256) hashed and unique ID for a machine.
>=8.0	Kernel version.
>=8.0	dotnet-scaffold version.
>=8.0	Hashed tool information (tool name, tool version, tool package name, tool package version, chosen scaffolder category, related scaffolding categories)
>=8.0	Tool level (global or local tool)
>=8.0	Hashed command invoked (for example, `mvccontroller`) and whether it succeeded.
>=8.0	dotnet-scaffold-aspnet scaffolder name, step names, and whether they succeeded.
>=8.0	dotnet-scaffold-aspire scaffolder name and whether it succeeded.
>=8.0	dotnet-scaffold aspnet scaffolder validation method name and whether they succeed.
>=8.0	dotnet-scaffold aspire scaffolder validation method name and whether they succeed.

Additional resources

- .NET Core SDK telemetry
- .NET CLI telemetry data

Choose an ASP.NET Core web UI

Article • 12/05/2023

ASP.NET Core is a complete UI framework. Choose which functionalities to combine that fit the app's web UI needs.

For new project development, we recommend ASP.NET Core Blazor.

ASP.NET Core Blazor

Blazor is a full-stack web UI framework and is recommended for most web UI scenarios.

Benefits of using Blazor:

- Reusable component model.
- Efficient diff-based component rendering.
- Flexibly render components from the server or client via WebAssembly.
- Build rich interactive web UI components in C#.
- Render components statically from the server.
- Progressively enhance server rendered components for smoother navigation and form handling and to enable streaming rendering.
- Share code for common logic on the client and server.
- Interop with JavaScript.
- Integrate components with existing MVC, Razor Pages, or JavaScript based apps.

For a complete overview of Blazor, its architecture and benefits, see ASP.NET Core Blazor and ASP.NET Core Blazor hosting models. To get started with your first Blazor app, see Build your first Blazor app.

ASP.NET Core Razor Pages

Razor Pages is a page-based model for building server rendered web UI. Razor pages UI are dynamically rendered on the server to generate the page's HTML and CSS in response to a browser request. The page arrives at the client ready to display. Support for Razor Pages is built on ASP.NET Core MVC.

Razor Pages benefits:

- Quickly build and update UI. Code for the page is kept with the page, while keeping UI and business logic concerns separate.
- Testable and scales to large apps.

- Keep your ASP.NET Core pages organized in a simpler way than ASP.NET MVC:
  - View specific logic and view models can be kept together in their own namespace and directory.
  - Groups of related pages can be kept in their own namespace and directory.

To get started with your first ASP.NET Core Razor Pages app, see Tutorial: Get started with Razor Pages in ASP.NET Core. For a complete overview of ASP.NET Core Razor Pages, its architecture and benefits, see: Introduction to Razor Pages in ASP.NET Core.

ASP.NET Core MVC

ASP.NET Core MVC renders UI on the server and uses a Model-View-Controller (MVC) architectural pattern. The MVC pattern separates an app into three main groups of components: models, views, and controllers. User requests are routed to a controller. The controller is responsible for working with the model to perform user actions or retrieve results of queries. The controller chooses the view to display to the user and provides it with any model data it requires.

ASP.NET Core MVC benefits:

- Based on a scalable and mature model for building large web apps.
- Clear separation of concerns for maximum flexibility.
- The Model-View-Controller separation of responsibilities ensures that the business model can evolve without being tightly coupled to low-level implementation details.

To get started with ASP.NET Core MVC, see Get started with ASP.NET Core MVC. For an overview of ASP.NET Core MVC's architecture and benefits, see Overview of ASP.NET Core MVC.

ASP.NET Core Single Page Applications (SPA) with frontend JavaScript frameworks

Build client-side logic for ASP.NET Core apps using popular JavaScript frameworks, like Angular, React, and Vue. ASP.NET Core provides project templates for Angular, React, and Vue, and it can be used with other JavaScript frameworks as well.

Benefits of ASP.NET Core SPA with JavaScript Frameworks, in addition to the client rendering benefits previously listed:

- The JavaScript runtime environment is already provided with the browser.
- Large community and mature ecosystem.

- Build client-side logic for ASP.NET Core apps using popular JS frameworks, like Angular, React, and Vue.

Downsides:

- More coding languages, frameworks, and tools required.
- Difficult to share code so some logic may be duplicated.

To get started, see:

- Create an ASP.NET Core app with Angular
- Create an ASP.NET Core app with React
- Create an ASP.NET Core app with Vue
- JavaScript and TypeScript in Visual Studio

Combine multiple web UI solutions: ASP.NET Core MVC or Razor Pages plus Blazor

MVC, Razor Pages, and Blazor are part of the ASP.NET Core framework and are designed to be used together. Razor components can be integrated into Razor Pages and MVC apps. When a view or page is rendered, components can be prerendered at the same time.

Benefits for MVC or Razor Pages plus Blazor, in addition to MVC or Razor Pages benefits:

- Prerendering executes Razor components on the server and renders them into a view or page, which improves the perceived load time of the app.
- Add interactivity to existing views or pages with the Component Tag Helper.

To get started with ASP.NET Core MVC or Razor Pages plus Blazor, see Integrate ASP.NET Core Razor components into ASP.NET Core apps.

Next steps

For more information, see:

- ASP.NET Core Blazor
- ASP.NET Core Blazor hosting models
- Integrate ASP.NET Core Razor components into ASP.NET Core apps
- Compare gRPC services with HTTP APIs

Introduction to Razor Pages in ASP.NET Core

Article • 10/07/2023

By Rick Anderson, Dave Brock, and Kirk Larkin

Razor Pages can make coding page-focused scenarios easier and more productive than using controllers and views.

If you're looking for a tutorial that uses the Model-View-Controller approach, see Get started with ASP.NET Core MVC.

This document provides an introduction to Razor Pages. It's not a step by step tutorial. If you find some of the sections too advanced, see Get started with Razor Pages. For an overview of ASP.NET Core, see the Introduction to ASP.NET Core.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

Create a Razor Pages project

Visual Studio

See Get started with Razor Pages for detailed instructions on how to create a Razor Pages project.

Razor Pages

Razor Pages is enabled in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In the preceding code:

- AddRazorPages adds services for Razor Pages to the app.
- MapRazorPages adds endpoints for Razor Pages to the IEndpointRouteBuilder.

Consider a basic page:

CSHTML

```
@page

<h1>Hello, world!</h1>
<h2>The time on the server is @DateTime.Now</h2>
```

The preceding code looks a lot like a Razor view file used in an ASP.NET Core app with controllers and views. What makes it different is the @page directive. `@page` makes the file into an MVC action, which means that it handles requests directly, without going through a controller. `@page` must be the first Razor directive on a page. `@page` affects the behavior of other Razor constructs. Razor Pages file names have a `.cshtml` suffix.

A similar page, using a `PageModel` class, is shown in the following two files. The `Pages/Index2.cshtml` file:

CSHTML

```
@page
@using RazorPagesIntro.Pages
@model Index2Model

<h2>Separate page model</h2>
<p>
    @Model.Message
</p>
```

The `Pages/Index2.cshtml.cs` page model:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System;

namespace RazorPagesIntro.Pages
{
    public class Index2Model : PageModel
    {
        public string Message { get; private set; } = "PageModel in C#";

        public void OnGet()
        {
            Message += $" Server time is { DateTime.Now }";
        }
    }
}
```

By convention, the `PageModel` class file has the same name as the Razor Page file with `.cs` appended. For example, the previous Razor Page is `Pages/Index2.cshtml`. The file containing the `PageModel` class is named `Pages/Index2.cshtml.cs`.

The associations of URL paths to pages are determined by the page's location in the file system. The following table shows a Razor Page path and the matching URL:

Expand table

File name and path	matching URL
`/Pages/Index.cshtml`	`/` or `/Index`
`/Pages/Contact.cshtml`	`/Contact`
`/Pages/Store/Contact.cshtml`	`/Store/Contact`
`/Pages/Store/Index.cshtml`	`/Store` or `/Store/Index`

Notes:

- The runtime looks for Razor Pages files in the *Pages* folder by default.
- `Index` is the default page when a URL doesn't include a page.

Write a basic form

Razor Pages is designed to make common patterns used with web browsers easy to implement when building an app. Model binding, Tag Helpers, and HTML helpers work with the properties defined in a Razor Page class. Consider a page that implements a basic "contact us" form for the `Contact` model:

For the samples in this document, the `DbContext` is initialized in the Program.cs file.

The in memory database requires the `Microsoft.EntityFrameworkCore.InMemory` NuGet package.

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The data model:

C#

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Models
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(10)]
        public string? Name { get; set; }
    }
}
```

The db context:

C#

```
using Microsoft.EntityFrameworkCore;

namespace RazorPagesContacts.Data
{
    public class CustomerDbContext : DbContext
    {
        public CustomerDbContext (DbContextOptions<CustomerDbContext>
options)
            : base(options)
        {
        }

        public DbSet<RazorPagesContacts.Models.Customer> Customer =>
Set<RazorPagesContacts.Models.Customer>();
    }
}
```

The `Pages/Customers/Create.cshtml` view file:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer!.Name" />
```

```
    <input type="submit" />
</form>
```

The `Pages/Customers/Create.cshtml.cs` page model:

C#

```
public class CreateModel : PageModel
{
    private readonly Data.CustomerDbContext _context;

    public CreateModel(Data.CustomerDbContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [BindProperty]
    public Customer? Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        if (Customer != null) _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

By convention, the `PageModel` class is called `<PageName>Model` and is in the same namespace as the page.

The `PageModel` class allows separation of the logic of a page from its presentation. It defines page handlers for requests sent to the page and the data used to render the page. This separation allows:

- Managing of page dependencies through dependency injection.
- Unit testing

The page has an `OnPostAsync` *handler method*, which runs on `POST` requests (when a user posts the form). Handler methods for any HTTP verb can be added. The most common handlers are:

- `OnGet` to initialize state needed for the page. In the preceding code, the `OnGet` method displays the `Create.cshtml` Razor Page.
- `OnPost` to handle form submissions.

The `Async` naming suffix is optional but is often used by convention for asynchronous functions. The preceding code is typical for Razor Pages.

If you're familiar with ASP.NET apps using controllers and views:

- The `OnPostAsync` code in the preceding example looks similar to typical controller code.
- Most of the MVC primitives like model binding, validation, and action results work the same with Controllers and Razor Pages.

The previous `OnPostAsync` method:

C#

```
[BindProperty]
public Customer? Customer { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    if (Customer != null) _context.Customer.Add(Customer);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

The basic flow of `OnPostAsync`:

Check for validation errors.

- If there are no errors, save the data and redirect.
- If there are errors, show the page again with validation messages. In many cases, validation errors would be detected on the client, and never submitted to the server.

The `Pages/Customers/Create.cshtml` view file:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>
```

The rendered HTML from `Pages/Customers/Create.cshtml`:

HTML

```
<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input type="text" data-val="true"
           data-val-length="The field Name must be a string with a maximum
length of 10."
           data-val-length-max="10" data-val-required="The Name field is
required."
           id="Customer_Name" maxlength="10" name="Customer.Name" value=""
/>
    <input type="submit" />
    <input name="__RequestVerificationToken" type="hidden"
           value="<Antiforgery token here>" />
</form>
```

In the previous code, posting the form:

- With valid data:
  - The `OnPostAsync` handler method calls the RedirectToPage helper method. `RedirectToPage` returns an instance of RedirectToPageResult. `RedirectToPage`:
    - Is an action result.
    - Is similar to `RedirectToAction` or `RedirectToRoute` (used in controllers and views).
    - Is customized for pages. In the preceding sample, it redirects to the root Index page (`/Index`). `RedirectToPage` is detailed in the URL generation for Pages section.

- With validation errors that are passed to the server:
  - The `OnPostAsync` handler method calls the Page helper method. `Page` returns an instance of PageResult. Returning `Page` is similar to how actions in controllers return `View`. `PageResult` is the default return type for a handler method. A handler method that returns `void` renders the page.
  - In the preceding example, posting the form with no value results in ModelState.IsValid returning false. In this sample, no validation errors are displayed on the client. Validation error handling is covered later in this document.

```
C#

[BindProperty]
public Customer? Customer { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    if (Customer != null) _context.Customer.Add(Customer);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

- With validation errors detected by client side validation:
  - Data is **not** posted to the server.
  - Client-side validation is explained later in this document.

The `Customer` property uses [BindProperty] attribute to opt in to model binding:

```
C#

[BindProperty]
public Customer? Customer { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    if (Customer != null) _context.Customer.Add(Customer);
    await _context.SaveChangesAsync();
```

```
    return RedirectToPage("./Index");
}
```

`[BindProperty]` should **not** be used on models containing properties that should not be changed by the client. For more information, see Overposting.

Razor Pages, by default, bind properties only with non-`GET` verbs. Binding to properties removes the need to writing code to convert HTTP data to the model type. Binding reduces code by using the same property to render form fields (`<input asp-for="Customer.Name">`) and accept the input.

> ⚠ **Warning**
>
> For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.
>
> To bind a property on `GET` requests, set the [BindProperty] attribute's `SupportsGet` property to `true`:
>
> C#
>
> ```
> [BindProperty(SupportsGet = true)]
> ```
>
> For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Reviewing the `Pages/Customers/Create.cshtml` view file:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>
```

- In the preceding code, the input tag helper `<input asp-for="Customer.Name" />` binds the HTML `<input>` element to the `Customer.Name` model expression.
- @addTagHelper makes Tag Helpers available.

The home page

`Index.cshtml` is the home page:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.IndexModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<h1>Contacts home page</h1>
<form method="post">
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Name</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
        @if (Model.Customers != null)
        {
            foreach (var contact in Model.Customers)
            {
                <tr>
                    <td> @contact.Id </td>
                    <td>@contact.Name</td>
                    <td>
                        <!-- <snippet_Edit> -->
                        <a asp-page="./Edit" asp-route-
id="@contact.Id">Edit</a> |
                        <!-- </snippet_Edit> -->
                        <!-- <snippet_Delete> -->
                        <button type="submit" asp-page-handler="delete" asp-
route-id="@contact.Id">delete</button>
                        <!-- </snippet_Delete> -->
                    </td>
                </tr>
            }
        }
        </tbody>
    </table>
    <a asp-page="Create">Create New</a>
</form>
```

The associated `PageModel` class (`Index.cshtml.cs`):

C#

```
public class IndexModel : PageModel
{
    private readonly Data.CustomerDbContext _context;
    public IndexModel(Data.CustomerDbContext context)
    {
        _context = context;
    }

    public IList<Customer>? Customers { get; set; }

    public async Task OnGetAsync()
    {
        Customers = await _context.Customer.ToListAsync();
    }

    public async Task<IActionResult> OnPostDeleteAsync(int id)
    {
        var contact = await _context.Customer.FindAsync(id);

        if (contact != null)
        {
            _context.Customer.Remove(contact);
            await _context.SaveChangesAsync();
        }

        return RedirectToPage();
    }
}
```

The `Index.cshtml` file contains the following markup:

CSHTML

```
<a asp-page="./Edit" asp-route-id="@contact.Id">Edit</a> |
```

The `<a /a>` Anchor Tag Helper used the `asp-route-{value}` attribute to generate a link to the Edit page. The link contains route data with the contact ID. For example, `https://localhost:5001/Edit/1`. Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.

The `Index.cshtml` file contains markup to create a delete button for each customer contact:

CSHTML

```
<button type="submit" asp-page-handler="delete" asp-route-
id="@contact.Id">delete</button>
```

The rendered HTML:

HTML

```
<button type="submit" formaction="/Customers?
id=1&amp;handler=delete">delete</button>
```

When the delete button is rendered in HTML, its formaction includes parameters for:

- The customer contact ID, specified by the `asp-route-id` attribute.
- The `handler`, specified by the `asp-page-handler` attribute.

When the button is selected, a form `POST` request is sent to the server. By convention, the name of the handler method is selected based on the value of the `handler` parameter according to the scheme `OnPost[handler]Async`.

Because the `handler` is `delete` in this example, the `OnPostDeleteAsync` handler method is used to process the `POST` request. If the `asp-page-handler` is set to a different value, such as `remove`, a handler method with the name `OnPostRemoveAsync` is selected.

C#

```
public async Task<IActionResult> OnPostDeleteAsync(int id)
{
    var contact = await _context.Customer.FindAsync(id);

    if (contact != null)
    {
        _context.Customer.Remove(contact);
        await _context.SaveChangesAsync();
    }

    return RedirectToPage();
}
```

The `OnPostDeleteAsync` method:

- Gets the `id` from the query string.
- Queries the database for the customer contact with `FindAsync`.
- If the customer contact is found, it's removed and the database is updated.
- Calls RedirectToPage to redirect to the root Index page (`/Index`).

The Edit.cshtml file

CSHTML

```
@page "{id:int}"
@model RazorPagesContacts.Pages.Customers.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Customer</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Customer!.Id" />
            <div class="form-group">
                <label asp-for="Customer!.Name" class="control-label">
</label>
                <input asp-for="Customer!.Name" class="form-control" />
                <span asp-validation-for="Customer!.Name" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The first line contains the `@page "{id:int}"` directive. The routing constraint `"{id:int}"` tells the page to accept requests to the page that contain `int` route data. If a request to the page doesn't contain route data that can be converted to an `int`, the runtime returns an HTTP 404 (not found) error. To make the ID optional, append `?` to the route constraint:

CSHTML

```
@page "{id:int?}"
```

The `Edit.cshtml.cs` file:

C#

```
public class EditModel : PageModel
{
    private readonly RazorPagesContacts.Data.CustomerDbContext _context;

    public EditModel(RazorPagesContacts.Data.CustomerDbContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Customer? Customer { get; set; }

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null)
        {
            return NotFound();
        }

        Customer = await _context.Customer.FirstOrDefaultAsync(m => m.Id ==
id);

        if (Customer == null)
        {
            return NotFound();
        }
        return Page();
    }

    // To protect from overposting attacks, enable the specific properties
you want to bind to.
    // For more details, see https://aka.ms/RazorPagesCRUD.
    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        if (Customer != null)
        {
            _context.Attach(Customer).State = EntityState.Modified;

            try
            {
                await _context.SaveChangesAsync();
```

```
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!CustomerExists(Customer.Id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }
        }

        return RedirectToPage("./Index");
    }

    private bool CustomerExists(int id)
    {
        return _context.Customer.Any(e => e.Id == id);
    }
}
```

Validation

Validation rules:

- Are declaratively specified in the model class.
- Are enforced everywhere in the app.

The System.ComponentModel.DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like [DataType] that help with formatting and don't provide any validation.

Consider the `Customer` model:

C#

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesContacts.Models
{
    public class Customer
    {
        public int Id { get; set; }

        [Required, StringLength(10)]
        public string? Name { get; set; }
```

```
    }
}
```

Using the following `Create.cshtml` view file:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Validation: customer name:</p>

<form method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <span asp-validation-for="Customer!.Name"></span>
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>
```

The preceding code:

- Includes jQuery and jQuery validation scripts.
- Uses the `<div />` and `<span />` Tag Helpers to enable:
  - Client-side validation.
  - Validation error rendering.
- Generates the following HTML:

  HTML

  ```
  <p>Enter a customer name:</p>

  <form method="post">
      Name:
      <input type="text" data-val="true"
             data-val-length="The field Name must be a string with a
  maximum length of 10."
             data-val-length-max="10" data-val-required="The Name field
  is required."
             id="Customer_Name" maxlength="10" name="Customer.Name"
  value="" />
      <input type="submit" />
  ```

```
    <input name="__RequestVerificationToken" type="hidden"
           value="<Antiforgery token here>" />
</form>

<script src="/lib/jquery/dist/jquery.js"></script>
<script src="/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>
```

Posting the Create form without a name value displays the error message "The Name field is required." on the form. If JavaScript is enabled on the client, the browser displays the error without posting to the server.

The `[StringLength(10)]` attribute generates `data-val-length-max="10"` on the rendered HTML. `data-val-length-max` prevents browsers from entering more than the maximum length specified. If a tool such as Fiddler is used to edit and replay the post:

- With the name longer than 10.
- The error message "The field Name must be a string with a maximum length of 10." is returned.

Consider the following `Movie` model:

C#

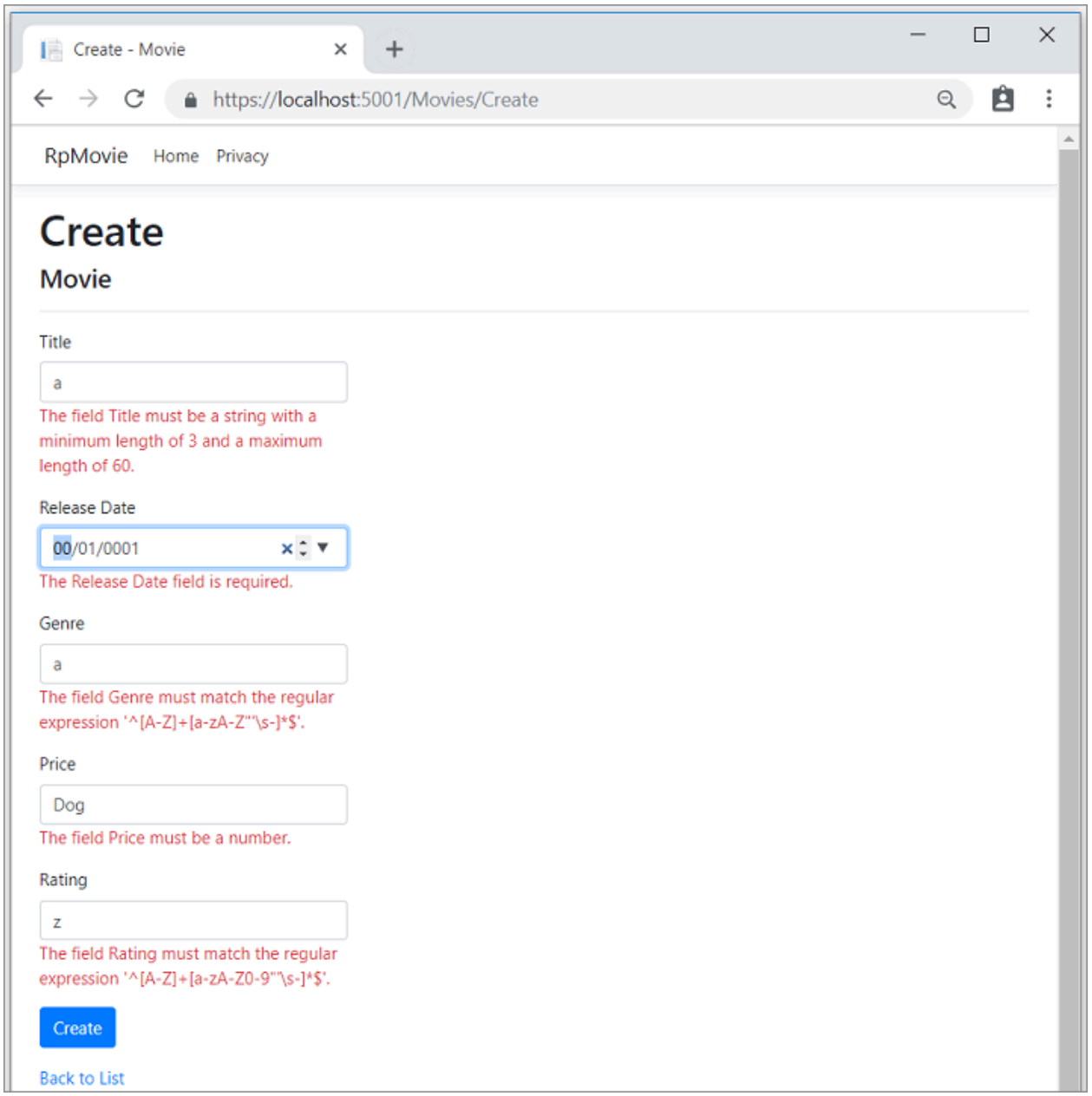
```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }

        [StringLength(60, MinimumLength = 3)]
        [Required]
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }

        [Range(1, 100)]
        [DataType(DataType.Currency)]
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
        [Required]
```

```
        [StringLength(30)]
        public string Genre { get; set; }

        [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
        [StringLength(5)]
        [Required]
        public string Rating { get; set; }
    }
}
```

The validation attributes specify behavior to enforce on the model properties they're applied to:

- The `Required` and `MinimumLength` attributes indicate that a property must have a value, but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White space, numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute sets the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

The Create page for the `Movie` model shows displays errors with invalid values:



For more information, see:

- Add validation to the Movie app
- Model validation in ASP.NET Core.

CSS isolation

Isolate CSS styles to individual pages, views, and components to reduce or avoid:

- Dependencies on global styles that can be challenging to maintain.
- Style conflicts in nested content.

To add a *scoped CSS file* for a page or view, place the CSS styles in a companion `.cshtml.css` file matching the name of the `.cshtml` file. In the following example, an `Index.cshtml.css` file supplies CSS styles that are only applied to the `Index.cshtml` page or view.

`Pages/Index.cshtml.css` (Razor Pages) or `Views/Index.cshtml.css` (MVC):

```css
h1 {
    color: red;
}
```

CSS isolation occurs at build time. The framework rewrites CSS selectors to match markup rendered by the app's pages or views. The rewritten CSS styles are bundled and produced as a static asset, `{APP ASSEMBLY}.styles.css`. The placeholder `{APP ASSEMBLY}` is the assembly name of the project. A link to the bundled CSS styles is placed in the app's layout.

In the `<head>` content of the app's `Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC), add or confirm the presence of the link to the bundled CSS styles:

```HTML
<link rel="stylesheet" href="~/{APP ASSEMBLY}.styles.css" />
```

In the following example, the app's assembly name is `WebApp`:

```HTML
<link rel="stylesheet" href="WebApp.styles.css" />
```

The styles defined in a scoped CSS file are only applied to the rendered output of the matching file. In the preceding example, any `h1` CSS declarations defined elsewhere in the app don't conflict with the `Index`'s heading style. CSS style cascading and inheritance rules remain in effect for scoped CSS files. For example, styles applied directly to an `<h1>` element in the `Index.cshtml` file override the scoped CSS file's styles in `Index.cshtml.css`.

> **Note**
>
> In order to guarantee CSS style isolation when bundling occurs, importing CSS in Razor code blocks isn't supported.
>
> CSS isolation only applies to HTML elements. CSS isolation isn't supported for Tag Helpers.

Within the bundled CSS file, each page, view, or Razor component is associated with a scope identifier in the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. The following example provides the style for the preceding `<h1>` element in the `Index` page of a Razor Pages app:

```css
/* /Pages/Index.cshtml.rz.scp.css */
h1[b-3xxtam6d07] {
    color: red;
}
```

In the `Index` page where the CSS style is applied from the bundled file, the scope identifier is appended as an HTML attribute:

```HTML
<h1 b-3xxtam6d07>
```

The identifier is unique to an app. At build time, a project bundle is created with the convention `{STATIC WEB ASSETS BASE PATH}/Project.lib.scp.css`, where the placeholder `{STATIC WEB ASSETS BASE PATH}` is the static web assets base path.

If other projects are utilized, such as NuGet packages or Razor class libraries, the bundled file:

- References the styles using CSS imports.
- Isn't published as a static web asset of the app that consumes the styles.

CSS preprocessor support

CSS preprocessors are useful for improving CSS development by utilizing features such as variables, nesting, modules, mixins, and inheritance. While CSS isolation doesn't natively support CSS preprocessors such as Sass or Less, integrating CSS preprocessors is seamless as long as preprocessor compilation occurs before the framework rewrites the CSS selectors during the build process. Using Visual Studio for example, configure existing preprocessor compilation as a **Before Build** task in the Visual Studio Task Runner Explorer.

Many third-party NuGet packages, such as AspNetCore.SassCompiler, can compile SASS/SCSS files at the beginning of the build process before CSS isolation occurs, and no additional configuration is required.

CSS isolation configuration

CSS isolation permits configuration for some advanced scenarios, such as when there are dependencies on existing tools or workflows.

Customize scope identifier format

In this section, the `{Pages|Views}` placeholder is either `Pages` for Razor Pages apps or `Views` for MVC apps.

By default, scope identifiers use the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. To customize the scope identifier format, update the project file to a desired pattern:

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/Index.cshtml.css" CssScope="custom-scope-
identifier" />
</ItemGroup>
```

In the preceding example, the CSS generated for `Index.cshtml.css` changes its scope identifier from `b-{STRING}` to `custom-scope-identifier`.

Use scope identifiers to achieve inheritance with scoped CSS files. In the following project file example, a `BaseView.cshtml.css` file contains common styles across views. A `DerivedView.cshtml.css` file inherits these styles.

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/BaseView.cshtml.css" CssScope="custom-scope-
identifier" />
  <None Update="{Pages|Views}/DerivedView.cshtml.css" CssScope="custom-
scope-identifier" />
</ItemGroup>
```

Use the wildcard (`*`) operator to share scope identifiers across multiple files:

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/*.cshtml.css" CssScope="custom-scope-
```

```
identifier" />
</ItemGroup>
```

Change base path for static web assets

The scoped CSS file is generated at the root of the app. In the project file, use the `StaticWebAssetBasePath` property to change the default path. The following example places the scoped CSS file, and the rest of the app's assets, at the `_content` path:

XML

```
<PropertyGroup>
  <StaticWebAssetBasePath>_content/$(PackageId)</StaticWebAssetBasePath>
</PropertyGroup>
```

Disable automatic bundling

To opt out of how framework publishes and loads scoped files at runtime, use the `DisableScopedCssBundling` property. When using this property, other tools or processes are responsible for taking the isolated CSS files from the `obj` directory and publishing and loading them at runtime:

XML

```
<PropertyGroup>
  <DisableScopedCssBundling>true</DisableScopedCssBundling>
</PropertyGroup>
```

Razor class library (RCL) support

When a Razor class library (RCL) provides isolated styles, the `<link>` tag's `href` attribute points to `{STATIC WEB ASSET BASE PATH}/{PACKAGE ID}.bundle.scp.css`, where the placeholders are:

- `{STATIC WEB ASSET BASE PATH}`: The static web asset base path.
- `{PACKAGE ID}`: The library's package identifier. The package identifier defaults to the project's assembly name if the package identifier isn't specified in the project file.

In the following example:

- The static web asset base path is `_content/ClassLib`.
- The class library's assembly name is `ClassLib`.

`Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC):

HTML

```
<link href="_content/ClassLib/ClassLib.bundle.scp.css" rel="stylesheet">
```

For more information on RCLs, see the following articles:

- Reusable Razor UI in class libraries with ASP.NET Core
- Consume ASP.NET Core Razor components from a Razor class library (RCL)

For information on Blazor CSS isolation, see ASP.NET Core Blazor CSS isolation.

Handle HEAD requests with an OnGet handler fallback

`HEAD` requests allow retrieving the headers for a specific resource. Unlike `GET` requests, `HEAD` requests don't return a response body.

Ordinarily, an `OnHead` handler is created and called for `HEAD` requests:

C#

```
public void OnHead()
{
    HttpContext.Response.Headers.Add("Head Test", "Handled by OnHead!");
}
```

Razor Pages falls back to calling the `OnGet` handler if no `OnHead` handler is defined.

XSRF/CSRF and Razor Pages

Razor Pages are protected by Antiforgery validation. The FormTagHelper injects antiforgery tokens into HTML form elements.

Using Layouts, partials, templates, and Tag Helpers with Razor Pages

Pages work with all the capabilities of the Razor view engine. Layouts, partials, templates, Tag Helpers, `_ViewStart.cshtml`, and `_ViewImports.cshtml` work in the same way they do for conventional Razor views.

Let's declutter this page by taking advantage of some of those capabilities.

Add a layout page to `Pages/Shared/_Layout.cshtml`:

CSHTML

```
<!DOCTYPE html>
<html>
<head>
    <title>RP Sample</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <a asp-page="/Index">Home</a>
    <a asp-page="/Customers/Create">Create</a>
    <a asp-page="/Customers/Index">Customers</a> <br />

    @RenderBody()
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>
</body>
</html>
```

The Layout:

- Controls the layout of each page (unless the page opts out of layout).
- Imports HTML structures such as JavaScript and stylesheets.
- The contents of the Razor page are rendered where `@RenderBody()` is called.

For more information, see layout page.

The Layout property is set in `Pages/_ViewStart.cshtml`:

CSHTML

```
@{
    Layout = "_Layout";
}
```

The layout is in the *Pages/Shared* folder. Pages look for other views (layouts, templates, partials) hierarchically, starting in the same folder as the current page. A layout in the *Pages/Shared* folder can be used from any Razor page under the *Pages* folder.

The layout file should go in the *Pages/Shared* folder.

We recommend you **not** put the layout file in the *Views/Shared* folder. *Views/Shared* is an MVC views pattern. Razor Pages are meant to rely on folder hierarchy, not path conventions.

View search from a Razor Page includes the *Pages* folder. The layouts, templates, and partials used with MVC controllers and conventional Razor views *just work*.

Add a `Pages/_ViewImports.cshtml` file:

CSHTML

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

`@namespace` is explained later in the tutorial. The `@addTagHelper` directive brings in the built-in Tag Helpers to all the pages in the *Pages* folder.

The `@namespace` directive set on a page:

CSHTML

```
@page
@namespace RazorPagesIntro.Pages.Customers

@model NameSpaceModel

<h2>Name space</h2>
<p>
    @Model.Message
</p>
```

The `@namespace` directive sets the namespace for the page. The `@model` directive doesn't need to include the namespace.

When the `@namespace` directive is contained in `_ViewImports.cshtml`, the specified namespace supplies the prefix for the generated namespace in the Page that imports the `@namespace` directive. The rest of the generated namespace (the suffix portion) is the dot-separated relative path between the folder containing `_ViewImports.cshtml` and the folder containing the page.

For example, the `PageModel` class `Pages/Customers/Edit.cshtml.cs` explicitly sets the namespace:

C#

```
namespace RazorPagesContacts.Pages
{
    public class EditModel : PageModel
    {
        private readonly AppDbContext _db;

        public EditModel(AppDbContext db)
        {
            _db = db;
        }

        // Code removed for brevity.
```

The `Pages/_ViewImports.cshtml` file sets the following namespace:

CSHTML

```
@namespace RazorPagesContacts.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The generated namespace for the `Pages/Customers/Edit.cshtml` Razor Page is the same as the `PageModel` class.

`@namespace` *also works with conventional Razor views.*

Consider the `Pages/Customers/Create.cshtml` view file:

CSHTML

```
@page
@model RazorPagesContacts.Pages.Customers.CreateModel
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

<p>Validation: customer name:</p>

<form method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <span asp-validation-for="Customer!.Name"></span>
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>

<script src="~/lib/jquery/dist/jquery.js"></script>
<script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
<script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>
```

The updated `Pages/Customers/Create.cshtml` view file with `_ViewImports.cshtml` and the preceding layout file:

CSHTML

```
@page
@model CreateModel

<p>Enter a customer name:</p>

<form method="post">
    Name:
    <input asp-for="Customer!.Name" />
    <input type="submit" />
</form>
```

In the preceding code, the `_ViewImports.cshtml` imported the namespace and Tag Helpers. The layout file imported the JavaScript files.

The Razor Pages starter project contains the `Pages/_ValidationScriptsPartial.cshtml`, which hooks up client-side validation.

For more information on partial views, see Partial views in ASP.NET Core.

URL generation for Pages

The `Create` page, shown previously, uses `RedirectToPage`:

C#

```
public class CreateModel : PageModel
{
    private readonly Data.CustomerDbContext _context;

    public CreateModel(Data.CustomerDbContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [BindProperty]
    public Customer? Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
```

```
            if (!ModelState.IsValid)
            {
                return Page();
            }

            if (Customer != null) _context.Customer.Add(Customer);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
```

The app has the following file/folder structure:

- */Pages*
  - `Index.cshtml`
  - `Privacy.cshtml`
  - */Customers*
    - `Create.cshtml`
    - `Edit.cshtml`
    - `Index.cshtml`

The `Pages/Customers/Create.cshtml` and `Pages/Customers/Edit.cshtml` pages redirect to `Pages/Customers/Index.cshtml` after success. The string `./Index` is a relative page name used to access the preceding page. It is used to generate URLs to the `Pages/Customers/Index.cshtml` page. For example:

- `Url.Page("./Index", ...)`
- `<a asp-page="./Index">Customers Index Page</a>`
- `RedirectToPage("./Index")`

The absolute page name `/Index` is used to generate URLs to the `Pages/Index.cshtml` page. For example:

- `Url.Page("/Index", ...)`
- `<a asp-page="/Index">Home Index Page</a>`
- `RedirectToPage("/Index")`

The page name is the path to the page from the root */Pages* folder including a leading `/` (for example, `/Index`). The preceding URL generation samples offer enhanced options and functional capabilities over hard-coding a URL. URL generation uses routing and

can generate and encode parameters according to how the route is defined in the destination path.

URL generation for pages supports relative names. The following table shows which Index page is selected using different `RedirectToPage` parameters in `Pages/Customers/Create.cshtml`.

Expand table

RedirectToPage(x)	Page
RedirectToPage("/Index")	*Pages/Index*
RedirectToPage("./Index");	*Pages/Customers/Index*
RedirectToPage("../Index")	*Pages/Index*
RedirectToPage("Index")	*Pages/Customers/Index*

`RedirectToPage("Index")`, `RedirectToPage("./Index")`, and `RedirectToPage("../Index")` are *relative names*. The `RedirectToPage` parameter is *combined* with the path of the current page to compute the name of the destination page.

Relative name linking is useful when building sites with a complex structure. When relative names are used to link between pages in a folder:

- Renaming a folder doesn't break the relative links.
- Links are not broken because they don't include the folder name.

To redirect to a page in a different Area, specify the area:

```
C#

RedirectToPage("/Index", new { area = "Services" });
```

For more information, see Areas in ASP.NET Core and Razor Pages route and app conventions in ASP.NET Core.

ViewData attribute

Data can be passed to a page with ViewDataAttribute. Properties with the `[ViewData]` attribute have their values stored and loaded from the ViewDataDictionary.

In the following example, the `AboutModel` applies the `[ViewData]` attribute to the `Title` property:

C#

```
public class AboutModel : PageModel
{
    [ViewData]
    public string Title { get; } = "About";

    public void OnGet()
    {
    }
}
```

In the About page, access the `Title` property as a model property:

CSHTML

```
<h1>@Model.Title</h1>
```

In the layout, the title is read from the ViewData dictionary:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

TempData

ASP.NET Core exposes the TempData. This property stores data until it's read. The Keep and Peek methods can be used to examine the data without deletion. `TempData` is useful for redirection, when data is needed for more than a single request.

The following code sets the value of `Message` using `TempData`:

C#

```
public class CreateDotModel : PageModel
{
    private readonly AppDbContext _db;

    public CreateDotModel(AppDbContext db)
```

```
    {
        _db = db;
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _db.Customers.Add(Customer);
        await _db.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";
        return RedirectToPage("./Index");
    }
}
```

The following markup in the `Pages/Customers/Index.cshtml` file displays the value of `Message` using `TempData`.

CSHTML

```
<h3>Msg: @Model.Message</h3>
```

The `Pages/Customers/Index.cshtml.cs` page model applies the `[TempData]` attribute to the `Message` property.

C#

```
[TempData]
public string Message { get; set; }
```

For more information, see TempData.

Multiple handlers per page

The following page generates markup for two handlers using the `asp-page-handler` Tag Helper:

CSHTML

```
@page
@model CreateFATHModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div><label>Name: <input asp-for="Customer.Name" /></label></div>
        <!-- <snippet_Handlers> -->
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC"
/>
        <!-- </snippet_Handlers> -->
    </form>
</body>
</html>
```

The form in the preceding example has two submit buttons, each using the `FormActionTagHelper` to submit to a different URL. The `asp-page-handler` attribute is a companion to `asp-page`. `asp-page-handler` generates URLs that submit to each of the handler methods defined by a page. `asp-page` isn't specified because the sample is linking to the current page.

The page model:

C#

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesContacts.Data;

namespace RazorPagesContacts.Pages.Customers
{
    public class CreateFATHModel : PageModel
    {
        private readonly AppDbContext _db;

        public CreateFATHModel(AppDbContext db)
        {
            _db = db;
        }

        [BindProperty]
        public Customer Customer { get; set; }

        public async Task<IActionResult> OnPostJoinListAsync()
```

```
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _db.Customers.Add(Customer);
            await _db.SaveChangesAsync();
            return RedirectToPage("/Index");
        }

        public async Task<IActionResult> OnPostJoinListUCAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }
            Customer.Name = Customer.Name?.ToUpperInvariant();
            return await OnPostJoinListAsync();
        }
    }
}
```

The preceding code uses *named handler methods*. Named handler methods are created by taking the text in the name after `On<HTTP Verb>` and before `Async` (if present). In the preceding example, the page methods are OnPost**JoinList**Async and OnPost**JoinListUC**Async. With *OnPost* and *Async* removed, the handler names are `JoinList` and `JoinListUC`.

CSHTML

```
<input type="submit" asp-page-handler="JoinList" value="Join" />
<input type="submit" asp-page-handler="JoinListUC" value="JOIN UC" />
```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH?handler=JoinListUC`.

Custom routes

Use the `@page` directive to:

- Specify a custom route to a page. For example, the route to the About page can be set to `/Some/Other/Path` with `@page "/Some/Other/Path"`.

- Append segments to a page's default route. For example, an "item" segment can be added to a page's default route with `@page "item"`.
- Append parameters to a page's default route. For example, an ID parameter, `id`, can be required for a page with `@page "{id}"`.

A root-relative path designated by a tilde (`~`) at the beginning of the path is supported. For example, `@page "~/Some/Other/Path"` is the same as `@page "/Some/Other/Path"`.

If you don't like the query string `?handler=JoinList` in the URL, change the route to put the handler name in the path portion of the URL. The route can be customized by adding a route template enclosed in double quotes after the `@page` directive.

CSHTML

```
@page "{handler?}"
@model CreateRouteModel

<html>
<body>
    <p>
        Enter your name.
    </p>
    <div asp-validation-summary="All"></div>
    <form method="POST">
        <div><label>Name: <input asp-for="Customer.Name" /></label></div>
        <input type="submit" asp-page-handler="JoinList" value="Join" />
        <input type="submit" asp-page-handler="JoinListUC" value="JOIN UC"
/>
    </form>
</body>
</html>
```

Using the preceding code, the URL path that submits to `OnPostJoinListAsync` is `https://localhost:5001/Customers/CreateFATH/JoinList`. The URL path that submits to `OnPostJoinListUCAsync` is `https://localhost:5001/Customers/CreateFATH/JoinListUC`.

The `?` following `handler` means the route parameter is optional.

Collocation of JavaScript (JS) files

Collocation of JavaScript (JS) files for pages and views is a convenient way to organize scripts in an app.

Collocate JS files using the following filename extension conventions:

- Pages of Razor Pages apps and views of MVC apps: `.cshtml.js`. Examples:

- `Pages/Index.cshtml.js` for the `Index` page of a Razor Pages app at `Pages/Index.cshtml`.
- `Views/Home/Index.cshtml.js` for the `Index` view of an MVC app at `Views/Home/Index.cshtml`.

Collocated JS files are publicly addressable using the ***path to the file in the project***:

- Pages and views from a collocated scripts file in the app:

  `{PATH}/{PAGE, VIEW, OR COMPONENT}.{EXTENSION}.js`

  - The `{PATH}` placeholder is the path to the page, view, or component.
  - The `{PAGE, VIEW, OR COMPONENT}` placeholder is the page, view, or component.
  - The `{EXTENSION}` placeholder matches the extension of the page, view, or component, either `razor` or `cshtml`.

  Razor Pages example:

  A JS file for the `Index` page is placed in the `Pages` folder (`Pages/Index.cshtml.js`) next to the `Index` page (`Pages/Index.cshtml`). In the `Index` page, the script is referenced at the path in the `Pages` folder:

  ```
  razor

  @section Scripts {
    <script src="~/Pages/Index.cshtml.js"></script>
  }
  ```

The default layout `Pages/Shared/_Layout.cshtml` can be configured to include collocated JS files, eliminating the need to configure each page individually:

```
razor

<script asp-src-include="@(ViewContext.View.Path).js"></script>
```

The sample download uses the preceding code snippet to include collocated JS files in the default layout.

When the app is published, the framework automatically moves the script to the web root. In the preceding example, the script is moved to `bin\Release\{TARGET FRAMEWORK MONIKER}\publish\wwwroot\Pages\Index.cshtml.js`, where the `{TARGET FRAMEWORK MONIKER}` placeholder is the Target Framework Moniker (TFM). No change is required to the script's relative URL in the `Index` page.

When the app is published, the framework automatically moves the script to the web root. In the preceding example, the script is moved to `bin\Release\{TARGET FRAMEWORK MONIKER}\publish\wwwroot\Components\Pages\Index.razor.js`, where the `{TARGET FRAMEWORK MONIKER}` placeholder is the Target Framework Moniker (TFM). No change is required to the script's relative URL in the `Index` component.

- For scripts provided by a Razor class library (RCL):

  `_content/{PACKAGE ID}/{PATH}/{PAGE, VIEW, OR COMPONENT}.{EXTENSION}.js`

  - The `{PACKAGE ID}` placeholder is the RCL's package identifier (or library name for a class library referenced by the app).
  - The `{PATH}` placeholder is the path to the page, view, or component. If a Razor component is located at the root of the RCL, the path segment isn't included.
  - The `{PAGE, VIEW, OR COMPONENT}` placeholder is the page, view, or component.
  - The `{EXTENSION}` placeholder matches the extension of page, view, or component, either `razor` or `cshtml`.

Advanced configuration and settings

The configuration and settings in following sections is not required by most apps.

To configure advanced options, use the AddRazorPages overload that configures RazorPagesOptions:

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
{
    options.RootDirectory = "/MyPages";
    options.Conventions.AuthorizeFolder("/MyPages/Admin");
});

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
```

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Use the RazorPagesOptions to set the root directory for pages, or add application model conventions for pages. For more information on conventions, see Razor Pages authorization conventions.

To precompile views, see Razor view compilation.

Specify that Razor Pages are at the content root

By default, Razor Pages are rooted in the */Pages* directory. Add WithRazorPagesAtContentRoot to specify that your Razor Pages are at the content root (ContentRootPath) of the app:

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/MyPages/Admin");
})
  .WithRazorPagesAtContentRoot();

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Specify that Razor Pages are at a custom root directory

Add WithRazorPagesRoot to specify that Razor Pages are at a custom root directory in the app (provide a relative path):

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesContacts.Data;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
{
    options.Conventions.AuthorizeFolder("/MyPages/Admin");
})
  .WithRazorPagesRoot("/path/to/razor/pages");

builder.Services.AddDbContext<CustomerDbContext>(options =>
    options.UseInMemoryDatabase("name"));

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Additional resources

- See Get started with Razor Pages, which builds on this introduction.
- Authorize attribute and Razor Pages
- Download or view sample code
- Overview of ASP.NET Core
- Razor syntax reference for ASP.NET Core
- Areas in ASP.NET Core
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Razor Pages authorization conventions in ASP.NET Core
- Razor Pages route and app conventions in ASP.NET Core
- Razor Pages unit tests in ASP.NET Core
- Partial views in ASP.NET Core

Tutorial: Create a Razor Pages web app with ASP.NET Core

Article • 06/23/2024

This series of tutorials explains the basics of building a Razor Pages web app.

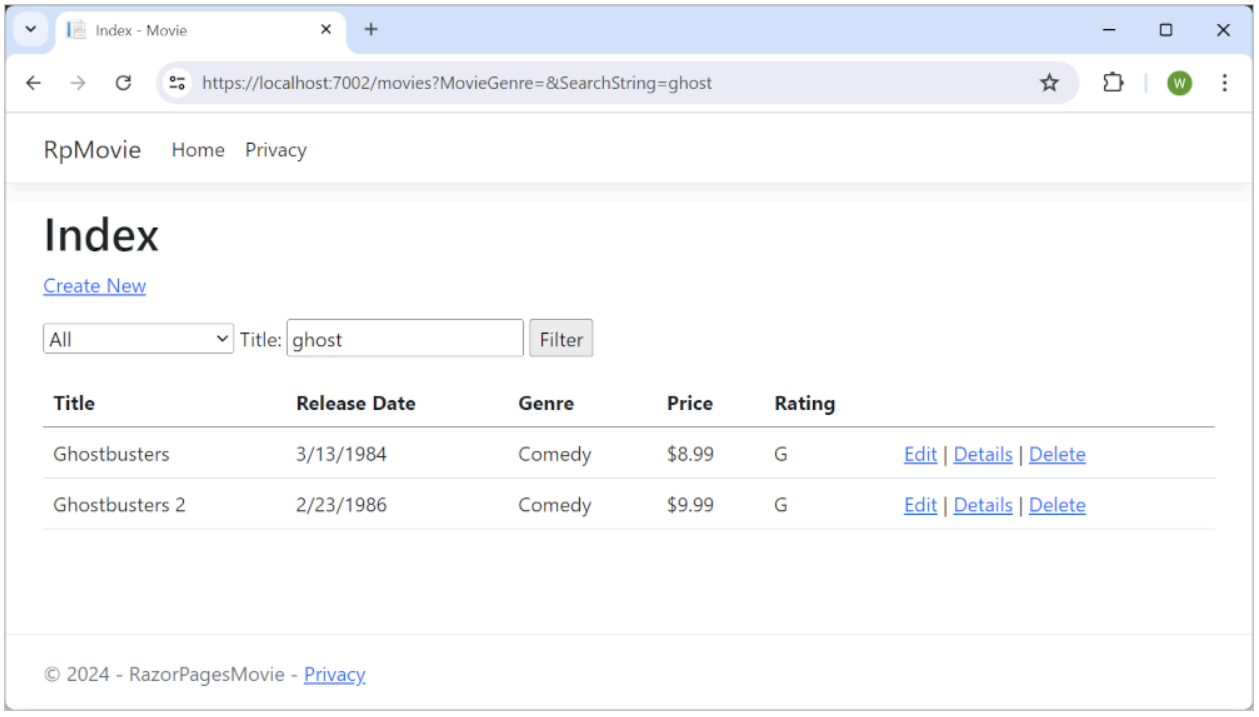
For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages in ASP.NET Core.

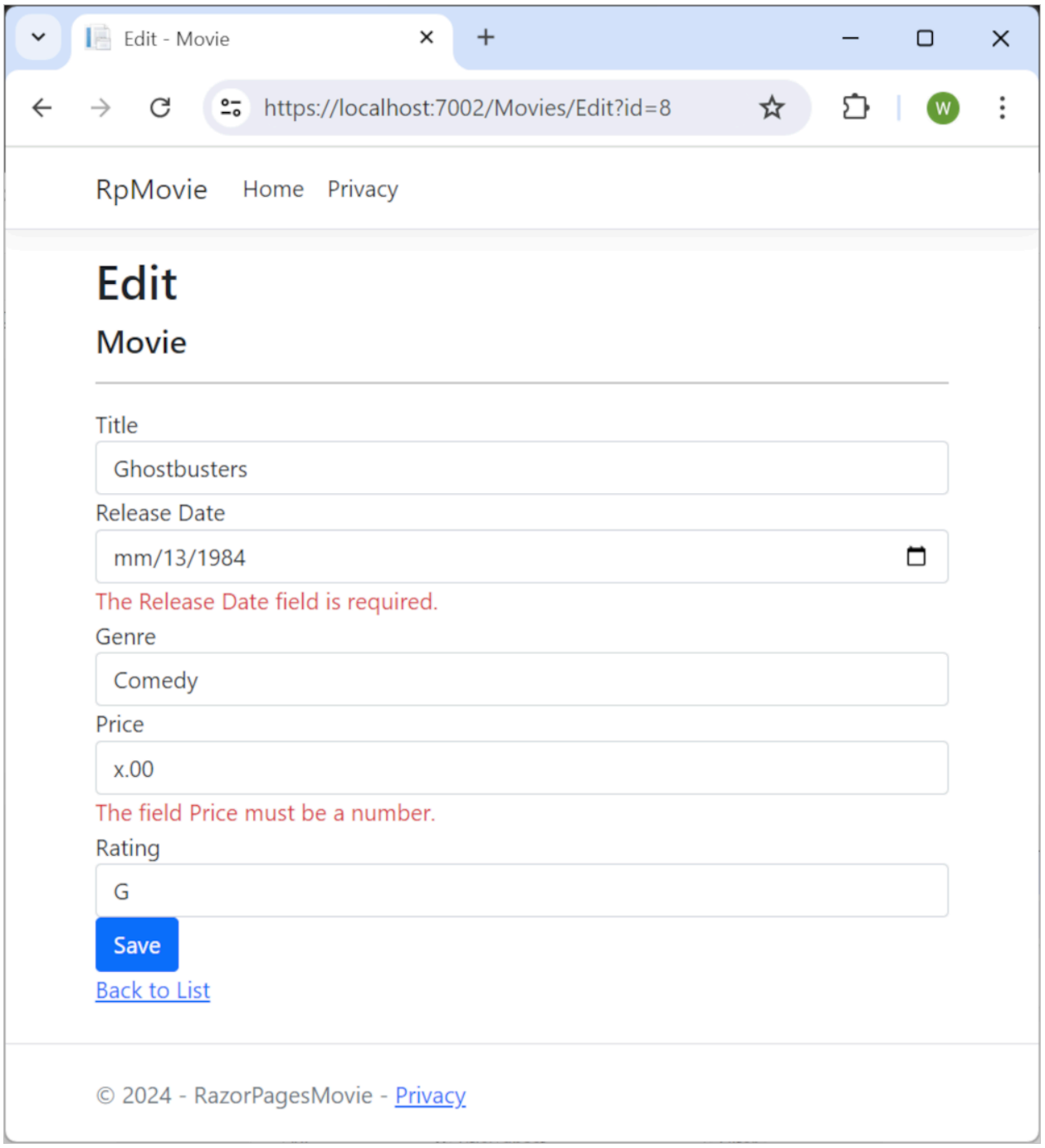
If you're new to ASP.NET Core development and are unsure of which ASP.NET Core web UI solution will best fit your needs, see Choose an ASP.NET Core UI.

This series includes the following tutorials:

1. Create a Razor Pages web app
2. Add a model to a Razor Pages app
3. Scaffold (generate) Razor pages
4. Work with a database
5. Update Razor pages
6. Add search
7. Add a new field
8. Add validation

At the end, you'll have an app that can display and manage a database of movies.





Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Get started with Razor Pages in ASP.NET Core

Article • 08/05/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

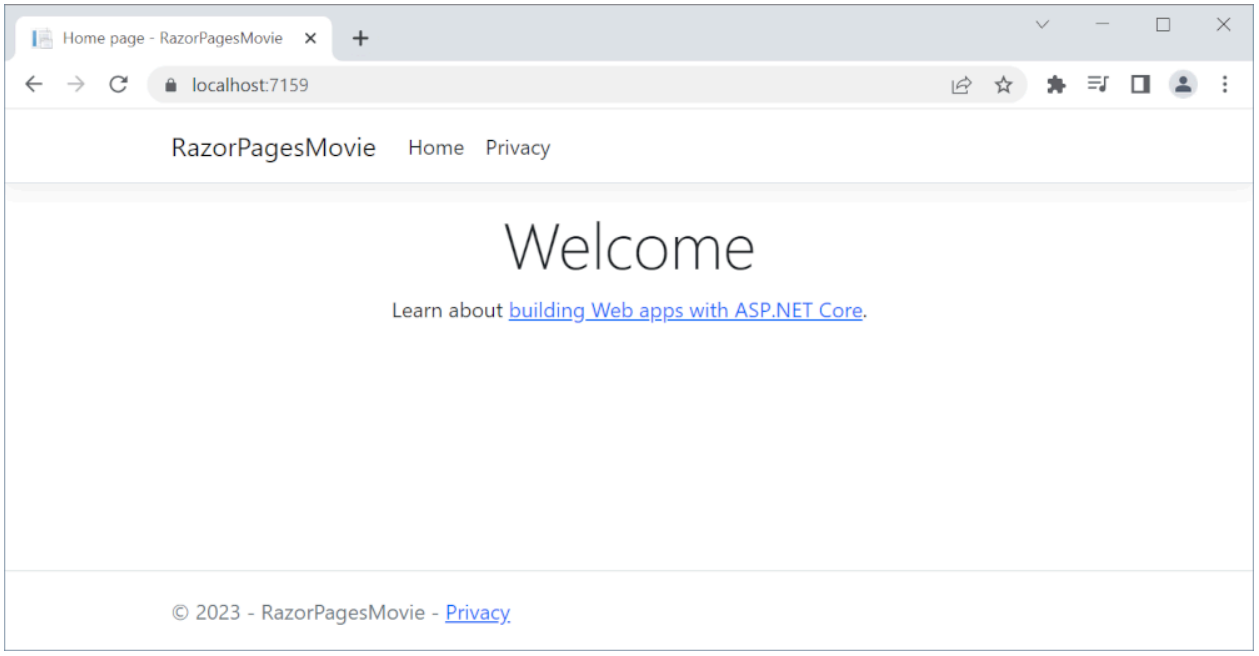
By Rick Anderson

This is the first tutorial of a series that teaches the basics of building an ASP.NET Core Razor Pages web app.

For a more advanced introduction aimed at developers who are familiar with controllers and views, see Introduction to Razor Pages. For a video introduction, see Entity Framework Core for Beginners.

If you're new to ASP.NET Core development and are unsure of which ASP.NET Core web UI solution will best fit your needs, see Choose an ASP.NET Core UI.

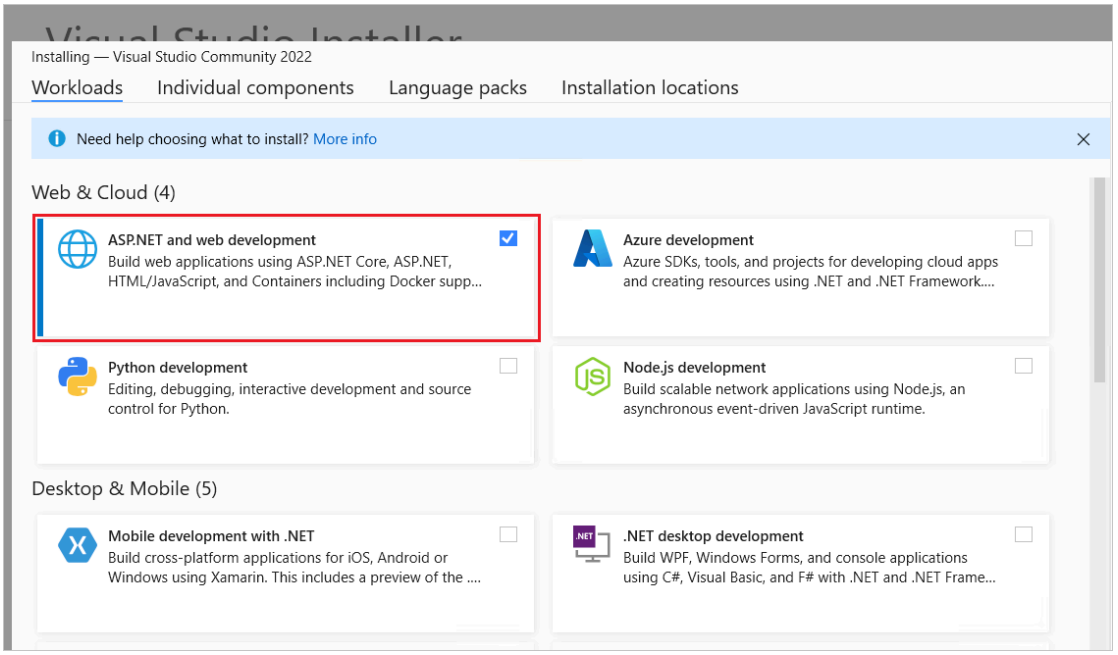
At the end of this tutorial, you'll have a Razor Pages web app that manages a database of movies.



Prerequisites

Visual Studio

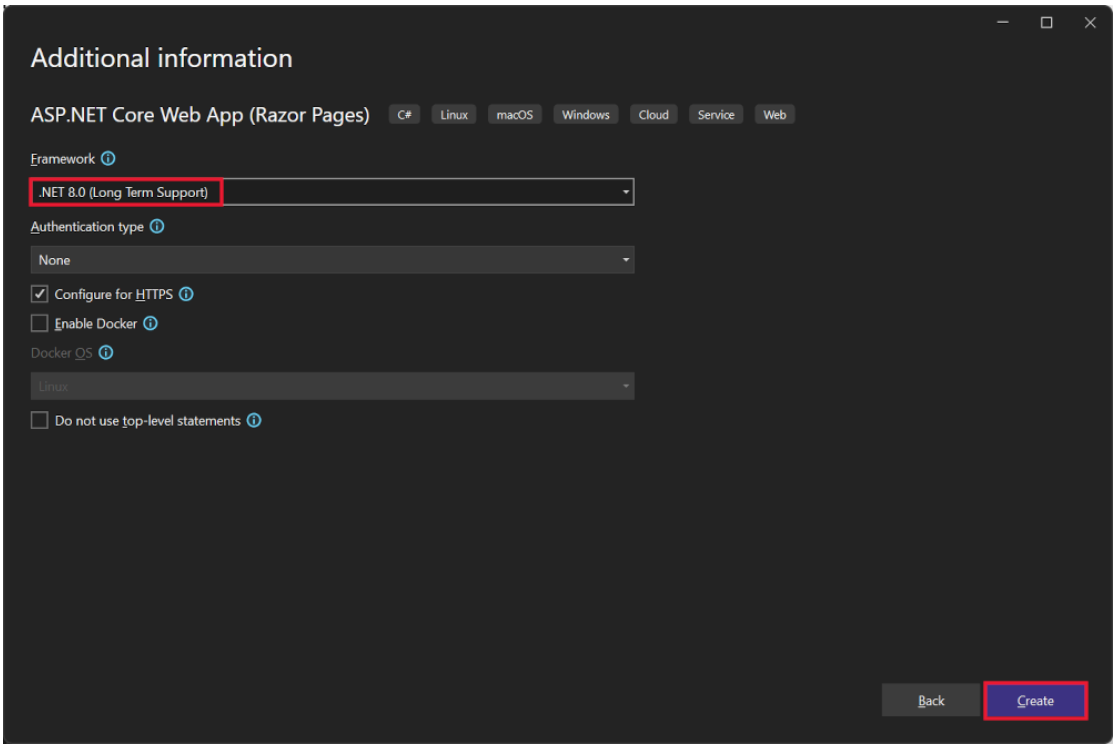
- Visual Studio 2022 with the **ASP.NET and web development** workload.



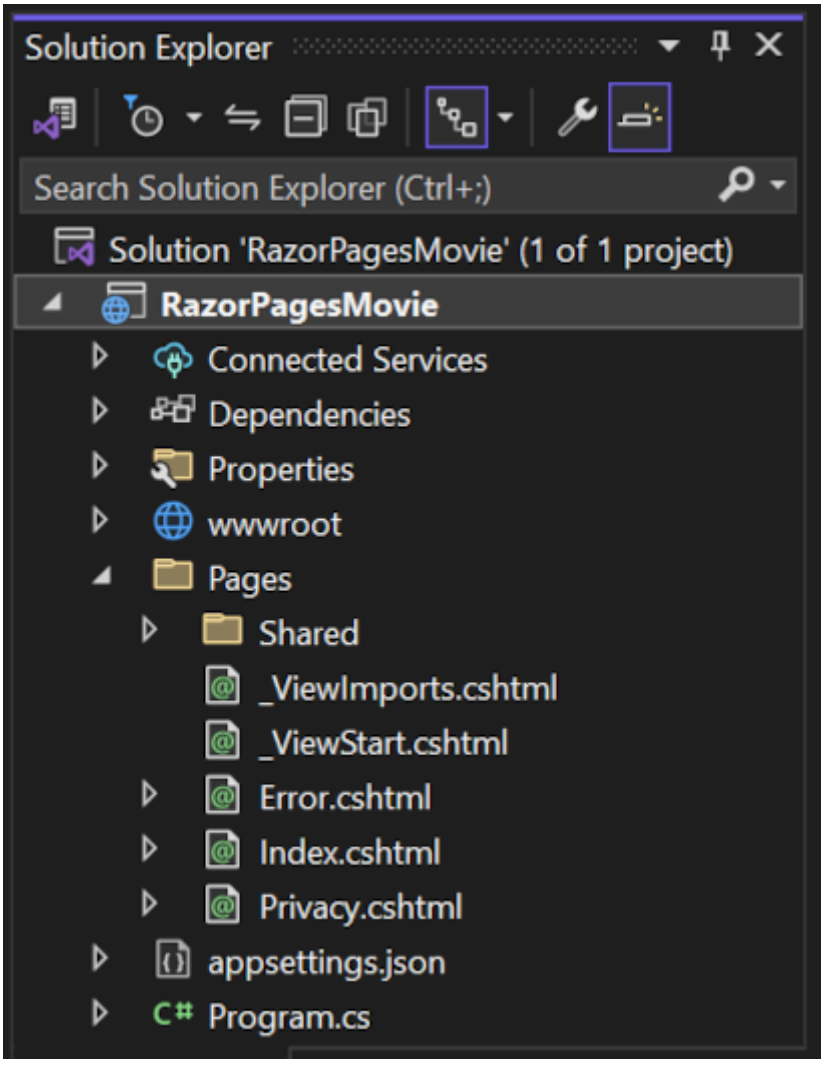
Create a Razor Pages web app

Visual Studio

- Start Visual Studio and select **New project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web App (Razor Pages)** > **Next**.
- In the **Configure your new project** dialog, enter `RazorPagesMovie` for **Project name**. It's important to name the project **RazorPagesMovie**, including matching the capitalization, so the namespaces will match when you copy and paste example code.
- Select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 8.0 (Long Term Support)**.
  - Verify: **Do not use top-level statements** is unchecked.
- Select **Create**.



The following starter project is created:



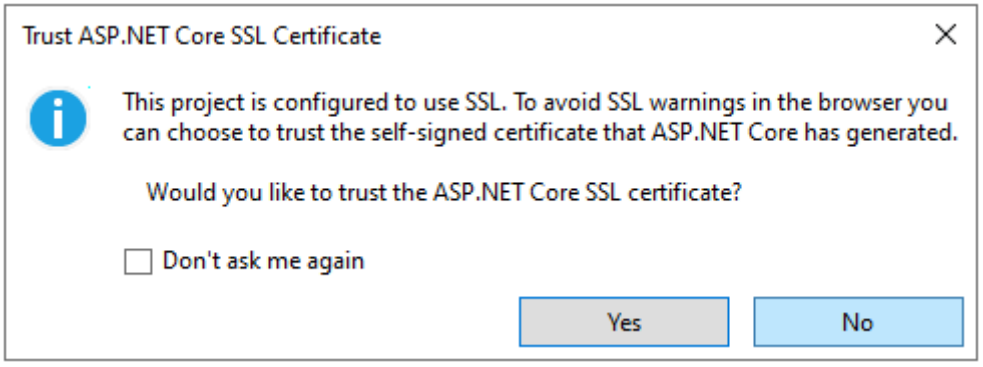
For alternative approaches to create the project, see Create a new project in Visual Studio.

Run the app

Visual Studio

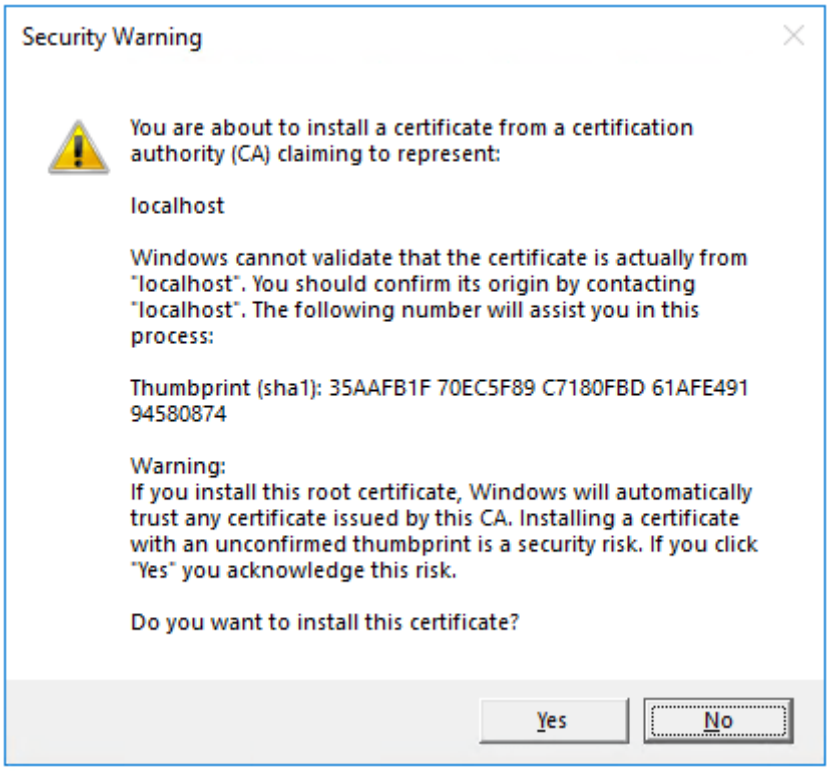
Select **RazorPagesMovie** in **Solution Explorer**, and then press `Ctrl`+`F5` to run the app without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio:

- Runs the app, which launches the Kestrel server.
- Launches the default browser at `https://localhost:<port>`, which displays the apps UI. `<port>` is the random port that is assigned when the app was created.

Close the browser window.

Examine the project files

The following sections contain an overview of the main project folders and files that you'll work with in later tutorials.

Pages folder

Contains Razor pages and supporting files. Each Razor page is a pair of files:

- A `.cshtml` file that has HTML markup with C# code using Razor syntax.
- A `.cshtml.cs` file that has C# code that handles page events.

Supporting files have names that begin with an underscore. For example, the `_Layout.cshtml` file configures UI elements common to all pages. `_Layout.cshtml` sets up the navigation menu at the top of the page and the copyright notice at the bottom of the page. For more information, see Layout in ASP.NET Core.

wwwroot folder

Contains static assets, like HTML files, JavaScript files, and CSS files. For more information, see Static files in ASP.NET Core.

`appsettings.json`

Contains configuration data, like connection strings. For more information, see Configuration in ASP.NET Core.

Program.cs

Contains the following code:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The following lines of code in this file create a `WebApplicationBuilder` with preconfigured defaults, add Razor Pages support to the Dependency Injection (DI) container, and builds the app:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();
```

The developer exception page is enabled by default and provides helpful information on exceptions. Production apps should not be run in development mode because the developer exception page can leak sensitive information.

The following code sets the exception endpoint to `/Error` and enables HTTP Strict Transport Security Protocol (HSTS) when the app is ***not*** running in development mode:

C#

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
```

For example, the preceding code runs when the app is in production or test mode. For more information, see Use multiple environments in ASP.NET Core.

The following code enables various Middleware:

- `app.UseHttpsRedirection();` : Redirects HTTP requests to HTTPS.
- `app.UseStaticFiles();` : Enables static files, such as HTML, CSS, images, and JavaScript to be served. For more information, see Static files in ASP.NET Core.
- `app.UseRouting();` : Adds route matching to the middleware pipeline. For more information, see Routing in ASP.NET Core
- `app.MapRazorPages();`: Configures endpoint routing for Razor Pages.
- `app.UseAuthorization();` : Authorizes a user to access secure resources. This app doesn't use authorization, therefore this line could be removed.
- `app.Run();` : Runs the app.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

Next: Add a model

Part 2, add a model to a Razor Pages app in ASP.NET Core

Article • 08/04/2024

In this tutorial, classes are added for managing movies in a database. The app's model classes use Entity Framework Core (EF Core) to work with the database. EF Core is an object-relational mapper (O/RM) that simplifies data access. You write the model classes first, and EF Core creates the database.

The model classes are known as POCO classes (from "**P**lain-**O**ld **C**LR **O**bjects") because they don't have a dependency on EF Core. They define the properties of the data that are stored in the database.

Add a data model

Visual Studio

1. In **Solution Explorer**, right-click the *RazorPagesMovie* project > **Add** > **New Folder**. Name the folder `Models`.
2. Right-click the `Models` folder. Select **Add** > **Class**. Name the class *Movie*.
3. Add the following properties to the `Movie` class:

C#

```
using System.ComponentModel.DataAnnotations;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    public decimal Price { get; set; }
}
```

The `Movie` class contains:

- The `ID` field is required by the database for the primary key.
- A [DataType] attribute that specifies the type of data in the `ReleaseDate` property. With this attribute:
  - The user isn't required to enter time information in the date field.
  - Only the date is displayed, not time information.
- The question mark after `string` indicates that the property is nullable. For more information, see Nullable reference types.

DataAnnotations are covered in a later tutorial.

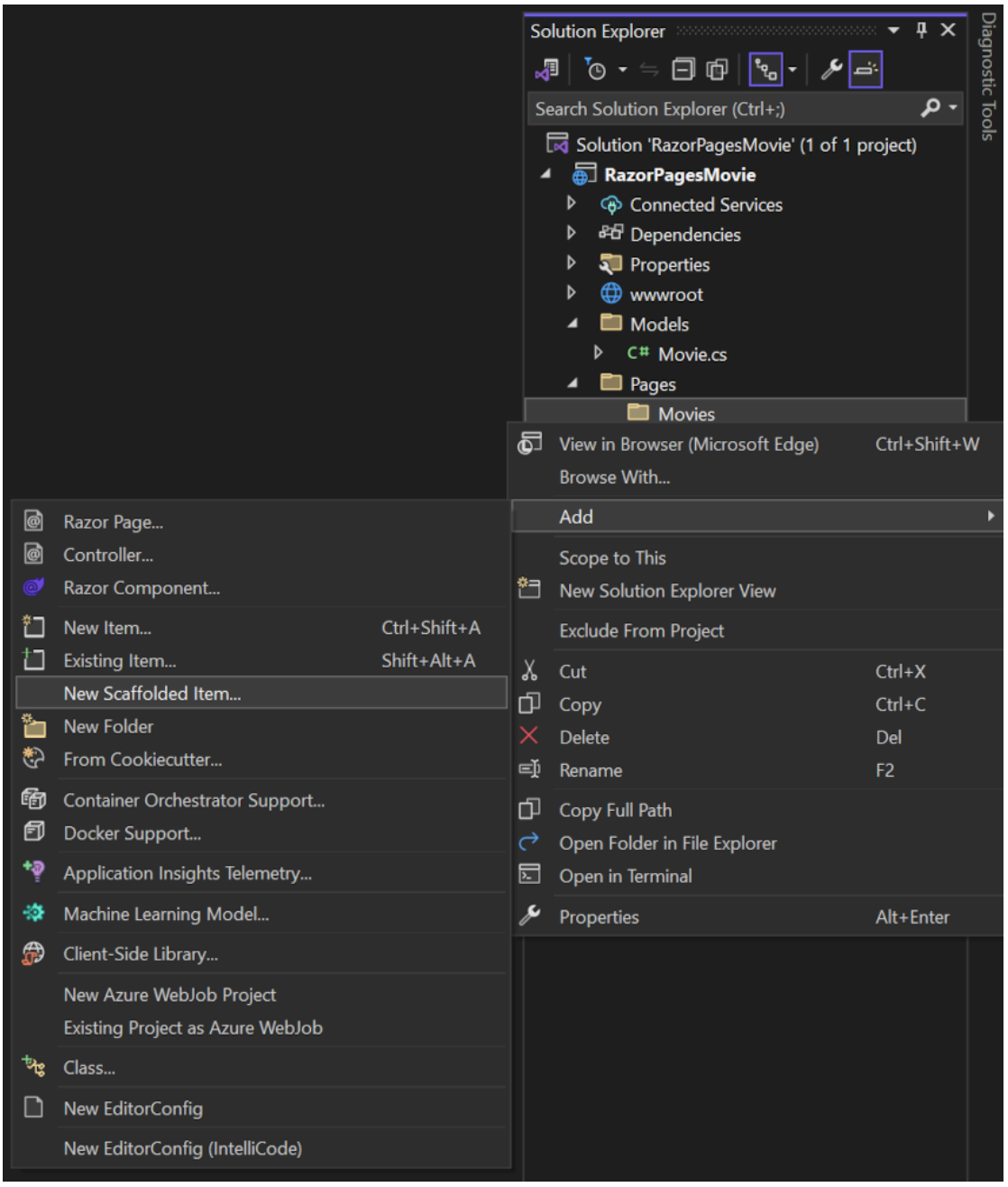
Build the project to verify there are no compilation errors.

Scaffold the movie model

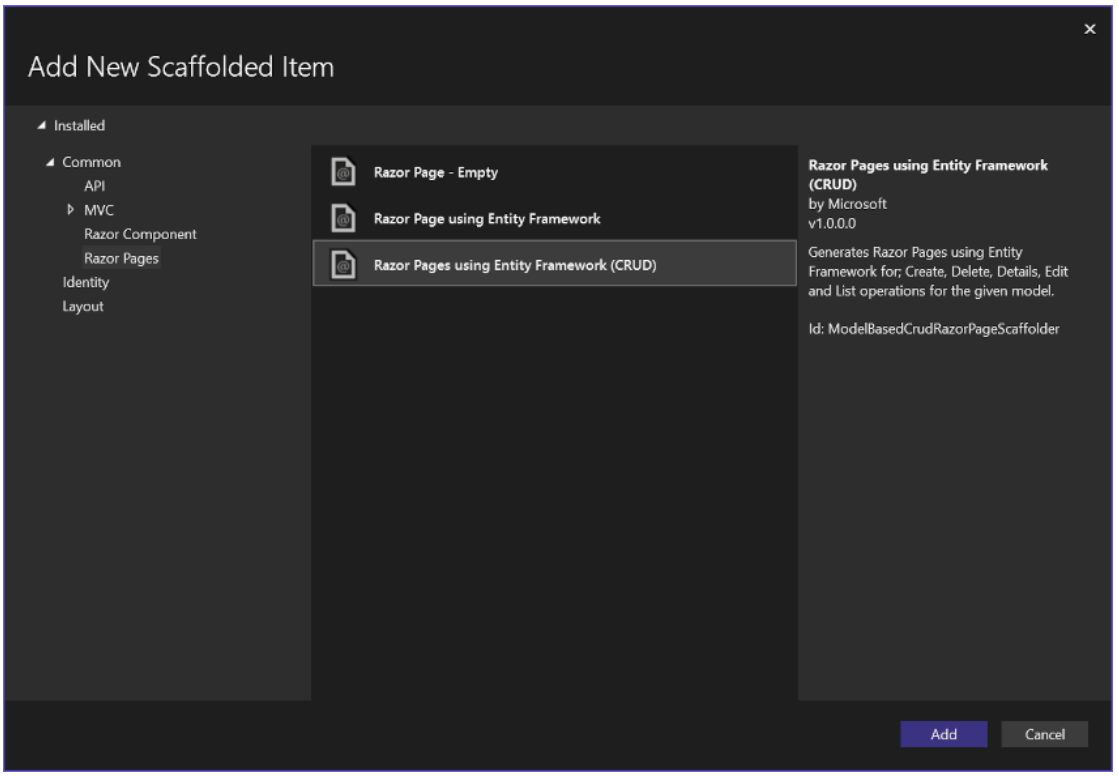
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

Visual Studio

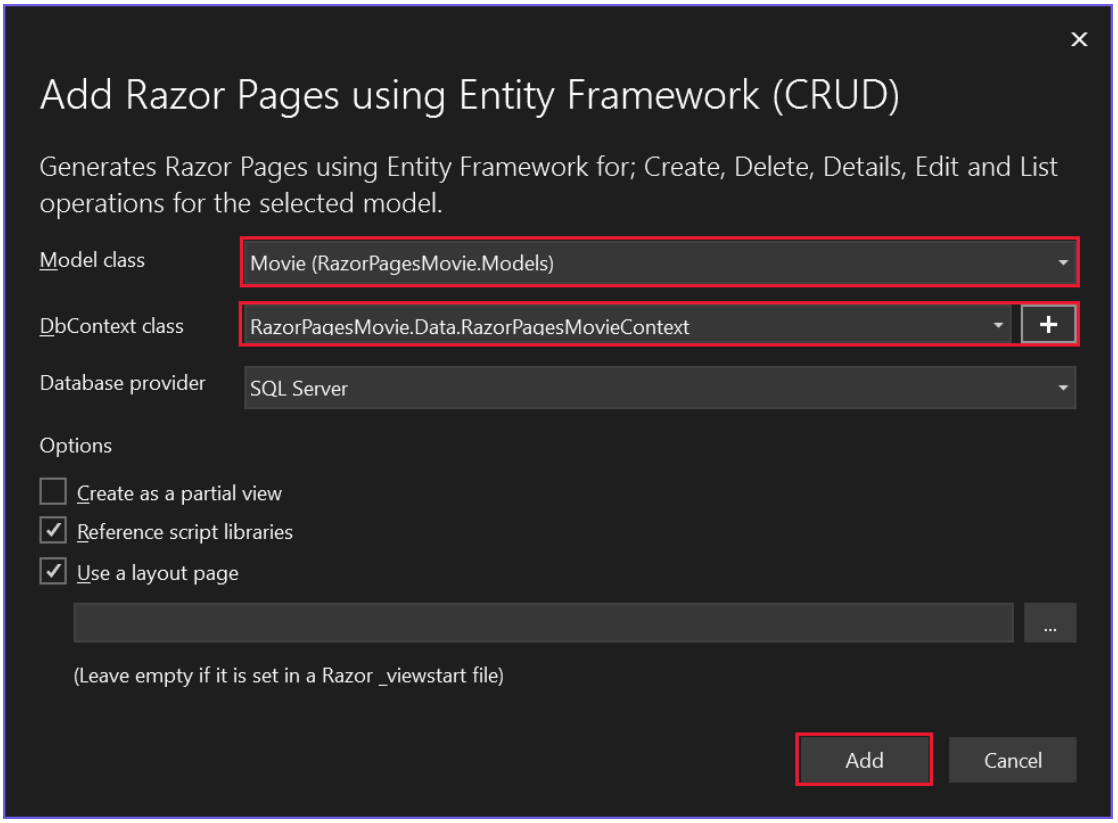
1. Create the *Pages/Movies* folder:
   a. Right-click on the *Pages* folder > **Add** > **New Folder**.
   b. Name the folder *Movies*.
2. Right-click on the *Pages/Movies* folder > **Add** > **New Scaffolded Item**.



3. In the **Add New Scaffold** dialog, select **Razor Pages using Entity Framework (CRUD)** > **Add**.



4. Complete the **Add Razor Pages using Entity Framework (CRUD)** dialog:
   a. In the **Model class** drop down, select **Movie (RazorPagesMovie.Models)**.
   b. In the **Data context class** row, select the + (plus) sign.
      i. In the **Add Data Context** dialog, the class name `RazorPagesMovie.Data.RazorPagesMovieContext` is generated.
      ii. In the **Database provider** drop down, select **SQL Server**.
   c. Select **Add**.



The `appsettings.json` file is updated with the connection string used to connect to a local database.

> ⚠ **Warning**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Files created and updated

The scaffold process creates the following files:

- *Pages/Movies*: Create, Delete, Details, Edit, and Index.
- `Data/RazorPagesMovieContext.cs`

The created files are explained in the next tutorial.

The scaffold process adds the following highlighted code to the `Program.cs` file:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
```

```
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The `Program.cs` changes are explained later in this tutorial.

Create the initial database schema using EF's migration feature

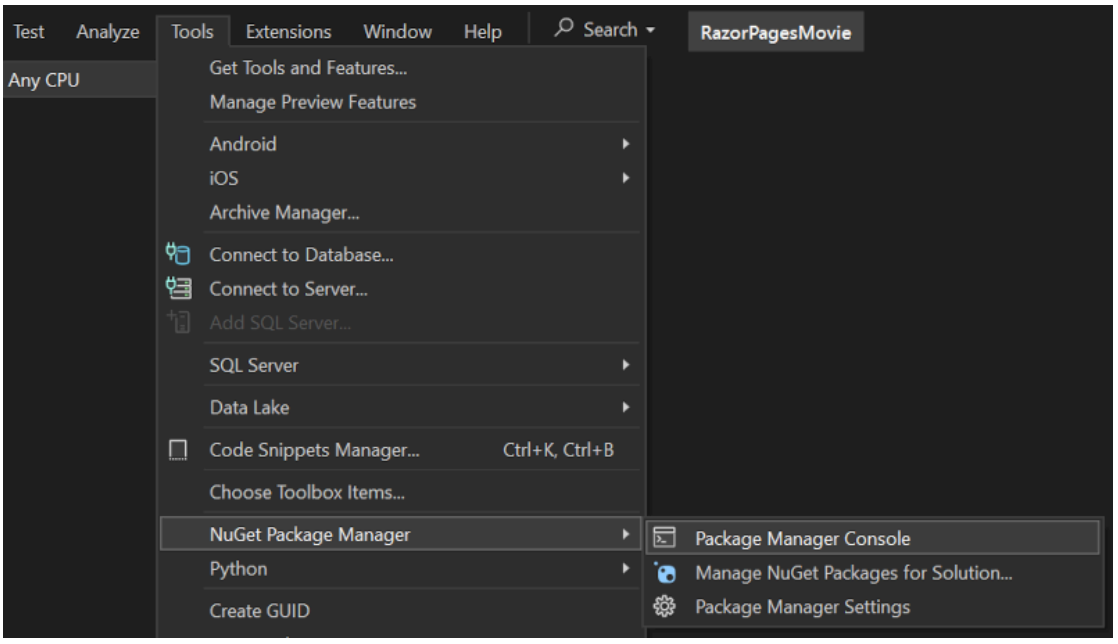
The migrations feature in Entity Framework Core provides a way to:

- Create the initial database schema.
- Incrementally update the database schema to keep it in sync with the app's data model. Existing data in the database is preserved.

Visual Studio

In this section, the **Package Manager Console** (PMC) window is used to:

- Add an initial migration.
- Update the database with the initial migration.
- From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



- In the PMC, enter the following command:

```
PowerShell

Add-Migration InitialCreate
```

- The `Add-Migration` command generates code to create the initial database schema. The schema is based on the model specified in `DbContext`. The `InitialCreate` argument is used to name the migration. Any name can be used, but by convention a name is selected that describes the migration.

The following warning is displayed, which is addressed in a later step:

> No type was specified for the decimal column 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values using 'HasColumnType()'.

- In the PMC, enter the following command:

```
PowerShell

Update-Database
```

The `Update-Database` command runs the `Up` method in migrations that have not been applied. In this case, the command runs the `Up` method in the `Migrations/<time-stamp>_InitialCreate.cs` file, which creates the database.

The data context `RazorPagesMovieContext`:

- Derives from Microsoft.EntityFrameworkCore.DbContext.
- Specifies which entities are included in the data model.
- Coordinates EF Core functionality, such as Create, Read, Update and Delete, for the `Movie` model.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Data
{
    public class RazorPagesMovieContext : DbContext
    {
        public RazorPagesMovieContext
(DbContextOptions<RazorPagesMovieContext> options)
            : base(options)
        {
        }

        public DbSet<RazorPagesMovie.Models.Movie> Movie { get; set; } =
default!;
    }
}
```

The preceding code creates a DbSet<Movie> property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the Configuration system reads the connection string from the `appsettings.json` file.

Test the app

1. Run the app and append `/Movies` to the URL in the browser (`http://localhost:port/movies`).

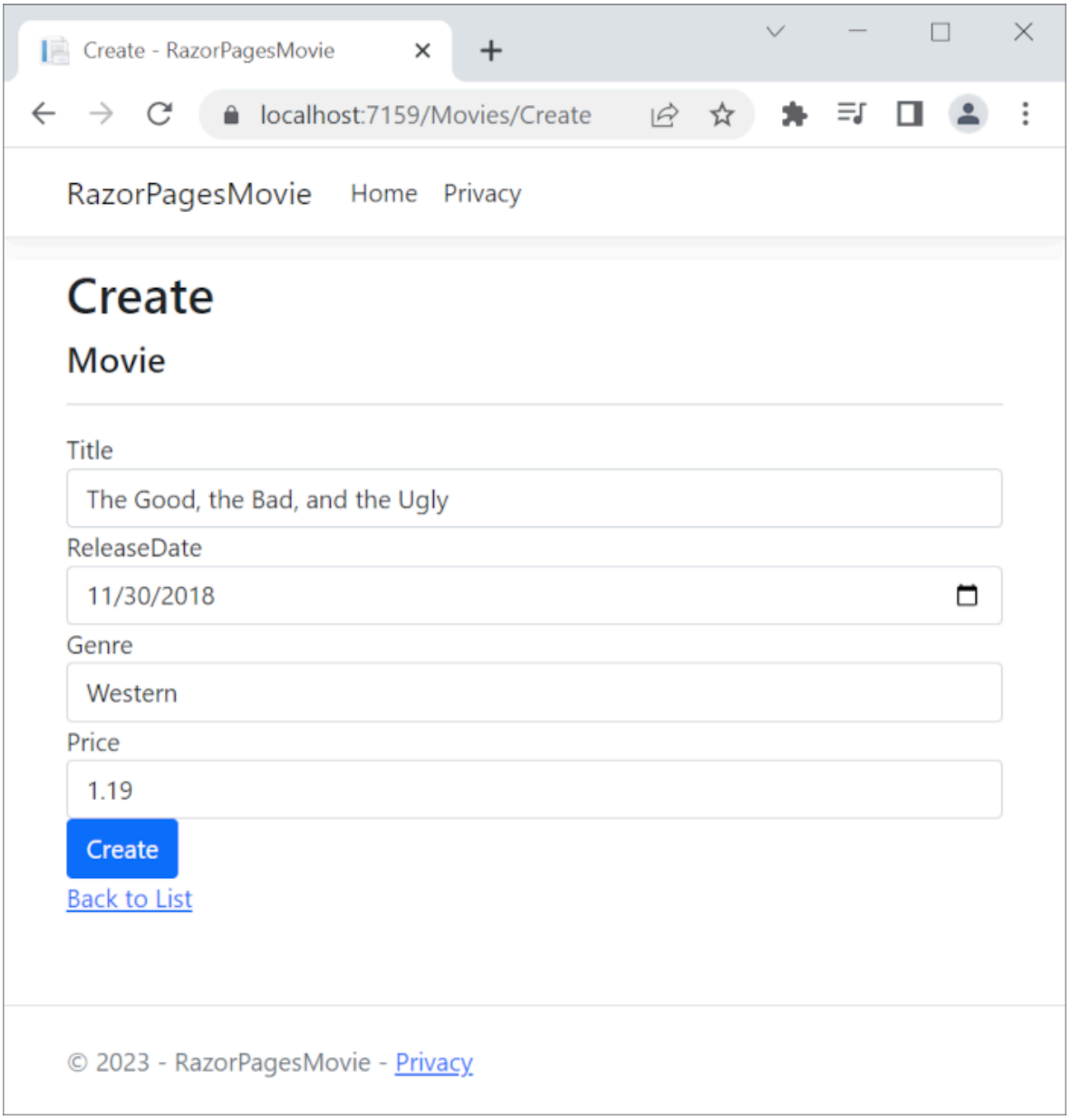
   If you receive the following error:

   Console

```
SqlException: Cannot open database "RazorPagesMovieContext-GUID"
requested by the login. The login failed.
Login failed for user 'User-name'.
```

You missed the migrations step.

2. Test the **Create New** link.



> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

3. Test the **Edit**, **Details**, and **Delete** links.

The next tutorial explains the files created by scaffolding.

Examine the context registered with dependency injection

ASP.NET Core is built with dependency injection. Services, such as the EF Core database context, are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically created a database context and registered it with the dependency injection container. The following highlighted code is added to the `Program.cs` file by the scaffolder:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));
var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();
```

```
app.MapRazorPages();

app.Run();
```

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

Previous: Get Started | Next: Scaffolded Razor Pages

Part 3, scaffolded Razor Pages in ASP.NET Core

Article • 06/23/2024

By Rick Anderson

This tutorial examines the Razor Pages created by scaffolding in the previous tutorial.

The Create, Delete, Details, and Edit pages

Examine the `Pages/Movies/Index.cshtml.cs` Page Model:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Models;

namespace RazorPagesMovie.Pages.Movies;

public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get;set; }  = default!;

    public async Task OnGetAsync()
    {
        if (_context.Movie != null)
        {
            Movie = await _context.Movie.ToListAsync();
        }
    }
}
```

Razor Pages are derived from PageModel. By convention, the `PageModel` derived class is named `PageNameModel`. For example, the Index page is named `IndexModel`.

The constructor uses dependency injection to add the `RazorPagesMovieContext` to the page:

C#

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }
```

See Asynchronous code for more information on asynchronous programming with Entity Framework.

When a `GET` request is made for the page, the `OnGetAsync` method returns a list of movies to the Razor Page. On a Razor Page, `OnGetAsync` or `OnGet` is called to initialize the state of the page. In this case, `OnGetAsync` gets a list of movies and displays them.

When `OnGet` returns `void` or `OnGetAsync` returns `Task`, no return statement is used. For example, examine the Privacy Page:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace RazorPagesMovie.Pages
{
    public class PrivacyModel : PageModel
    {
        private readonly ILogger<PrivacyModel> _logger;

        public PrivacyModel(ILogger<PrivacyModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

When the return type is IActionResult or `Task<IActionResult>`, a return statement must be provided. For example, the `Pages/Movies/Create.cshtml.cs OnPostAsync` method:

C#

```
public async Task<IActionResult> OnPostAsync()
{
  if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Examine the `Pages/Movies/Index.cshtml` Razor Page:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
```

```
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor can transition from HTML into C# or into Razor-specific markup. When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup, otherwise it transitions into C#.

The @page directive

The `@page` Razor directive makes the file an MVC action, which means that it can handle requests. `@page` must be the first Razor directive on a page. `@page` and `@model` are examples of transitioning into Razor-specific markup. See Razor syntax for more information.

The @model directive

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel
```

The `@model` directive specifies the type of the model passed to the Razor Page. In the preceding example, the `@model` line makes the `PageModel` derived class available to the Razor Page. The model is used in the `@Html.DisplayNameFor` and `@Html.DisplayFor` HTML Helpers on the page.

Examine the lambda expression used in the following HTML Helper:

CSHTML

```
@Html.DisplayNameFor(model => model.Movie[0].Title)
```

The DisplayNameFor HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. The lambda expression is inspected rather than evaluated. That means there is no access violation when `model`, `model.Movie`, or `model.Movie[0]` is `null` or empty. When the lambda expression is evaluated, for example, with `@Html.DisplayFor(modelItem => item.Title)`, the model's property values are evaluated.

The layout page

Select the menu links **RazorPagesMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the `Pages/Shared/_Layout.cshtml` file.

Open and examine the `Pages/Shared/_Layout.cshtml` file.

Layout templates allow the HTML container layout to be:

- Specified in one place.
- Applied in multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the page-specific views show up, *wrapped* in the layout page. For example, select the **Privacy** link and the `Pages/Privacy.cshtml` view is rendered inside the `RenderBody` method.

ViewData and layout

Consider the following markup from the `Pages/Movies/Index.cshtml` file:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

The preceding highlighted markup is an example of Razor transitioning into C#. The `{` and `}` characters enclose a block of C# code.

The `PageModel` base class contains a `ViewData` dictionary property that can be used to pass data to a View. Objects are added to the `ViewData` dictionary using a ***key value*** pattern. In the preceding sample, the `Title` property is added to the `ViewData` dictionary.

The `Title` property is used in the `Pages/Shared/_Layout.cshtml` file. The following markup shows the first few lines of the `_Layout.cshtml` file.

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - RazorPagesMovie</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css"
/>
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/RazorPagesMovie.styles.css" asp-append-
version="true" />
```

Update the layout

1. Change the `<title>` element in the `Pages/Shared/_Layout.cshtml` file to display **Movie** rather than **RazorPagesMovie**.

   CSHTML

   ```
   <!DOCTYPE html>
   <html lang="en">
   <head>
       <meta charset="utf-8" />
       <meta name="viewport" content="width=device-width, initial-
   scale=1.0" />
       <title>@ViewData["Title"] - Movie</title>
   ```

2. Find the following anchor element in the `Pages/Shared/_Layout.cshtml` file.

   CSHTML

   ```
   <a class="navbar-brand" asp-area="" asp-
   page="/Index">RazorPagesMovie</a>
   ```

3. Replace the preceding element with the following markup:

```cshtml
<a class="navbar-brand" asp-page="/Movies/Index">RpMovie</a>
```

The preceding anchor element is a Tag Helper. In this case, it's the Anchor Tag Helper. The `asp-page="/Movies/Index"` Tag Helper attribute and value creates a link to the `/Movies/Index` Razor Page. The `asp-area` attribute value is empty, so the area isn't used in the link. See Areas for more information.

4. Save the changes and test the app by selecting the **RpMovie** link. See the _Layout.cshtml file in GitHub if you have any problems.

5. Test the **Home**, **RpMovie**, **Create**, **Edit**, and **Delete** links. Each page sets the title, which you can see in the browser tab. When you bookmark a page, the title is used for the bookmark.

> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize the app. See this GitHub issue 4076 for instructions on adding decimal comma.

The `Layout` property is set in the `Pages/_ViewStart.cshtml` file:

```cshtml
@{
    Layout = "_Layout";
}
```

The preceding markup sets the layout file to `Pages/Shared/_Layout.cshtml` for all Razor files under the *Pages* folder. See Layout for more information.

The Create page model

Examine the `Pages/Movies/Create.cshtml.cs` page model:

```C#
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesMovie.Models;
```

```
namespace RazorPagesMovie.Pages.Movies
{
    public class CreateModel : PageModel
    {
        private readonly RazorPagesMovie.Data.RazorPagesMovieContext
_context;

        public CreateModel(RazorPagesMovie.Data.RazorPagesMovieContext
context)
        {
            _context = context;
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Movie Movie { get; set; } = default!;


        // To protect from overposting attacks, see
https://aka.ms/RazorPagesCRUD
        public async Task<IActionResult> OnPostAsync()
        {
          if (!ModelState.IsValid || _context.Movie == null || Movie ==
null)
            {
                return Page();
            }

            _context.Movie.Add(Movie);
            await _context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

The `OnGet` method initializes any state needed for the page. The Create page doesn't have any state to initialize, so `Page` is returned. Later in the tutorial, an example of `OnGet` initializing state is shown. The `Page` method creates a `PageResult` object that renders the `Create.cshtml` page.

The `Movie` property uses the [BindProperty] attribute to opt-in to model binding. When the Create form posts the form values, the ASP.NET Core runtime binds the posted values to the `Movie` model.

The `OnPostAsync` method is run when the page posts form data:

C#

```
public async Task<IActionResult> OnPostAsync()
{
  if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movie.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

If there are any model errors, the form is redisplayed, along with any form data posted. Most model errors can be caught on the client-side before the form is posted. An example of a model error is posting a value for the date field that cannot be converted to a date. Client-side validation and model validation are discussed later in the tutorial.

If there are no model errors:

- The data is saved.
- The browser is redirected to the Index page.

The Create Razor Page

Examine the `Pages/Movies/Create.cshtml` Razor Page file:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
```

```
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label">
</label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Visual Studio

Visual Studio displays the following tags in a distinctive bold font used for Tag Helpers:

- `<form method="post">`
- `<div asp-validation-summary="ModelOnly" class="text-danger"></div>`
- `<label asp-for="Movie.Title" class="control-label"></label>`
- `<input asp-for="Movie.Title" class="form-control" />`
- `<span asp-validation-for="Movie.Title" class="text-danger"></span>`

Create.cshtml

```
@page
@model RazorPagesMovie.Pages.Movies.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Movie.Title" class="control-label"></label>
                <input asp-for="Movie.Title" class="form-control" />
                <span asp-validation-for="Movie.Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.ReleaseDate" class="control-label"></label>
                <input asp-for="Movie.ReleaseDate" class="form-control" />
                <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Genre" class="control-label"></label>
                <input asp-for="Movie.Genre" class="form-control" />
                <span asp-validation-for="Movie.Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Movie.Price" class="control-label"></label>
                <input asp-for="Movie.Price" class="form-control" />
                <span asp-validation-for="Movie.Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
```

The `<form method="post">` element is a Form Tag Helper. The Form Tag Helper automatically includes an antiforgery token.

The scaffolding engine creates Razor markup for each field in the model, except the ID, similar to the following:

CSHTML

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<div class="form-group">
    <label asp-for="Movie.Title" class="control-label"></label>
    <input asp-for="Movie.Title" class="form-control" />
    <span asp-validation-for="Movie.Title" class="text-danger"></span>
</div>
```

The Validation Tag Helpers (`<div asp-validation-summary` and `<span asp-validation-for`) display validation errors. Validation is covered in more detail later in this series.

The Label Tag Helper (`<label asp-for="Movie.Title" class="control-label"></label>`) generates the label caption and `[for]` attribute for the `Title` property.

The Input Tag Helper (`<input asp-for="Movie.Title" class="form-control">`) uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side.

For more information on Tag Helpers such as `<form method="post">`, see Tag Helpers in ASP.NET Core.

Next steps

Previous: Add a model | Next: Work with a database

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 4 of tutorial series on Razor Pages

Article • 08/26/2024

By Joe Audette

The `RazorPagesMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in `Program.cs`:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using RazorPagesMovie.Data;
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();
```

The ASP.NET Core Configuration system reads the `ConnectionString` key. For local development, configuration gets the connection string from the `appsettings.json` file.

Visual Studio

The generated connection string is similar to the following JSON:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
```

```
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "RazorPagesMovieContext": "Server=
(localdb)\\mssqllocaldb;Database=RazorPagesMovieContext-
bc;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

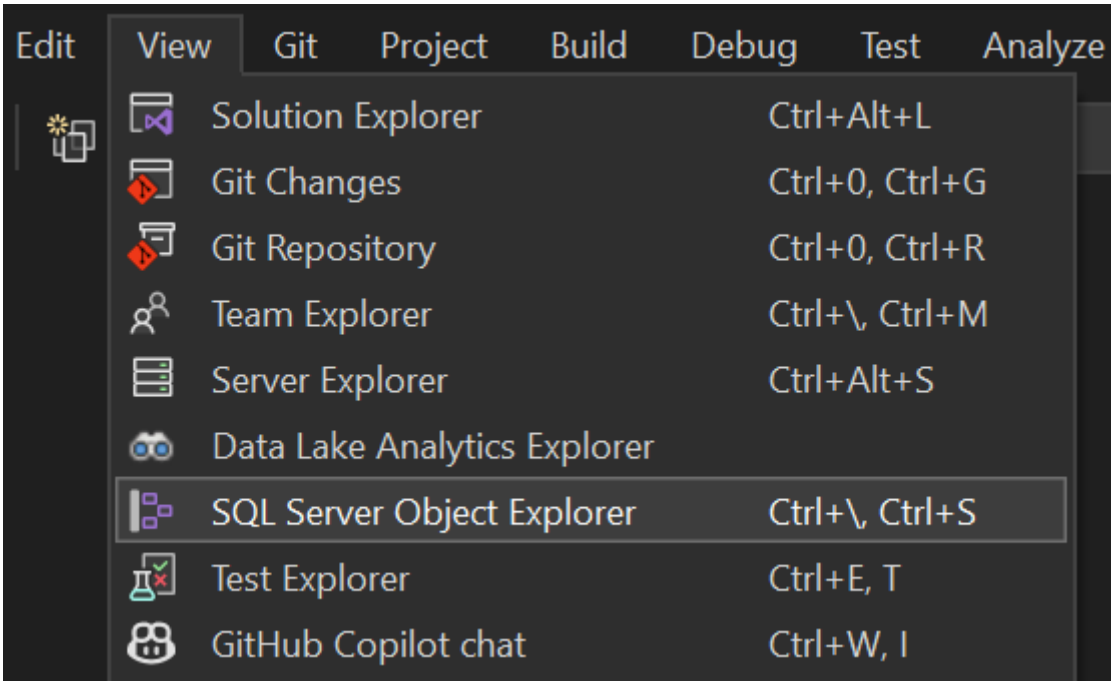
> **Note**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Visual Studio

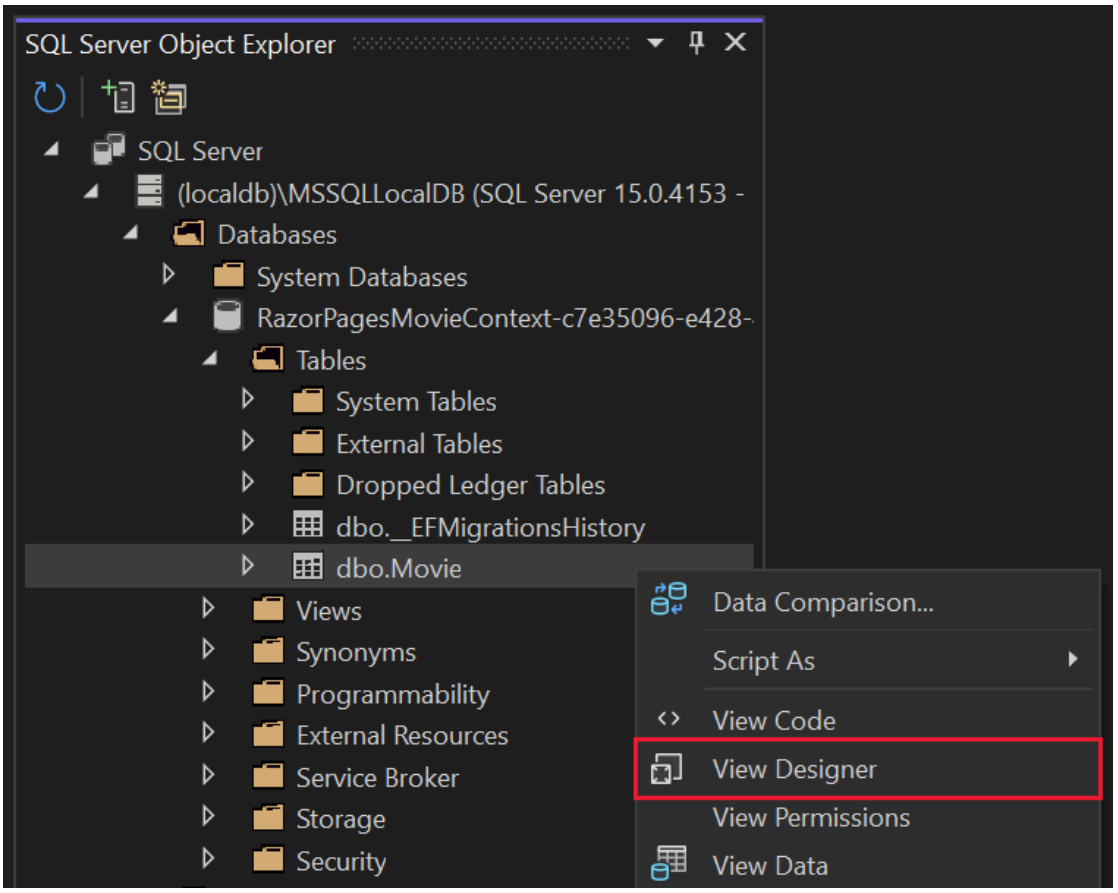
SQL Server Express LocalDB

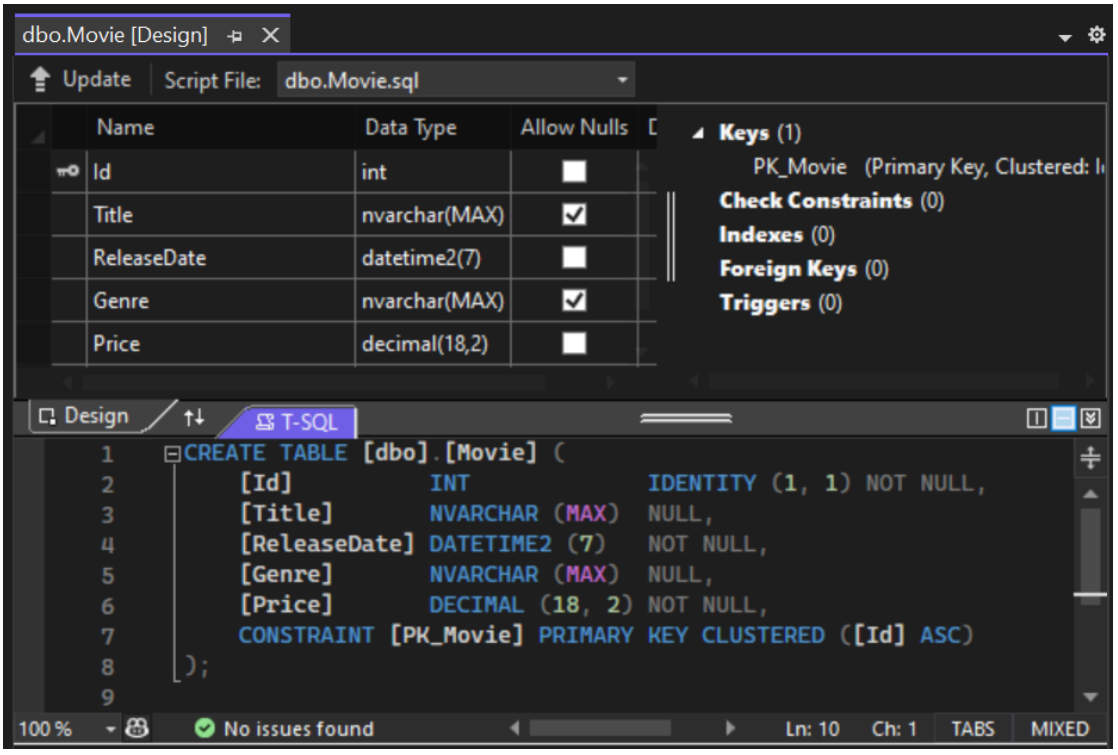
LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates `*.mdf` files in the `C:\Users\<user>\` directory.

1. From the **View** menu, open **SQL Server Object Explorer** (SSOX).



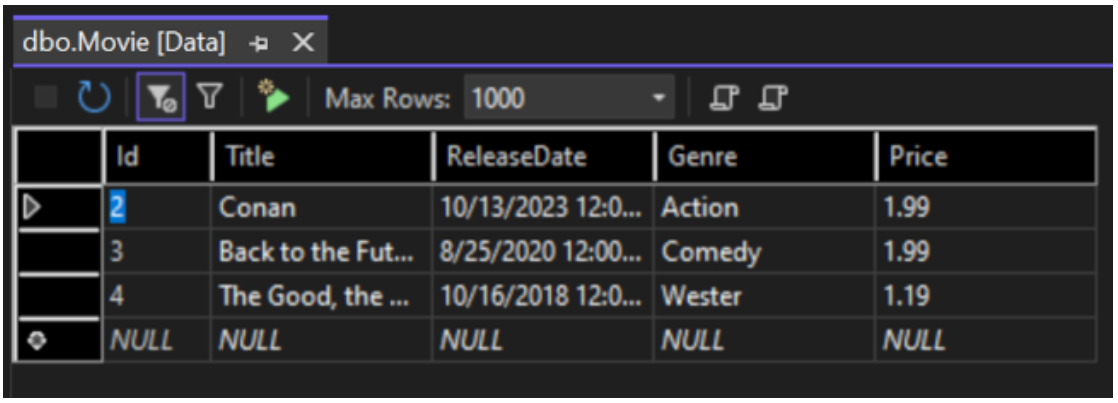
2. Right-click on the `Movie` table and select **View Designer**:





Note the key icon next to `ID`. By default, EF creates a property named `ID` for the primary key.

3. Right-click on the `Movie` table and select **View Data**:



Seed the database

Create a new class named `SeedData` in the *Models* folder with the following code:

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;

namespace RazorPagesMovie.Models;

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new RazorPagesMovieContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<RazorPagesMovieContext>>()))
        {
            if (context == null || context.Movie == null)
            {
                throw new ArgumentNullException("Null
RazorPagesMovieContext");
            }

            // Look for any movies.
            if (context.Movie.Any())
            {
                return;   // DB has been seeded
            }

            context.Movie.AddRange(
                new Movie
                {
                    Title = "When Harry Met Sally",
                    ReleaseDate = DateTime.Parse("1989-2-12"),
                    Genre = "Romantic Comedy",
                    Price = 7.99M
                },
```

```
                new Movie
                {
                    Title = "Ghostbusters ",
                    ReleaseDate = DateTime.Parse("1984-3-13"),
                    Genre = "Comedy",
                    Price = 8.99M
                },

                new Movie
                {
                    Title = "Ghostbusters 2",
                    ReleaseDate = DateTime.Parse("1986-2-23"),
                    Genre = "Comedy",
                    Price = 9.99M
                },

                new Movie
                {
                    Title = "Rio Bravo",
                    ReleaseDate = DateTime.Parse("1959-4-15"),
                    Genre = "Western",
                    Price = 3.99M
                }
            );
            context.SaveChanges();
        }
    }
}
```

If there are any movies in the database, the seed initializer returns and no movies are added.

C#

```
if (context.Movie.Any())
{
    return;
}
```

Add the seed initializer

Update the `Program.cs` with the following highlighted code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;
using RazorPagesMovie.Data;
using RazorPagesMovie.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddDbContext<RazorPagesMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("RazorPag
esMovieContext") ?? throw new InvalidOperationException("Connection
string 'RazorPagesMovieContext' not found.")));

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In the previous code, `Program.cs` has been modified to do the following:

- Get a database context instance from the dependency injection (DI) container.
- Call the `seedData.Initialize` method, passing to it the database context instance.
- Dispose the context when the seed method completes. The using statement ensures the context is disposed.

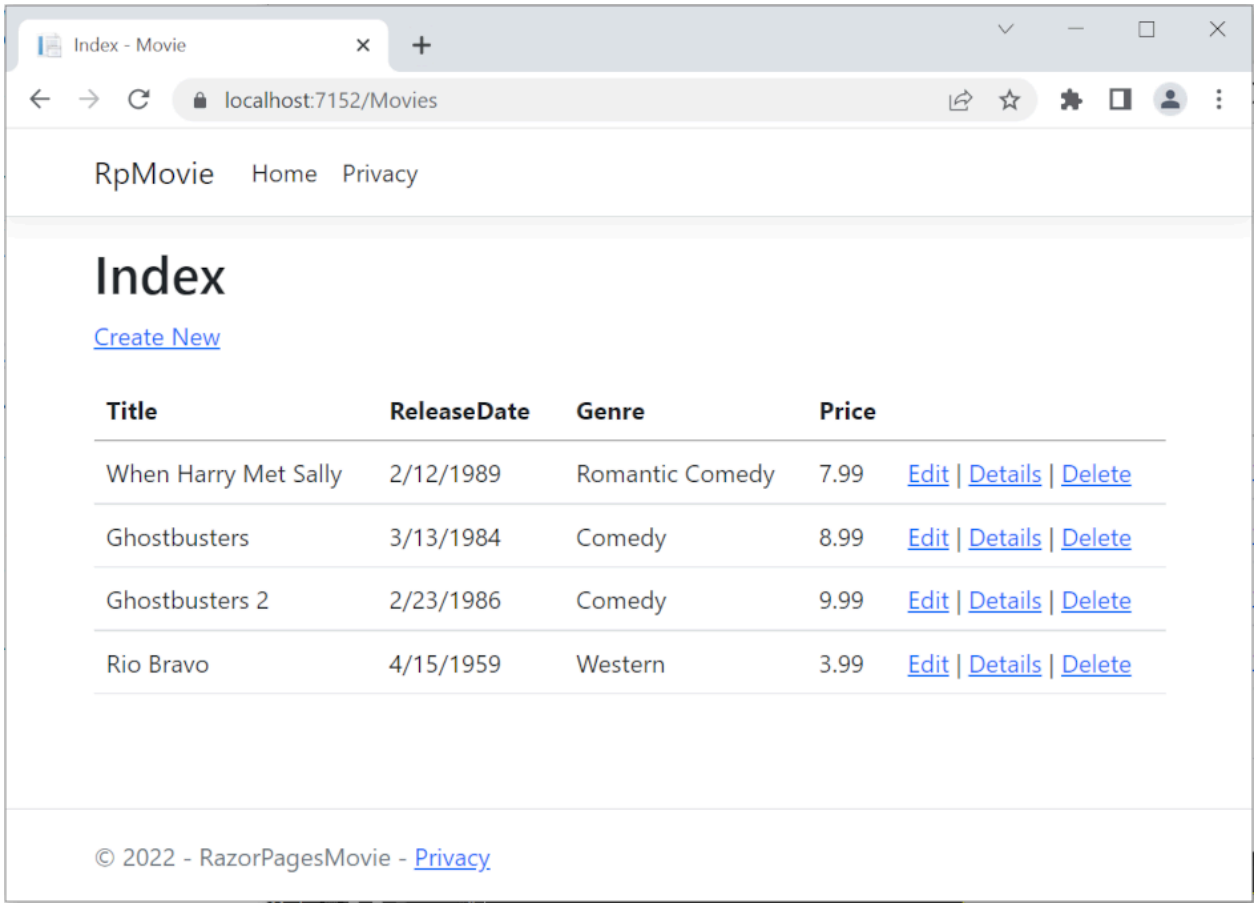
The following exception occurs when `Update-Database` has not been run:

> `SqlException: Cannot open database "RazorPagesMovieContext-" requested by the login. The login failed.` `Login failed for user 'user name'.`

Test the app

Delete all the records in the database so the seed method will run. Stop and start the app to seed the database. If the database isn't seeded, put a breakpoint on `if (context.Movie.Any())` and step through the code.

The app shows the seeded data:



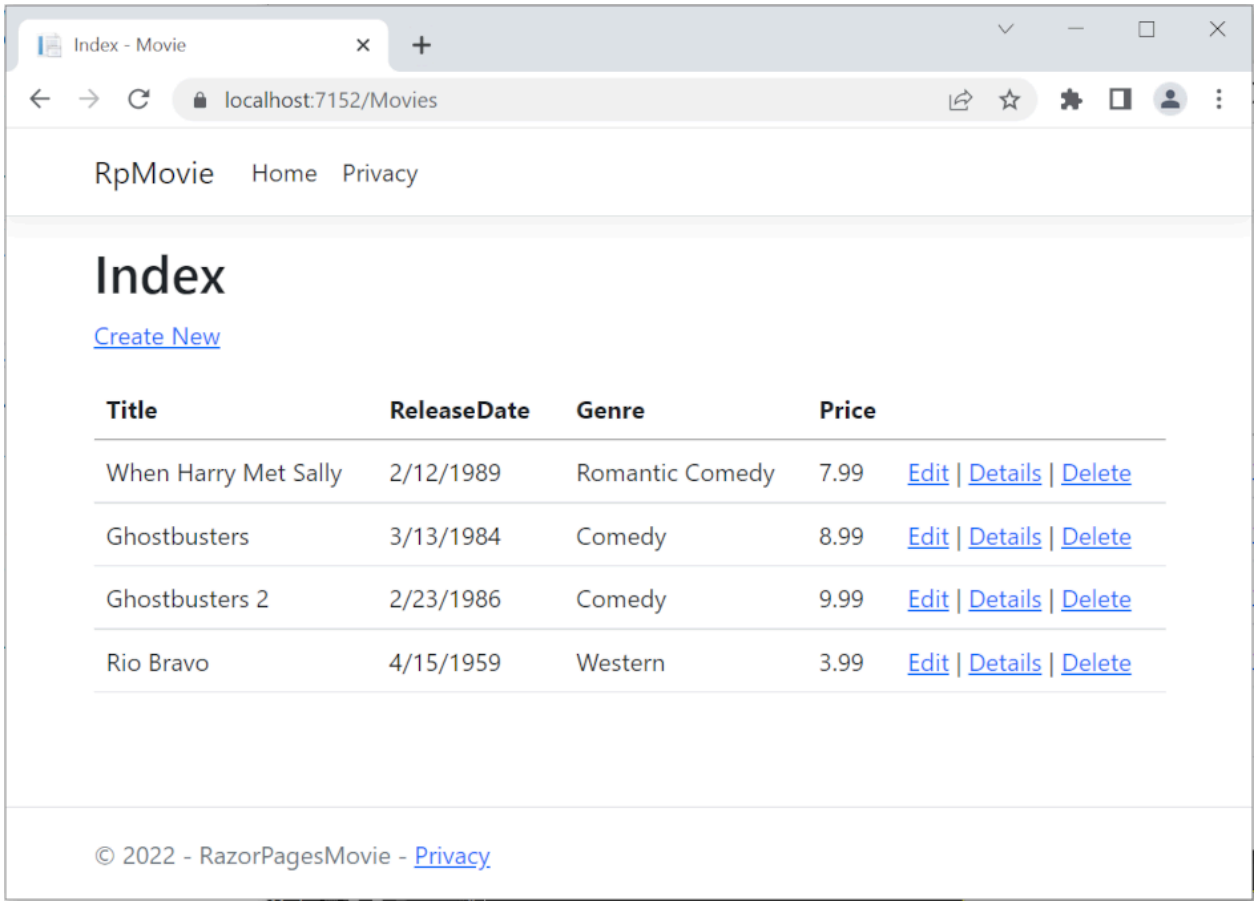
Next steps

Previous: Scaffolded Razor Pages | Next: Update the pages

Part 5, update the generated pages in an ASP.NET Core app

Article • 06/23/2024

The scaffolded movie app has a good start, but the presentation isn't ideal. **ReleaseDate** should be two words, **Release Date**.



Update the model

Update `Models/Movie.cs` with the following highlighted code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
```

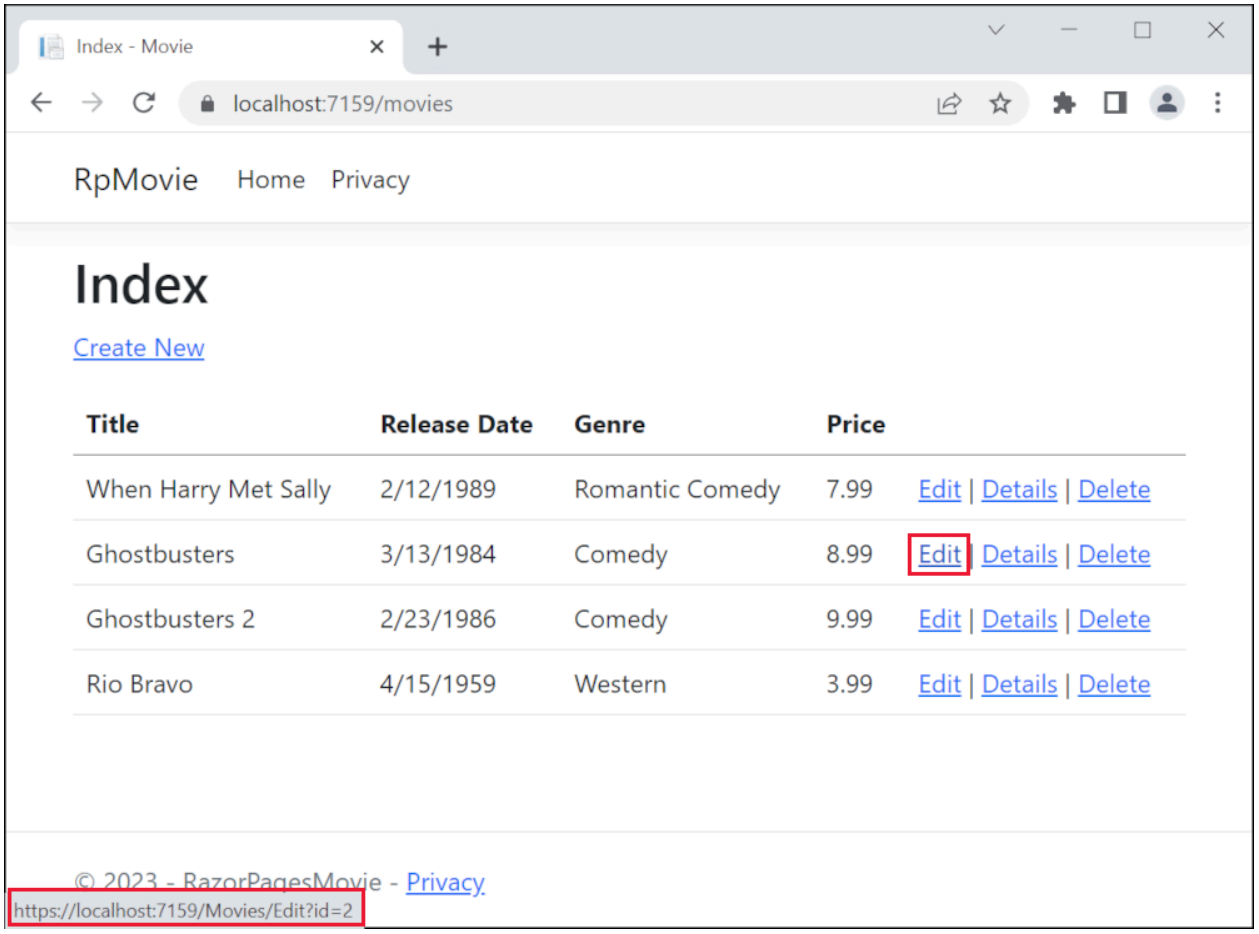
```
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; } = string.Empty;

        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
```

In the previous code:

- The `[Column(TypeName = "decimal(18, 2)")]` data annotation enables Entity Framework Core to correctly map `Price` to currency in the database. For more information, see Data Types.
- The [Display] attribute specifies the display name of a field. In the preceding code, `Release Date` instead of `ReleaseDate`.
- The [DataType] attribute specifies the type of the data (`Date`). The time information stored in the field isn't displayed.

DataAnnotations is covered in the next tutorial.

Browse to *Pages/Movies* and hover over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Anchor Tag Helper in the `Pages/Movies/Index.cshtml` file.

CSHTML

```
@foreach (var item in Model.Movie) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.

In the preceding code, the Anchor Tag Helper dynamically generates the HTML `href` attribute value from the Razor Page (the route is relative), the `asp-page`, and the route identifier (`asp-route-id`). For more information, see URL generation for Pages.

Use **View Source** from a browser to examine the generated markup. A portion of the generated HTML is shown below:

HTML

```
<td>
  <a href="/Movies/Edit?id=1">Edit</a> |
  <a href="/Movies/Details?id=1">Details</a> |
  <a href="/Movies/Delete?id=1">Delete</a>
</td>
```

The dynamically generated links pass the movie ID with a query string. For example, the `?id=1` in `https://localhost:5001/Movies/Details?id=1`.

Add route template

Update the Edit, Details, and Delete Razor Pages to use the `{id:int}` route template. Change the page directive for each of these pages from `@page` to `@page "{id:int}"`. Run the app and then view source.

The generated HTML adds the ID to the path portion of the URL:

HTML

```
<td>
  <a href="/Movies/Edit/1">Edit</a> |
  <a href="/Movies/Details/1">Details</a> |
  <a href="/Movies/Delete/1">Delete</a>
</td>
```

A request to the page with the `{id:int}` route template that does **not** include the integer returns an HTTP 404 (not found) error. For example, `https://localhost:5001/Movies/Details` returns a 404 error. To make the ID optional, append `?` to the route constraint:

CSHTML

```
@page "{id:int?}"
```

Test the behavior of `@page "{id:int?}"`:

1. Set the page directive in `Pages/Movies/Details.cshtml` to `@page "{id:int?}"`.
2. Set a break point in `public async Task<IActionResult> OnGetAsync(int? id)`, in `Pages/Movies/Details.cshtml.cs`.
3. Navigate to `https://localhost:5001/Movies/Details/`.

With the `@page "{id:int}"` directive, the break point is never hit. The routing engine returns HTTP 404. Using `@page "{id:int?}"`, the `OnGetAsync` method returns `NotFound` (HTTP 404):

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
```

```
    if (id == null)
    {
        return NotFound();
    }

    Movie = await _context.Movie.FirstOrDefaultAsync(m => m.ID == id);

    if (Movie == null)
    {
        return NotFound();
    }
    return Page();
}
```

Review concurrency exception handling

Review the `OnPostAsync` method in the `Pages/Movies/Edit.cshtml.cs` file:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Attach(Movie).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie.Id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return RedirectToPage("./Index");
}

private bool MovieExists(int id)
{
```

```
    return _context.Movie.Any(e => e.Id == id);
}
```

The previous code detects concurrency exceptions when one client deletes the movie and the other client posts changes to the movie.

To test the `catch` block:

1. Set a breakpoint on `catch (DbUpdateConcurrencyException)`.
2. Select **Edit** for a movie, make changes, but don't enter **Save**.
3. In another browser window, select the **Delete** link for the same movie, and then delete the movie.
4. In the previous browser window, post changes to the movie.

Production code may want to detect concurrency conflicts. See Handle concurrency conflicts for more information.

Posting and binding review

Examine the `Pages/Movies/Edit.cshtml.cs` file:

C#

```
public class EditModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public EditModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Movie Movie { get; set; } = default!;

    public async Task<IActionResult> OnGetAsync(int? id)
    {
        if (id == null || _context.Movie == null)
        {
            return NotFound();
        }

        var movie =  await _context.Movie.FirstOrDefaultAsync(m => m.Id ==
id);
        if (movie == null)
        {
            return NotFound();
        }
        Movie = movie;
```

```
            return Page();
        }

        // To protect from overposting attacks, enable the specific properties
    you want to bind to.
        // For more details, see https://aka.ms/RazorPagesCRUD.
        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            _context.Attach(Movie).State = EntityState.Modified;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!MovieExists(Movie.Id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }

            return RedirectToPage("./Index");
        }

        private bool MovieExists(int id)
        {
          return _context.Movie.Any(e => e.Id == id);
        }
```

When an HTTP GET request is made to the Movies/Edit page, for example, `https://localhost:5001/Movies/Edit/3`:

- The `OnGetAsync` method fetches the movie from the database and returns the `Page` method.
- The `Page` method renders the `Pages/Movies/Edit.cshtml` Razor Page. The `Pages/Movies/Edit.cshtml` file contains the model directive `@model RazorPagesMovie.Pages.Movies.EditModel`, which makes the movie model available on the page.
- The Edit form is displayed with the values from the movie.

When the Movies/Edit page is posted:

- The form values on the page are bound to the `Movie` property. The `[BindProperty]` attribute enables Model binding.

  ```
  C#

  [BindProperty]
  public Movie Movie { get; set; }
  ```

- If there are errors in the model state, for example, `ReleaseDate` cannot be converted to a date, the form is redisplayed with the submitted values.
- If there are no model errors, the movie is saved.

The HTTP GET methods in the Index, Create, and Delete Razor pages follow a similar pattern. The HTTP POST `OnPostAsync` method in the Create Razor Page follows a similar pattern to the `OnPostAsync` method in the Edit Razor Page.

Next steps

Previous: Work with a database | Next: Add search

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 6, add search to ASP.NET Core Razor Pages

Article • 06/23/2024

By Rick Anderson

In the following sections, searching movies by *genre* or *name* is added.

Add the following highlighted code to `Pages/Movies/Index.cshtml.cs`:

C#

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovie.Data.RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovie.Data.RazorPagesMovieContext context)
    {
        _context = context;
    }

    public IList<Movie> Movie { get;set; }  = default!;

    [BindProperty(SupportsGet = true)]
    public string? SearchString { get; set; }

    public SelectList? Genres { get; set; }

    [BindProperty(SupportsGet = true)]
    public string? MovieGenre { get; set; }
```

In the previous code:

- `SearchString`: Contains the text users enter in the search text box. `SearchString` has the [BindProperty] attribute. `[BindProperty]` binds form values and query strings with the same name as the property. `[BindProperty(SupportsGet = true)]` is required for binding on HTTP GET requests.
- `Genres`: Contains the list of genres. `Genres` allows the user to select a genre from the list. `SelectList` requires `using Microsoft.AspNetCore.Mvc.Rendering;`
- `MovieGenre`: Contains the specific genre the user selects. For example, "Western".
- `Genres` and `MovieGenre` are used later in this tutorial.

> ⚠ **Warning**

For security reasons, you must opt in to binding `GET` request data to page model properties. Verify user input before mapping it to properties. Opting into `GET` binding is useful when addressing scenarios that rely on query string or route values.

To bind a property on `GET` requests, set the [BindProperty] attribute's `SupportsGet` property to `true`:

C#

```
[BindProperty(SupportsGet = true)]
```

For more information, see ASP.NET Core Community Standup: Bind on GET discussion (YouTube).

Update the Index page's `OnGetAsync` method with the following code:

C#

```
public async Task OnGetAsync()
{
    var movies = from m in _context.Movie
                 select m;
    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    Movie = await movies.ToListAsync();
}
```

The first line of the `OnGetAsync` method creates a LINQ query to select the movies:

C#

```
// using System.Linq;
var movies = from m in _context.Movie
             select m;
```

The query is only ***defined*** at this point, it has ***not*** been run against the database.

If the `SearchString` property is not `null` or empty, the movies query is modified to filter on the search string:

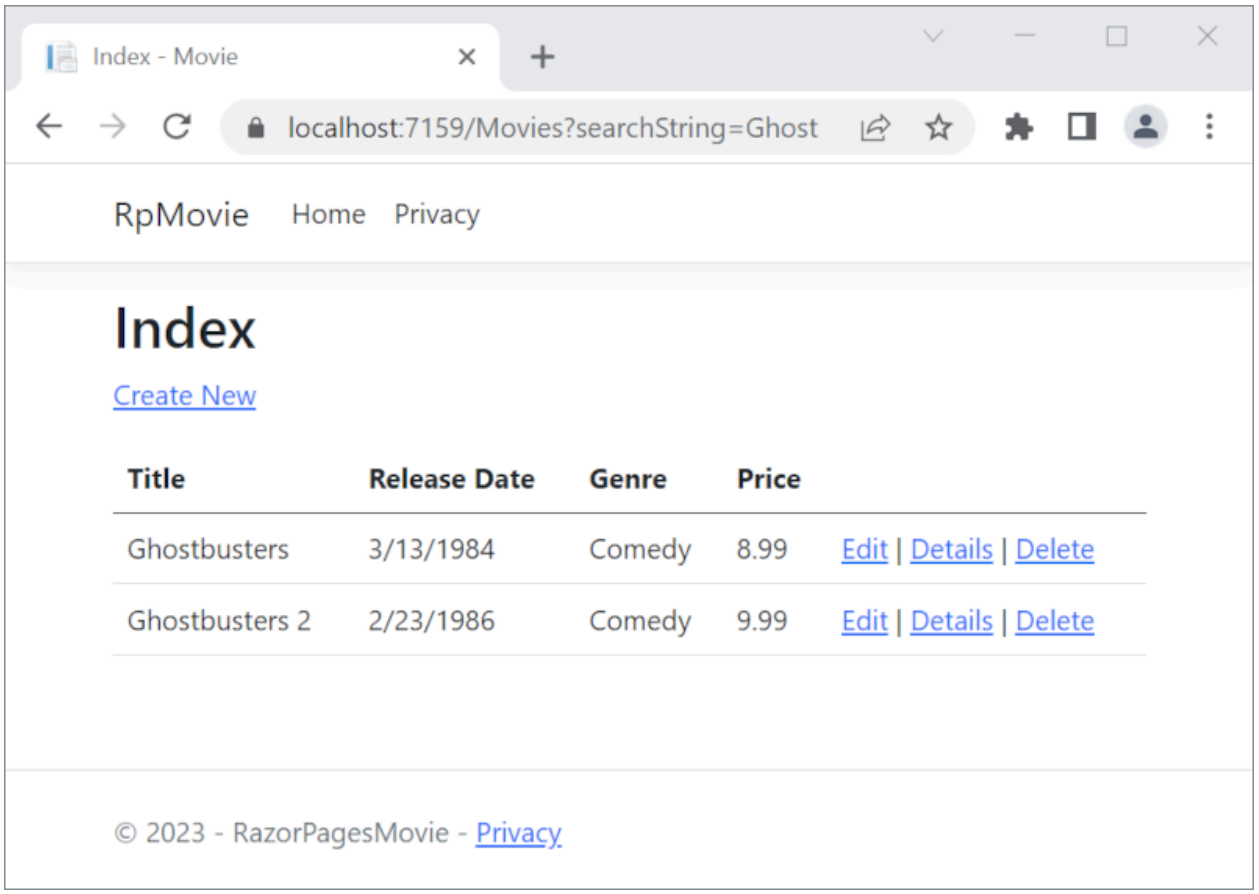
C#

```
if (!string.IsNullOrEmpty(SearchString))
{
    movies = movies.Where(s => s.Title.Contains(SearchString));
}
```

The `s => s.Title.Contains()` code is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains`. LINQ queries are not executed when they're defined or when they're modified by calling a method, such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. The evaluation of an expression is delayed until its realized value is iterated over or the `ToListAsync` method is called. See Query Execution for more information.

> **Note**
>
> The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. SQLite with the default collation is a mixture of case sensitive and case ***IN***sensitive, depending on the query. For information on making case insensitive SQLite queries, see the following:
>
> - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
> - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
> - Collations and Case Sensitivity

Navigate to the Movies page and append a query string such as `?searchString=Ghost` to the URL. For example, `https://localhost:5001/Movies?searchString=Ghost`. The filtered movies are displayed.

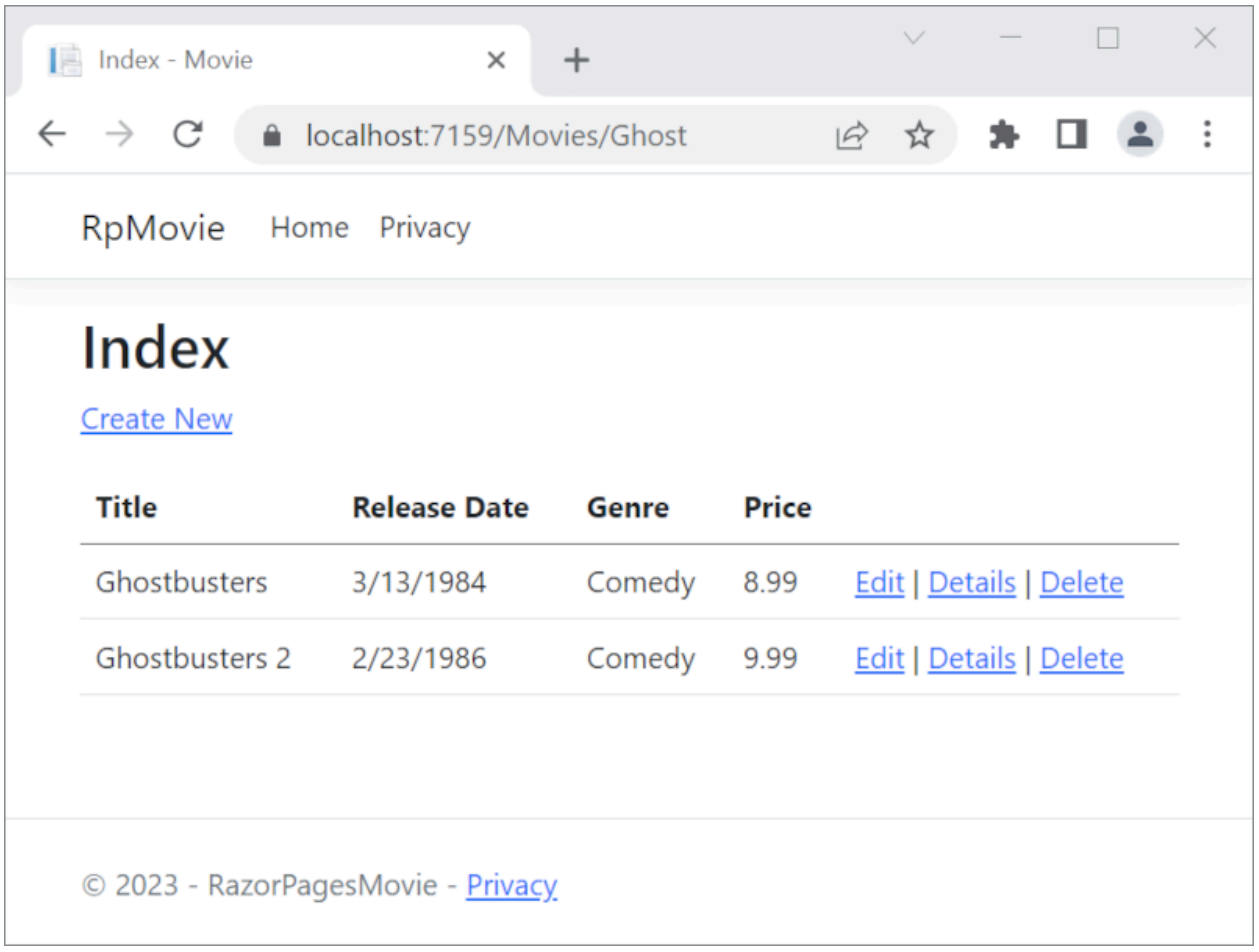


If the following route template is added to the Index page, the search string can be passed as a URL segment. For example, `https://localhost:5001/Movies/Ghost`.

CSHTML

```
@page "{searchString?}"
```

The preceding route constraint allows searching the title as route data (a URL segment) instead of as a query string value. The `?` in `"{searchString?}"` means this is an optional route parameter.



The ASP.NET Core runtime uses model binding to set the value of the `SearchString` property from the query string (`?searchString=Ghost`) or route data (`https://localhost:5001/Movies/Ghost`). Model binding is ***not*** case sensitive.

However, users cannot be expected to modify the URL to search for a movie. In this step, UI is added to filter movies. If you added the route constraint `"{searchString?}"`, remove it.

Open the `Pages/Movies/Index.cshtml` file, and add the markup highlighted in the following code:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
```

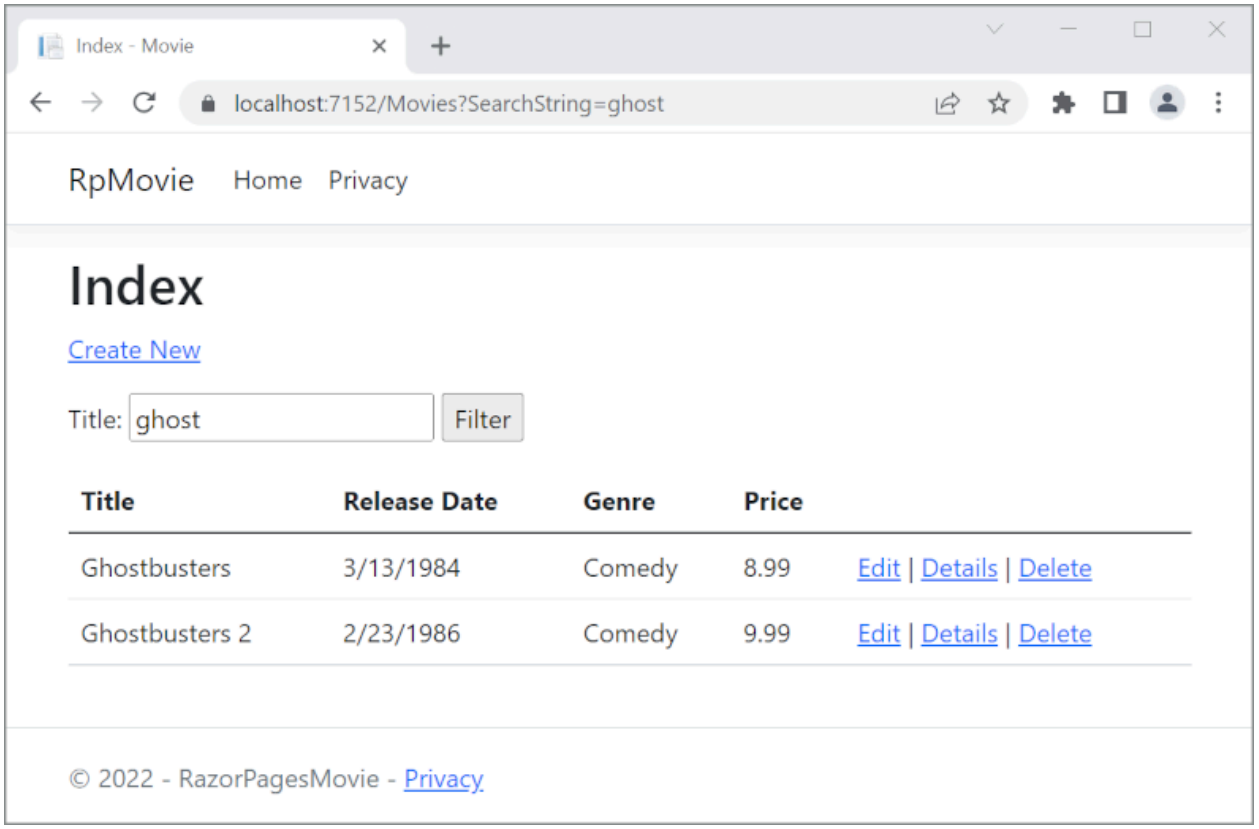
```
<form>
    <p>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

```
<table class="table">
    @*Markup removed for brevity.*@
```

The HTML `<form>` tag uses the following Tag Helpers:

- Form Tag Helper. When the form is submitted, the filter string is sent to the *Pages/Movies/Index* page via query string.
- Input Tag Helper

Save the changes and test the filter.



Search by genre

Update the `Movies/Index.cshtml.cs` page `OnGetAsync` method with the following code:

C#

```
public async Task OnGetAsync()
{
```

```
    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;

    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(SearchString))
    {
        movies = movies.Where(s => s.Title.Contains(SearchString));
    }

    if (!string.IsNullOrEmpty(MovieGenre))
    {
        movies = movies.Where(x => x.Genre == MovieGenre);
    }
    Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
    Movie = await movies.ToListAsync();
}
```

The following code is a LINQ query that retrieves all the genres from the database.

C#

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres.

C#

```
Genres = new SelectList(await genreQuery.Distinct().ToListAsync());
```

Add search by genre to the Razor Page

Update the `Index.cshtml` <form> element as highlighted in the following markup:

CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}
```

```
<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
```

Test the app by searching by genre, by movie title, and by both.

Next steps

Previous: Update the pages | Next: Add a new field

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 7, add a new field to a Razor Page in ASP.NET Core

Article • 06/23/2024

By Rick Anderson

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field schema change to the database.

When using EF Code First to automatically create and track a database, Code First:

- Adds an __EFMigrationsHistory table to the database to track whether the schema of the database is in sync with the model classes it was generated from.
- Throws an exception if the model classes aren't in sync with the database.

Automatic verification that the schema and model are in sync makes it easier to find inconsistent database code issues.

Adding a Rating Property to the Movie Model

1. Open the `Models/Movie.cs` file and add a `Rating` property:

   C#

   ```
   public class Movie
   {
       public int Id { get; set; }
       public string Title { get; set; } = string.Empty;

       [Display(Name = "Release Date")]
       [DataType(DataType.Date)]
       public DateTime ReleaseDate { get; set; }
       public string Genre { get; set; } = string.Empty;

       [Column(TypeName = "decimal(18, 2)")]
       public decimal Price { get; set; }
       public string Rating { get; set; } = string.Empty;
   }
   ```

2. Edit `Pages/Movies/Index.cshtml`, and add a `Rating` field:

   CSHTML

```
@page
@model RazorPagesMovie.Pages.Movies.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form>
    <p>
        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">

    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Movie[0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movie[0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movie)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
```

```
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-
id="@item.Id">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

3. Update the following pages with a `Rating` field:

   - *Pages/Movies/Create.cshtml*.
   - *Pages/Movies/Delete.cshtml*.
   - *Pages/Movies/Details.cshtml*.
   - *Pages/Movies/Edit.cshtml*.

The app won't work until the database is updated to include the new field. Running the app without an update to the database throws a `SqlException`:

```
SqlException: Invalid column name 'Rating'.
```

The `SqlException` exception is caused by the updated Movie model class being different than the schema of the Movie table of the database. There's no `Rating` column in the database table.

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database using the new model class schema. This approach is convenient early in the development cycle, it allows developers to quickly evolve the model and database schema together. The downside is that existing data in the database is lost. Don't use this approach on a production database! Dropping the database on schema changes

and using an initializer to automatically seed the database with test data is often a productive way to develop an app.
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is to keep the data. Make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

For this tutorial, use Code First Migrations.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but make this change for each `new Movie` block.

C#

```
context.Movie.AddRange(
    new Movie
    {
        Title = "When Harry Met Sally",
        ReleaseDate = DateTime.Parse("1989-2-12"),
        Genre = "Romantic Comedy",
        Price = 7.99M,
        Rating = "R"
    },
```

See the completed SeedData.cs file.

Build the app

Visual Studio

Press `Ctrl`+`Shift`+`B`

Visual Studio

Add a migration for the rating field

1. From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.

2. In the PMC, enter the following commands:

PowerShell

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the framework to:

- Compare the `Movie` model with the `Movie` database schema.
- Create code to migrate the database schema to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

The `Update-Database` command tells the framework to apply the schema changes to the database and to preserve existing data.

Delete all the records in the database, the initializer will seed the database and include the `Rating` field. Deleting can be done with the delete links in the browser or from Sql Server Object Explorer (SSOX).

Another option is to delete the database and use migrations to re-create the database. To delete the database in SSOX:

1. Select the database in SSOX.
2. Right-click on the database, and select **Delete**.
3. Check **Close existing connections**.
4. Select **OK**.
5. In the PMC, update the database:

```
PowerShell
Update-Database
```

Run the app and verify you can create, edit, and display movies with a `Rating` field. If the database isn't seeded, set a break point in the `SeedData.Initialize` method.

Next steps

Previous: Add Search | Next: Add Validation

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 8 of tutorial series on Razor Pages

Article • 06/23/2024

By Rick Anderson

In this section, validation logic is added to the `Movie` model. The validation rules are enforced any time a user creates or edits a movie.

Validation

A key tenet of software development is called DRY ("**D**on't **R**epeat **Y**ourself"). Razor Pages encourages development where functionality is specified once, and it's reflected throughout the app. DRY can help:

- Reduce the amount of code in an app.
- Make the code less error prone, and easier to test and maintain.

The validation support provided by Razor Pages and Entity Framework is a good example of the DRY principle:

- Validation rules are declaratively specified in one place, in the model class.
- Rules are enforced everywhere in the app.

Add validation rules to the movie model

The System.ComponentModel.DataAnnotations namespace provides:

- A set of built-in validation attributes that are applied declaratively to a class or property.
- Formatting attributes like `[DataType]` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in `[Required]`, `[StringLength]`, `[RegularExpression]`, and `[Range]` validation attributes.

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models;

public class Movie
```

```
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string Title { get; set; } = string.Empty;

    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
    [Required]
    [StringLength(30)]
    public string Genre { get; set; } = string.Empty;

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string Rating { get; set; } = string.Empty;
}
```

The validation attributes specify behavior to enforce on the model properties they're applied to:

- The `[Required]` and `[MinimumLength]` attributes indicate that a property must have a value. Nothing prevents a user from entering white space to satisfy this validation.
- The `[RegularExpression]` attribute is used to limit what characters can be input. In the preceding code, `Genre`:
  - Must only use letters.
  - The first letter must be uppercase. White spaces are allowed, while numbers and special characters aren't allowed.
- The `RegularExpression` `Rating`:
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a `Genre`.
- The `[Range]` attribute constrains a value to within a specified range.
- The `[StringLength]` attribute can set a maximum length of a string property, and optionally its minimum length.

- Value types, such as `decimal`, `int`, `float`, `DateTime`, are inherently required and don't need the `[Required]` attribute.

The preceding validation rules are used for demonstration, they are not optimal for a production system. For example, the preceding prevents entering a movie with only two chars and doesn't allow special characters in `Genre`.

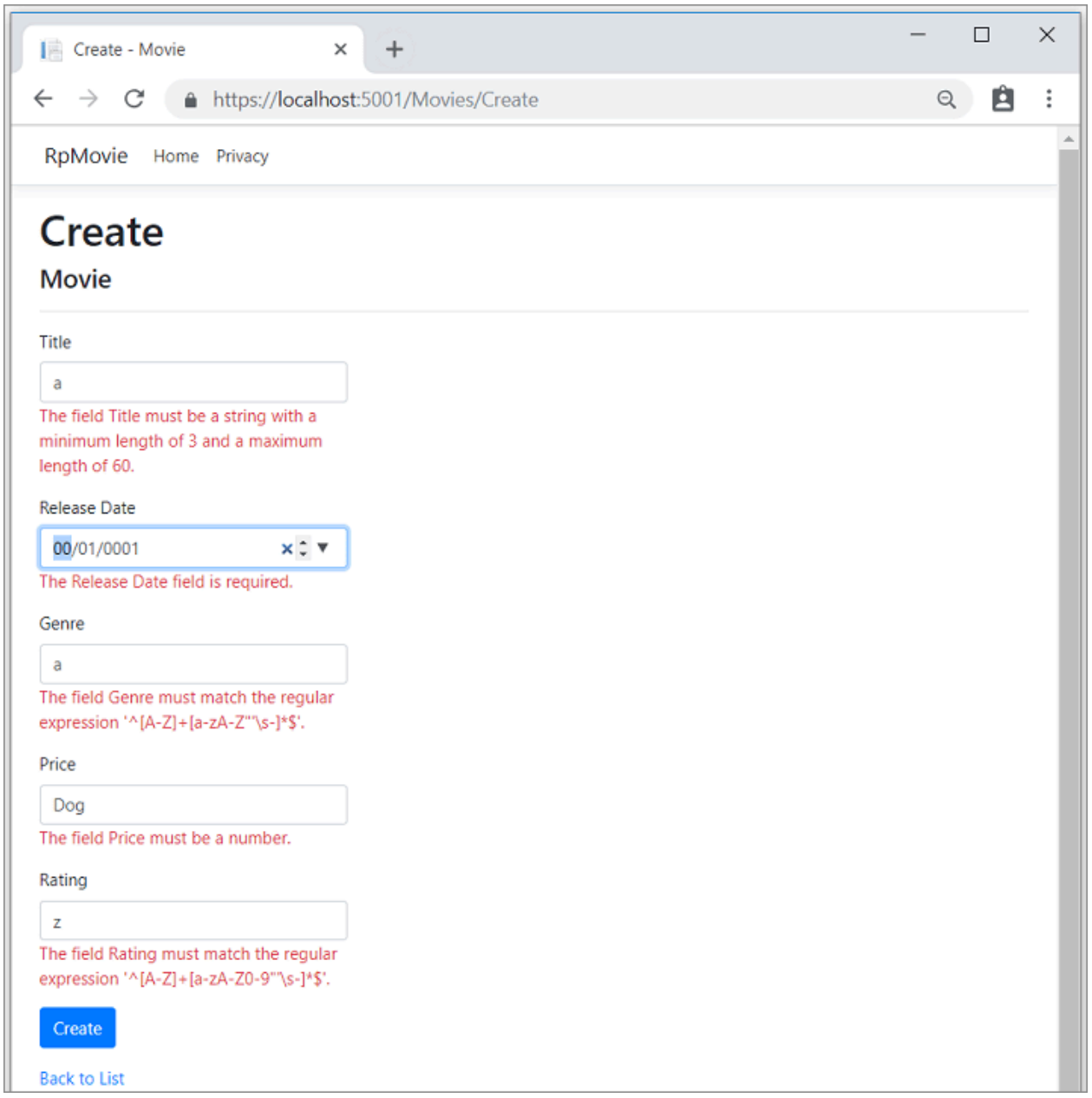
Having validation rules automatically enforced by ASP.NET Core helps:

- Make the app more robust.
- Reduce chances of saving invalid data to the database.

Validation Error UI in Razor Pages

Run the app and navigate to Pages/Movies.

Select the **Create New** link. Complete the form with some invalid values. When jQuery client-side validation detects the error, it displays an error message.



> **Note**
>
> You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered a validation error message in each field containing an invalid value. The errors are enforced both client-side, using JavaScript and jQuery, and server-side, when a user has JavaScript disabled.

A significant benefit is that **no** code changes were necessary in the Create or Edit pages. Once data annotations were applied to the model, the validation UI was enabled. The Razor Pages created in this tutorial automatically picked up the validation rules, using

validation attributes on the properties of the `Movie` model class. Test validation using the Edit page, the same validation is applied.

The form data isn't posted to the server until there are no client-side validation errors. Verify form data isn't posted by one or more of the following approaches:

- Put a break point in the `OnPostAsync` method. Submit the form by selecting **Create** or **Save**. The break point is never hit.
- Use the Fiddler tool.
- Use the browser developer tools to monitor network traffic.

Server-side validation

When JavaScript is disabled in the browser, submitting the form with errors will post to the server.

Optional, test server-side validation:

1. Disable JavaScript in the browser. JavaScript can be disabled using browser's developer tools. If JavaScript cannot be disabled in the browser, try another browser.
2. Set a break point in the `OnPostAsync` method of the Create or Edit page.
3. Submit a form with invalid data.
4. Verify the model state is invalid:

```
C#
```

```
if (!ModelState.IsValid)
{
    return Page();
}
```

Alternatively, Disable client-side validation on the server.

The following code shows a portion of the `Create.cshtml` page scaffolded earlier in the tutorial. It's used by the Create and Edit pages to:

- Display the initial form.
- Redisplay the form in the event of an error.

```
CSHTML
```

```
<form method="post">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Movie.Title" class="control-label"></label>
        <input asp-for="Movie.Title" class="form-control" />
        <span asp-validation-for="Movie.Title" class="text-danger"></span>
    </div>
```

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client-side. The Validation Tag Helper displays validation errors. See Validation for more information.

The Create and Edit pages have no validation rules in them. The validation rules and the error strings are specified only in the `Movie` class. These validation rules are automatically applied to Razor Pages that edit the `Movie` model.

When validation logic needs to change, it's done only in the model. Validation is applied consistently throughout the app, validation logic is defined in one place. Validation in one place helps keep the code clean, and makes it easier to maintain and update.

Use DataType Attributes

Examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. The `[DataType]` attribute is applied to the `ReleaseDate` and `Price` properties.

C#

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

The `[DataType]` attributes provide:

- Hints for the view engine to format the data.
- Supplies attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email.

Use the `[RegularExpression]` attribute to validate the format of the data. The `[DataType]` attribute is used to specify a data type that's more specific than the

database intrinsic type. `[DataType]` attributes aren't validation attributes. In the sample app, only the date is displayed, without time.

The `DataType` enumeration provides many data types, such as `Date`, `Time`, `PhoneNumber`, `Currency`, `EmailAddress`, and more.

The `[DataType]` attributes:

- Can enable the app to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`.
- Can provide a date selector `DataType.Date` in browsers that support HTML5.
- Emit HTML 5 `data-`, pronounced "data dash", attributes that HTML 5 browsers consume.
- Do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

The `[DisplayFormat]` attribute is used to explicitly specify the date format:

```
C#

[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting will be applied when the value is displayed for editing. That behavior may not be wanted for some fields. For example, in currency values, the currency symbol is usually not wanted in the edit UI.

The `[DisplayFormat]` attribute can be used by itself, but it's generally a good idea to use the `[DataType]` attribute. The `[DataType]` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `[DataType]` attribute provides the following benefits that aren't available with `[DisplayFormat]`:

- The browser can enable HTML5 features, for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.
- By default, the browser renders data using the correct format based on its locale.

- The `[DataType]` attribute can enable the ASP.NET Core framework to choose the right field template to render the data. The `DisplayFormat`, if used by itself, uses the string template.

Note: jQuery validation doesn't work with the `[Range]` attribute and `DateTime`. For example, the following code will always display a client-side validation error, even when the date is in the specified range:

C#

```
[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]
```

It's a best practice to avoid compiling hard dates in models, so using the `[Range]` attribute and `DateTime` is discouraged. Use Configuration for date ranges and other values that are subject to frequent change rather than specifying it in code.

The following code shows combining attributes on one line:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace RazorPagesMovie.Models;

public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; } = string.Empty;

    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$"), Required, StringLength(30)]
    public string Genre { get; set; } = string.Empty;

    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; } = string.Empty;
}
```

Get started with Razor Pages and EF Core shows advanced EF Core operations with Razor Pages.

Apply migrations

The DataAnnotations applied to the class changes the schema. For example, the DataAnnotations applied to the `Title` field:

C#

```
[StringLength(60, MinimumLength = 3)]
[Required]
public string Title { get; set; } = string.Empty;
```

- Limits the characters to 60.
- Doesn't allow a `null` value.

The `Movie` table currently has the following schema:

SQL

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (MAX)  NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (MAX)  NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (MAX)  NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

The preceding schema changes don't cause EF to throw an exception. However, create a migration so the schema is consistent with the model.

Visual Studio

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**. In the PMC, enter the following commands:

PowerShell

```
Add-Migration New_DataAnnotations
Update-Database
```

`Update-Database` runs the `Up` method of the `New_DataAnnotations` class.

Examine the `Up` method:

C#

```
public partial class New_DataAnnotations : Migration
{
    /// <inheritdoc />
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AlterColumn<string>(
            name: "Title",
            table: "Movie",
            type: "nvarchar(60)",
            maxLength: 60,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");

        migrationBuilder.AlterColumn<string>(
            name: "Rating",
            table: "Movie",
            type: "nvarchar(5)",
            maxLength: 5,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");

        migrationBuilder.AlterColumn<string>(
            name: "Genre",
            table: "Movie",
            type: "nvarchar(30)",
            maxLength: 30,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(max)");
    }
```

The updated `Movie` table has the following schema:

SQL

```
CREATE TABLE [dbo].[Movie] (
    [ID]          INT             IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (60)   NOT NULL,
    [ReleaseDate] DATETIME2 (7)   NOT NULL,
    [Genre]       NVARCHAR (30)   NOT NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    [Rating]      NVARCHAR (5)    NOT NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([ID] ASC)
);
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core app in Azure with SQL Database.

Thanks for completing this introduction to Razor Pages. Get started with Razor Pages and EF Core is an excellent follow up to this tutorial.

Additional resources

- Tag Helpers in forms in ASP.NET Core
- Globalization and localization in ASP.NET Core
- Tag Helpers in ASP.NET Core
- Author Tag Helpers in ASP.NET Core

Next steps

Previous: Add a new field

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Filter methods for Razor Pages in ASP.NET Core

Article • 04/11/2023

By Rick Anderson

Razor Page filters IPageFilter and IAsyncPageFilter allow Razor Pages to run code before and after a Razor Page handler is run. Razor Page filters are similar to ASP.NET Core MVC action filters, except they can't be applied to individual page handler methods.

Razor Page filters:

- Run code after a handler method has been selected, but before model binding occurs.
- Run code before the handler method executes, after model binding is complete.
- Run code after the handler method executes.
- Can be implemented on a page or globally.
- Cannot be applied to specific page handler methods.
- Can have constructor dependencies populated by Dependency Injection (DI). For more information, see ServiceFilterAttribute and TypeFilterAttribute.

While page constructors and middleware enable executing custom code before a handler method executes, only Razor Page filters enable access to HttpContext and the page. Middleware has access to the `HttpContext`, but not to the "page context". Filters have a FilterContext derived parameter, which provides access to `HttpContext`. Here's a sample for a page filter: Implement a filter attribute that adds a header to the response, something that can't be done with constructors or middleware. Access to the page context, which includes access to the instances of the page and it's model, are only available when executing filters, handlers, or the body of a Razor Page.

View or download sample code (how to download)

Razor Page filters provide the following methods, which can be applied globally or at the page level:

- Synchronous methods:
  - OnPageHandlerSelected : Called after a handler method has been selected, but before model binding occurs.
  - OnPageHandlerExecuting : Called before the handler method executes, after model binding is complete.

    - OnPageHandlerExecuted : Called after the handler method executes, before the action result.

- Asynchronous methods:
    - OnPageHandlerSelectionAsync : Called asynchronously after the handler method has been selected, but before model binding occurs.
    - OnPageHandlerExecutionAsync : Called asynchronously before the handler method is invoked, after model binding is complete.

Implement **either** the synchronous or the async version of a filter interface, **not** both. The framework checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both interfaces are implemented, only the async methods are called. The same rule applies to overrides in pages, implement the synchronous or the async version of the override, not both.

Implement Razor Page filters globally

The following code implements `IAsyncPageFilter`:

C#

```
public class SampleAsyncPageFilter : IAsyncPageFilter
{
    private readonly IConfiguration _config;

    public SampleAsyncPageFilter(IConfiguration config)
    {
        _config = config;
    }

    public Task OnPageHandlerSelectionAsync(PageHandlerSelectedContext
context)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent",
                                                        out StringValues
value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,

"SampleAsyncPageFilter.OnPageHandlerSelectionAsync",
                               value, key.ToString());

        return Task.CompletedTask;
    }

    public async Task
OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,

PageHandlerExecutionDelegate next)
```

```
    {
        // Do post work.
        await next.Invoke();
    }
}
```

In the preceding code, `ProcessUserAgent.Write` is user supplied code that works with the user agent string.

The following code enables the `SampleAsyncPageFilter` in the `Startup` class:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddMvcOptions(options =>
        {
            options.Filters.Add(new SampleAsyncPageFilter(Configuration));
        });
}
```

The following code calls AddFolderApplicationModelConvention to apply the `SampleAsyncPageFilter` to only pages in */Movies*:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages(options =>
    {
        options.Conventions.AddFolderApplicationModelConvention(
            "/Movies",
            model => model.Filters.Add(new
SampleAsyncPageFilter(Configuration)));
    });
}
```

The following code implements the synchronous `IPageFilter`:

C#

```
public class SamplePageFilter : IPageFilter
{
    private readonly IConfiguration _config;

    public SamplePageFilter(IConfiguration config)
    {
        _config = config;
```

```
    }

    public void OnPageHandlerSelected(PageHandlerSelectedContext context)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent", out
StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
                               "SamplePageFilter.OnPageHandlerSelected",
                               value, key.ToString());
    }

    public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        Debug.WriteLine("Global sync OnPageHandlerExecuting called.");
    }

    public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
    {
        Debug.WriteLine("Global sync OnPageHandlerExecuted called.");
    }
}
```

The following code enables the `SamplePageFilter`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddMvcOptions(options =>
        {
            options.Filters.Add(new SamplePageFilter(Configuration));
        });
}
```

Implement Razor Page filters by overriding filter methods

The following code overrides the asynchronous Razor Page filters:

C#

```
public class IndexModel : PageModel
{
    private readonly IConfiguration _config;

    public IndexModel(IConfiguration config)
    {
```

```
        _config = config;
    }

    public override Task
OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
        Debug.WriteLine("/IndexModel OnPageHandlerSelectionAsync");
        return Task.CompletedTask;
    }

    public async override Task
OnPageHandlerExecutionAsync(PageHandlerExecutingContext context,

PageHandlerExecutionDelegate next)
    {
        var key = _config["UserAgentID"];
        context.HttpContext.Request.Headers.TryGetValue("user-agent", out
StringValues value);
        ProcessUserAgent.Write(context.ActionDescriptor.DisplayName,
                               "/IndexModel-OnPageHandlerExecutionAsync",
                                value, key.ToString());

        await next.Invoke();
    }
}
```

Implement a filter attribute

The built-in attribute-based filter OnResultExecutionAsync filter can be subclassed. The following filter adds a header to the response:

C#

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace PageFilter.Filters
{
    public class AddHeaderAttribute  : ResultFilterAttribute
    {
        private readonly string _name;
        private readonly string _value;

        public AddHeaderAttribute (string name, string value)
        {
            _name = name;
            _value = value;
        }

        public override void OnResultExecuting(ResultExecutingContext
context)
```

```
        {
            context.HttpContext.Response.Headers.Add(_name, new string[] {
_value });
        }
    }
}
```

The following code applies the `AddHeader` attribute:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using PageFilter.Filters;

namespace PageFilter.Movies
{
    [AddHeader("Author", "Rick")]
    public class TestModel : PageModel
    {
        public void OnGet()
        {

        }
    }
}
```

Use a tool such as the browser developer tools to examine the headers. Under **Response Headers**, `author: Rick` is displayed.

See Overriding the default order for instructions on overriding the order.

See Cancellation and short circuiting for instructions to short-circuit the filter pipeline from a filter.

Authorize filter attribute

The Authorize attribute can be applied to a `PageModel`:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace PageFilter.Pages
{
    [Authorize]
    public class ModelWithAuthFilterModel : PageModel
```

```
    {
        public IActionResult OnGet() => Page();
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Razor Pages route and app conventions in ASP.NET Core

Article • 04/11/2023

Learn how to use page route and app model provider conventions to control page routing, discovery, and processing in Razor Pages apps.

To specify a page route, add route segments, or add parameters to a route, use the page's `@page` directive. For more information, see Custom routes.

There are reserved words that can't be used as route segments or parameter names. For more information, see Routing: Reserved routing names.

View or download sample code (how to download)

Expand table

Scenario	The sample demonstrates
Model conventions Conventions.Add • IPageRouteModelConvention • IPageApplicationModelConvention • IPageHandlerModelConvention	Add a route template and header to an app's pages.
Page route action conventions • AddFolderRouteModelConvention • AddPageRouteModelConvention • AddPageRoute	Add a route template to pages in a folder and to a single page.
Page model action conventions • AddFolderApplicationModelConvention • AddPageApplicationModelConvention • ConfigureFilter (filter class, lambda expression, or filter factory)	Add a header to pages in a folder, add a header to a single page, and configure a filter factory to add a header to an app's pages.

Razor Pages conventions are configured using an AddRazorPages overload that configures RazorPagesOptions. The following convention examples are explained later in this topic:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages(options =>
    {
        options.Conventions.Add( ... );
        options.Conventions.AddFolderRouteModelConvention(
            "/OtherPages", model => { ... });
        options.Conventions.AddPageRouteModelConvention(
            "/About", model => { ... });
        options.Conventions.AddPageRoute(
            "/Contact", "TheContactPage/{text?}");
        options.Conventions.AddFolderApplicationModelConvention(
            "/OtherPages", model => { ... });
        options.Conventions.AddPageApplicationModelConvention(
            "/About", model => { ... });
        options.Conventions.ConfigureFilter(model => { ... });
        options.Conventions.ConfigureFilter( ... );
    });
}
```

Route order

Routes specify an Order for processing (route matching).

Expand table

Route order	Behavior
-1	The route is processed before other routes are processed.
0	Order isn't specified (default value). Not assigning `Order` (`Order = null`) defaults the route `Order` to 0 (zero) for processing.
1, 2, ... n	Specifies the route processing order.

Route processing is established by convention:

- Routes are processed in sequential order (-1, 0, 1, 2, ... n).
- When routes have the same `Order`, the most specific route is matched first followed by less specific routes.
- When routes with the same `Order` and the same number of parameters match a request URL, routes are processed in the order that they're added to the PageConventionCollection.

If possible, avoid depending on an established route processing order. Generally, routing selects the correct route with URL matching. If you must set route `Order` properties to route requests correctly, the app's routing scheme is probably confusing to clients and fragile to maintain. Seek to simplify the app's routing scheme. The sample app requires an explicit route processing order to demonstrate several routing scenarios using a single app. However, you should attempt to avoid the practice of setting route `Order` in production apps.

Razor Pages routing and MVC controller routing share an implementation. Information on route order in the MVC topics is available at Routing to controller actions: Ordering attribute routes.

Model conventions

Add a delegate for IPageConvention to add model conventions that apply to Razor Pages.

Add a route model convention to all pages

Use Conventions to create and add an IPageRouteModelConvention to the collection of IPageConvention instances that are applied during page route model construction.

The sample app contains the `GlobalTemplatePageRouteModelConvention` class to add a `{globalTemplate?}` route template to all of the pages in the app:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
namespace SampleApp.Conventions;

public class GlobalTemplatePageRouteModelConvention :
IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 1,
```

```
                    Template = AttributeRouteModel.CombineTemplates(
                        selector.AttributeRouteModel!.Template,
                        "{globalTemplate?}"),
                };
            });
        }
    }
}
```

In the preceding code:

- The PageRouteModel is passed to the Apply method.
- The PageRouteModel.Selectors gets the count of selectors.
- A new SelectorModel is added which contains a AttributeRouteModel

Razor Pages options, such as adding Conventions, are added when Razor Pages is added to the service collection. For an example, see the sample app.

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.EntityFrameworkCore;
using SampleApp.Conventions;
using SampleApp.Data;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseInMemoryDatabase("InMemoryDb"));

builder.Services.AddRazorPages(options =>
   {
       options.Conventions.Add(new
GlobalTemplatePageRouteModelConvention());

       options.Conventions.AddFolderRouteModelConvention("/OtherPages",
model =>
       {
           var selectorCount = model.Selectors.Count;
           for (var i = 0; i < selectorCount; i++)
           {
               var selector = model.Selectors[i];
               model.Selectors.Add(new SelectorModel
               {
                   AttributeRouteModel = new AttributeRouteModel
                   {
                       Order = 2,
                       Template = AttributeRouteModel.CombineTemplates(
                           selector.AttributeRouteModel!.Template,
                           "{otherPagesTemplate?}"),
                   }
```

```
                });
            }
        });

        options.Conventions.AddPageRouteModelConvention("/About", model =>
        {
            var selectorCount = model.Selectors.Count;
            for (var i = 0; i < selectorCount; i++)
            {
                var selector = model.Selectors[i];
                model.Selectors.Add(new SelectorModel
                {
                    AttributeRouteModel = new AttributeRouteModel
                    {
                        Order = 2,
                        Template = AttributeRouteModel.CombineTemplates(
                            selector.AttributeRouteModel!.Template,
                            "{aboutTemplate?}"),
                    }
                });
            }
        });

    });

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseAuthorization();
app.MapRazorPages();
app.Run();
```

Consider the `GlobalTemplatePageRouteModelConvention` class:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
namespace SampleApp.Conventions;

public class GlobalTemplatePageRouteModelConvention :
IPageRouteModelConvention
{
    public void Apply(PageRouteModel model)
    {
        var selectorCount = model.Selectors.Count;
        for (var i = 0; i < selectorCount; i++)
```

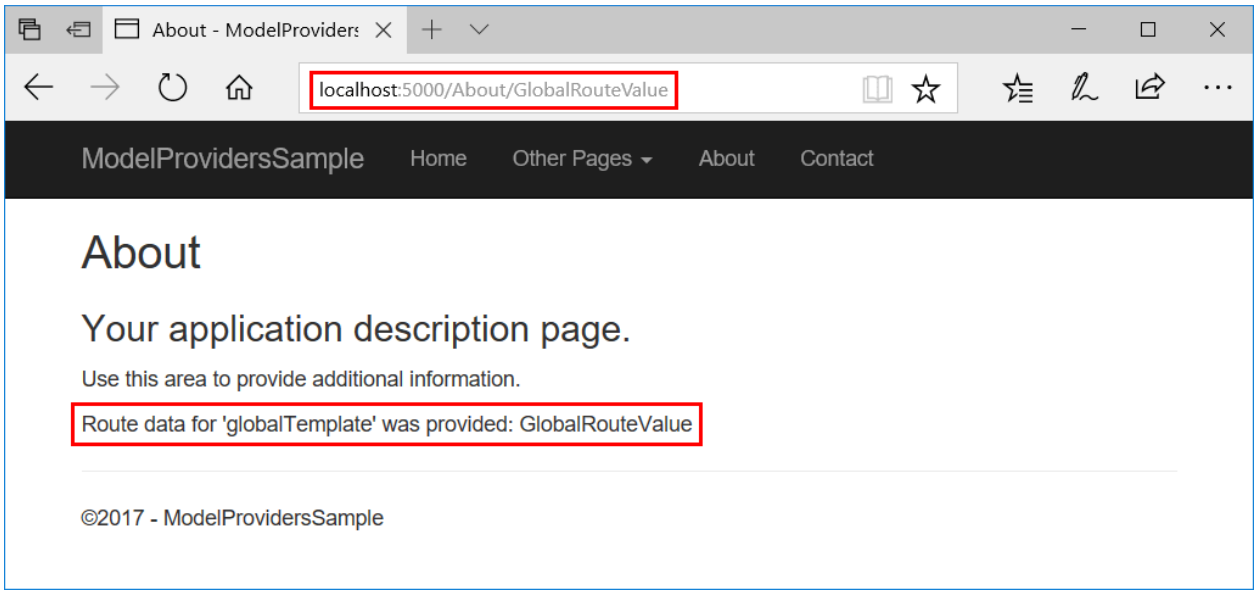
```
        {
            var selector = model.Selectors[i];
            model.Selectors.Add(new SelectorModel
            {
                AttributeRouteModel = new AttributeRouteModel
                {
                    Order = 1,
                    Template = AttributeRouteModel.CombineTtemplates(
                        selector.AttributeRouteModel!.Template,
                        "{globalTemplate?}"),
                }
            });
        }
    }
}
```

The Order property for the AttributeRouteModel is set to `1`. This ensures the following route matching behavior in the sample app:

- A route template for `TheContactPage/{text?}` is added later in this topic. The `Contact Page` route has a default order of `null` (`Order = 0`), so it matches before the `{globalTemplate?}` route template which has `Order = 1`.
- The `{aboutTemplate?}` route template is show in the preceding code. The `{aboutTemplate?}` template is given an `Order` of `2`. When the About page is requested at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.
- The `{otherPagesTemplate?}` route template is shown in the preceding code. The `{otherPagesTemplate?}` template is given an `Order` of `2`. When any page in the *Pages/OtherPages* folder is requested with a route parameter:
- For example, `/OtherPages/Page1/xyz`
- The route data value `"xyz"` is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`).
- `RouteData.Values["otherPagesTemplate"]` with (`Order = 2`) is not loaded due to the `Order` property `2` having a higher value.

When possible, don't set the `Order`. When `Order` is not set, it defaults to `Order = 0`. Rely on routing to select the correct route rather than the `Order` property.

Request the sample's `About` page at `localhost:{port}/About/GlobalRouteValue` and inspect the result:



The sample app uses the Rick.Docs.Samples.RouteInfo NuGet package to display routing information in the logging output. Using `localhost:{port}/About/GlobalRouteValue`, the logger displays the request, the `Order`, and the template used:

.NET CLI

```
info: SampleApp.Pages.AboutModel[0]
       /About/GlobalRouteValue   Order = 1 Template =
About/{globalTemplate?}
```

Add an app model convention to all pages

Use Conventions to create and add an IPageApplicationModelConvention to the collection of IPageConvention instances that are applied during page app model construction.

To demonstrate this and other conventions later in the topic, the sample app includes an `AddHeaderAttribute` class. The class constructor accepts a `name` string and a `values` string array. These values are used in its `OnResultExecuting` method to set a response header. The full class is shown in the Page model action conventions section later in the topic.

The sample app uses the `AddHeaderAttribute` class to add a header, `GlobalHeader`, to all of the pages in the app:

C#

```
public class GlobalHeaderPageApplicationModelConvention
    : IPageApplicationModelConvention
```

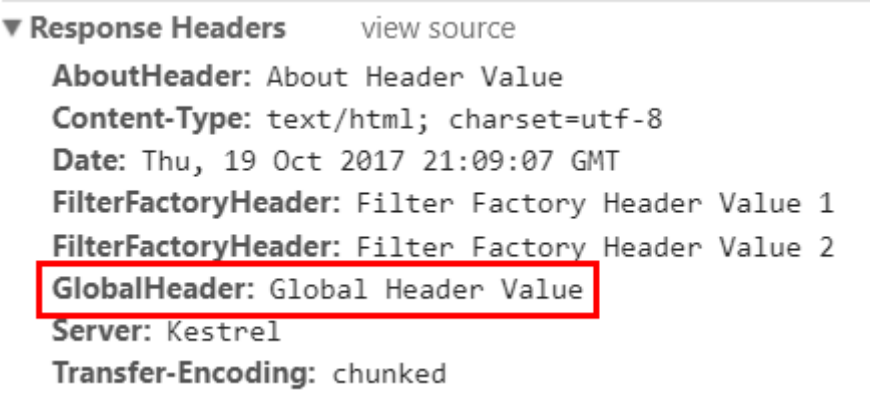
```
{
    public void Apply(PageApplicationModel model)
    {
        model.Filters.Add(new AddHeaderAttribute(
            "GlobalHeader", new string[] { "Global Header Value" }));
    }
}
```

`Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseInMemoryDatabase("InMemoryDb"));

builder.Services.AddRazorPages(options =>
   {
       options.Conventions.Add(new
GlobalTemplatePageRouteModelConvention());

       options.Conventions.Add(new
GlobalHeaderPageApplicationModelConvention());
```

Request the sample's About page at `localhost:{port}/About` and inspect the headers to view the result:



Add a handler model convention to all pages

Use Conventions to create and add an IPageHandlerModelConvention to the collection of IPageConvention instances that are applied during page handler model construction.

C#

```
public class GlobalPageHandlerModelConvention
    : IPageHandlerModelConvention
```

```
{
    public void Apply(PageHandlerModel model)
    {
        // Access the PageHandlerModel
    }
}
```

Page route action conventions

The default route model provider that derives from IPageRouteModelProvider invokes conventions which are designed to provide extensibility points for configuring page routes.

Folder route model convention

Use AddFolderRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for all of the pages under the specified folder.

The sample app uses AddFolderRouteModelConvention to add an `{otherPagesTemplate?}` route template to the pages in the *OtherPages* folder:

C#

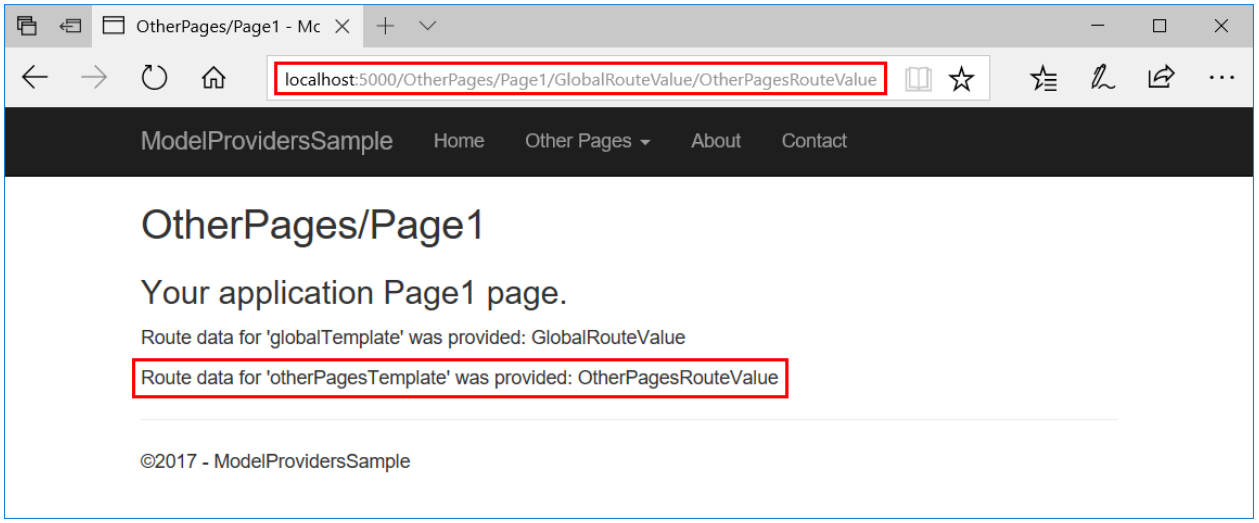
```
options.Conventions.AddFolderRouteModelConvention("/OtherPages", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
                    selector.AttributeRouteModel!.Template,
                    "{otherPagesTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If a page in the

Pages/OtherPages folder is requested with a route parameter value (for example, `/OtherPages/Page1/RouteDataValue`), "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["otherPagesTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1/GlobalRouteValue/OtherPagesRouteValue` and inspect the result:



Page route model convention

Use AddPageRouteModelConvention to create and add an IPageRouteModelConvention that invokes an action on the PageRouteModel for the page with the specified name.

The sample app uses `AddPageRouteModelConvention` to add an `{aboutTemplate?}` route template to the About page:

C#

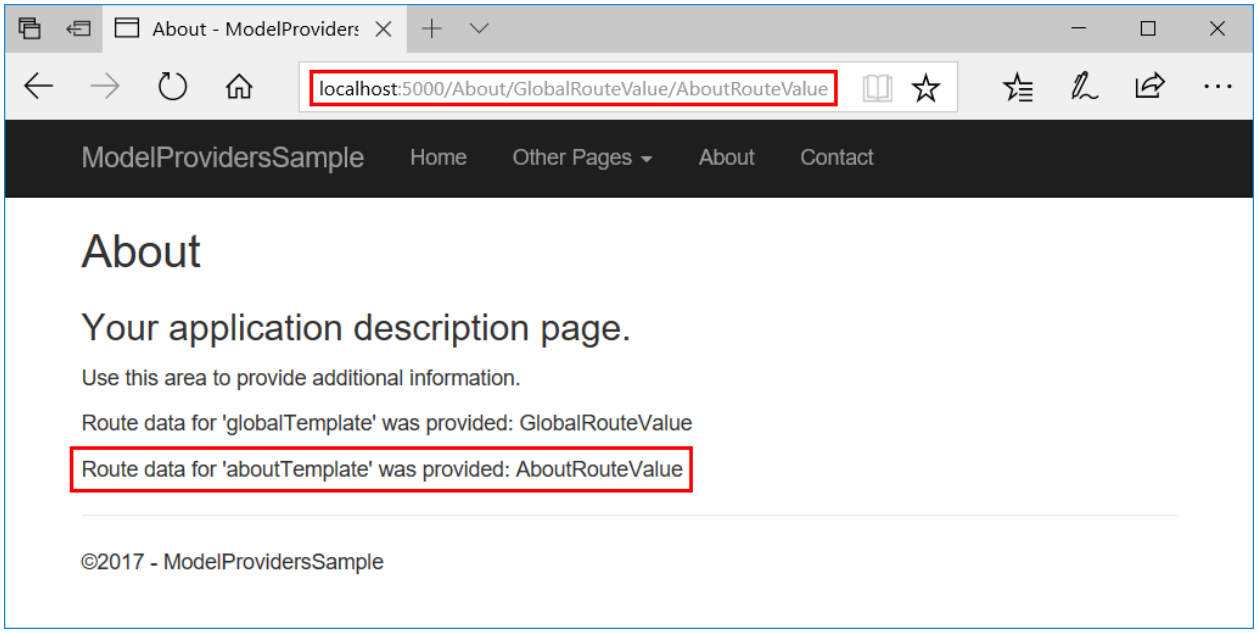
```
options.Conventions.AddPageRouteModelConvention("/About", model =>
{
    var selectorCount = model.Selectors.Count;
    for (var i = 0; i < selectorCount; i++)
    {
        var selector = model.Selectors[i];
        model.Selectors.Add(new SelectorModel
        {
            AttributeRouteModel = new AttributeRouteModel
            {
                Order = 2,
                Template = AttributeRouteModel.CombineTemplates(
```

```
                    selector.AttributeRouteModel!.Template,
                    "{aboutTemplate?}"),
            }
        });
    }
});
```

The Order property for the AttributeRouteModel is set to `2`. This ensures that the template for `{globalTemplate?}` (set earlier in the topic to `1`) is given priority for the first route data value position when a single route value is provided. If the About page is requested with a route parameter value at `/About/RouteDataValue`, "RouteDataValue" is loaded into `RouteData.Values["globalTemplate"]` (`Order = 1`) and not `RouteData.Values["aboutTemplate"]` (`Order = 2`) due to setting the `Order` property.

Wherever possible, don't set the `Order`, which results in `Order = 0`. Rely on routing to select the correct route.

Request the sample's About page at `localhost:{port}/About/GlobalRouteValue/AboutRouteValue` and inspect the result:



The logger output displays:

.NET CLI

```
info: SampleApp.Pages.AboutModel[0]
      /About/GlobalRouteValue/AboutRouteValue   Order = 2 Template =
About/{globalTemplate?}/{aboutTemplate?}
```

Use a parameter transformer to customize page routes

See Parameter transformers.

Configure a page route

Use AddPageRoute to configure a route to a page at the specified page path. Generated links to the page use the specified route. AddPageRoute uses AddPageRouteModelConvention to establish the route.

The sample app creates a route to `/TheContactPage` for the `Contact` Razor Page:

C#

```
options.Conventions.AddPageRoute("/Contact", "TheContactPage/{text?}");
```

The `Contact` page can also be reached at `/`Contact1` via its default route.

The sample app's custom route to the `Contact` page allows for an optional `text` route segment (`{text?}`). The page also includes this optional segment in its `@page` directive in case the visitor accesses the page at its `/Contact` route:

CSHTML

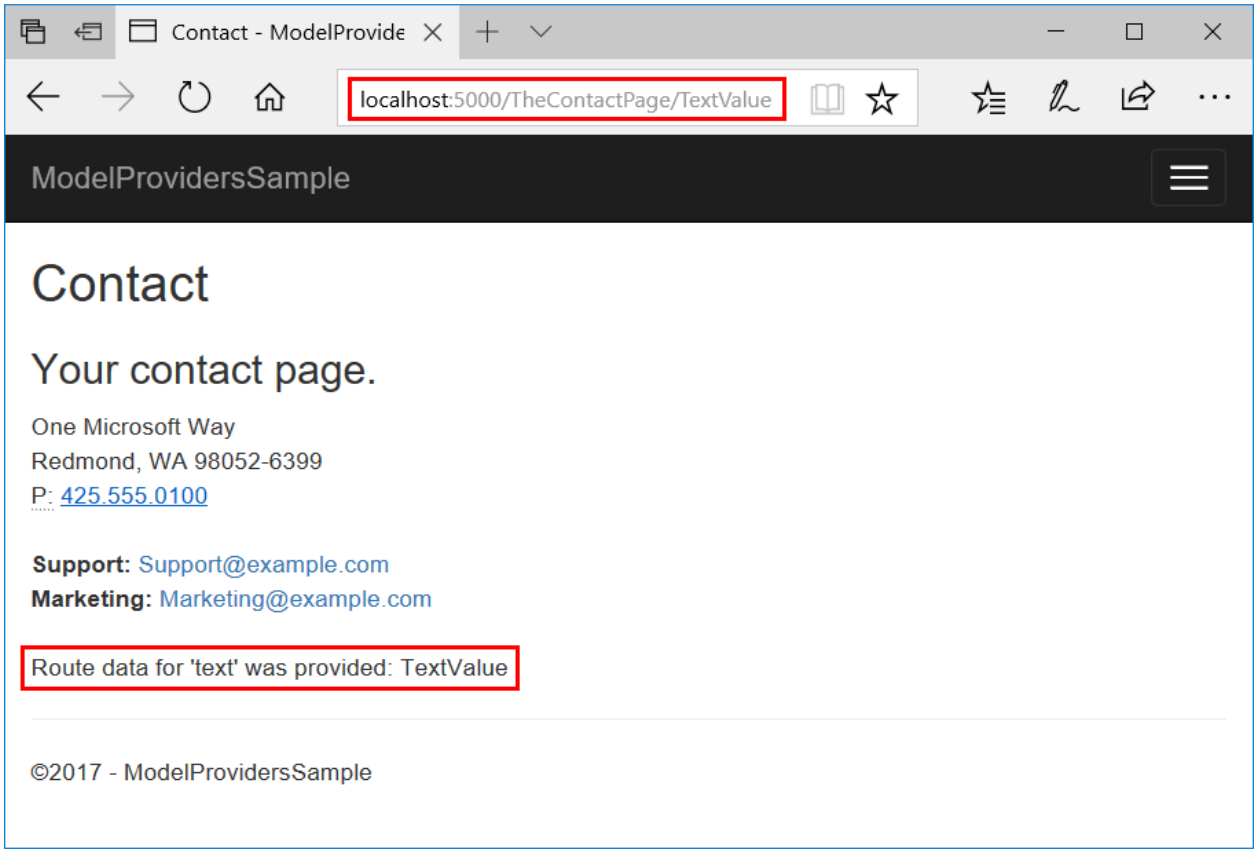
```
@page "{text?}"
@model ContactModel
@{
    ViewData["Title"] = "Contact";
}

<h1>@ViewData["Title"]</h1>
<h2>@Model.Message</h2>

<address>
    One Microsoft Way<br>
    Redmond, WA 98052-6399<br>
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong> <a
href="mailto:Support@example.com">Support@example.com</a><br>
    <strong>Marketing:</strong> <a
href="mailto:Marketing@example.com">Marketing@example.com</a>
</address>
```

```
<p>@Model.RouteDataTextTemplateValue</p>
```

Note that the URL generated for the **Contact** link in the rendered page reflects the updated route:



Visit the `Contact` page at either its ordinary route, `/Contact`, or the custom route, `/TheContactPage`. If you supply an additional `text` route segment, the page shows the HTML-encoded segment that you provide:



Page model action conventions

The default page model provider that implements IPageApplicationModelProvider invokes conventions which are designed to provide extensibility points for configuring

page models. These conventions are useful when building and modifying page discovery and processing scenarios.

For the examples in this section, the sample app uses an `AddHeaderAttribute` class, which is a ResultFilterAttribute, that applies a response header:

C#

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string[] _values;

    public AddHeaderAttribute(string name, string[] values)
    {
        _name = name;
        _values = values;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _values);
        base.OnResultExecuting(context);
    }
}
```

Using conventions, the sample demonstrates how to apply the attribute to all of the pages in a folder and to a single page.

Folder app model convention

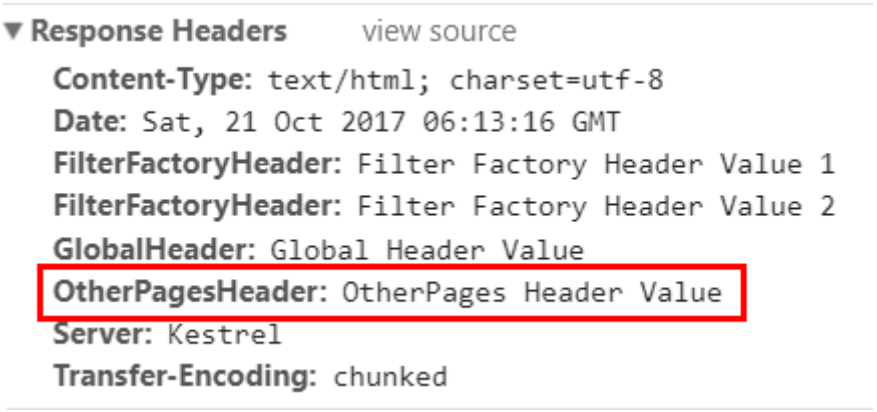
Use AddFolderApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on PageApplicationModel instances for all pages under the specified folder.

The sample demonstrates the use of `AddFolderApplicationModelConvention` by adding a header, `OtherPagesHeader`, to the pages inside the *OtherPages* folder of the app:

C#

```
options.Conventions.AddFolderApplicationModelConvention("/OtherPages", model
=>
{
    model.Filters.Add(new AddHeaderAttribute(
        "OtherPagesHeader", new string[] { "OtherPages Header Value" }));
});
```

Request the sample's Page1 page at `localhost:5000/OtherPages/Page1` and inspect the headers to view the result:



Page app model convention

Use AddPageApplicationModelConvention to create and add an IPageApplicationModelConvention that invokes an action on the PageApplicationModel for the page with the specified name.

The sample demonstrates the use of `AddPageApplicationModelConvention` by adding a header, `AboutHeader`, to the About page:

C#

```
options.Conventions.AddPageApplicationModelConvention("/About", model =>
{
    model.Filters.Add(new AddHeaderAttribute(
        "AboutHeader", new string[] { "About Header Value" }));
});
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



Configure a filter

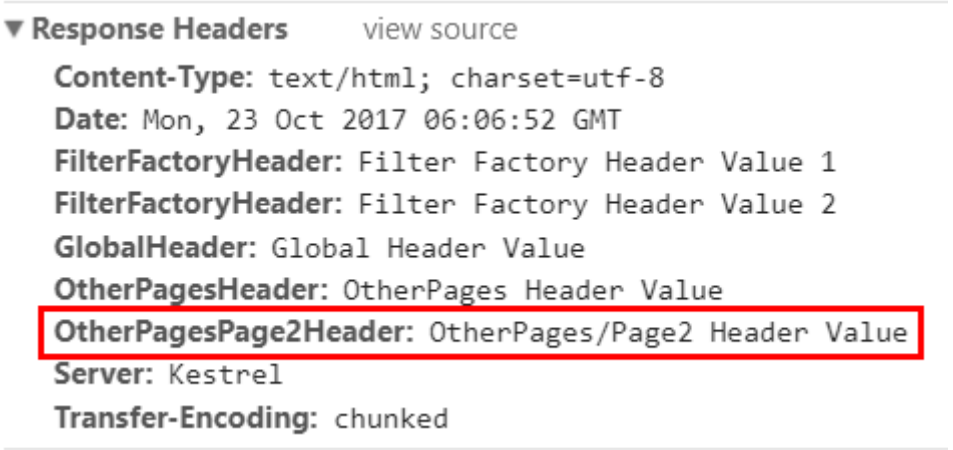
ConfigureFilter configures the specified filter to apply. You can implement a filter class, but the sample app shows how to implement a filter in a lambda expression, which is implemented behind-the-scenes as a factory that returns a filter:

```
C#

options.Conventions.ConfigureFilter(model =>
{
    if (model.RelativePath.Contains("OtherPages/Page2"))
    {
        return new AddHeaderAttribute(
            "OtherPagesPage2Header",
            new string[] { "OtherPages/Page2 Header Value" });
    }
    return new EmptyFilter();
});
```

The page app model is used to check the relative path for segments that lead to the Page2 page in the *OtherPages* folder. If the condition passes, a header is added. If not, the `EmptyFilter` is applied.

`EmptyFilter` is an Action filter. Since Action filters are ignored by Razor Pages, the `EmptyFilter` has no effect as intended if the path doesn't contain `OtherPages/Page2`.

Request the sample's Page2 page at `localhost:5000/OtherPages/Page2` and inspect the headers to view the result:



Configure a filter factory

ConfigureFilter configures the specified factory to apply filters to all Razor Pages.

The sample app provides an example of using a filter factory by adding a header, `FilterFactoryHeader`, with two values to the app's pages:

```
C#
```

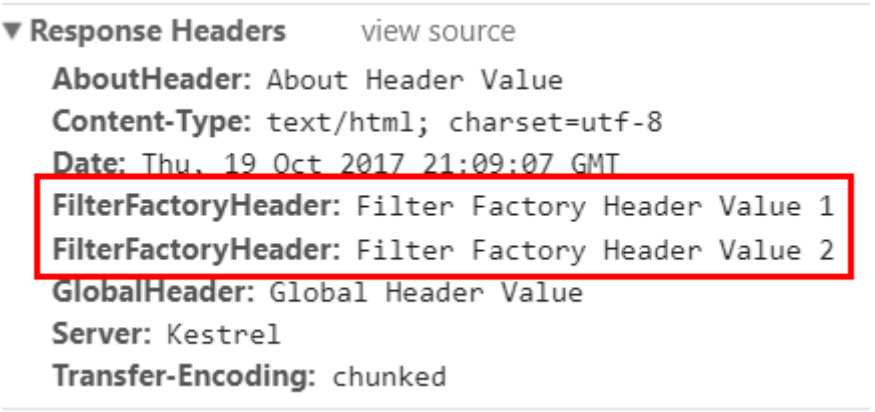
```
options.Conventions.ConfigureFilter(new AddHeaderWithFactory());
```

`AddHeaderWithFactory.cs`:

C#

```
public class AddHeaderWithFactory : IFilterFactory
{
    // Implement IFilterFactory
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new AddHeaderFilter();
    }

    private class AddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "FilterFactoryHeader",
                new string[]
                {
                    "Filter Factory Header Value 1",
                    "Filter Factory Header Value 2"
                });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

Request the sample's About page at `localhost:5000/About` and inspect the headers to view the result:



MVC Filters and the Page filter (IPageFilter)

MVC Action filters are ignored by Razor Pages, since Razor Pages use handler methods. Other types of MVC filters are available for you to use: Authorization, Exception, Resource, and Result. For more information, see the Filters topic.

The Page filter (IPageFilter) is a filter that applies to Razor Pages. For more information, see Filter methods for Razor Pages.

Additional resources

- Razor Pages Routing
- Razor Pages authorization conventions in ASP.NET Core
- Areas in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Overview of ASP.NET Core MVC

Article • 09/25/2023

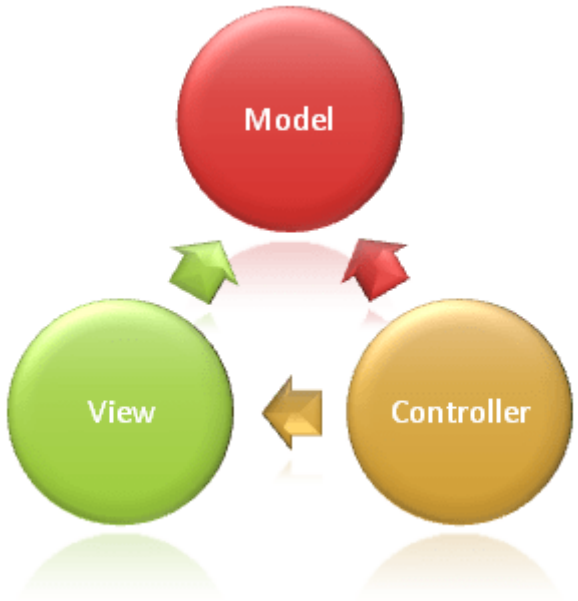
By Steve Smith

ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern.

MVC pattern

The Model-View-Controller (MVC) architectural pattern separates an application into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns. Using this pattern, user requests are routed to a Controller which is responsible for working with the Model to perform user actions and/or retrieve results of queries. The Controller chooses the View to display to the user, and provides it with any Model data it requires.

The following diagram shows the three main components and which ones reference the others:



This delineation of responsibilities helps you scale the application in terms of complexity because it's easier to code, debug, and test something (model, view, or controller) that has a single job. It's more difficult to update, test, and debug code that has dependencies spread across two or more of these three areas. For example, user interface logic tends to change more frequently than business logic. If presentation code and business logic are combined in a single object, an object containing business logic

must be modified every time the user interface is changed. This often introduces errors and requires the retesting of business logic after every minimal user interface change.

> **Note**
>
> Both the view and the controller depend on the model. However, the model depends on neither the view nor the controller. This is one of the key benefits of the separation. This separation allows the model to be built and tested independent of the visual presentation.

Model Responsibilities

The Model in an MVC application represents the state of the application and any business logic or operations that should be performed by it. Business logic should be encapsulated in the model, along with any implementation logic for persisting the state of the application. Strongly-typed views typically use ViewModel types designed to contain the data to display on that view. The controller creates and populates these ViewModel instances from the model.

View Responsibilities

Views are responsible for presenting content through the user interface. They use the Razor view engine to embed .NET code in HTML markup. There should be minimal logic within views, and any logic in them should relate to presenting content. If you find the need to perform a great deal of logic in view files in order to display data from a complex model, consider using a View Component, ViewModel, or view template to simplify the view.

Controller Responsibilities

Controllers are the components that handle user interaction, work with the model, and ultimately select a view to render. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction. In the MVC pattern, the controller is the initial entry point, and is responsible for selecting which model types to work with and which view to render (hence its name - it controls how the app responds to a given request).

> **Note**

Controllers shouldn't be overly complicated by too many responsibilities. To keep controller logic from becoming overly complex, push business logic out of the controller and into the domain model.

> **Tip**
>
> If you find that your controller actions frequently perform the same kinds of actions, move these common actions into **filters**.

ASP.NET Core MVC

The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core.

ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enables a clean separation of concerns. It gives you full control over markup, supports TDD-friendly development and uses the latest web standards.

Routing

ASP.NET Core MVC is built on top of ASP.NET Core's routing, a powerful URL-mapping component that lets you build applications that have comprehensible and searchable URLs. This enables you to define your application's URL naming patterns that work well for search engine optimization (SEO) and for link generation, without regard for how the files on your web server are organized. You can define your routes using a convenient route template syntax that supports route value constraints, defaults and optional values.

Convention-based routing enables you to globally define the URL formats that your application accepts and how each of those formats maps to a specific action method on a given controller. When an incoming request is received, the routing engine parses the URL and matches it to one of the defined URL formats, and then calls the associated controller's action method.

C#

```
routes.MapRoute(name: "Default", template: "
{controller=Home}/{action=Index}/{id?}");
```

Attribute routing enables you to specify routing information by decorating your controllers and actions with attributes that define your application's routes. This means that your route definitions are placed next to the controller and action with which they're associated.

C#

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
      ...
    }
}
```

Model binding

ASP.NET Core MVC model binding converts client request data (form values, route data, query string parameters, HTTP headers) into objects that the controller can handle. As a result, your controller logic doesn't have to do the work of figuring out the incoming request data; it simply has the data as parameters to its action methods.

C#

```
public async Task<IActionResult> Login(LoginViewModel model, string
returnUrl = null) { ... }
```

Model validation

ASP.NET Core MVC supports validation by decorating your model object with data annotation validation attributes. The validation attributes are checked on the client side before values are posted to the server, as well as on the server before the controller action is called.

C#

```
using System.ComponentModel.DataAnnotations;
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }
```

```
    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}
```

A controller action:

C#

```
public async Task<IActionResult> Login(LoginViewModel model, string
returnUrl = null)
{
    if (ModelState.IsValid)
    {
      // work with the model
    }
    // At this point, something failed, redisplay form
    return View(model);
}
```

The framework handles validating request data both on the client and on the server. Validation logic specified on model types is added to the rendered views as unobtrusive annotations and is enforced in the browser with jQuery Validation.

Dependency injection

ASP.NET Core has built-in support for dependency injection (DI). In ASP.NET Core MVC, controllers can request needed services through their constructors, allowing them to follow the Explicit Dependencies Principle.

Your app can also use dependency injection in view files, using the `@inject` directive:

CSHTML

```
@inject SomeService ServiceName

<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ServiceName.GetTitle</title>
</head>
<body>
    <h1>@ServiceName.GetTitle</h1>
```

```
</body>
</html>
```

Filters

Filters help developers encapsulate cross-cutting concerns, like exception handling or authorization. Filters enable running custom pre- and post-processing logic for action methods, and can be configured to run at certain points within the execution pipeline for a given request. Filters can be applied to controllers or actions as attributes (or can be run globally). Several filters (such as `Authorize`) are included in the framework. `[Authorize]` is the attribute that is used to create MVC authorization filters.

C#

```
[Authorize]
public class AccountController : Controller
```

Areas

Areas provide a way to partition a large ASP.NET Core MVC Web app into smaller functional groupings. An area is an MVC structure inside an application. In an MVC project, logical components like Model, Controller, and View are kept in different folders, and MVC uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search etc. Each of these units have their own logical component views, controllers, and models.

Web APIs

In addition to being a great platform for building web sites, ASP.NET Core MVC has great support for building Web APIs. You can build services that reach a broad range of clients including browsers and mobile devices.

The framework includes support for HTTP content-negotiation with built-in support to format data as JSON or XML. Write custom formatters to add support for your own formats.

Use link generation to enable support for hypermedia. Easily enable support for Cross-Origin Resource Sharing (CORS) so that your Web APIs can be shared across multiple

Web applications.

Testability

The framework's use of interfaces and dependency injection make it well-suited to unit testing, and the framework includes features (like a TestHost and InMemory provider for Entity Framework) that make integration tests quick and easy as well. Learn more about how to test controller logic.

Razor view engine

ASP.NET Core MVC views use the Razor view engine to render views. Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code. Razor is used to dynamically generate web content on the server. You can cleanly mix server code with client side content and code.

CSHTML

```
<ul>
    @for (int i = 0; i < 5; i++) {
        <li>List item @i</li>
    }
</ul>
```

Using the Razor view engine you can define layouts, partial views and replaceable sections.

Strongly typed views

Razor views in MVC can be strongly typed based on your model. Controllers can pass a strongly typed model to views enabling your views to have type checking and IntelliSense support.

For example, the following view renders a model of type `IEnumerable<Product>`:

CSHTML

```
@model IEnumerable<Product>
<ul>
    @foreach (Product p in Model)
    {
        <li>@p.Name</li>
```

```
    }
</ul>
```

Tag Helpers

Tag Helpers enable server side code to participate in creating and rendering HTML elements in Razor files. You can use tag helpers to define custom tags (for example, `<environment>`) or to modify the behavior of existing tags (for example, `<label>`). Tag Helpers bind to specific elements based on the element name and its attributes. They provide the benefits of server-side rendering while still preserving an HTML editing experience.

There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in LinkTagHelper can be used to create a link to the `Login` action of the `AccountsController`:

CSHTML

```
<p>
    Thank you for confirming your email.
    Please <a asp-controller="Account" asp-action="Login">Click here to Log
in</a>.
</p>
```

The `EnvironmentTagHelper` can be used to include different scripts in your views (for example, raw or minified) based on the runtime environment, such as Development, Staging, or Production:

CSHTML

```
<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.1.4.js"
            asp-fallback-src="~/lib/jquery/dist/jquery.js"
            asp-fallback-test="window.jQuery">
    </script>
</environment>
```

Tag Helpers provide an HTML-friendly development experience and a rich IntelliSense environment for creating HTML and Razor markup. Most of the built-in Tag Helpers

target existing HTML elements and provide server-side attributes for the element.

View Components

View Components allow you to package rendering logic and reuse it throughout the application. They're similar to partial views, but with associated logic.

Compatibility version

The SetCompatibilityVersion method allows an app to opt-in or opt-out of potentially breaking behavior changes introduced in ASP.NET Core MVC 2.1 or later.

For more information, see Compatibility version for ASP.NET Core MVC.

Additional resources

- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)
- Dependency injection in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Get started with ASP.NET Core MVC

Article • 07/30/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Rick Anderson

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point. See Choose an ASP.NET Core UI, which compares Razor Pages, MVC, and Blazor for UI development.

This is the first tutorial of a series that teaches ASP.NET Core MVC web development with controllers and views.

At the end of the series, you'll have an app that manages and displays movie data. You learn how to:

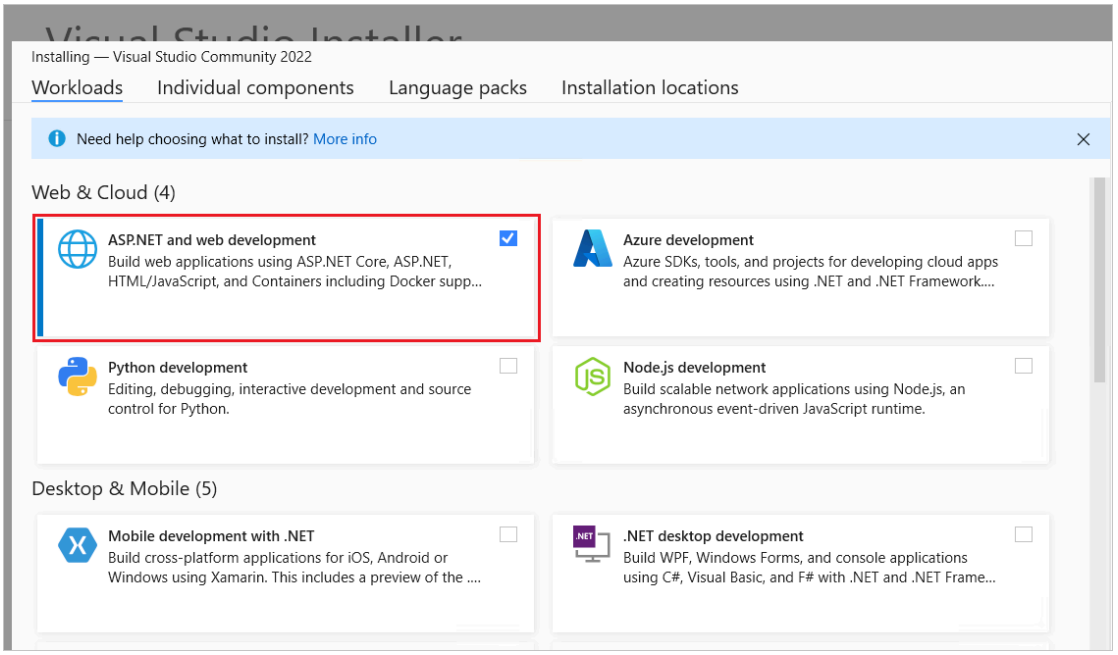
- ✓ Create a web app.
- ✓ Add and scaffold a model.
- ✓ Work with a database.
- ✓ Add search and validation.

View or download sample code (how to download).

Prerequisites

Visual Studio

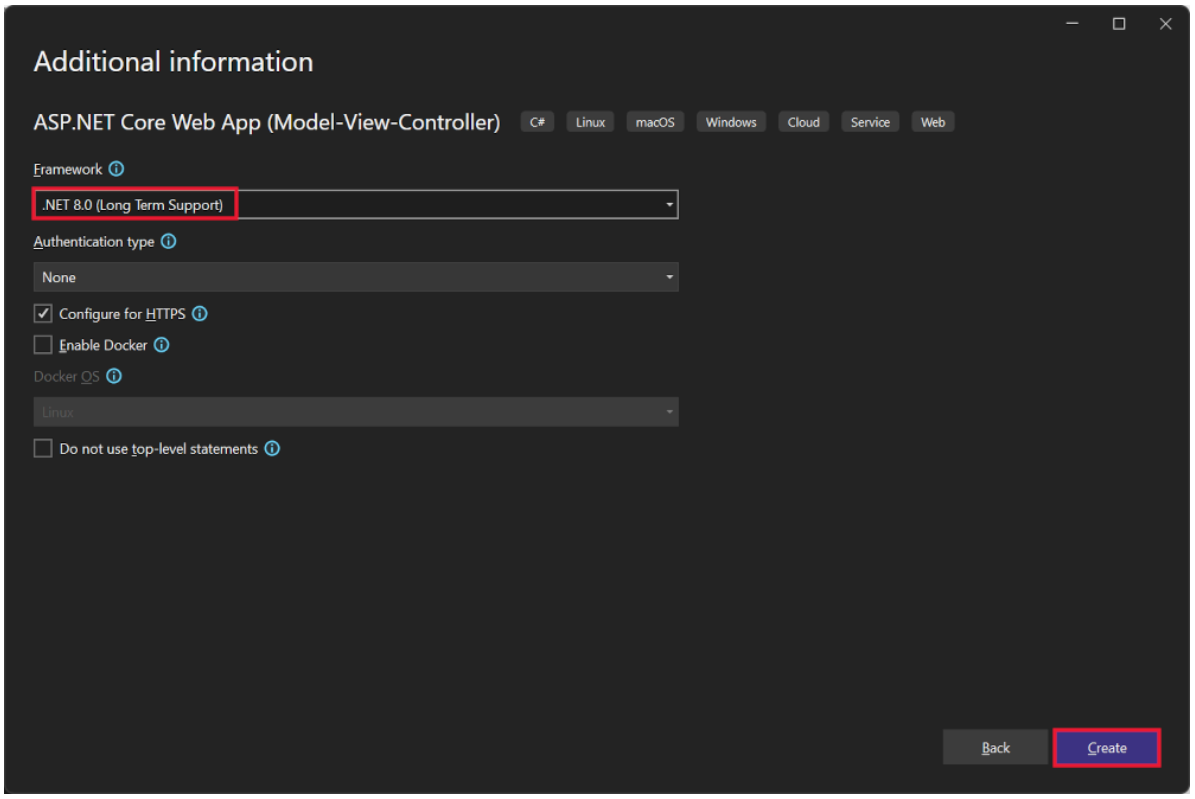
- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a web app

Visual Studio

- Start Visual Studio and select **Create a new project**.
- In the **Create a new project** dialog, select **ASP.NET Core Web App (Model-View-Controller)** > **Next**.
- In the **Configure your new project** dialog:
  - Enter `MvcMovie` for **Project name**. It's important to name the project *MvcMovie*. Capitalization needs to match each `namespace` when code is copied.
  - The **Location** for the project can be set to anywhere.
- Select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 8.0 (Long Term Support)**.
  - Verify that **Do not use top-level statements** is unchecked.
- Select **Create**.



For more information, including alternative approaches to create the project, see Create a new project in Visual Studio.

Visual Studio uses the default project template for the created MVC project. The created project:

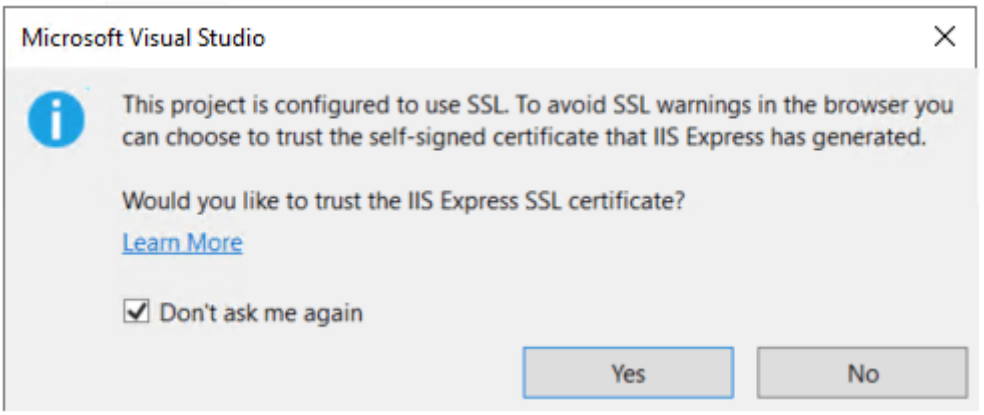
- Is a working app.
- Is a basic starter project.

Run the app

Visual Studio

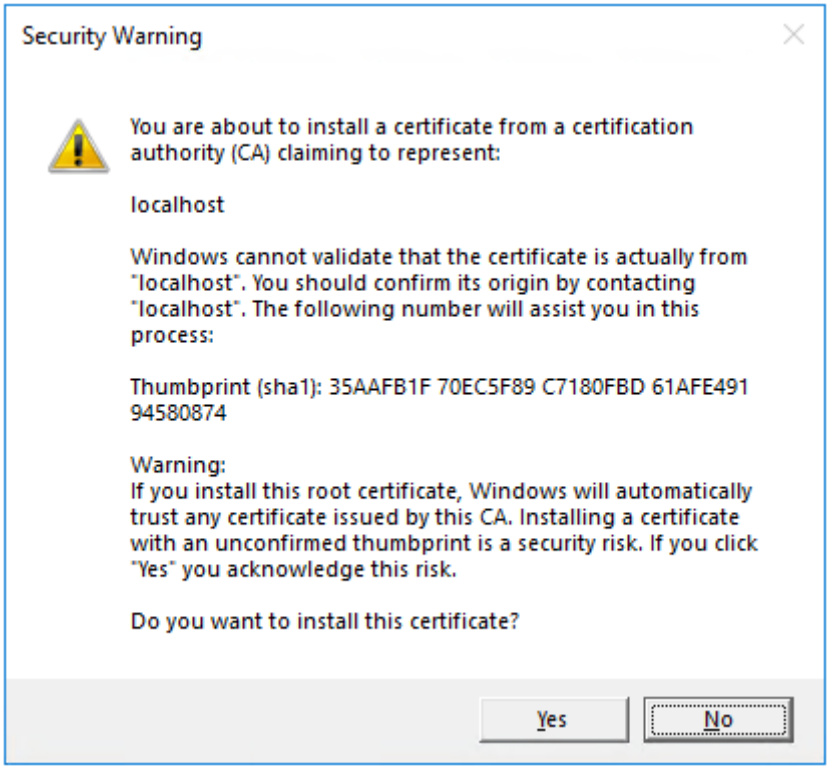
- Press `Ctrl`+`F5` to run the app without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio runs the app and opens the default browser.

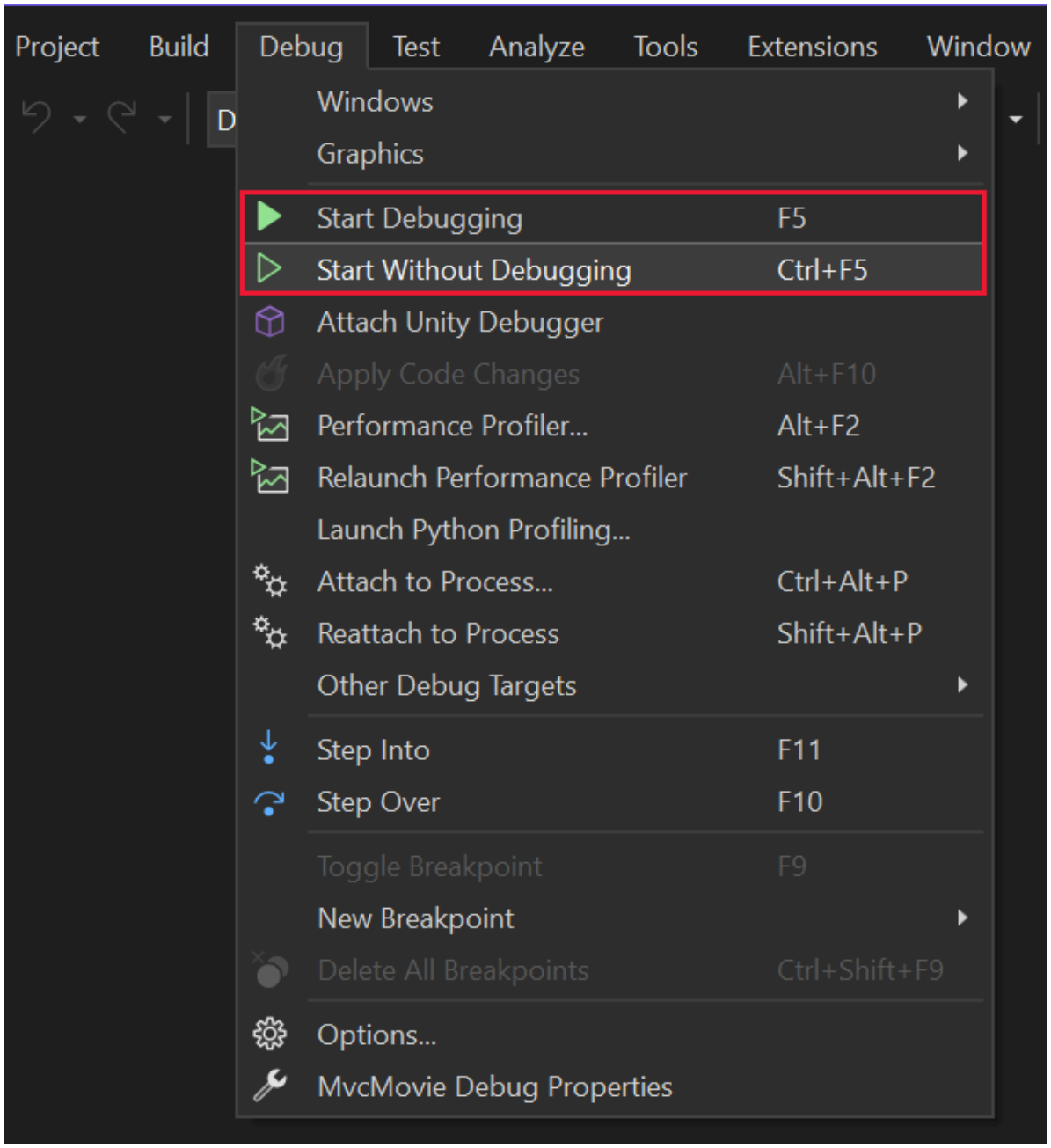
The address bar shows `localhost:<port#>` and not something like `example.com`. The standard hostname for your local computer is `localhost`. When Visual Studio creates a web project, a random port is used for the web server.

Launching the app without debugging by pressing `Ctrl`+`F5` allows you to:

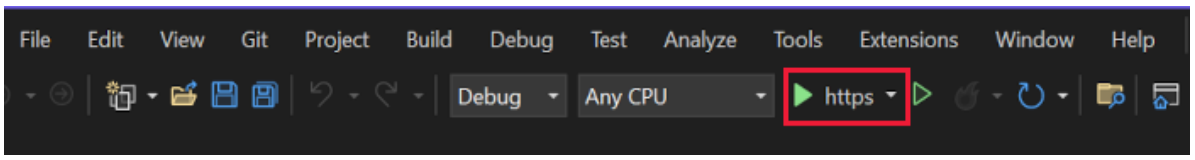
- Make code changes.

- Save the file.
- Quickly refresh the browser and see the code changes.

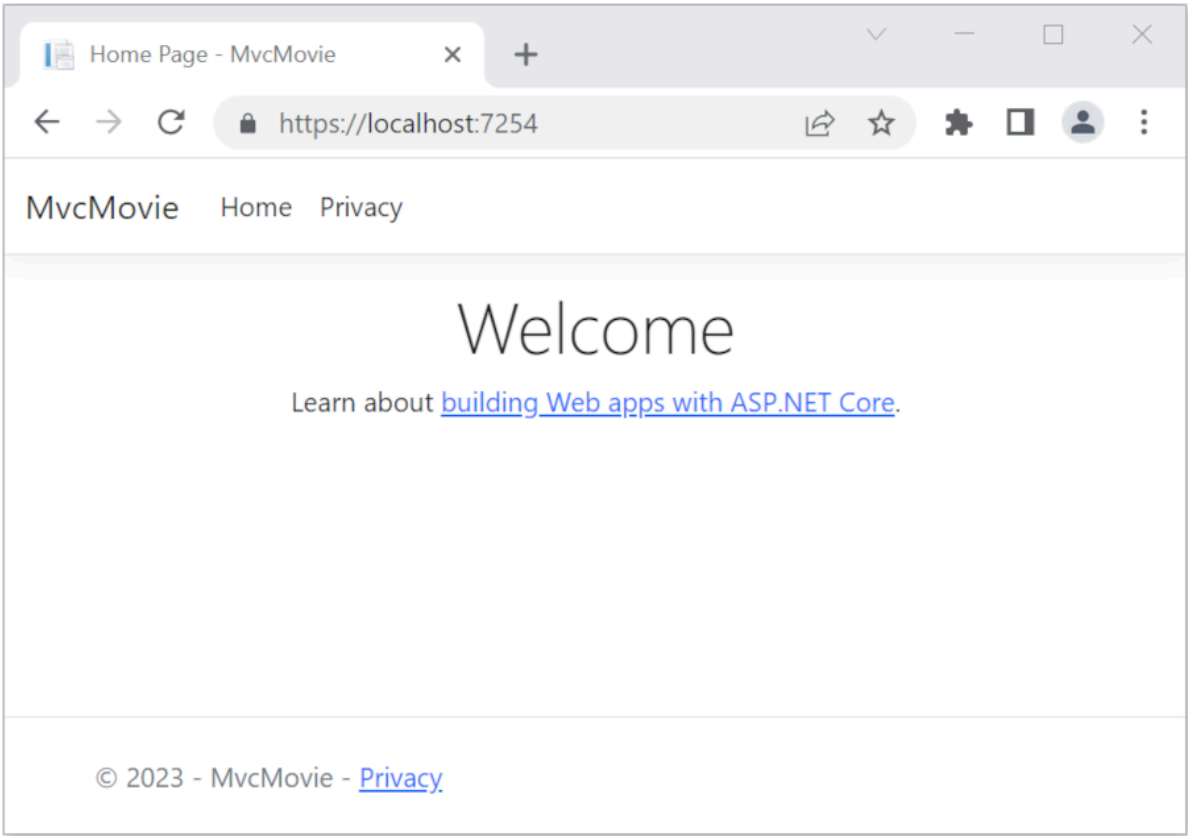
You can launch the app in debug or non-debug mode from the **Debug** menu:



You can debug the app by selecting the **https** button in the toolbar:



The following image shows the app:



- Close the browser window. Visual Studio will stop the application.

Visual Studio

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next tutorial in this series, you learn about MVC and start writing some code.

Next: Add a controller

Part 2, add a controller to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.

MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that:
  - Handle browser requests.
  - Retrieve model data.
  - Call view templates that return a response.

In an MVC app, the view only displays information. The controller handles and responds to user input and interaction. For example, the controller handles URL segments and query-string values, and passes these values to the model. The model might use these values to query the database. For example:

- `https://localhost:5001/Home/Privacy`: specifies the `Home` controller and the `Privacy` action.
- `https://localhost:5001/Movies/Edit/5`: is a request to edit the movie with ID=5 using the `Movies` controller and the `Edit` action, which are detailed later in the tutorial.

Route data is explained later in the tutorial.

The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps manage complexity when building an app,

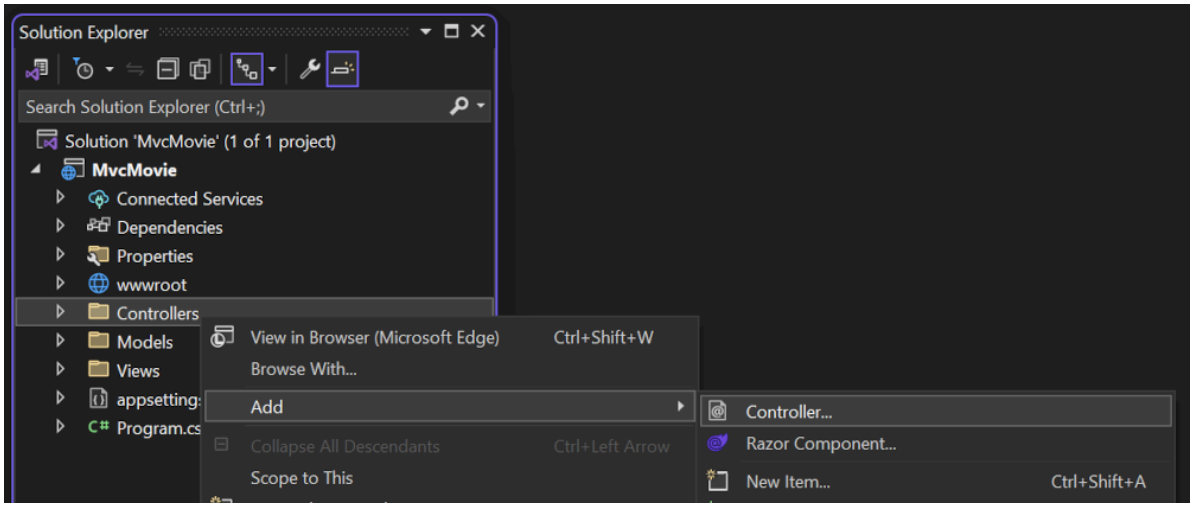
because it enables work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

These concepts are introduced and demonstrated in this tutorial series while building a movie app. The MVC project contains folders for the *Controllers* and *Views*.

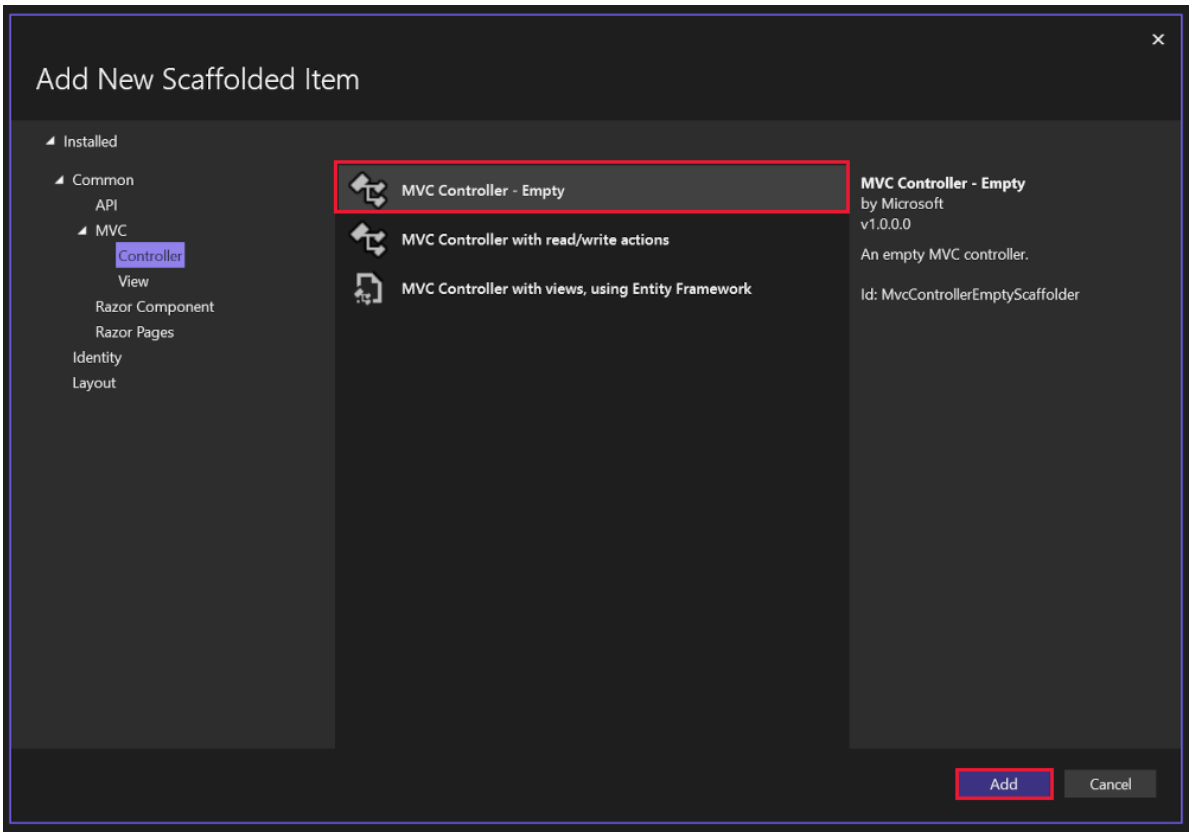
Add a controller

Visual Studio

In **Solution Explorer**, right-click **Controllers** > **Add** > **Controller**.



In the **Add New Scaffolded Item** dialog box, select **MVC Controller - Empty** > **Add**.



In the **Add New Item - MvcMovie** dialog, enter `HelloWorldController.cs` and select **Add**.

Replace the contents of `Controllers/HelloWorldController.cs` with the following code:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    //
    // GET: /HelloWorld/
    public string Index()
    {
        return "This is my default action...";
    }
    //
    // GET: /HelloWorld/Welcome/
    public string Welcome()
    {
        return "This is the Welcome action method...";
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint:

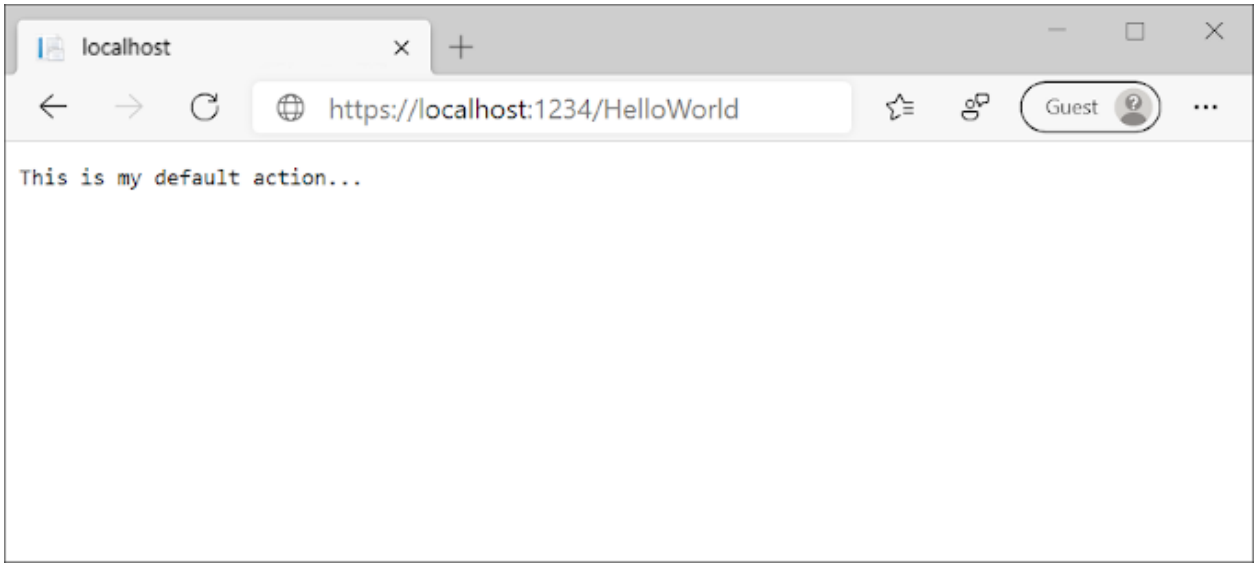
- Is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`.
- Combines:
  - The protocol used: `HTTPS`.
  - The network location of the web server, including the TCP port: `localhost:5001`.
  - The target URI: `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL.

The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial, the scaffolding engine is used to generate `HTTP POST` methods, which update data.

Run the app without the debugger by pressing `Ctrl`+`F5` (Windows) or `⌘`+`F5` (macOS).

Append `/HelloWorld` to the path in the address bar. The `Index` method returns a string.



MVC invokes controller classes, and the action methods within them, depending on the incoming URL. The default URL routing logic used by MVC, uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

The routing format is set in the `Program.cs` file.

C#

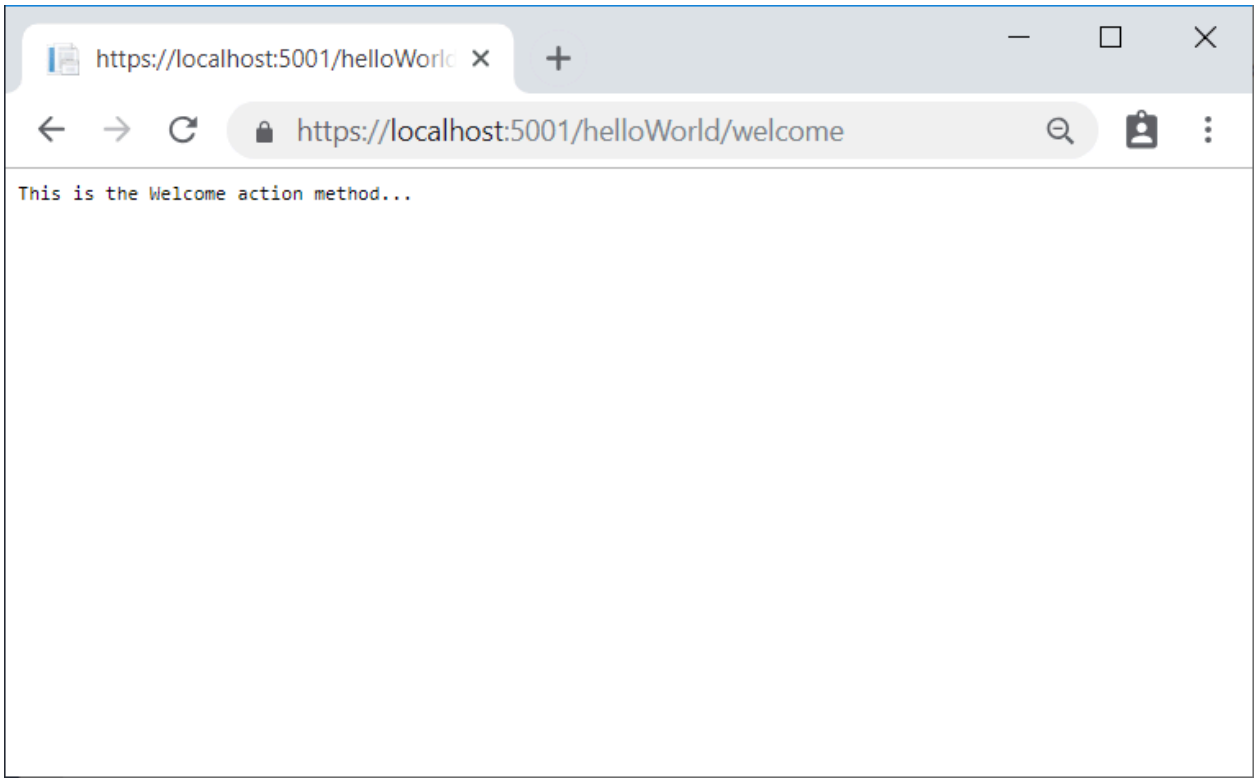
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:

- The first URL segment determines the controller class to run. So `localhost:5001/HelloWorld` maps to the **HelloWorld** Controller class.
- The second part of the URL segment determines the action method on the class. So `localhost:5001/HelloWorld/Index` causes the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:5001/HelloWorld` and the `Index` method was called by default. `Index` is the default method that will be called on a controller if a method name isn't explicitly specified.
- The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to: `https://localhost:{PORT}/HelloWorld/Welcome`. Replace `{PORT}` with your port number.

The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`.

Change the `Welcome` method to include two parameters as shown in the following code.

C#

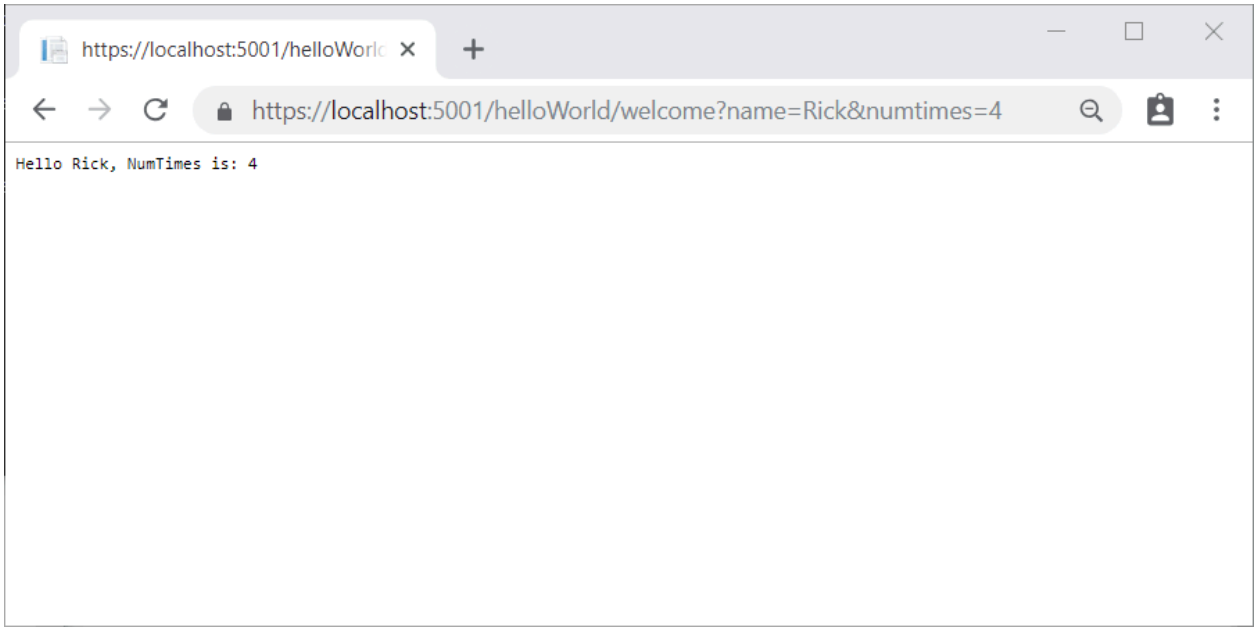
```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is:
{numTimes}");
}
```

The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input, such as through JavaScript.
- Uses Interpolated Strings in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to: `https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`. Replace `{PORT}` with your port number.

Try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string to parameters in the method. See Model Binding for more information.



In the previous image:

- The URL segment `Parameters` isn't used.
- The `name` and `numTimes` parameters are passed in the query string.
- The `?` (question mark) in the above URL is a separator, and the query string follows.
- The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

C#

```
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick`

In the preceding URL:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` starts the query string.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

In the preceding example:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` (in `id?`) indicates the `id` parameter is optional.

Previous: Get Started Next: Add a View

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 3, add a view to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section, you modify the `HelloWorldController` class to use Razor view files. This cleanly encapsulates the process of generating HTML responses to a client.

View templates are created using Razor. Razor-based view templates:

- Have a `.cshtml` file extension.
- Provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

C#

```
public IActionResult Index()
{
    return View();
}
```

The preceding code:

- Calls the controller's View method.
- Uses a view template to generate an HTML response.

Controller methods:

- Are referred to as *action methods*. For example, the `Index` action method in the preceding code.
- Generally return an IActionResult or a class derived from ActionResult, not a type like `string`.

Add a view

Visual Studio

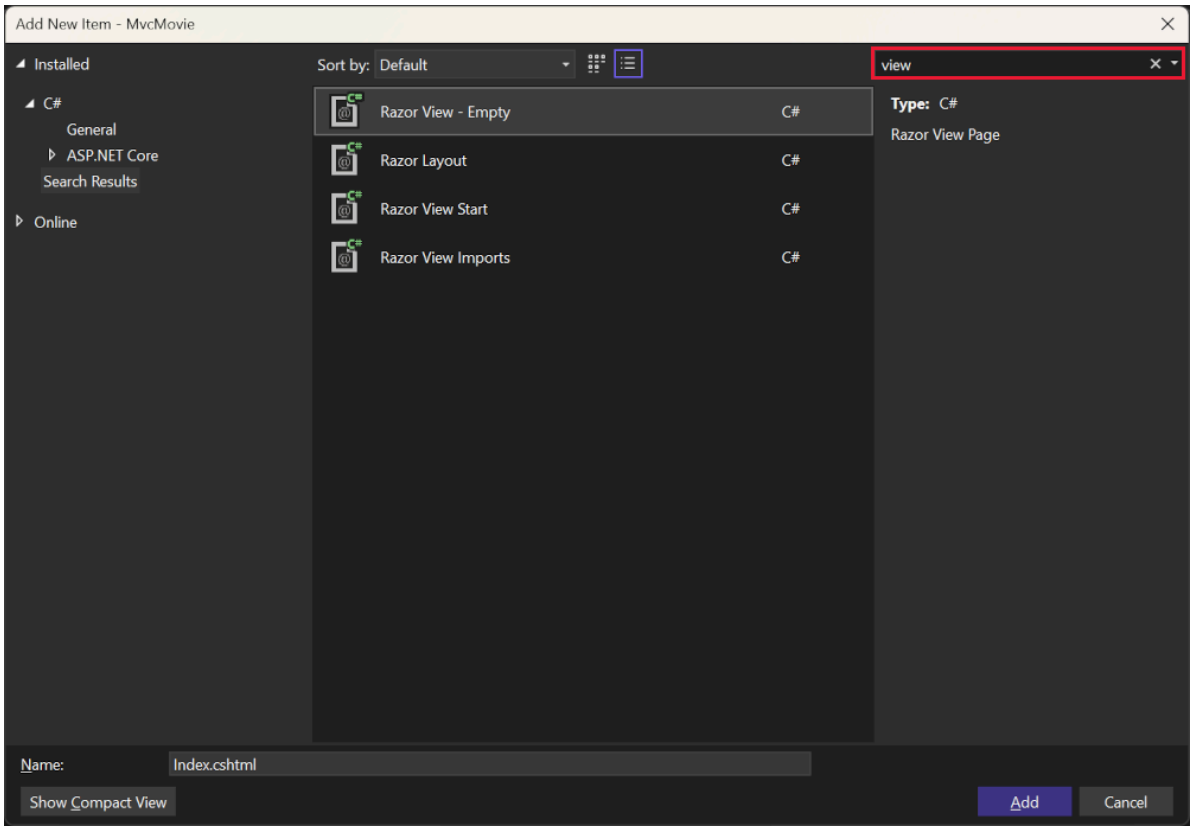
Right-click on the *Views* folder, and then **Add** > **New Folder** and name the folder *HelloWorld*.

Right-click on the *Views/HelloWorld* folder, and then **Add** > **New Item**.

In the **Add New Item** dialog select **Show All Templates**.

In the **Add New Item - MvcMovie** dialog:

- In the search box in the upper-right, enter *view*
- Select **Razor View - Empty**
- Keep the **Name** box value, `Index.cshtml`.
- Select **Add**



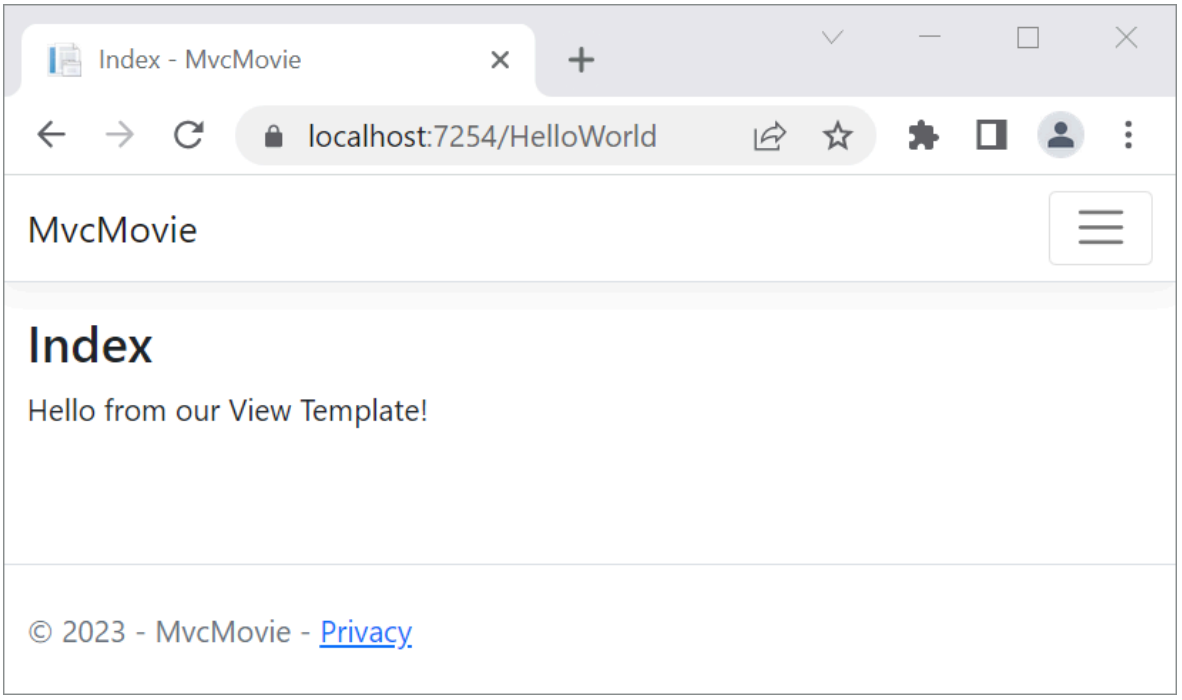
Replace the contents of the `Views/HelloWorld/Index.cshtml` Razor view file with the following:

CSHTML

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://localhost:{PORT}/HelloWorld`:

- The `Index` method in the `HelloWorldController` ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser.
- A view template file name wasn't specified, so MVC defaulted to using the default view file. When the view file name isn't specified, the default view is returned. The default view has the same name as the action method, `Index` in this example. The view template `/Views/HelloWorld/Index.cshtml` is used.
- The following image shows the string "Hello from our View Template!" hard-coded in the view:



Change views and layout pages

Select the menu links **MvcMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the `Views/Shared/_Layout.cshtml` file.

Open the `Views/Shared/_Layout.cshtml` file.

Layout templates allow:

- Specifying the HTML container layout of a site in one place.
- Applying the HTML container layout across multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the *`Views/Home/Privacy.cshtml`* view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

Replace the content of the `Views/Shared/_Layout.cshtml` file with the following markup. The changes are highlighted:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Movies"
asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
```

```
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2023 - Movie App - <a asp-area="" asp-controller="Home"
asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

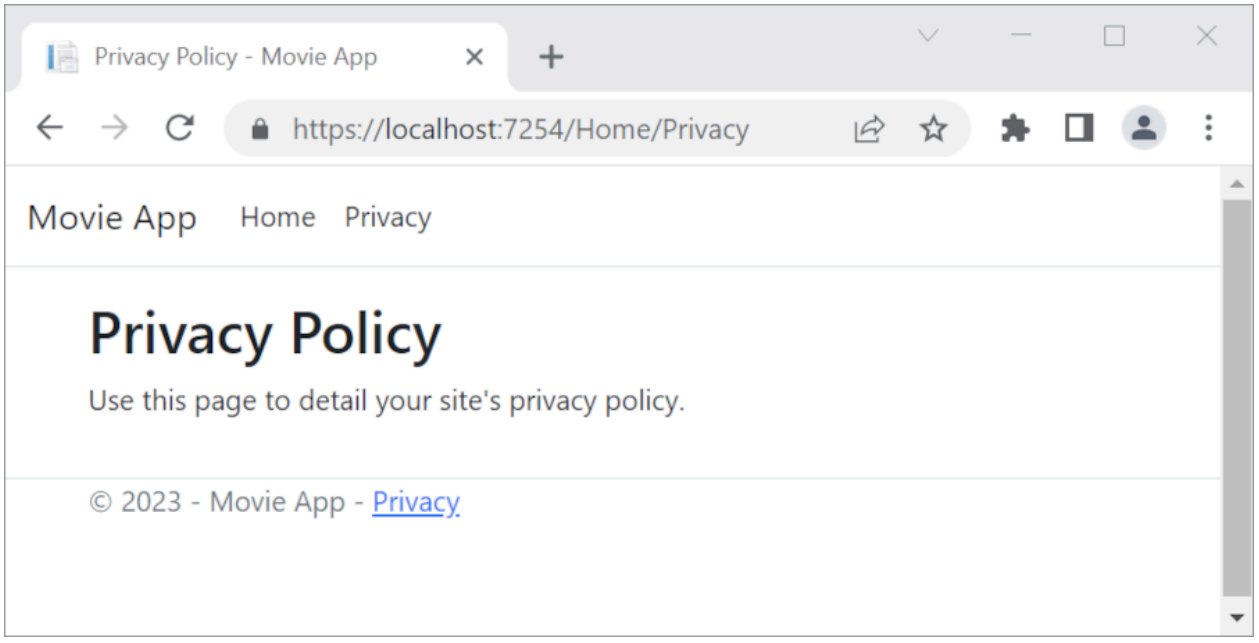
The preceding markup made the following changes:

- Three occurrences of `MvcMovie` to `Movie App`.
- The anchor element `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

In the preceding markup, the `asp-area=""` anchor Tag Helper attribute and attribute value was omitted because this app isn't using Areas.

Note: The `Movies` controller hasn't been implemented. At this point, the `Movie App` link isn't functional.

Save the changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - MvcMovie**



Select the **Home** link.

Notice that the title and anchor text display **Movie App**. The changes were made once in the layout template and all pages on the site reflect the new link text and new title.

Examine the `Views/_ViewStart.cshtml` file:

CSHTML

```
@{
    Layout = "_Layout";
}
```

The `Views/_ViewStart.cshtml` file brings in the `Views/Shared/_Layout.cshtml` file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Open the `Views/HelloWorld/Index.cshtml` view file.

Change the title and `<h2>` element as highlighted in the following:

CSHTML

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>

<p>Hello from our View Template!</p>
```

The title and `<h2>` element are slightly different so it's clear which part of the code changes the display.

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

CSHTML

```
<title>@ViewData["Title"] - Movie App</title>
```

Save the change and navigate to `https://localhost:{PORT}/HelloWorld`.

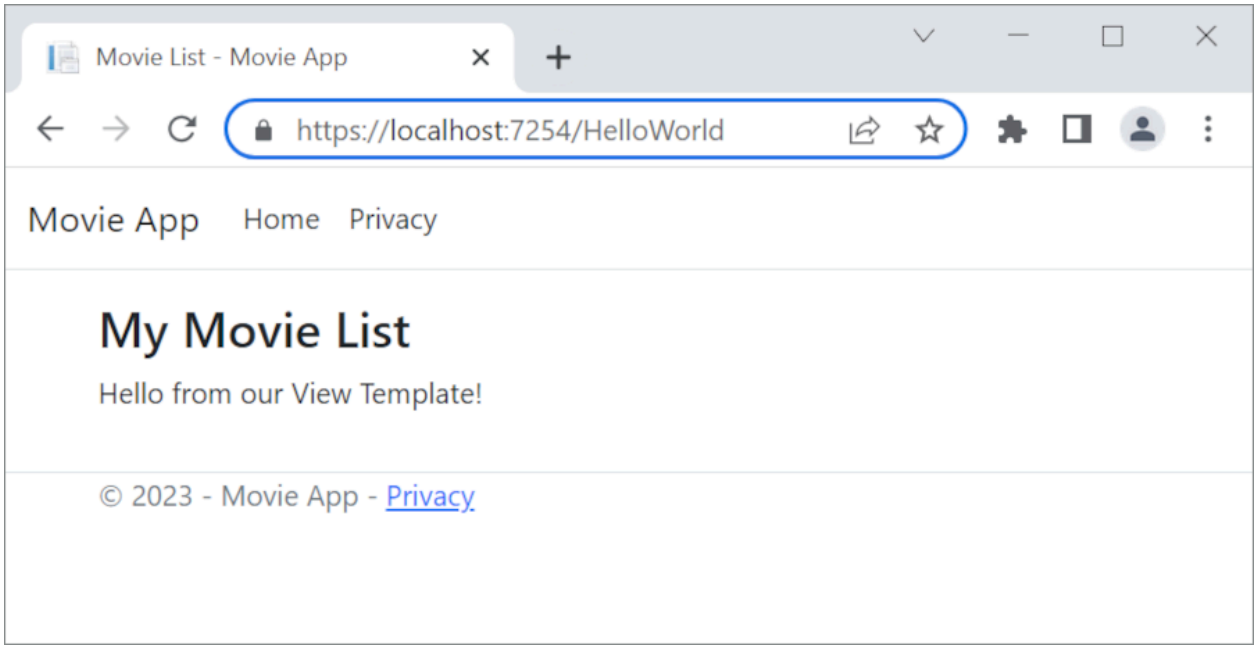
Notice that the following have changed:

- Browser title.

- Primary heading.
- Secondary headings.

If there are no changes in the browser, it could be cached content that is being viewed. Press Ctrl+F5 in the browser to force the response from the server to be loaded. The browser title is created with `ViewData["Title"]` we set in the `Index.cshtml` view template and the additional "- Movie App" added in the layout file.

The content in the `Index.cshtml` view template is merged with the `Views/Shared/_Layout.cshtml` view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app. To learn more, see Layout.



The small bit of "data", the "Hello from our View Template!" message, is hard-coded however. The MVC application has a "V" (view), a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response.

View templates should **not**:

- Do business logic
- Interact with a database directly.

A view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code:

- Clean.
- Testable.
- Maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and an `ID` parameter and then outputs the values directly to the browser.

Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate data must be passed from the controller to the view to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary. The view template can then access the dynamic data.

In `HelloWorldController.cs`, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary.

The `ViewData` dictionary is a dynamic object, which means any type can be used. The `ViewData` object has no defined properties until something is added. The MVC model binding system automatically maps the named parameters `name` and `numTimes` from the query string to parameters in the method. The complete `HelloWorldController`:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
```

```
    public IActionResult Welcome(string name, int numTimes = 1)
    {
        ViewData["Message"] = "Hello " + name;
        ViewData["NumTimes"] = numTimes;
        return View();
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named `Views/HelloWorld/Welcome.cshtml`.

You'll create a loop in the `Welcome.cshtml` view template that displays "Hello" `NumTimes`. Replace the contents of `Views/HelloWorld/Welcome.cshtml` with the following:

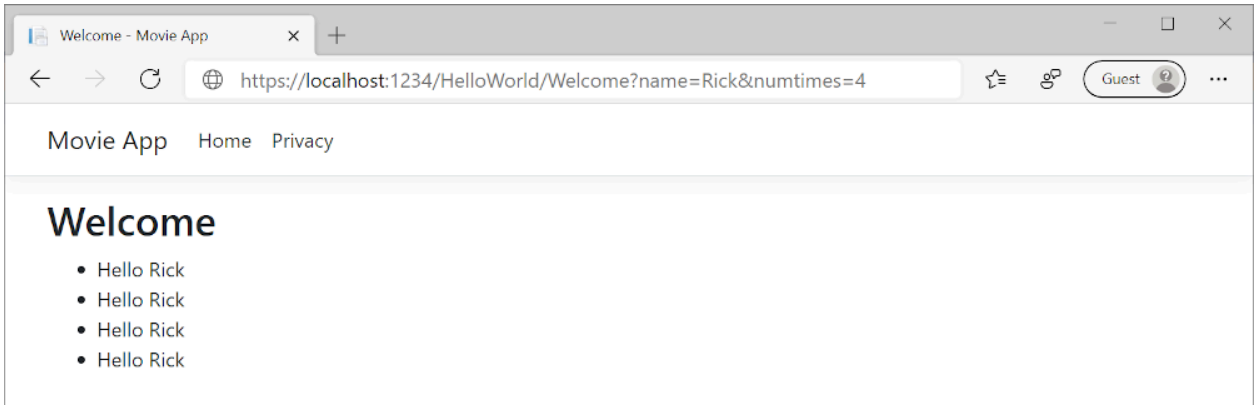
CSHTML

```
@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder. The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.



In the preceding sample, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is preferred over the `ViewData` dictionary approach.

In the next tutorial, a database of movies is created.

Previous: Add a Controller | Next: Add a Model

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 4, add a model to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson and Jon P Smith.

In this tutorial, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.

These model classes are used with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes created are known as ***POCO*** classes, from **P**lain **O**ld **C**LR **O**bjects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.

In this tutorial, model classes are created first, and EF Core creates the database.

Add a data model class

Visual Studio

Right-click the *Models* folder > **Add** > **Class**. Name the file `Movie.cs`.

Update the `Models/Movie.cs` file with the following code:

C#

```
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    public decimal Price { get; set; }
}
```

The `Movie` class contains an `Id` field, which is required by the database for the primary key.

The DataType attribute on `ReleaseDate` specifies the type of the data (`Date`). With this attribute:

- The user isn't required to enter time information in the date field.
- Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

The question mark after `string` indicates that the property is nullable. For more information, see Nullable reference types.

Add NuGet packages

Visual Studio

Visual Studio automatically installs the required packages.

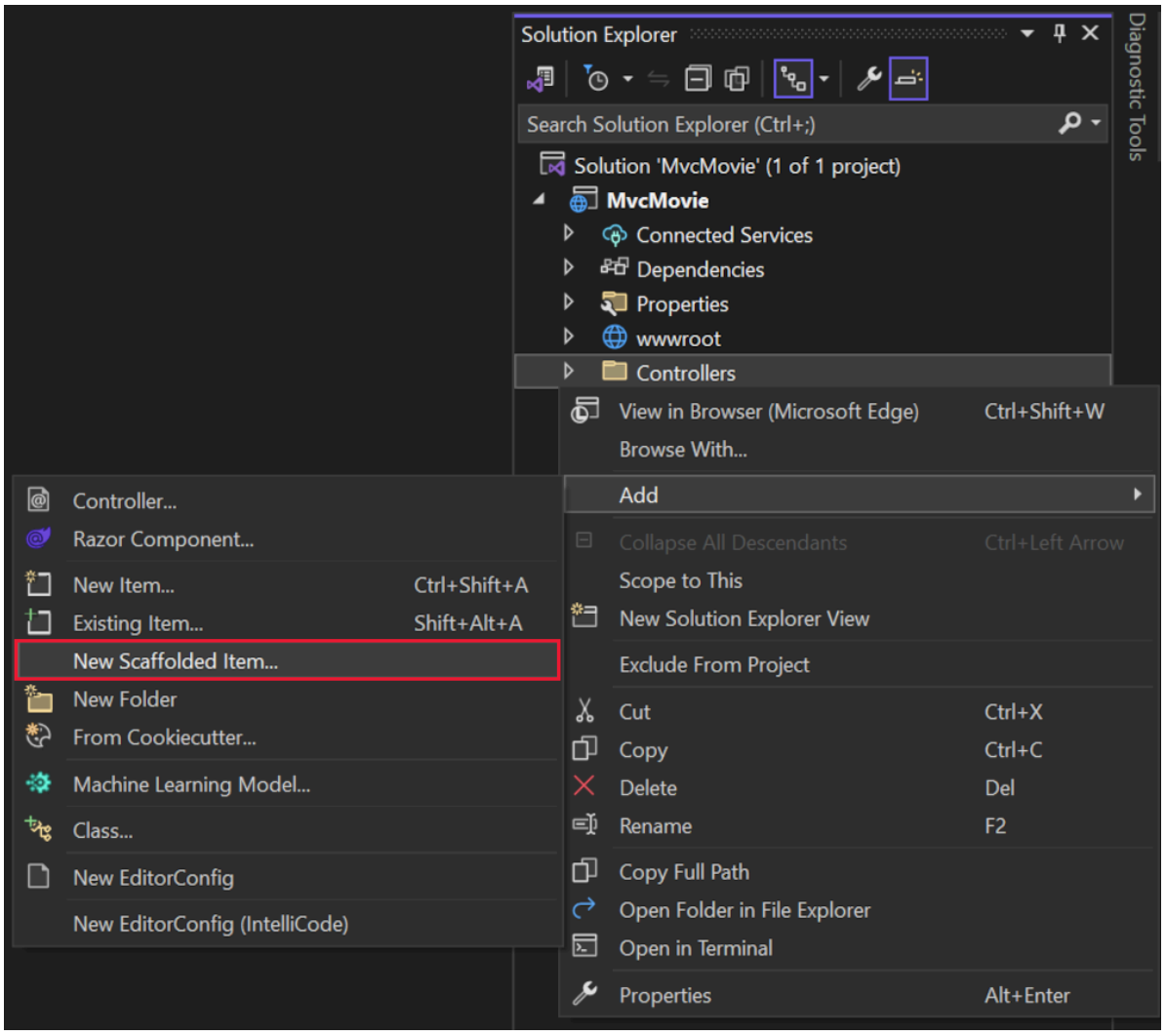
Build the project as a check for compiler errors.

Scaffold movie pages

Use the scaffolding tool to produce `Create`, `Read`, `Update`, and `Delete` (CRUD) pages for the movie model.

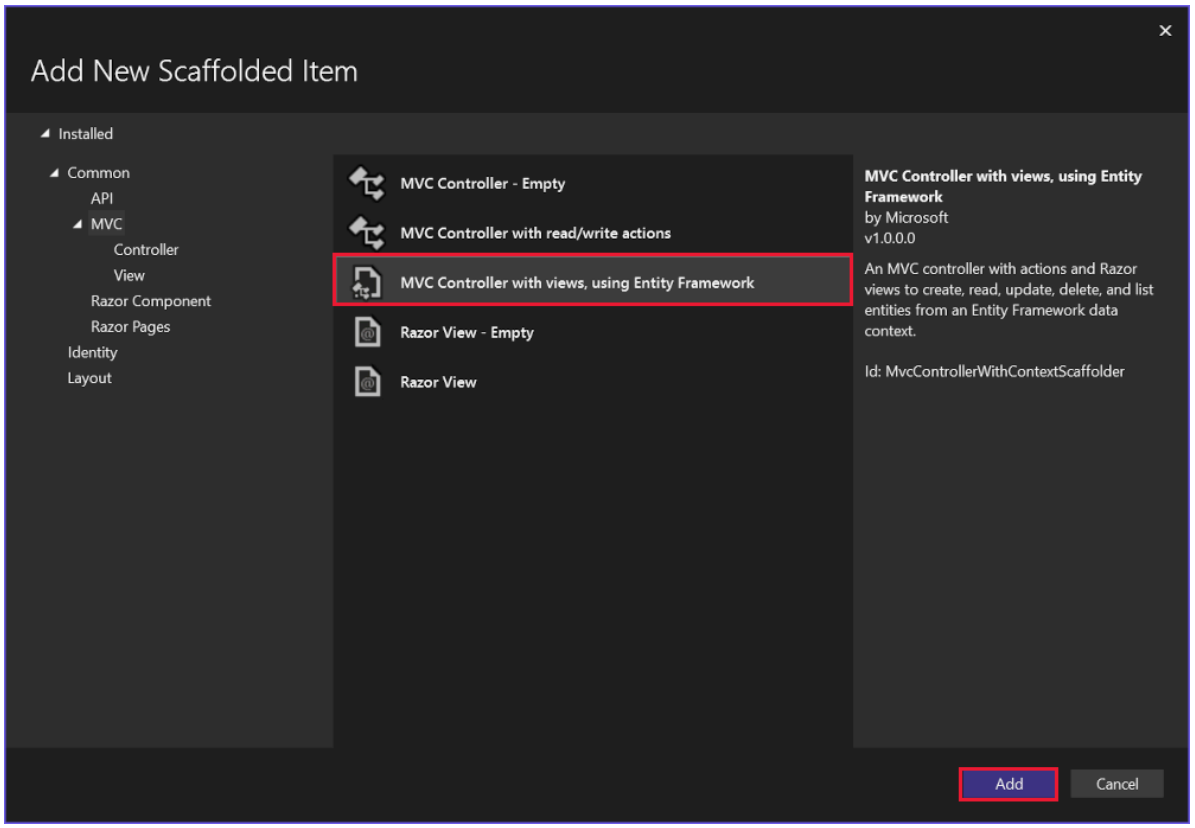
Visual Studio

In **Solution Explorer**, right-click the *Controllers* folder and select **Add** > **New Scaffolded Item**.



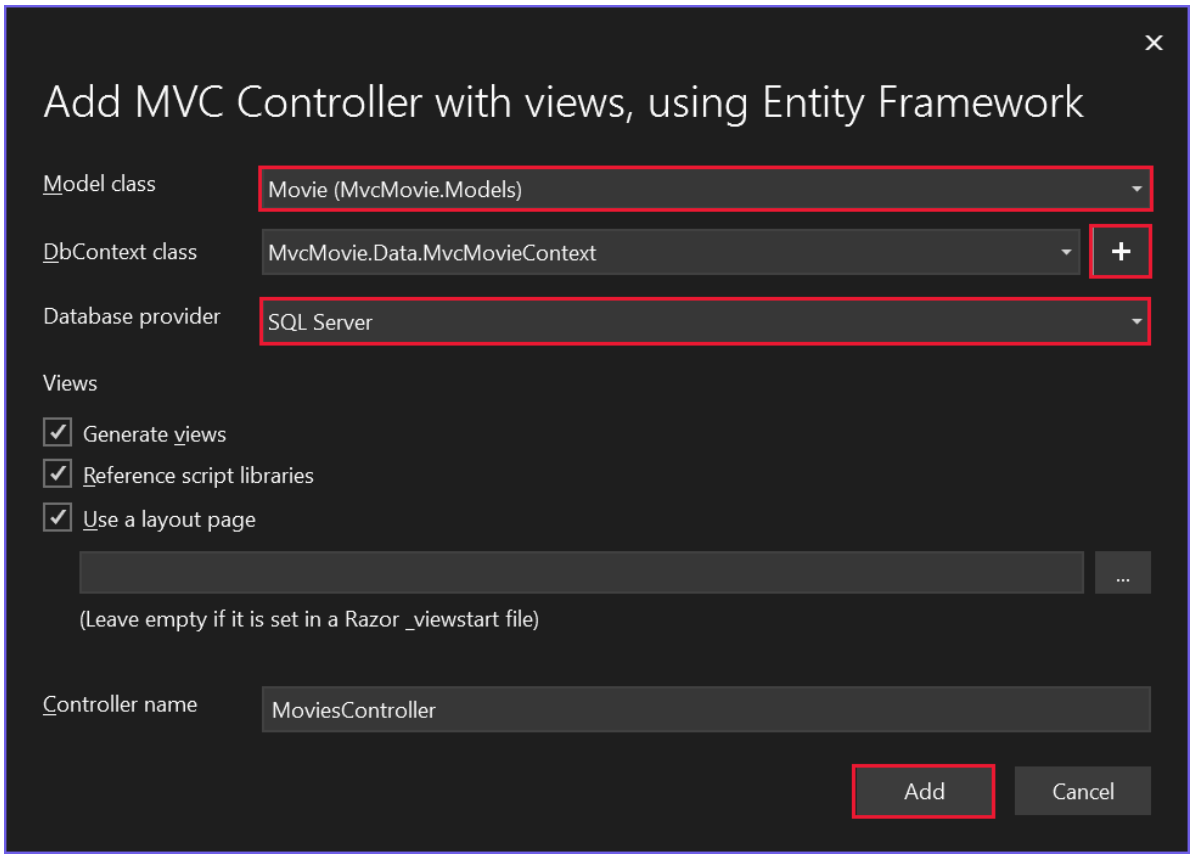
In the **Add New Scaffolded Item** dialog:

- In the left pane, select **Installed** > **Common** > **MVC**.
- Select **MVC Controller with views, using Entity Framework**.
- Select **Add**.



Complete the **Add MVC Controller with views, using Entity Framework** dialog:

- In the **Model class** drop down, select **Movie (MvcMovie.Models)**.
- In the **Data context class** row, select the + (plus) sign.
  - In the **Add Data Context** dialog, the class name *MvcMovie.Data.MvcMovieContext* is generated.
  - Select **Add**.
- In the **Database provider** drop down, select **SQL Server**.
- **Views** and **Controller name**: Keep the default.
- Select **Add**.



If you get an error message, select **Add** a second time to try it again.

Scaffolding adds the following packages:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

Scaffolding creates the following:

- A movies controller: `Controllers/MoviesController.cs`
- Razor view files for **Create**, **Delete**, **Details**, **Edit**, and **Index** pages: `Views/Movies/*.cshtml`
- A database context class: `Data/MvcMovieContext.cs`

Scaffolding updates the following:

- Inserts required package references in the `MvcMovie.csproj` project file.
- Registers the database context in the `Program.cs` file.
- Adds a database connection string to the `appsettings.json` file.

The automatic creation of these files and file updates is known as *scaffolding*.

The scaffolded pages can't be used yet because the database doesn't exist. Running the app and selecting the **Movie App** link results in a *Cannot open database* or *no*

such table: Movie error message.

Build the app to verify that there are no errors.

Initial migration

Use the EF Core Migrations feature to create the database. *Migrations* is a set of tools that create and update a database to match the data model.

Visual Studio

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console** .

In the Package Manager Console (PMC), enter the following commands:

```
PowerShell

Add-Migration InitialCreate
Update-Database
```

- `Add-Migration InitialCreate`: Generates a `Migrations/{timestamp}_InitialCreate.cs` migration file. The `InitialCreate` argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the `MvcMovieContext` class.
- `Update-Database`: Updates the database to the latest migration, which the previous command created. This command runs the `Up` method in the `Migrations/{time-stamp}_InitialCreate.cs` file, which creates the database.

The `Update-Database` command generates the following warning:

> No store type was specified for the decimal property 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.

Ignore the preceding warning, it's fixed in a later tutorial.

For more information on the PMC tools for EF Core, see EF Core tools reference - PMC in Visual Studio.

Test the app

Visual Studio

Run the app and select the **Movie App** link.

If you get an exception similar to the following, you may have missed the `Update-Database` command in the migrations step:

```
Console

SqlException: Cannot open database "MvcMovieContext-1" requested by the
login. The login failed.
```

> **Note**
>
> You may not be able to enter decimal commas in the `Price` field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point and for non US-English date formats, the app must be globalized. For globalization instructions, see this GitHub issue.

Examine the generated database context class and registration

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. The database context is derived from Microsoft.EntityFrameworkCore.DbContext and specifies the entities to include in the data model.

Scaffolding creates the `Data/MvcMovieContext.cs` database context class:

```
C#
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<MvcMovie.Models.Movie> Movie { get; set; }
    }
}
```

The preceding code creates a DbSet<Movie> property that represents the movies in the database.

Dependency injection

ASP.NET Core is built with dependency injection (DI). Services, such as the database context, are registered with DI in `Program.cs`. These services are provided to components that require them via constructor parameters.

In the `Controllers/MoviesController.cs` file, the constructor uses Dependency Injection to inject the `MvcMovieContext` database context into the controller. The database context is used in each of the CRUD methods in the controller.

Scaffolding generated the following highlighted code in `Program.cs`:

Visual Studio

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context")));
```

The ASP.NET Core configuration system reads the "MvcMovieContext" database connection string.

Examine the generated database connection string

Scaffolding added a connection string to the `appsettings.json` file:

Visual Studio

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Data Source=MvcMovieContext-ea7a4069-f366-4742-
bd1c-3f753a804ce1.db"
  }
}
```

For local development, the ASP.NET Core configuration system reads the `ConnectionString` key from the `appsettings.json` file.

The `InitialCreate` class

Examine the `Migrations/{timestamp}_InitialCreate.cs` migration file:

C#

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace MvcMovie.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Movie",
```

```
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Title = table.Column<string>(type: "nvarchar(max)",
nullable: true),
                    ReleaseDate = table.Column<DateTime>(type: "datetime2",
nullable: false),
                    Genre = table.Column<string>(type: "nvarchar(max)",
nullable: true),
                    Price = table.Column<decimal>(type: "decimal(18,2)",
nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Movie", x => x.Id);
                });
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Movie");
        }
    }
}
```

In the preceding code:

- `InitialCreate.Up` creates the Movie table and configures `Id` as the primary key.
- `InitialCreate.Down` reverts the schema changes made by the `Up` migration.

Dependency injection in the controller

Open the `Controllers/MoviesController.cs` file and examine the constructor:

C#

```
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the

CRUD methods in the controller.

Test the **Create** page. Enter and submit data.

Test the **Edit**, **Details**, and **Delete** pages.

Strongly typed models and the `@model` directive

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism passed a strongly typed model in the `MoviesController` class and views.

Examine the generated `Details` method in the `Controllers/MoviesController.cs` file:

C#

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example, `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller, the first URL segment.
- The action to `details`, the second URL segment.
- The `id` to 1, the last URL segment.

The `id` can be passed in with a query string, as in the following example:

`https://localhost:5001/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in cases when the `id` value isn't provided.

A lambda expression is passed in to the FirstOrDefaultAsync method to select movie entities that match the route data or query string value.

C#

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

C#

```
return View(movie);
```

Examine the contents of the `Views/Movies/Details.cshtml` file:

CSHTML

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
```

```
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

The `@model` statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following `@model` statement was included:

CSHTML

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows access to the movie that the controller passed to the view. The `Model` object is strongly typed. For example, in the `Details.cshtml` view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the `Index.cshtml` view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this `Movies` list from the `Index` action method to the view:

C#

```
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

The code returns problem details if the `Movie` property of the data context is null.

When the movies controller was created, scaffolding included the following `@model` statement at the top of the `Index.cshtml` file:

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows access to the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the `Index.cshtml` view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
```

```
                @Html.DisplayFor(modelItem => item.ReleaseDate)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Genre)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Price)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a>
|
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Because the `Model` object is strongly typed as an `IEnumerable<Movie>` object, each item in the loop is typed as `Movie`. Among other benefits, the compiler validates the types used in the code.

Additional resources

- Entity Framework Core for Beginners
- Tag Helpers
- Globalization and localization

Previous: Adding a View | Next: Working with SQL

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 5, work with a database in an ASP.NET Core MVC app

Article • 09/10/2024

By Rick Anderson and Jon P Smith.

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `Program.cs` file:

Visual Studio

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context")));
```

The ASP.NET Core Configuration system reads the `ConnectionString` key. For local development, it gets the connection string from the `appsettings.json` file:

JSON

```
"ConnectionStrings": {
  "MvcMovieContext": "Data Source=MvcMovieContext-ea7a4069-f366-4742-
bd1c-3f753a804ce1.db"
}
```

> **Note**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Visual Studio

SQL Server Express LocalDB

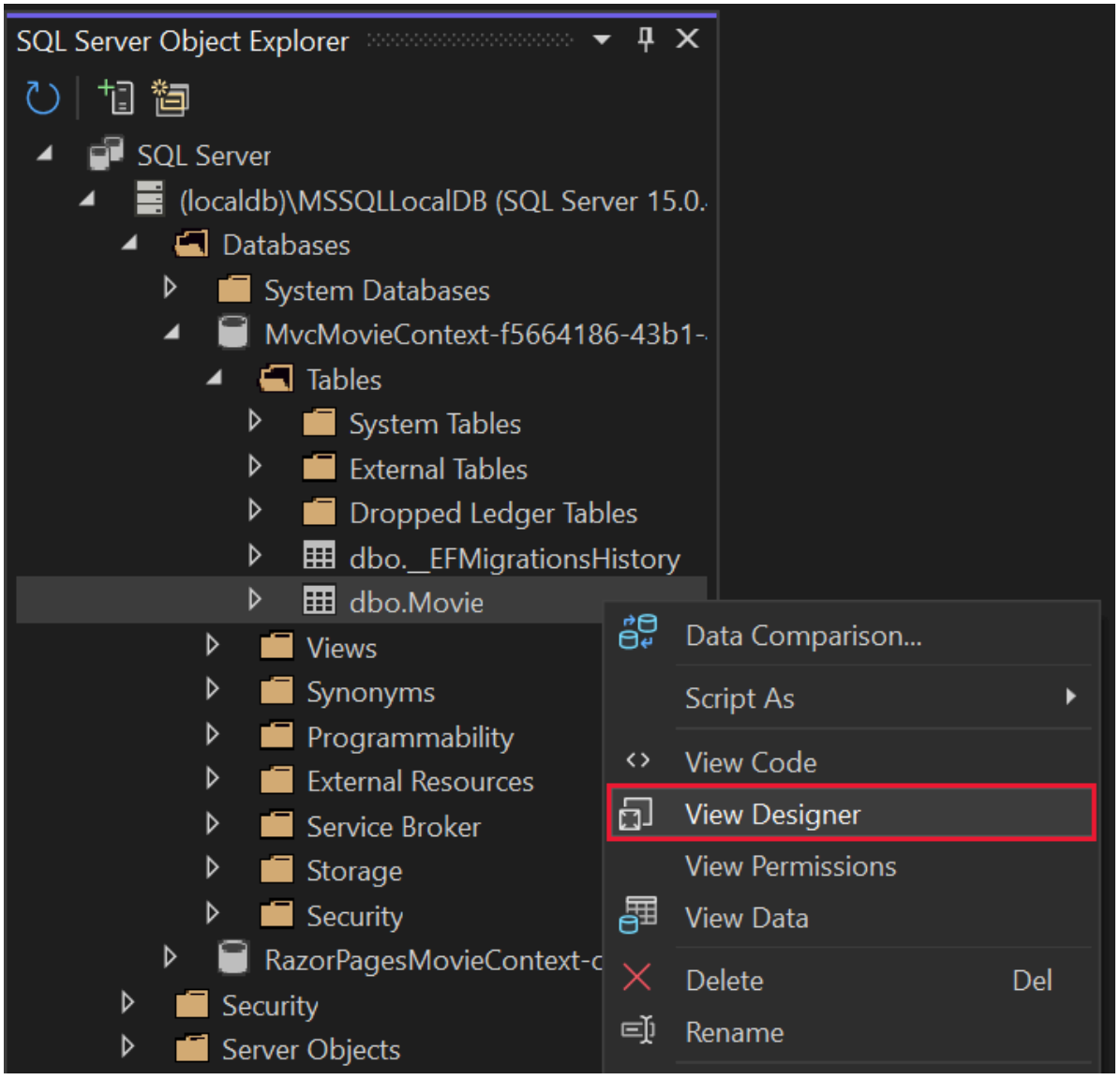
LocalDB:

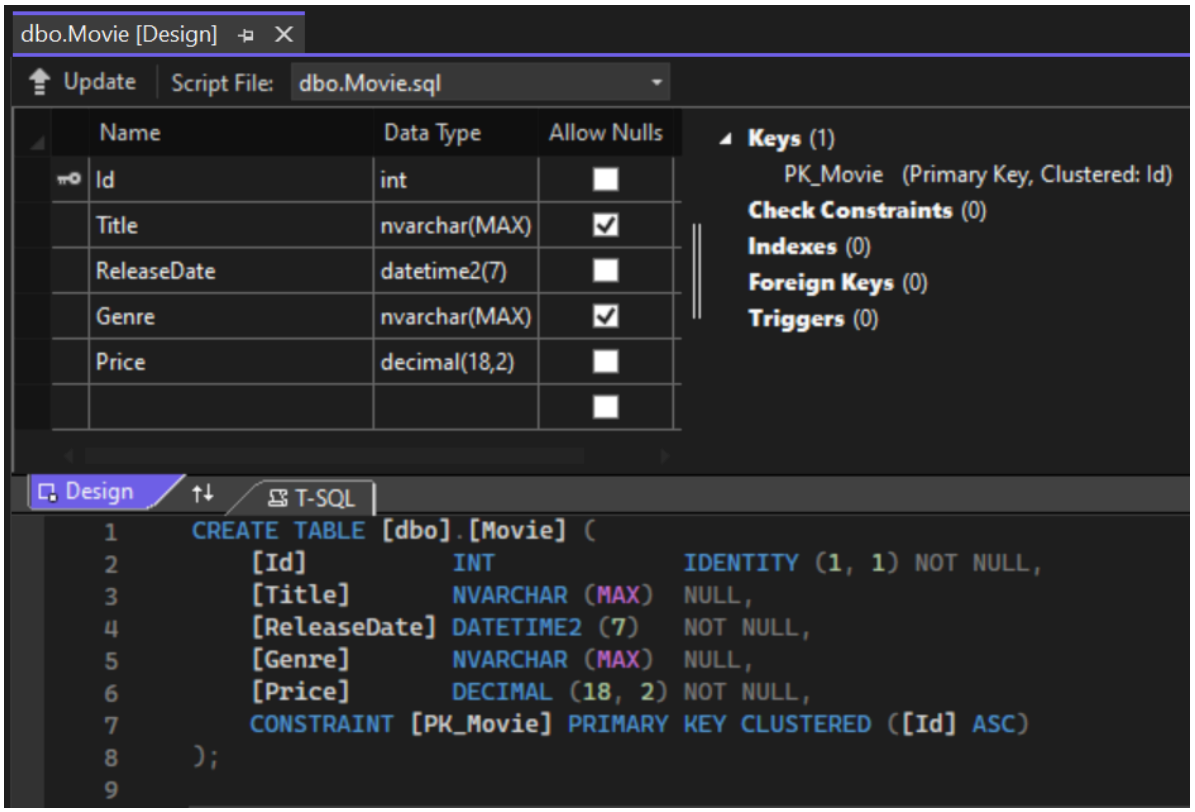
- Is a lightweight version of the SQL Server Express Database Engine, installed by default with Visual Studio.
- Starts on demand by using a connection string.
- Is targeted for program development. It runs in user mode, so there's no complex configuration.
- By default creates *.mdf* files in the *C:/Users/{user}* directory.

Examine the database

From the **View** menu, open **SQL Server Object Explorer** (SSOX).

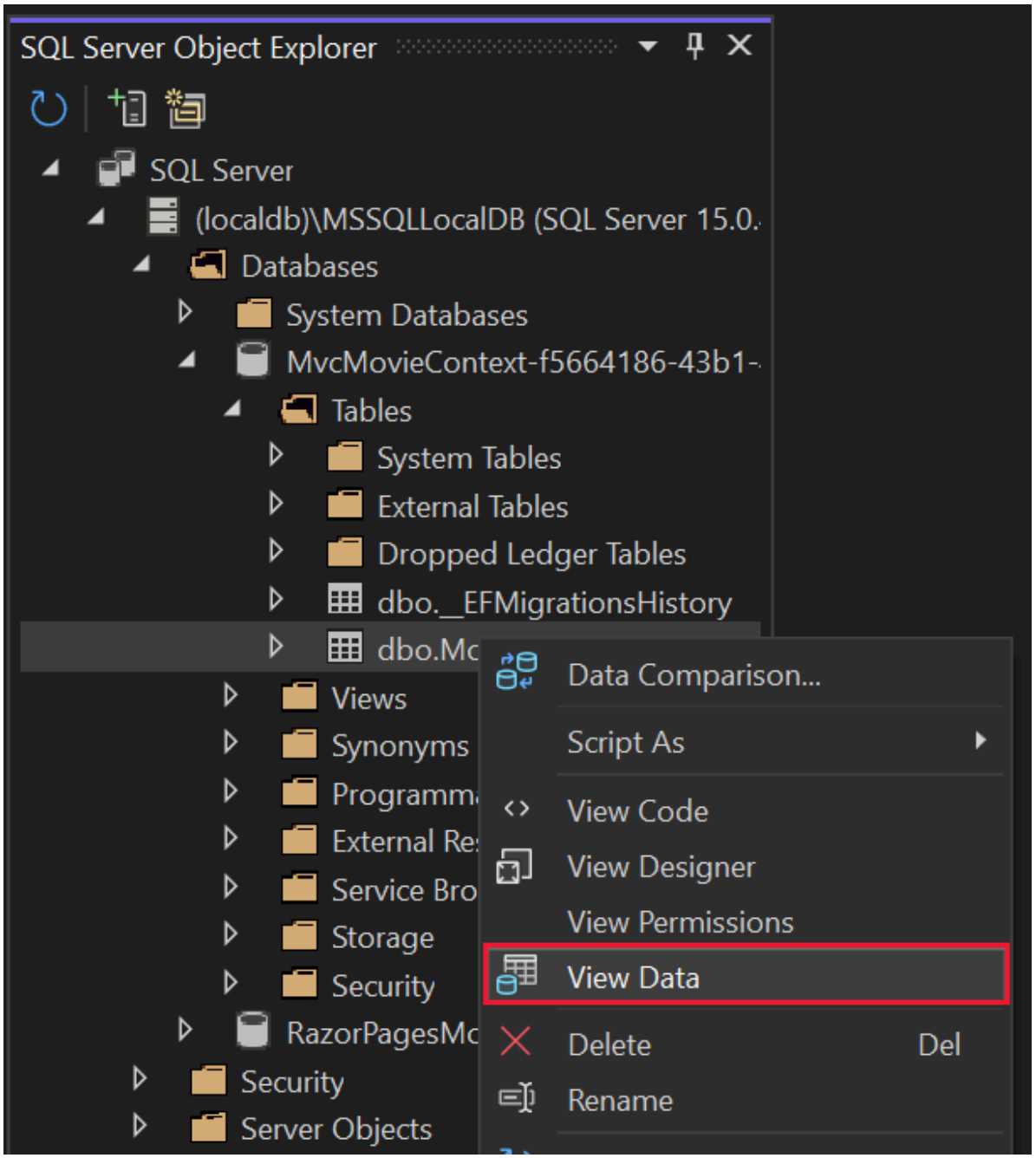
Right-click on the `Movie` table (`dbo.Movie`) > **View Designer**

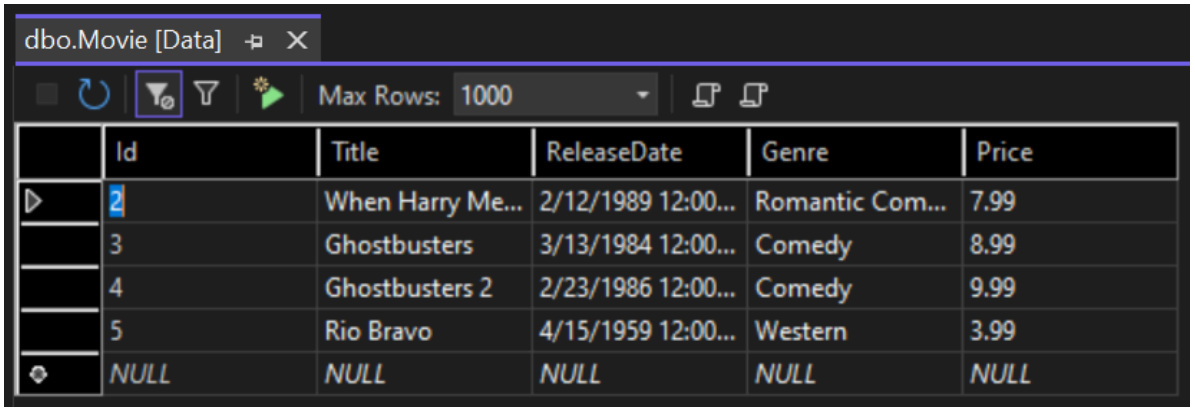




Note the key icon next to `ID`. By default, EF makes a property named `ID` the primary key.

Right-click on the `Movie` table > **View Data**





Id	Title	ReleaseDate	Genre	Price
2	When Harry Me...	2/12/1989 12:00...	Romantic Com...	7.99
3	Ghostbusters	3/13/1984 12:00...	Comedy	8.99
4	Ghostbusters 2	2/23/1986 12:00...	Comedy	9.99
5	Rio Bravo	4/15/1959 12:00...	Western	3.99
NULL	NULL	NULL	NULL	NULL

Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using System;
using System.Linq;

namespace MvcMovie.Models;

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new MvcMovieContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<MvcMovieContext>>()))
        {
            // Look for any movies.
            if (context.Movie.Any())
            {
                return;   // DB has been seeded
            }
            context.Movie.AddRange(
                new Movie
                {
                    Title = "When Harry Met Sally",
                    ReleaseDate = DateTime.Parse("1989-2-12"),
                    Genre = "Romantic Comedy",
                    Price = 7.99M
                },
                new Movie
                {
                    Title = "Ghostbusters ",
                    ReleaseDate = DateTime.Parse("1984-3-13"),
                    Genre = "Comedy",
                    Price = 8.99M
                },
                new Movie
                {
                    Title = "Ghostbusters 2",
                    ReleaseDate = DateTime.Parse("1986-2-23"),
                    Genre = "Comedy",
                    Price = 9.99M
                },
                new Movie
                {
                    Title = "Rio Bravo",
                    ReleaseDate = DateTime.Parse("1959-4-15"),
                    Genre = "Western",
                    Price = 3.99M
```

```
                }
            );
            context.SaveChanges();
        }
    }
}
```

If there are any movies in the database, the seed initializer returns and no movies are added.

C#

```
if (context.Movie.Any())
{
    return;  // DB has been seeded.
}
```

Add the seed initializer

Visual Studio

Replace the contents of `Program.cs` with the following code. The new code is highlighted.

C#

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using MvcMovie.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context")));

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();
```

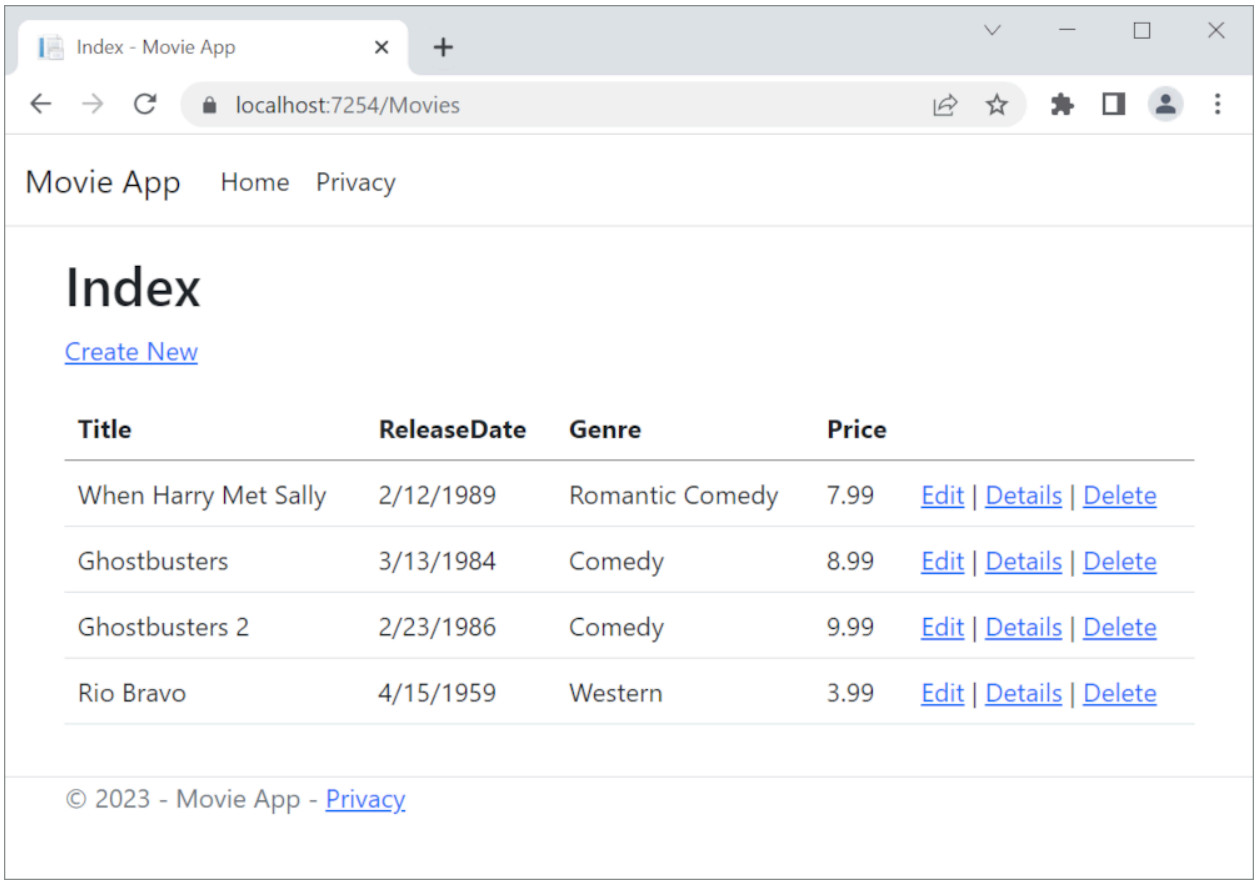
```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}
```

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Delete all the records in the database. You can do this with the delete links in the browser or from SSOX.

Test the app. Force the app to initialize, calling the code in the `Program.cs` file, so the seed method runs. To force initialization, close the command prompt window that Visual Studio opened, and restart by pressing Ctrl+F5.

The app shows the seeded data.



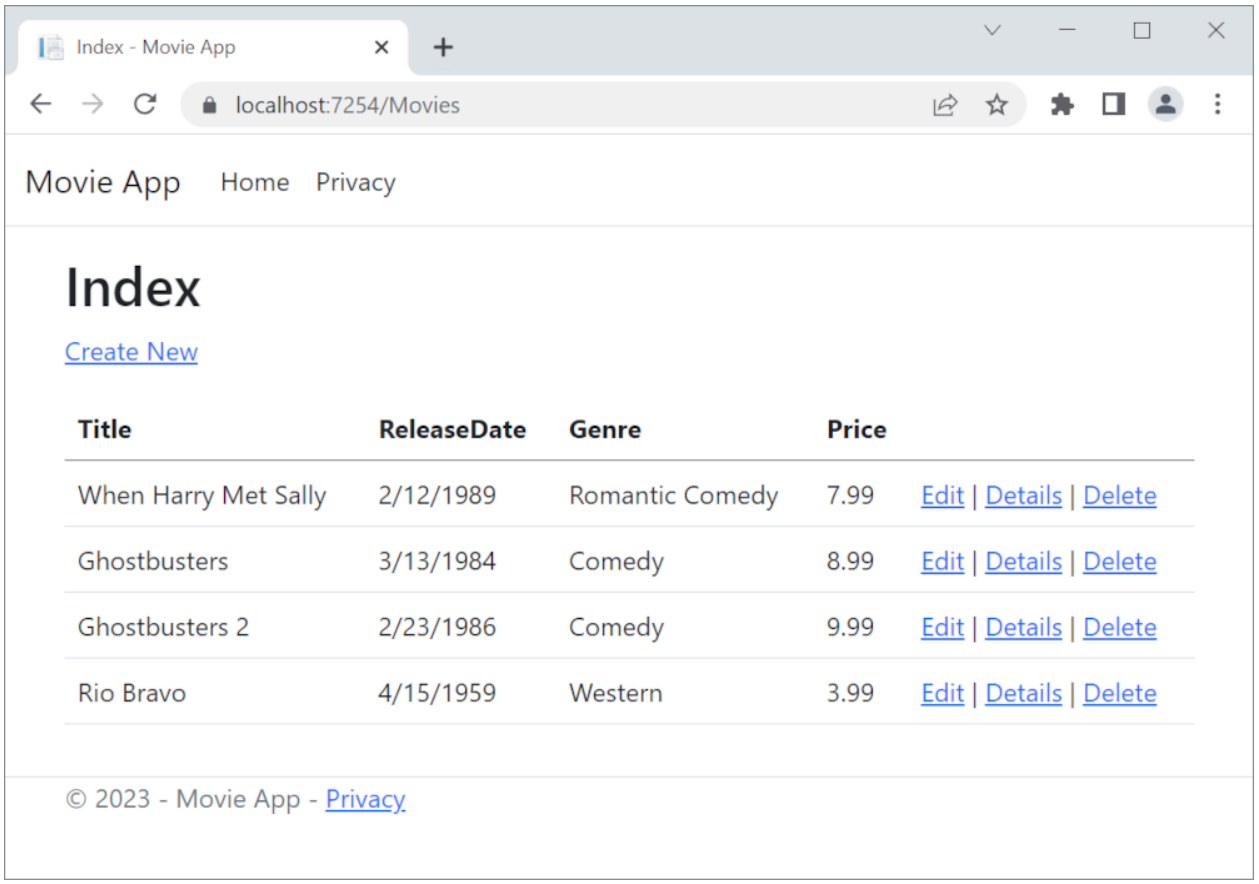
Previous: Adding a model
Next: Adding controller methods and views

Part 6, controller methods and views in ASP.NET Core

Article • 07/26/2024

By Rick Anderson

We have a good start to the movie app, but the presentation isn't ideal, for example, **ReleaseDate** should be two words.



Open the `Models/Movie.cs` file and add the highlighted lines shown below:

C#

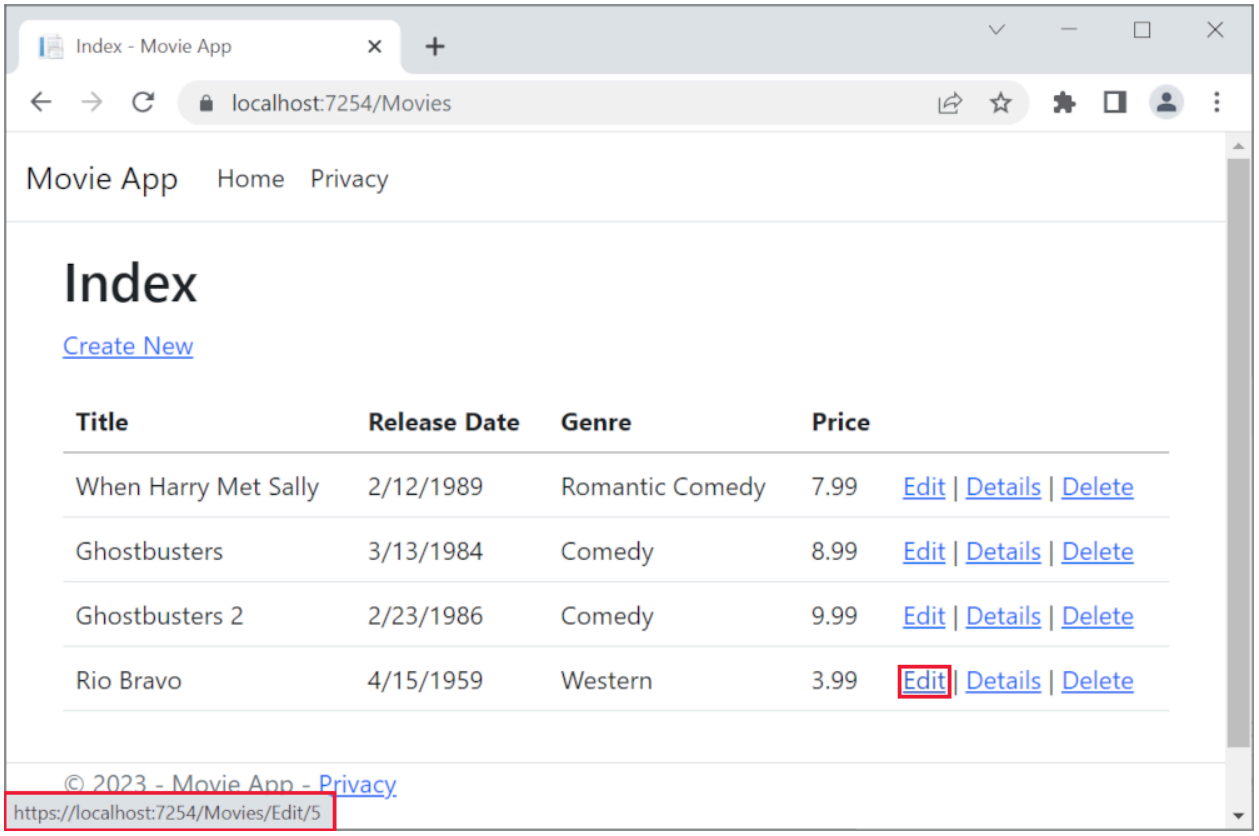
```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
```

```
        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        public DateTime ReleaseDate { get; set; }
        public string? Genre { get; set; }
        [Column(TypeName = "decimal(18, 2)")]
        public decimal Price { get; set; }
    }
```

`DataAnnotations` are explained in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data (Date), so the time information stored in the field isn't displayed.

The `[Column(TypeName = "decimal(18, 2)")]` data annotation is required so Entity Framework Core can correctly map `Price` to currency in the database. For more information, see Data Types.

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.



The **Edit**, **Details**, and **Delete** links are generated by the Core MVC Anchor Tag Helper in the `Views/Movies/Index.cshtml` file.

CSHTML

```
            <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
        </td>
    </tr>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

HTML

```
 <td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for routing set in the `Program.cs` file:

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

ASP.NET Core translates `https://localhost:5001/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. For more information, see Additional resources.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the `Edit.cshtml` Razor file.

C#

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
```

```
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

C#

```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
```

```
    return View(movie);
}
```

The `[Bind]` attribute is one way to protect against over-posting. You should only include properties in the `[Bind]` attribute that you want to change. For more information, see Protect your controller from over-posting. ViewModels provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

C#

```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to prevent forgery of a request and is paired up with an antiforgery token generated in the edit view file (`Views/Movies/Edit.cshtml`). The edit view file generates the antiforgery token with the Form Tag Helper.

CSHTML

```
<form asp-action="Edit">
```

The Form Tag Helper generates a hidden antiforgery token that must match the `[ValidateAntiForgeryToken]` generated antiforgery token in the `Edit` method of the Movies controller. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `FindAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

C#

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

CSHTML

```
@model MvcMovie.Models.Movie

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Id" />
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ReleaseDate" class="control-label"></label>
                <input asp-for="ReleaseDate" class="form-control" />
                <span asp-validation-for="ReleaseDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Genre" class="control-label"></label>
                <input asp-for="Genre" class="form-control" />
                <span asp-validation-for="Genre" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Price" class="control-label"></label>
                <input asp-for="Price" class="form-control" />
                <span asp-validation-for="Price" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file. `@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The Label Tag Helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML `<input>` element. The Validation Tag Helper displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

HTML

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field
is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
                <input class="form-control" type="text" id="Genre"
name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Genre" data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true"
data-val-number="The field Price must be a number." data-val-required="The
Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Price" data-valmsg-replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
```

```
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmq
UyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The `<input>` elements are in an `HTML <form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the `Save` button is clicked. The last line before the closing `</form>` element shows the hidden XSRF token generated by the Form Tag Helper.

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

C#

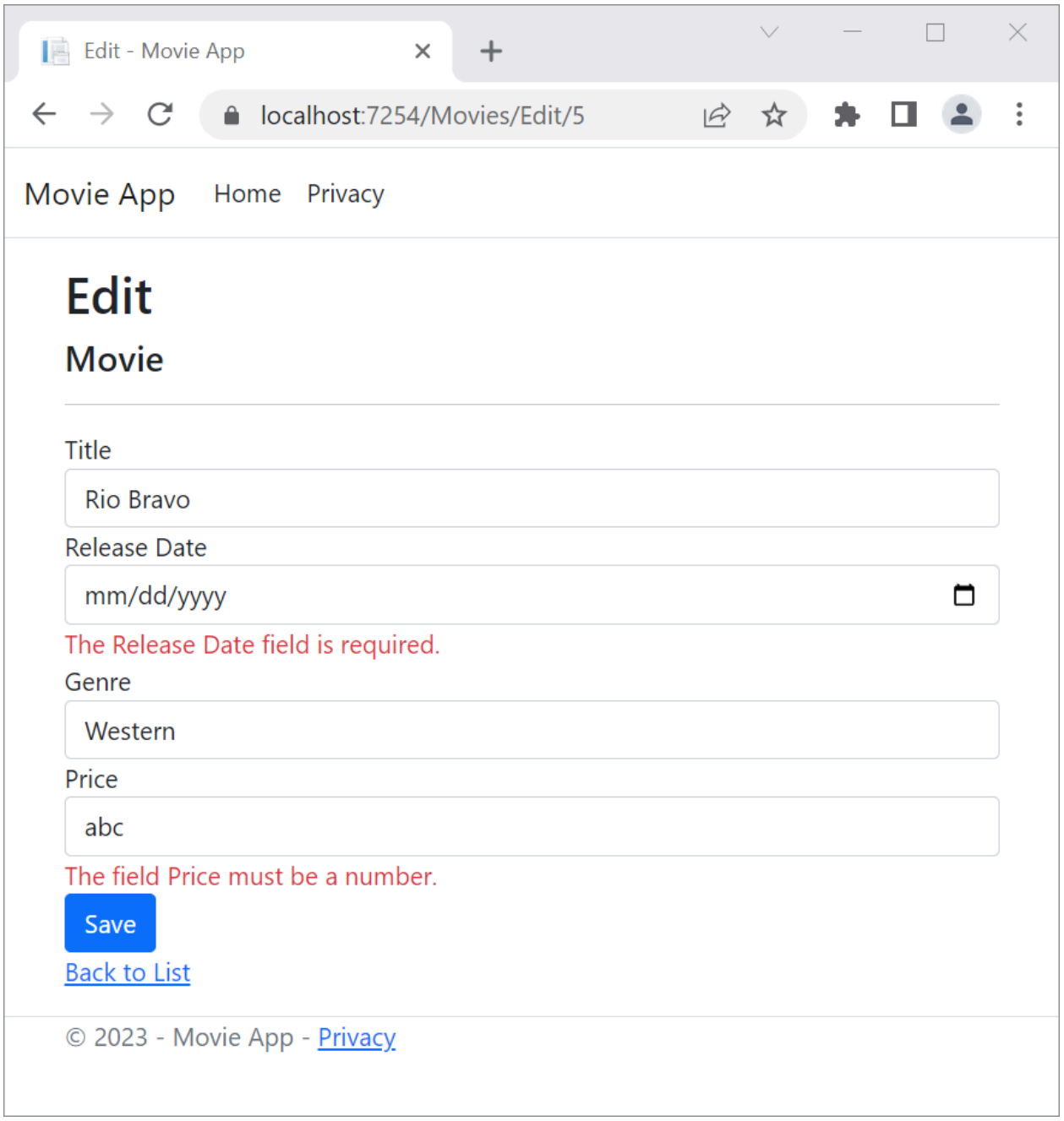
```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
```

```
        }
        return View(movie);
    }
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the antiforgery token generator in the Form Tag Helper

The model binding system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` property verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine Model Validation in more detail. The Validation Tag Helper in the `Views/Movies/Edit.cshtml` view template takes care of displaying appropriate error messages.



All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests shouldn't change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- Globalization and localization

- Introduction to Tag Helpers
- Author Tag Helpers
- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core
- Protect your controller from over-posting
- ViewModels
- Form Tag Helper
- Input Tag Helper
- Label Tag Helper
- Select Tag Helper
- Validation Tag Helper

Previous Next

Part 7, add search to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section, you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method found inside `Controllers/MoviesController.cs` with the following code:

C#

```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

The following line in the `Index` action method creates a LINQ query to select the movies:

C#

```
var movies = from m in _context.Movie
             select m;
```

The query is *only defined* at this point, it has **not** been run against the database.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

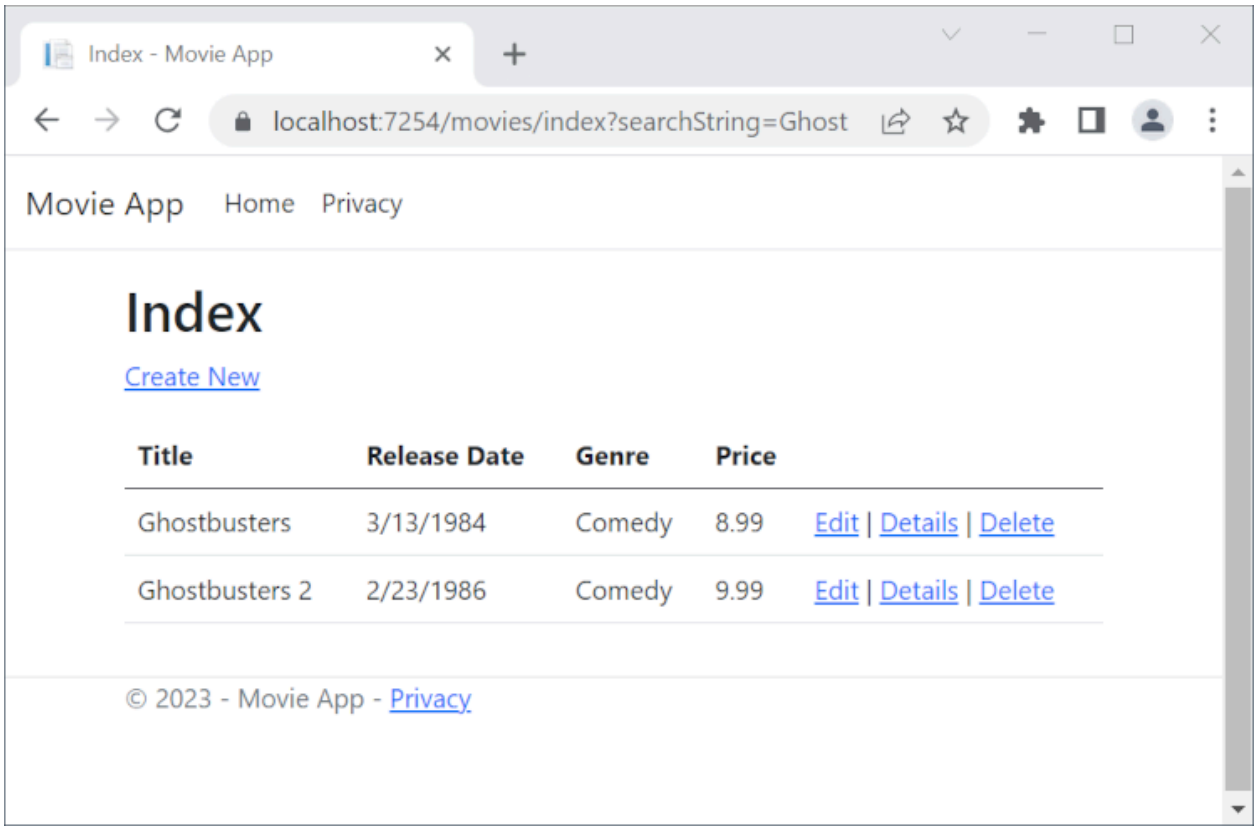
C#

```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
}
```

The `s => s.Title!.ToUpper().Contains(searchString.ToUpper())` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see Query Execution.

> **Note**
>
> The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. SQLite with the default collation is a mixture of case sensitive and case ***IN***sensitive, depending on the query. For information on making case insensitive SQLite queries, see the following:
>
> - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
> - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
> - Collations and Case Sensitivity

Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in `Program.cs`.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Change the parameter to `id` and change all occurrences of `searchString` to `id`.

The previous `Index` method:

C#

```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
```

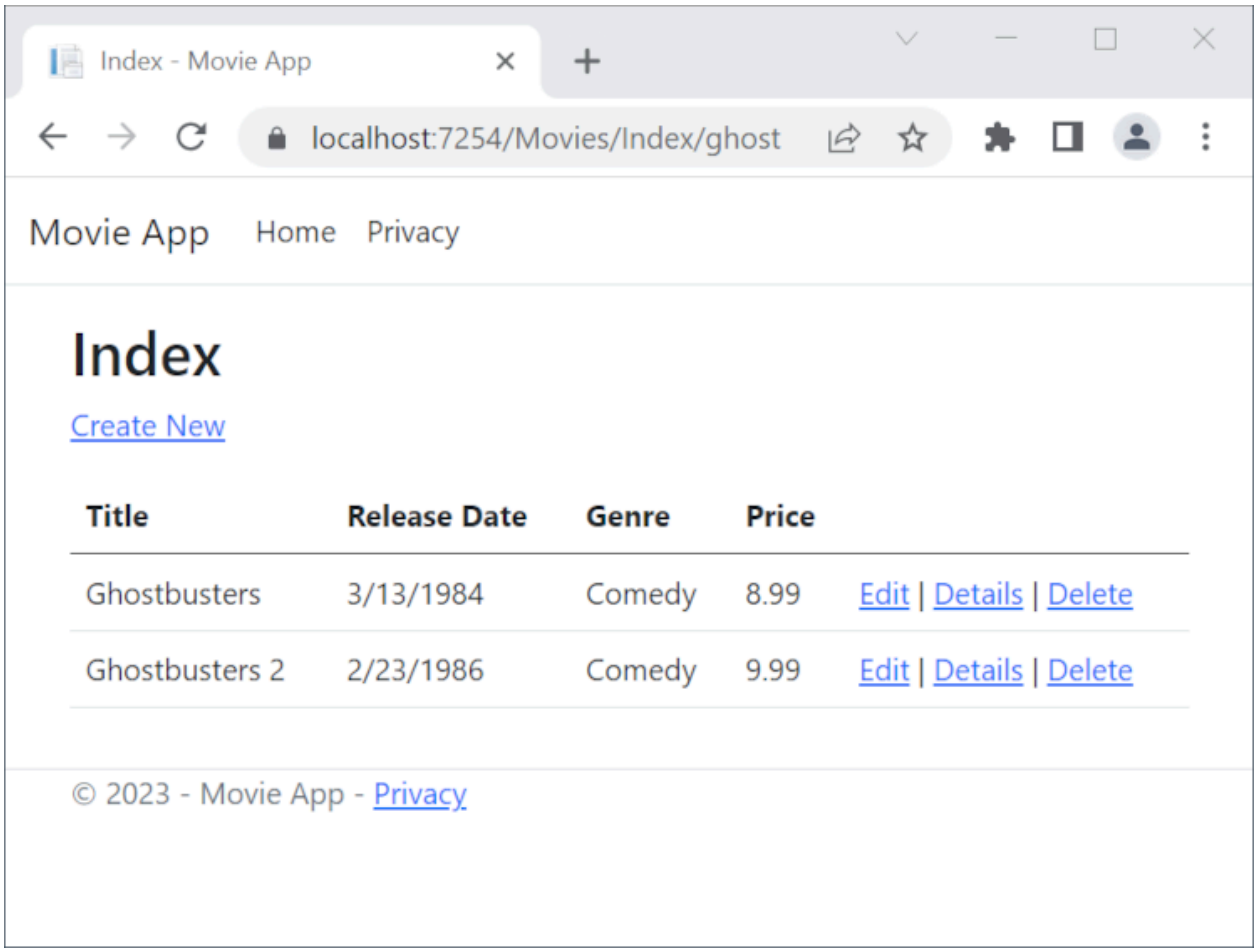
```
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

The updated `Index` method with `id` parameter:

C#

```
public async Task<IActionResult> Index(string id)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(id.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.



However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

C#

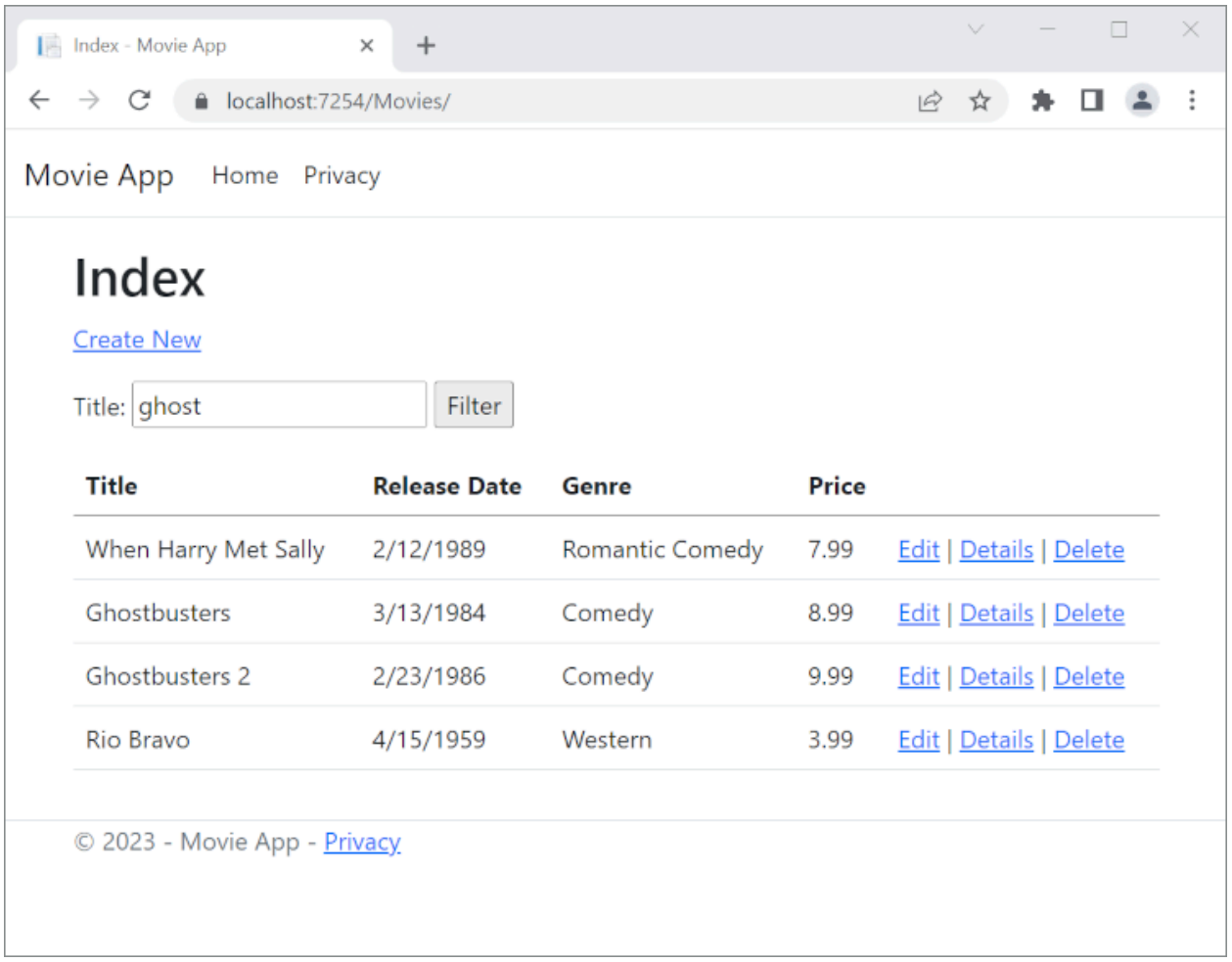
```
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

Open the `Views/Movies/Index.cshtml` file, and add the `<form>` markup highlighted below:

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

The HTML `<form>` tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.



There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

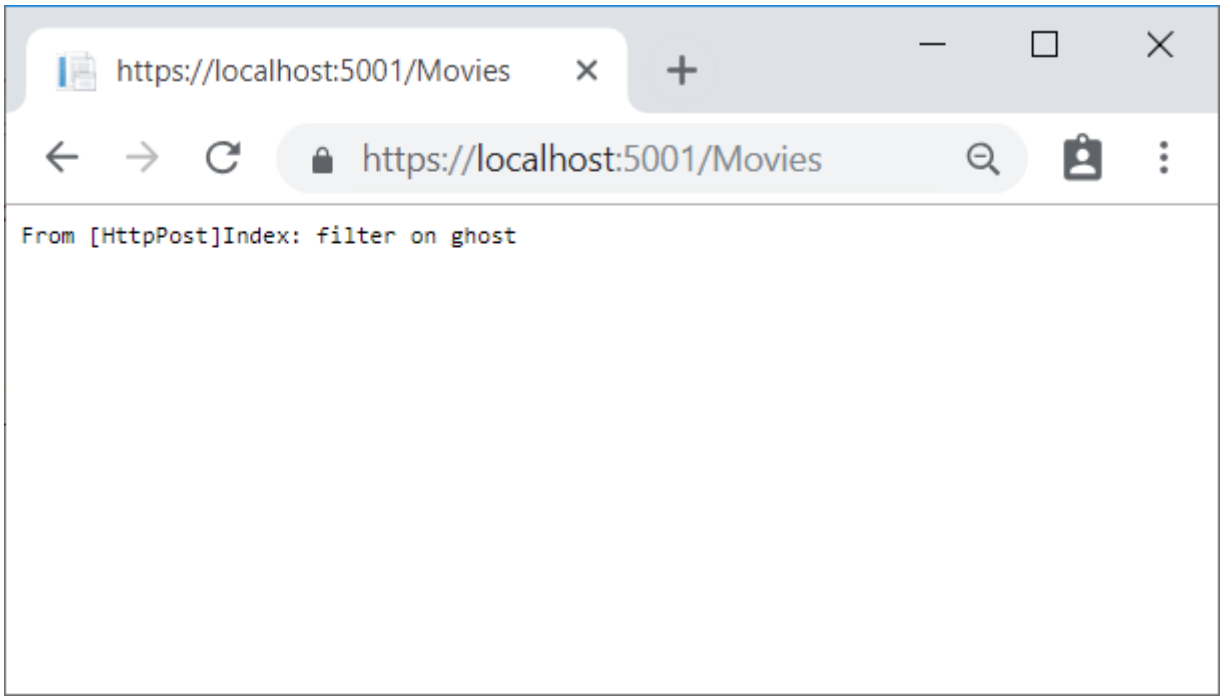
You could add the following `[HttpPost] Index` method.

C#

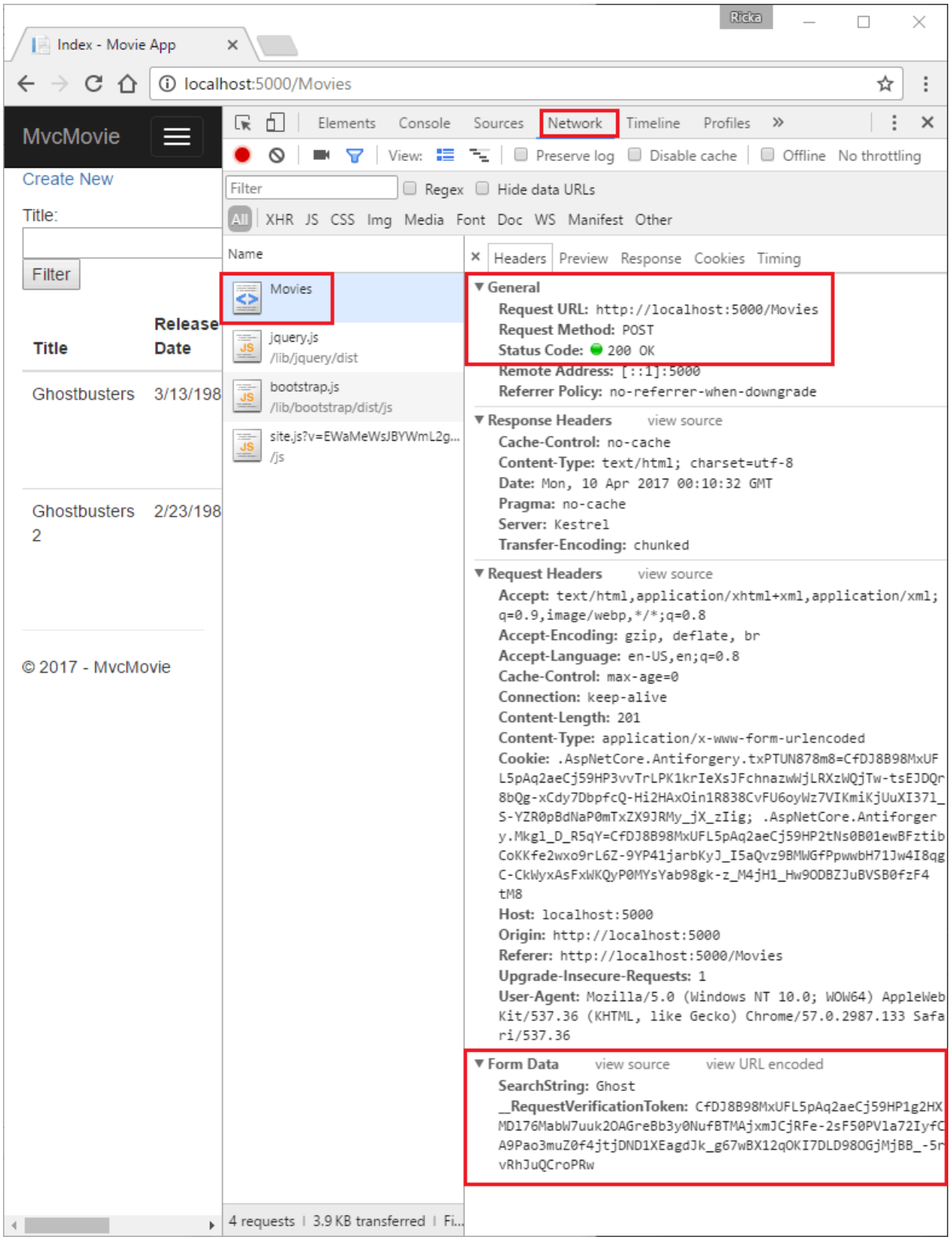
```
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:{PORT}/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool. The image below shows the Chrome browser Developer tools:



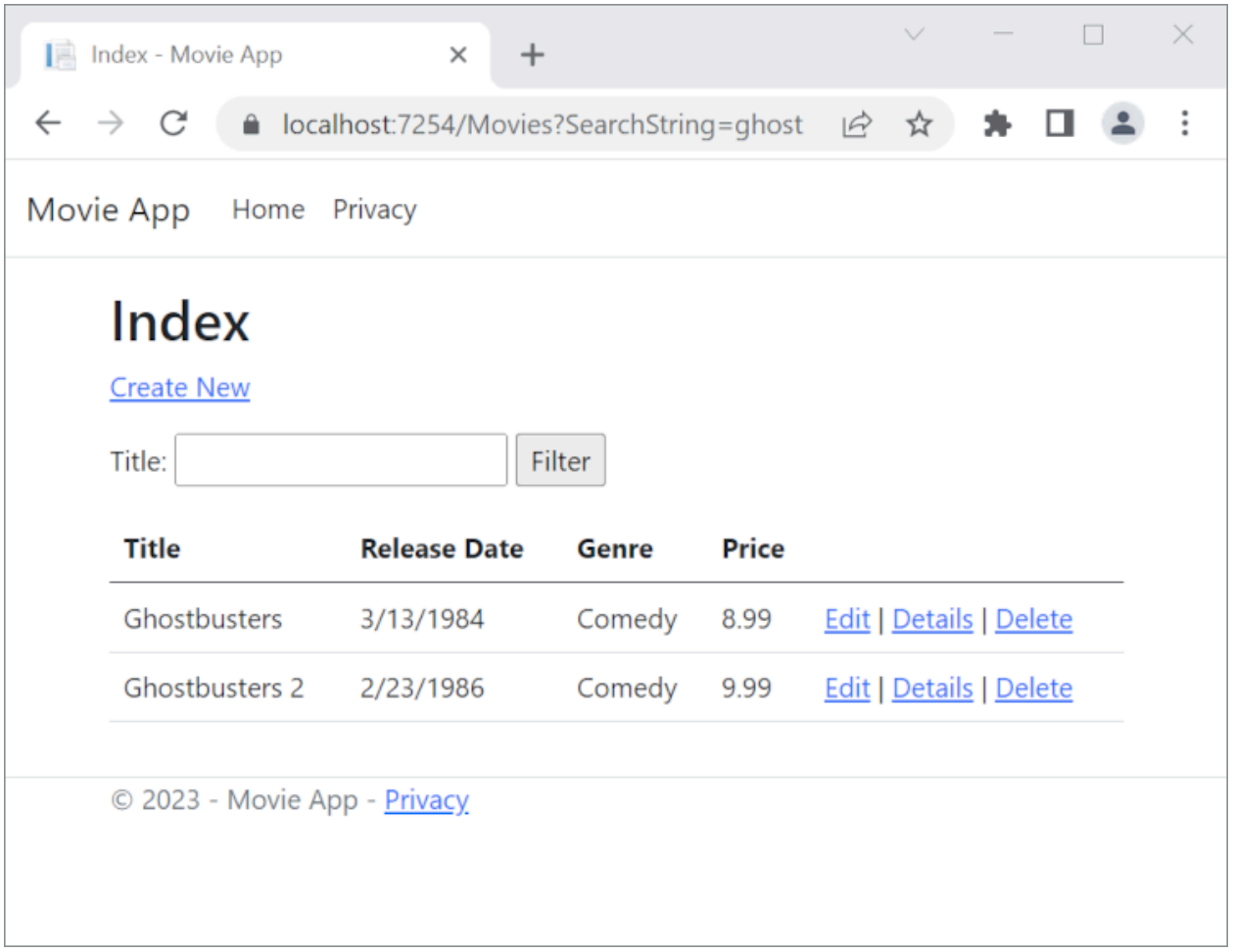
You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF antiforgery token. We're not modifying data, so we don't need to validate the token in the controller method.

Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be `HTTP GET` found in the `Views/Movies/Index.cshtml` file.

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        Title: <input type="text" name="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



The following markup shows the change to the `form` tag:

CSHTML

```
<form asp-controller="Movies" asp-action="Index" method="get">
```

Add Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models;

public class MovieGenreViewModel
{
    public List<Movie>? Movies { get; set; }
    public SelectList? Genres { get; set; }
    public string? MovieGenre { get; set; }
    public string? SearchString { get; set; }
}
```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This allows the user to select a genre from the list.
- `MovieGenre`, which contains the selected genre.
- `SearchString`, which contains the text users enter in the search text box.

Replace the `Index` method in `MoviesController.cs` with the following code:

C#

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string
searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    // Use LINQ to get list of genres.
```

```
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;
    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

The following code is a `LINQ` query that retrieves all the genres from the database.

C#

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

When the user searches for the item, the search value is retained in the search box.

Add search by genre to the Index view

Update `Index.cshtml` found in *Views/Movies/* as follows:

CSHTML

```
@model MvcMovie.Models.MovieGenreViewModel

@{
```

```
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>

        Title: <input type="text" asp-for="SearchString" />
        <input type="submit" value="Filter" />
    </p>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Price)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies!)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
```

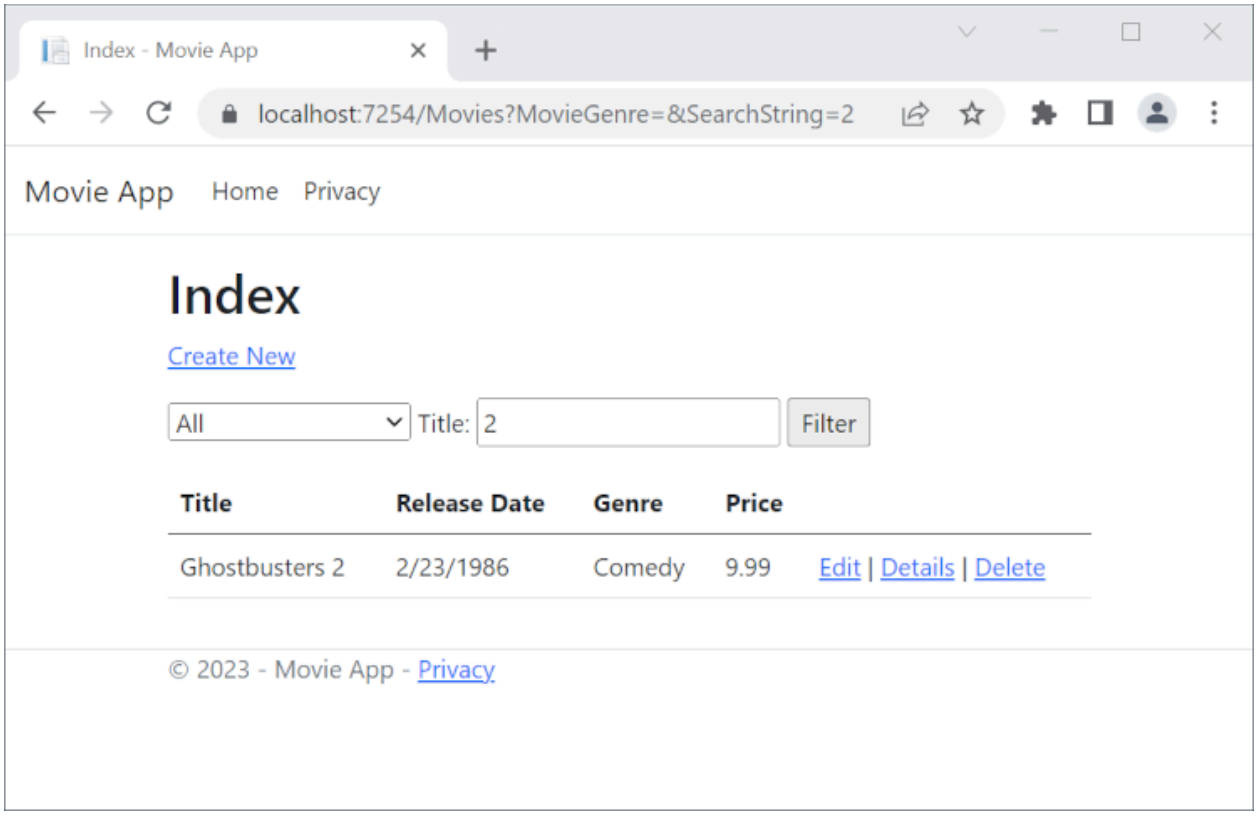
```
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Examine the lambda expression used in the following HTML Helper:

```
@Html.DisplayNameFor(model => model.Movies![0].Title)
```

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.Movies`, or `model.Movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated. The `!` after `model.Movies` is the null-forgiving operator, which is used to declare that `Movies` isn't null.

Test the app by searching by genre, by movie title, and by both:



Previous Next

Part 8, add a new field to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section Entity Framework Code First Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When EF Code First is used to automatically create a database, Code First:

- Adds a table to the database to track the schema of the database.
- Verifies the database is in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Add a Rating Property to the Movie Model

Add a `Rating` property to `Models/Movie.cs`:

C#

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }

    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    public string? Rating {  get; set; }
}
```

Build the app

Visual Studio

Press `Ctrl`+`Shift`+`B`

Because you've added a new field to the `Movie` class, you need to update the property binding list so this new property will be included. In `MoviesController.cs`, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

C#

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

Update the view templates in order to display, create, and edit the new `Rating` property in the browser view.

Edit the `/Views/Movies/Index.cshtml` file and add a `Rating` field:

CSHTML

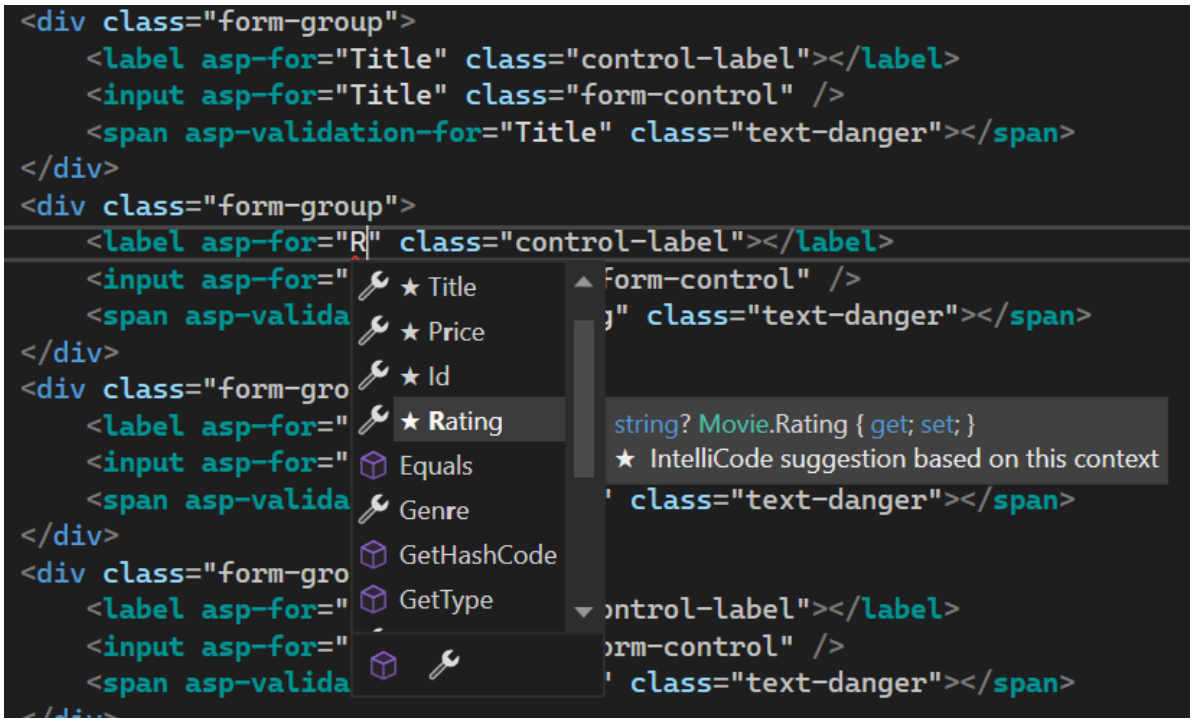
```
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Price)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Movies![0].Rating)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Movies!)
        {
            <tr>
                <td>
```

```
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Rating)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.Id">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.Id">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the `/Views/Movies/Create.cshtml` with a `Rating` field.

Visual Studio / Visual Studio for Mac

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers.



Update the remaining templates.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

C#

```
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

The app won't work until the DB is updated to include the new field. If it's run now, the following `SqlException` is thrown:

```
SqlException: Invalid column name 'Rating'.
```

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the

development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.

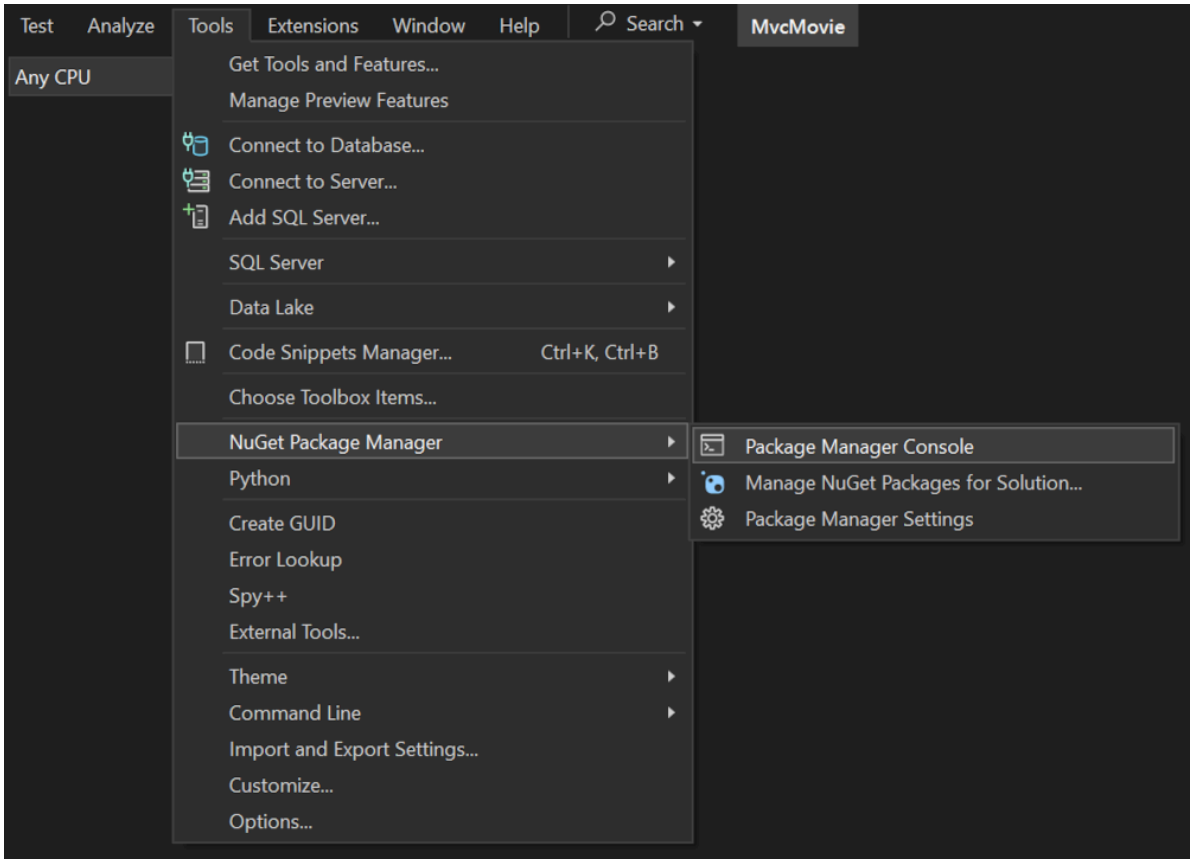
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

3. Use Code First Migrations to update the database schema.

For this tutorial, Code First Migrations is used.

Visual Studio

From the **Tools** menu, select **NuGet Package Manager** > **Package Manager Console**.



In the PMC, enter the following commands:

```
PowerShell
```

```
Add-Migration Rating
Update-Database
```

The `Add-Migration` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the `Rating` field.

Run the app and verify you can create, edit, and display movies with a `Rating` field.

Previous Next

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 9, add validation to an ASP.NET Core MVC app

Article • 07/26/2024

By Rick Anderson

In this section:

- Validation logic is added to the `Movie` model.
- You ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core Code First is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Delete the previously edited data

In the next step, validation rules are added that don't allow null values. Run the app, navigate to `/Movies/Index`, delete all listed movies, and stop the app. The app will use the seed data the next time it is run.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in validation attributes `Required`, `StringLength`, `RegularExpression`, `Range` and the `DataType` formatting attribute.

```
C#
```

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string? Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
    [Required]
    [StringLength(30)]
    public string? Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string? Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White spaces are allowed while numbers, and special characters are not allowed.
- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.

    - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".

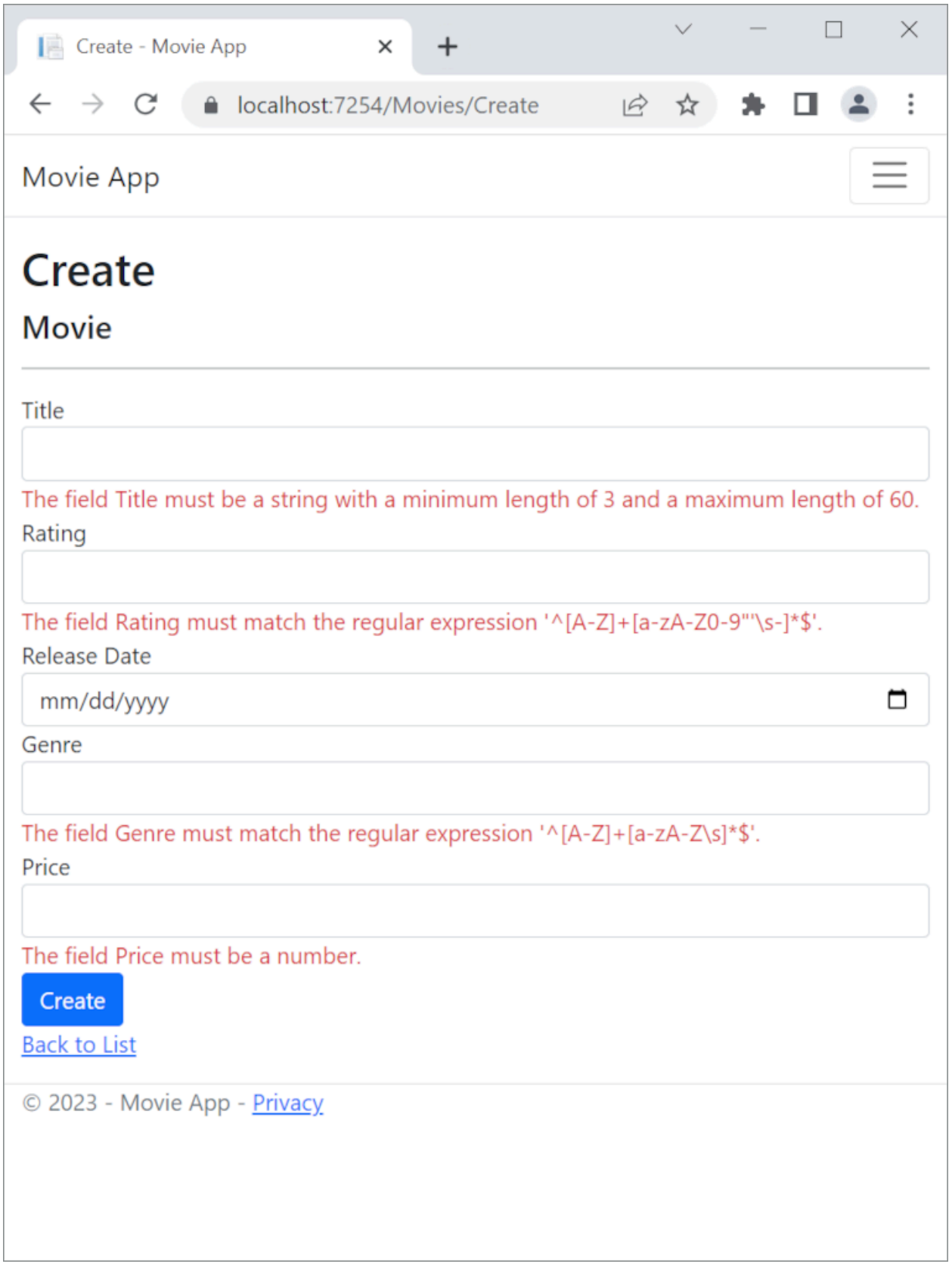
- The `Range` attribute constrains a value to within a specified range.
- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI

Run the app and navigate to the Movies controller.

Select the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



Note

You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the `Create.cshtml` view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data isn't sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the Fiddler tool, or the F12 Developer tools.

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

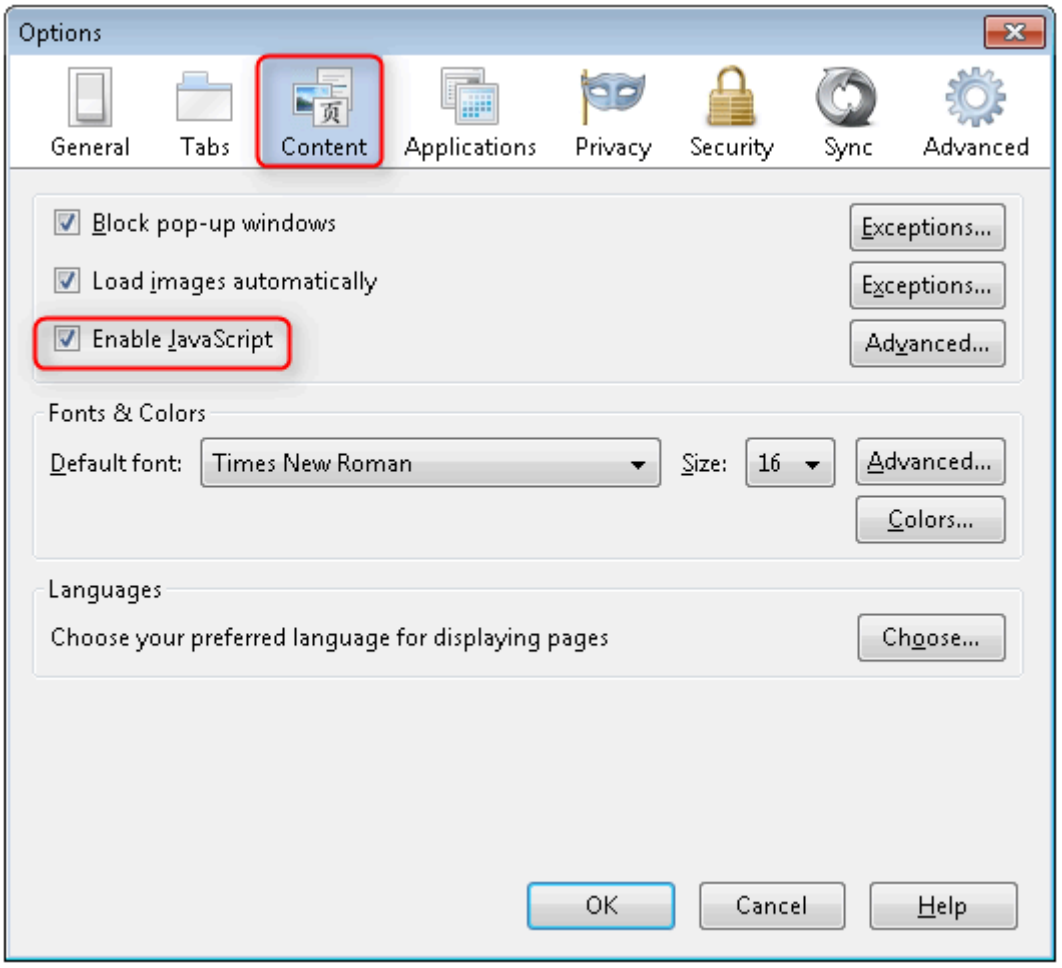
C#

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

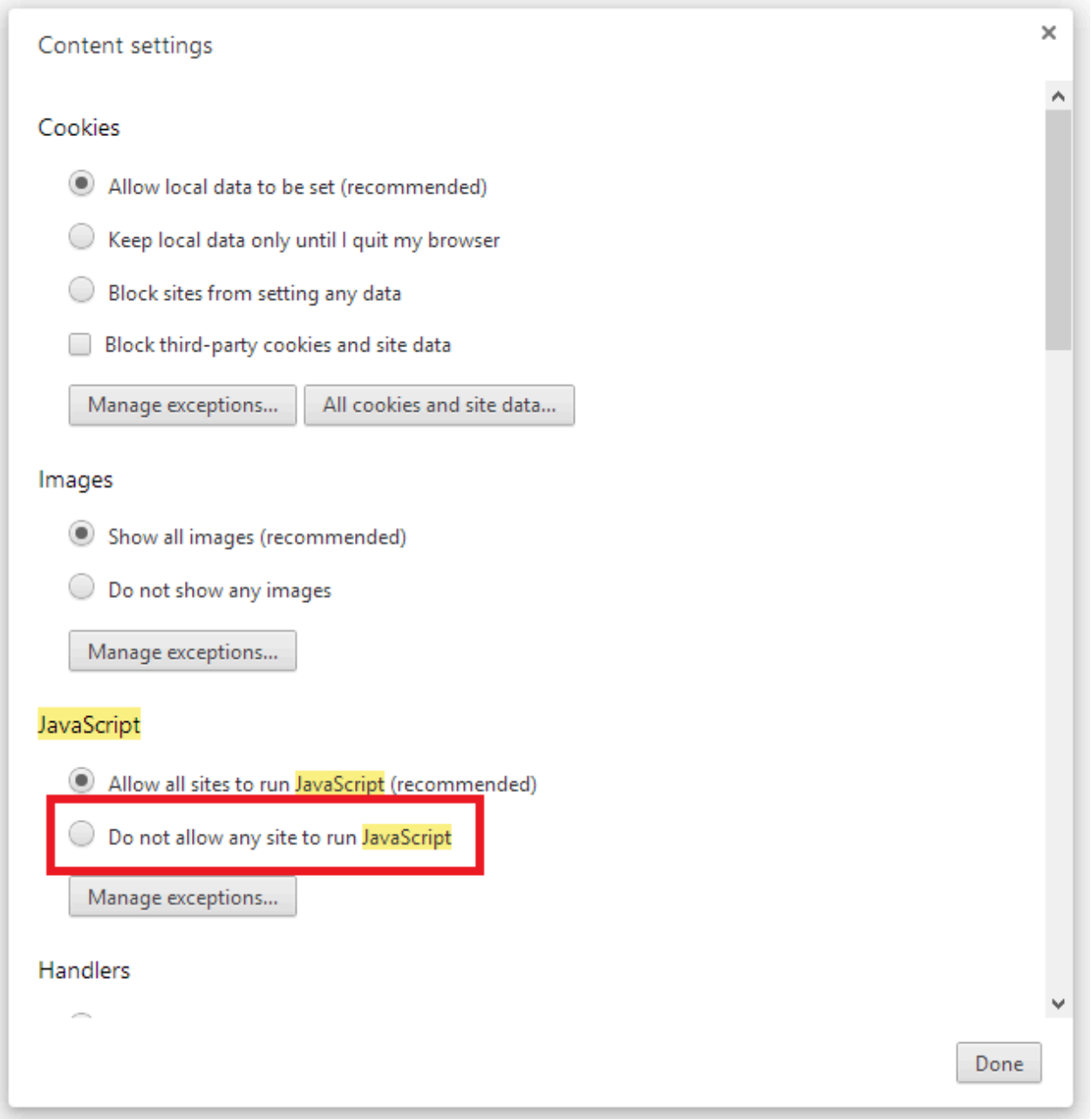
The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation won't submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

The following image shows how to disable JavaScript in the Firefox browser.



The following image shows how to disable JavaScript in the Chrome browser.



After you disable JavaScript, post invalid data and step through the debugger.

```
74          // POST: Movies/Create
75          // To protect from overposting attacks, please enable t
76          // more details see http://go.microsoft.com/fwlink/?Lin
77          [HttpPost]
78          [ValidateAntiForgeryToken]
            0 references
79          public async Task<IAc          t> Create([Bind("ID,Title
80          {                              false
81              if (ModelState.IsValid)
82              {
83                  _context.Add(movie);
84                  await _context.SaveChangesAsync();
85                  return RedirectToAction("Index");
86              }
87              return View(movie);
88          }
89
```

A portion of the `Create.cshtml` view template is shown in the following markup:

HTML

```
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>

            @*Markup removed for brevity.*@
```

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The Validation Tag Helper displays validation errors. See Validation for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the `Movie.cs` file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType`

enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

C#

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data and supplies elements/attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

C#

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some

fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

> ⓘ **Note**
>
> jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:
>
> `[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]`

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

C#

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }
    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
```

```
    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$"), Required, StringLength(30)]
    public string Genre { get; set; }
    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

In the next part of the series, we review the app and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- Working with Forms
- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers

Previous Next

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 10, examine the Details and Delete methods of an ASP.NET Core app

Article • 08/04/2024

By Rick Anderson

Open the Movie controller and examine the `Details` method:

C#

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method, and an `id` value. Recall these segments are defined in `Program.cs`.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

EF makes it easy to search for data using the `FirstOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:{PORT}/Movies/Details/1` to something like `http://localhost:`

`{PORT}/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

C#

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{

    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    if (movie != null)
    {
        _context.Movie.Remove(movie);
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

C#

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

C#

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

C#

```
// POST: Movies/Delete/6
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Previous

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Views in ASP.NET Core MVC

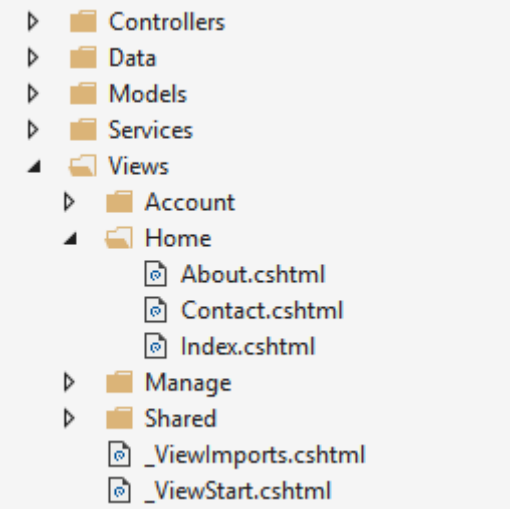
Article • 06/03/2022

By Steve Smith and Dave Brock

This document explains views used in ASP.NET Core MVC applications. For information on Razor Pages, see Introduction to Razor Pages in ASP.NET Core.

In the Model-View-Controller (MVC) pattern, the *view* handles the app's data presentation and user interaction. A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.

In ASP.NET Core MVC, views are `.cshtml` files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in a `Views` folder at the root of the app:



The `Home` controller is represented by a `Home` folder inside the `Views` folder. The `Home` folder contains the views for the `About`, `Contact`, and `Index` (homepage) webpages. When a user requests one of these three webpages, controller actions in the `Home` controller determine which of the three views is used to build and return a webpage to the user.

Use layouts to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

Partial views reduce code duplication by managing reusable parts of views. For example, a partial view is useful for an author biography on a blog website that appears in several views. An author biography is ordinary view content and doesn't require code to

execute in order to produce the content for the webpage. Author biography content is available to the view by model binding alone, so using a partial view for this type of content is ideal.

View components are similar to partial views in that they allow you to reduce repetitive code, but they're appropriate for view content that requires code to run on the server in order to render the webpage. View components are useful when the rendered content requires database interaction, such as for a website shopping cart. View components aren't limited to model binding in order to produce webpage output.

Benefits of using views

Views help to establish separation of concerns within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes your app modular, which provides several benefits:

- The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
- The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
- It's easier to test the user interface parts of the app because the views are separate units.
- Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

Creating a view

Views that are specific to a controller are created in the `Views/[ControllerName]` folder. Views that are shared among controllers are placed in the `Views/Shared` folder. To create a view, add a new file and give it the same name as its associated controller action with the `.cshtml` file extension. To create a view that corresponds with the `About` action in the `Home` controller, create an `About.cshtml` file in the `Views/Home` folder:

CSHTML

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>
```

```
<p>Use this area to provide additional information.</p>
```

Razor markup starts with the `@` symbol. Run C# statements by placing C# code within Razor code blocks set off by curly braces (`{ ... }`). For example, see the assignment of "About" to `ViewData["Title"]` shown above. You can display values within HTML by simply referencing the value with the `@` symbol. See the contents of the `<h2>` and `<h3>` elements above.

The view content shown above is only part of the entire webpage that's rendered to the user. The rest of the page's layout and other common aspects of the view are specified in other view files. To learn more, see the Layout topic.

How controllers specify views

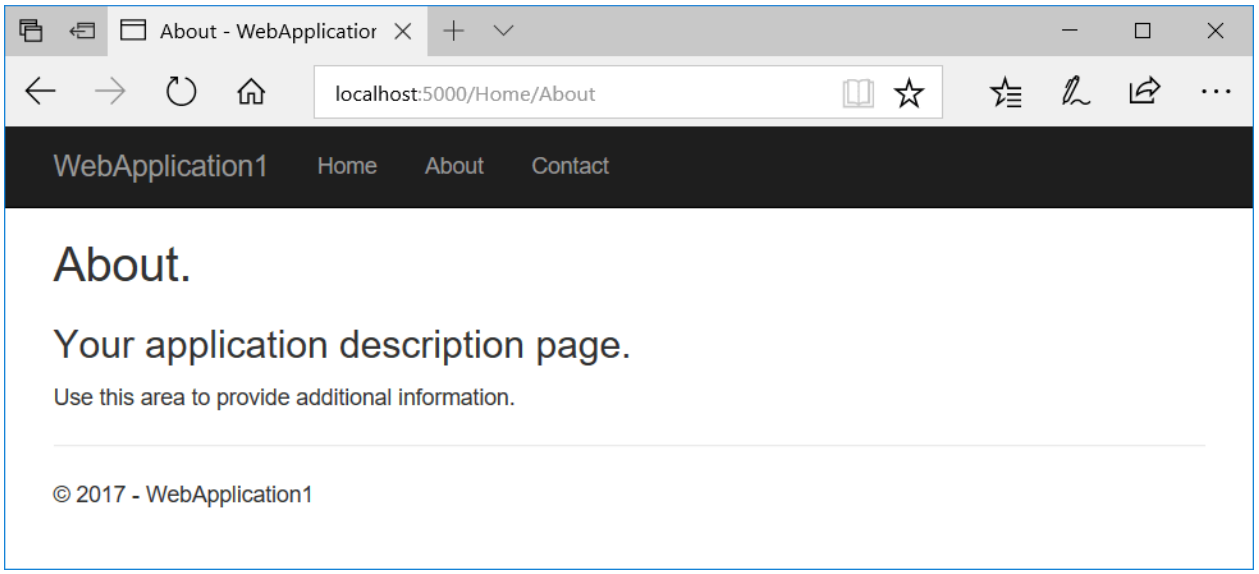
Views are typically returned from actions as a ViewResult, which is a type of ActionResult. Your action method can create and return a `ViewResult` directly, but that isn't commonly done. Since most controllers inherit from Controller, you simply use the `View` helper method to return the `ViewResult`:

`HomeController.cs`:

C#

```
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

When this action returns, the `About.cshtml` view shown in the last section is rendered as the following webpage:



The `View` helper method has several overloads. You can optionally specify:

- An explicit view to return:

```
C#

return View("Orders");
```

- A model to pass to the view:

```
C#

return View(Orders);
```

- Both a view and a model:

```
C#

return View("Orders", Orders);
```

View discovery

When an action returns a view, a process called *view discovery* takes place. This process determines which view file is used based on the view name.

The default behavior of the `View` method (`return View();`) is to return a view with the same name as the action method from which it's called. For example, the `About` `ActionResult` method name of the controller is used to search for a view file named `About.cshtml`. First, the runtime looks in the `Views/[ControllerName]` folder for the view. If it doesn't find a matching view there, it searches the `Shared` folder for the view.

It doesn't matter if you implicitly return the `ViewResult` with `return View();` or explicitly pass the view name to the `View` method with `return View("<ViewName>");`. In both cases, view discovery searches for a matching view file in this order:

1. `Views/\[ControllerName]/\[ViewName].cshtml`
2. `Views/Shared/\[ViewName].cshtml`

A view file path can be provided instead of a view name. If using an absolute path starting at the app root (optionally starting with "/" or "~/"), the `.cshtml` extension must be specified:

```
C#

return View("Views/Home/About.cshtml");
```

You can also use a relative path to specify views in different directories without the `.cshtml` extension. Inside the `HomeController`, you can return the `Index` view of your `Manage` views with a relative path:

```
C#

return View("../Manage/Index");
```

Similarly, you can indicate the current controller-specific directory with the "./" prefix:

```
C#

return View("./About");
```

Partial views and view components use similar (but not identical) discovery mechanisms.

You can customize the default convention for how views are located within the app by using a custom IViewLocationExpander.

View discovery relies on finding view files by file name. If the underlying file system is case sensitive, view names are probably case sensitive. For compatibility across operating systems, match case between controller and action names and associated view folders and file names. If you encounter an error that a view file can't be found while working with a case-sensitive file system, confirm that the casing matches between the requested view file and the actual view file name.

Follow the best practice of organizing the file structure for your views to reflect the relationships among controllers, actions, and views for maintainability and clarity.

Pass data to views

Pass data to views using several approaches:

- Strongly typed data: viewmodel
- Weakly typed data
  - `ViewData` (`ViewDataAttribute`)
  - `ViewBag`

Strongly-typed data (viewmodel)

The most robust approach is to specify a model type in the view. This model is commonly referred to as a *viewmodel*. You pass an instance of the viewmodel type to the view from the action.

Using a viewmodel to pass data to a view allows the view to take advantage of *strong* type checking. *Strong typing* (or *strongly typed*) means that every variable and constant has an explicitly defined type (for example, `string`, `int`, or `DateTime`). The validity of types used in a view is checked at compile time.

Visual Studio and Visual Studio Code list strongly typed class members using a feature called IntelliSense. When you want to see the properties of a viewmodel, type the variable name for the viewmodel followed by a period (`.`). This helps you write code faster with fewer errors.

Specify a model using the `@model` directive. Use the model with `@Model`:

CSHTML

```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

To provide the model to the view, the controller passes it as a parameter:

C#

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";
```

```
    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

There are no restrictions on the model types that you can provide to a view. We recommend using Plain Old CLR Object (POCO) viewmodels with little or no behavior (methods) defined. Usually, viewmodel classes are either stored in the `Models` folder or a separate `ViewModels` folder at the root of the app. The `Address` viewmodel used in the example above is a POCO viewmodel stored in a file named `Address.cs`:

C#

```
namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}
```

Nothing prevents you from using the same classes for both your viewmodel types and your business model types. However, using separate models allows your views to vary independently from the business logic and data access parts of your app. Separation of models and viewmodels also offers security benefits when models use model binding and validation for data sent to the app by the user.

Weakly typed data (`ViewData`, `[ViewData]` attribute, and `ViewBag`)

`ViewBag` *isn't available by default for use in Razor Pages* `PageModel` *classes.*

In addition to strongly typed views, views have access to a *weakly typed* (also called *loosely typed*) collection of data. Unlike strong types, *weak types* (or *loose types*) means

that you don't explicitly declare the type of data you're using. You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views.

Expand table

Passing data between a ...	Example
Controller and a view	Populating a dropdown list with data.
View and a layout view	Setting the `<title>` element content in the layout view from a view file.
Partial view and a view	A widget that displays data based on the webpage that the user requested.

This collection can be referenced through either the `ViewData` or `ViewBag` properties on controllers and views. The `ViewData` property is a dictionary of weakly typed objects. The `ViewBag` property is a wrapper around `ViewData` that provides dynamic properties for the underlying `ViewData` collection. Note: Key lookups are case-insensitive for both `ViewData` and `ViewBag`.

`ViewData` and `ViewBag` are dynamically resolved at runtime. Since they don't offer compile-time type checking, both are generally more error-prone than using a viewmodel. For that reason, some developers prefer to minimally or never use `ViewData` and `ViewBag`.

`ViewData`

`ViewData` is a ViewDataDictionary object accessed through `string` keys. String data can be stored and used directly without the need for a cast, but you must cast other `ViewData` object values to specific types when you extract them. You can use `ViewData` to pass data from controllers to views and within views, including partial views and layouts.

The following is an example that sets values for a greeting and an address using `ViewData` in an action:

C#

```
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
```

```
    ViewData["Address"]  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

Work with the data in a view:

CSHTML

```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```

`[ViewData]` attribute

Another approach that uses the ViewDataDictionary is ViewDataAttribute. Properties on controllers or Razor Page models marked with the `[ViewData]` attribute have their values stored and loaded from the dictionary.

In the following example, the Home controller contains a `Title` property marked with `[ViewData]`. The `About` method sets the title for the About view:

C#

```
public class HomeController : Controller
{
    [ViewData]
    public string Title { get; set; }

    public IActionResult About()
    {
        Title = "About Us";
        ViewData["Message"] = "Your application description page.";
```

```
        return View();
    }
}
```

In the layout, the title is read from the ViewData dictionary:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

`ViewBag`

`ViewBag` *isn't available by default for use in Razor Pages* `PageModel` *classes.*

`ViewBag` is a `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.DynamicViewData` object that provides dynamic access to the objects stored in `ViewData`. `ViewBag` can be more convenient to work with, since it doesn't require casting. The following example shows how to use `ViewBag` with the same result as using `ViewData` above:

C#

```
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

CSHTML

```
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
```

```
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State
@ViewBag.Address.PostalCode
</address>
```

Using `ViewData` and `ViewBag` simultaneously

`ViewBag` *isn't available by default for use in Razor Pages* `PageModel` *classes.*

Since `ViewData` and `ViewBag` refer to the same underlying `ViewData` collection, you can use both `ViewData` and `ViewBag` and mix and match between them when reading and writing values.

Set the title using `ViewBag` and the description using `ViewData` at the top of an `About.cshtml` view:

CSHTML

```
@{
    Layout = "/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Contoso";
    ViewData["Description"] = "Let us tell you about Contoso's philosophy
and mission.";
}
```

Read the properties but reverse the use of `ViewData` and `ViewBag`. In the `_Layout.cshtml` file, obtain the title using `ViewData` and obtain the description using `ViewBag`:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"]</title>
    <meta name="description" content="@ViewBag.Description">
    ...
```

Remember that strings don't require a cast for `ViewData`. You can use `@ViewData["Title"]` without casting.

Using both `ViewData` and `ViewBag` at the same time works, as does mixing and matching reading and writing the properties. The following markup is rendered:

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>About Contoso</title>
    <meta name="description" content="Let us tell you about Contoso's
philosophy and mission.">
    ...
```

Summary of the differences between `ViewData` and `ViewBag`

`ViewBag` *isn't available by default for use in Razor Pages* `PageModel` *classes.*

- `ViewData`
  - Derives from ViewDataDictionary, so it has dictionary properties that can be useful, such as `ContainsKey`, `Add`, `Remove`, and `Clear`.
  - Keys in the dictionary are strings, so whitespace is allowed. Example: `ViewData["Some Key With Whitespace"]`
  - Any type other than a `string` must be cast in the view to use `ViewData`.
- `ViewBag`
  - Derives from `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.DynamicViewData`, so it allows the creation of dynamic properties using dot notation (`@ViewBag.SomeKey = <value or object>`), and no casting is required. The syntax of `ViewBag` makes it quicker to add to controllers and views.
  - Simpler to check for null values. Example: `@ViewBag.Person?.Name`

When to use `ViewData` or `ViewBag`

Both `ViewData` and `ViewBag` are equally valid approaches for passing small amounts of data among controllers and views. The choice of which one to use is based on preference. You can mix and match `ViewData` and `ViewBag` objects, however, the code is easier to read and maintain with one approach used consistently. Both approaches are dynamically resolved at runtime and thus prone to causing runtime errors. Some development teams avoid them.

Dynamic views

Views that don't declare a model type using `@model` but that have a model instance passed to them (for example, `return View(Address);`) can reference the instance's properties dynamically:

CSHTML

```
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

This feature offers flexibility but doesn't offer compilation protection or IntelliSense. If the property doesn't exist, webpage generation fails at runtime.

More view features

Tag Helpers make it easy to add server-side behavior to existing HTML tags. Using Tag Helpers avoids the need to write custom code or helpers within your views. Tag helpers are applied as attributes to HTML elements and are ignored by editors that can't process them. This allows you to edit and render view markup in a variety of tools.

Generating custom HTML markup can be achieved with many built-in HTML Helpers. More complex user interface logic can be handled by View Components. View components provide the same SoC that controllers and views offer. They can eliminate the need for actions and views that deal with data used by common user interface elements.

Like many other aspects of ASP.NET Core, views support dependency injection, allowing services to be injected into views.

CSS isolation

Isolate CSS styles to individual pages, views, and components to reduce or avoid:

- Dependencies on global styles that can be challenging to maintain.
- Style conflicts in nested content.

To add a *scoped CSS file* for a page or view, place the CSS styles in a companion `.cshtml.css` file matching the name of the `.cshtml` file. In the following example, an `Index.cshtml.css` file supplies CSS styles that are only applied to the `Index.cshtml` page or view.

`Pages/Index.cshtml.css` (Razor Pages) or `Views/Index.cshtml.css` (MVC):

CSS

```
h1 {
    color: red;
}
```

CSS isolation occurs at build time. The framework rewrites CSS selectors to match markup rendered by the app's pages or views. The rewritten CSS styles are bundled and produced as a static asset, `{APP ASSEMBLY}.styles.css`. The placeholder `{APP ASSEMBLY}` is the assembly name of the project. A link to the bundled CSS styles is placed in the app's layout.

In the `<head>` content of the app's `Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC), add or confirm the presence of the link to the bundled CSS styles:

HTML

```
<link rel="stylesheet" href="~/{APP ASSEMBLY}.styles.css" />
```

In the following example, the app's assembly name is `WebApp`:

HTML

```
<link rel="stylesheet" href="WebApp.styles.css" />
```

The styles defined in a scoped CSS file are only applied to the rendered output of the matching file. In the preceding example, any `h1` CSS declarations defined elsewhere in the app don't conflict with the `Index`'s heading style. CSS style cascading and inheritance rules remain in effect for scoped CSS files. For example, styles applied directly to an `<h1>` element in the `Index.cshtml` file override the scoped CSS file's styles in `Index.cshtml.css`.

> **Note**
>
> In order to guarantee CSS style isolation when bundling occurs, importing CSS in Razor code blocks isn't supported.
>
> CSS isolation only applies to HTML elements. CSS isolation isn't supported for Tag Helpers.

Within the bundled CSS file, each page, view, or Razor component is associated with a scope identifier in the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-

character string generated by the framework. The following example provides the style for the preceding `<h1>` element in the `Index` page of a Razor Pages app:

```css
/* /Pages/Index.cshtml.rz.scp.css */
h1[b-3xxtam6d07] {
    color: red;
}
```

In the `Index` page where the CSS style is applied from the bundled file, the scope identifier is appended as an HTML attribute:

```HTML
<h1 b-3xxtam6d07>
```

The identifier is unique to an app. At build time, a project bundle is created with the convention `{STATIC WEB ASSETS BASE PATH}/Project.lib.scp.css`, where the placeholder `{STATIC WEB ASSETS BASE PATH}` is the static web assets base path.

If other projects are utilized, such as NuGet packages or Razor class libraries, the bundled file:

- References the styles using CSS imports.
- Isn't published as a static web asset of the app that consumes the styles.

CSS preprocessor support

CSS preprocessors are useful for improving CSS development by utilizing features such as variables, nesting, modules, mixins, and inheritance. While CSS isolation doesn't natively support CSS preprocessors such as Sass or Less, integrating CSS preprocessors is seamless as long as preprocessor compilation occurs before the framework rewrites the CSS selectors during the build process. Using Visual Studio for example, configure existing preprocessor compilation as a **Before Build** task in the Visual Studio Task Runner Explorer.

Many third-party NuGet packages, such as AspNetCore.SassCompiler, can compile SASS/SCSS files at the beginning of the build process before CSS isolation occurs, and no additional configuration is required.

CSS isolation configuration

CSS isolation permits configuration for some advanced scenarios, such as when there are dependencies on existing tools or workflows.

Customize scope identifier format

In this section, the `{Pages|Views}` placeholder is either `Pages` for Razor Pages apps or `Views` for MVC apps.

By default, scope identifiers use the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. To customize the scope identifier format, update the project file to a desired pattern:

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/Index.cshtml.css" CssScope="custom-scope-
identifier" />
</ItemGroup>
```

In the preceding example, the CSS generated for `Index.cshtml.css` changes its scope identifier from `b-{STRING}` to `custom-scope-identifier`.

Use scope identifiers to achieve inheritance with scoped CSS files. In the following project file example, a `BaseView.cshtml.css` file contains common styles across views. A `DerivedView.cshtml.css` file inherits these styles.

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/BaseView.cshtml.css" CssScope="custom-scope-
identifier" />
  <None Update="{Pages|Views}/DerivedView.cshtml.css" CssScope="custom-
scope-identifier" />
</ItemGroup>
```

Use the wildcard (`*`) operator to share scope identifiers across multiple files:

XML

```
<ItemGroup>
  <None Update="{Pages|Views}/*.cshtml.css" CssScope="custom-scope-
identifier" />
</ItemGroup>
```

Change base path for static web assets

The scoped CSS file is generated at the root of the app. In the project file, use the `StaticWebAssetBasePath` property to change the default path. The following example places the scoped CSS file, and the rest of the app's assets, at the `_content` path:

XML

```
<PropertyGroup>
  <StaticWebAssetBasePath>_content/$(PackageId)</StaticWebAssetBasePath>
</PropertyGroup>
```

Disable automatic bundling

To opt out of how framework publishes and loads scoped files at runtime, use the `DisableScopedCssBundling` property. When using this property, other tools or processes are responsible for taking the isolated CSS files from the `obj` directory and publishing and loading them at runtime:

XML

```
<PropertyGroup>
  <DisableScopedCssBundling>true</DisableScopedCssBundling>
</PropertyGroup>
```

Razor class library (RCL) support

When a Razor class library (RCL) provides isolated styles, the `<link>` tag's `href` attribute points to `{STATIC WEB ASSET BASE PATH}/{PACKAGE ID}.bundle.scp.css`, where the placeholders are:

- `{STATIC WEB ASSET BASE PATH}`: The static web asset base path.
- `{PACKAGE ID}`: The library's package identifier. The package identifier defaults to the project's assembly name if the package identifier isn't specified in the project file.

In the following example:

- The static web asset base path is `_content/ClassLib`.
- The class library's assembly name is `ClassLib`.

`Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC):

HTML

```
<link href="_content/ClassLib/ClassLib.bundle.scp.css" rel="stylesheet">
```

For more information on RCLs, see the following articles:

- Reusable Razor UI in class libraries with ASP.NET Core
- Consume ASP.NET Core Razor components from a Razor class library (RCL)

For information on Blazor CSS isolation, see ASP.NET Core Blazor CSS isolation.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Partial views in ASP.NET Core

Article • 05/17/2023

By Steve Smith, Maher JENDOUBI, Rick Anderson, and Scott Sauber

A partial view is a Razor markup file (`.cshtml`) without a @page directive that renders HTML output *within* another markup file's rendered output.

The term *partial view* is used when developing either an MVC app, where markup files are called *views*, or a Razor Pages app, where markup files are called *pages*. This topic generically refers to MVC views and Razor Pages pages as *markup files*.

View or download sample code (how to download)

When to use partial views

Partial views are an effective way to:

- Break up large markup files into smaller components.

  In a large, complex markup file composed of several logical pieces, there's an advantage to working with each piece isolated into a partial view. The code in the markup file is manageable because the markup only contains the overall page structure and references to partial views.

- Reduce the duplication of common markup content across markup files.

  When the same markup elements are used across markup files, a partial view removes the duplication of markup content into one partial view file. When the markup is changed in the partial view, it updates the rendered output of the markup files that use the partial view.

Partial views shouldn't be used to maintain common layout elements. Common layout elements should be specified in _Layout.cshtml files.

Don't use a partial view where complex rendering logic or code execution is required to render the markup. Instead of a partial view, use a view component.

Declare partial views

A partial view is a `.cshtml` markup file without an @page directive maintained within the *Views* folder (MVC) or *Pages* folder (Razor Pages).

In ASP.NET Core MVC, a controller's ViewResult is capable of returning either a view or a partial view. In Razor Pages, a PageModel can return a partial view represented as a PartialViewResult object. Referencing and rendering partial views is described in the Reference a partial view section.

Unlike MVC view or page rendering, a partial view doesn't run `_ViewStart.cshtml`. For more information on `_ViewStart.cshtml`, see Layout in ASP.NET Core.

Partial view file names often begin with an underscore (`_`). This naming convention isn't required, but it helps to visually differentiate partial views from views and pages.

Reference a partial view

Use a partial view in a Razor Pages PageModel

In ASP.NET Core 2.0 or 2.1, the following handler method renders the *_AuthorPartialRP.cshtml* partial view to the response:

```
C#

public IActionResult OnGetPartial() =>
    new PartialViewResult
    {
        ViewName = "_AuthorPartialRP",
        ViewData = ViewData,
    };
```

In ASP.NET Core 2.2 or later, a handler method can alternatively call the Partial method to produce a `PartialViewResult` object:

```
C#

public IActionResult OnGetPartial() =>
    Partial("_AuthorPartialRP");
```

Use a partial view in a markup file

Within a markup file, there are several ways to reference a partial view. We recommend that apps use one of the following asynchronous rendering approaches:

- Partial Tag Helper
- Asynchronous HTML Helper

Partial Tag Helper

The Partial Tag Helper requires ASP.NET Core 2.1 or later.

The Partial Tag Helper renders content asynchronously and uses an HTML-like syntax:

```
CSHTML

<partial name="_PartialName" />
```

When a file extension is present, the Tag Helper references a partial view that must be in the same folder as the markup file calling the partial view:

```
CSHTML

<partial name="_PartialName.cshtml" />
```

The following example references a partial view from the app root. Paths that start with a tilde-slash (`~/`) or a slash (`/`) refer to the app root:

Razor Pages

```
CSHTML

<partial name="~/Pages/Folder/_PartialName.cshtml" />
<partial name="/Pages/Folder/_PartialName.cshtml" />
```

MVC

```
CSHTML

<partial name="~/Views/Folder/_PartialName.cshtml" />
<partial name="/Views/Folder/_PartialName.cshtml" />
```

The following example references a partial view with a relative path:

```
CSHTML

<partial name="../Account/_PartialName.cshtml" />
```

For more information, see Partial Tag Helper in ASP.NET Core.

Asynchronous HTML Helper

When using an HTML Helper, the best practice is to use PartialAsync. `PartialAsync` returns an IHtmlContent type wrapped in a Task<TResult>. The method is referenced by prefixing the awaited call with an `@` character:

CSHTML

```
@await Html.PartialAsync("_PartialName")
```

When the file extension is present, the HTML Helper references a partial view that must be in the same folder as the markup file calling the partial view:

CSHTML

```
@await Html.PartialAsync("_PartialName.cshtml")
```

The following example references a partial view from the app root. Paths that start with a tilde-slash (`~/`) or a slash (`/`) refer to the app root:

Razor Pages

CSHTML

```
@await Html.PartialAsync("~/Pages/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Pages/Folder/_PartialName.cshtml")
```

MVC

CSHTML

```
@await Html.PartialAsync("~/Views/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Views/Folder/_PartialName.cshtml")
```

The following example references a partial view with a relative path:

CSHTML

```
@await Html.PartialAsync("../Account/_LoginPartial.cshtml")
```

Alternatively, you can render a partial view with RenderPartialAsync. This method doesn't return an IHtmlContent. It streams the rendered output directly to the response. Because the method doesn't return a result, it must be called within a Razor code block:

CSHTML

```
@{
    await Html.RenderPartialAsync("_AuthorPartial");
}
```

Since `RenderPartialAsync` streams rendered content, it provides better performance in some scenarios. In performance-critical situations, benchmark the page using both approaches and use the approach that generates a faster response.

Synchronous HTML Helper

Partial and RenderPartial are the synchronous equivalents of `PartialAsync` and `RenderPartialAsync`, respectively. The synchronous equivalents aren't recommended because there are scenarios in which they deadlock. The synchronous methods are targeted for removal in a future release.

> **ⓘ Important**
>
> If you need to execute code, use a **view component** instead of a partial view.

Calling `Partial` or `RenderPartial` results in a Visual Studio analyzer warning. For example, the presence of `Partial` yields the following warning message:

> Use of IHtmlHelper.Partial may result in application deadlocks. Consider using <partial> Tag Helper or IHtmlHelper.PartialAsync.

Replace calls to `@Html.Partial` with `@await Html.PartialAsync` or the Partial Tag Helper. For more information on Partial Tag Helper migration, see Migrate from an HTML Helper.

Partial view discovery

When a partial view is referenced by name without a file extension, the following locations are searched in the stated order:

Razor Pages

1. Currently executing page's folder
2. Directory graph above the page's folder
3. `/Shared`
4. `/Pages/Shared`

5. `/Views/Shared`

MVC

1. `/Areas/<Area-Name>/Views/<Controller-Name>`
2. `/Areas/<Area-Name>/Views/Shared`
3. `/Views/Shared`
4. `/Pages/Shared`

The following conventions apply to partial view discovery:

- Different partial views with the same file name are allowed when the partial views are in different folders.
- When referencing a partial view by name without a file extension and the partial view is present in both the caller's folder and the *Shared* folder, the partial view in the caller's folder supplies the partial view. If the partial view isn't present in the caller's folder, the partial view is provided from the *Shared* folder. Partial views in the *Shared* folder are called *shared partial views* or *default partial views*.
- Partial views can be *chained*—a partial view can call another partial view if a circular reference isn't formed by the calls. Relative paths are always relative to the current file, not to the root or parent of the file.

> **Note**
>
> A Razor `section` defined in a partial view is invisible to parent markup files. The `section` is only visible to the partial view in which it's defined.

Access data from partial views

When a partial view is instantiated, it receives a *copy* of the parent's `ViewData` dictionary. Updates made to the data within the partial view aren't persisted to the parent view. `ViewData` changes in a partial view are lost when the partial view returns.

The following example demonstrates how to pass an instance of ViewDataDictionary to a partial view:

CSHTML

```
@await Html.PartialAsync("_PartialName", customViewData)
```

You can pass a model into a partial view. The model can be a custom object. You can pass a model with `PartialAsync` (renders a block of content to the caller) or `RenderPartialAsync` (streams the content to the output):

CSHTML

```
@await Html.PartialAsync("_PartialName", model)
```

Razor Pages

The following markup in the sample app is from the `Pages/ArticlesRP/ReadRP.cshtml` page. The page contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. The `ViewDataDictionary` constructor overload is used to pass a new `ViewData` dictionary while retaining the existing `ViewData` dictionary.

CSHTML

```
@model ReadRPModel

<h2>@Model.Article.Title</h2>
@* Pass the author's name to Pages\Shared\_AuthorPartialRP.cshtml *@
@await Html.PartialAsync("../Shared/_AuthorPartialRP",
Model.Article.AuthorName)
@Model.Article.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Pages\ArticlesRP\_ArticleSectionRP.cshtml partial
view. *@
@{
    var index = 0;

    foreach (var section in Model.Article.Sections)
    {
        await Html.PartialAsync("_ArticleSectionRP",
                                section,
                                new ViewDataDictionary(ViewData)
                                {
                                    { "index", index }
                                });

        index++;
    }
}
```

`Pages/Shared/_AuthorPartialRP.cshtml` is the first partial view referenced by the `ReadRP.cshtml` markup file:

CSHTML

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Pages/Shared/_AuthorPartialRP.cshtml.
</div>
```

`Pages/ArticlesRP/_ArticleSectionRP.cshtml` is the second partial view referenced by the `ReadRP.cshtml` markup file:

CSHTML

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

MVC

The following markup in the sample app shows the `Views/Articles/Read.cshtml` view. The view contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. The `ViewDataDictionary` constructor overload is used to pass a new `ViewData` dictionary while retaining the existing `ViewData` dictionary.

CSHTML

```
@model PartialViewsSample.ViewModels.Article

<h2>@Model.Title</h2>
@* Pass the author's name to Views\Shared\_AuthorPartial.cshtml *@
@await Html.PartialAsync("_AuthorPartial", Model.AuthorName)
@Model.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Views\Articles\_ArticleSection.cshtml partial view. *@
@{
    var index = 0;

    foreach (var section in Model.Sections)
    {
        @(await Html.PartialAsync("_ArticleSection",
                                  section,
                                  new ViewDataDictionary(ViewData)
                                  {
                                      { "index", index }
                                  }))
```

```
            index++;
        }
    }
```

`Views/Shared/_AuthorPartial.cshtml` is the first partial view referenced by the `Read.cshtml` markup file:

CSHTML

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Views/Shared/_AuthorPartial.cshtml.
</div>
```

`Views/Articles/_ArticleSection.cshtml` is the second partial view referenced by the `Read.cshtml` markup file:

CSHTML

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

At runtime, the partials are rendered into the parent markup file's rendered output, which itself is rendered within the shared `_Layout.cshtml`. The first partial view renders the article author's name and publication date:

> Abraham Lincoln
>
> This partial view from <shared partial view file path>. 11/19/1863 12:00:00 AM

The second partial view renders the article's sections:

> Section One Index: 0
>
> Four score and seven years ago ...
>
> Section Two Index: 1
>
> Now we are engaged in a great civil war, testing ...

> Section Three Index: 2
>
> But, in a larger sense, we can not dedicate ...

Additional resources

- Razor syntax reference for ASP.NET Core
- Tag Helpers in ASP.NET Core
- Partial Tag Helper in ASP.NET Core
- View components in ASP.NET Core
- Areas in ASP.NET Core

Handle requests with controllers in ASP.NET Core MVC

Article • 04/25/2023

By Steve Smith and Scott Addie

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET Core MVC.

What is a Controller?

A controller is used to define and group a set of actions. An action (or *action method*) is a method on a controller which handles requests. Controllers logically group similar actions together. This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively. Requests are mapped to actions through routing. Controllers are activated and disposed on a per request basis.

By convention, controller classes:

- Reside in the project's root-level *Controllers* folder.
- Inherit from `Microsoft.AspNetCore.Mvc.Controller`.

A controller is an instantiable class, usually public, in which at least one of the following conditions is true:

- The class name is suffixed with `Controller`.
- The class inherits from a class whose name is suffixed with `Controller`.
- The `[Controller]` attribute is applied to the class.

A controller class must not have an associated `[NonController]` attribute.

Controllers should follow the Explicit Dependencies Principle. There are a couple of approaches to implementing this principle. If multiple controller actions require the same service, consider using constructor injection to request those dependencies. If the service is needed by only a single action method, consider using Action Injection to request the dependency.

Within the **M**odel-**V**iew-**C**ontroller pattern, a controller is responsible for the initial processing of the request and instantiation of the model. Generally, business decisions should be performed within the model.

The controller takes the result of the model's processing (if any) and returns either the proper view and its associated view data or the result of the API call. Learn more at Overview of ASP.NET Core MVC and Get started with ASP.NET Core MVC and Visual Studio.

The controller is a *UI-level* abstraction. Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned. In well-factored apps, it doesn't directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.

Defining Actions

Public methods on a controller, except those with the `[NonAction]` attribute, are actions. Parameters on actions are bound to request data and are validated using model binding. Model validation occurs for everything that's model-bound. The `ModelState.IsValid` property value indicates whether model binding and validation succeeded.

Action methods should contain logic for mapping a request to a business concern. Business concerns should typically be represented as services that the controller accesses through dependency injection. Actions then map the result of the business action to an application state.

Actions can return anything, but frequently return an instance of `IActionResult` (or `Task<IActionResult>` for async methods) that produces a response. The action method is responsible for choosing *what kind of response*. The action result *does the responding*.

Controller Helper Methods

Controllers usually inherit from Controller, although this isn't required. Deriving from `Controller` provides access to three categories of helper methods:

1. Methods resulting in an empty response body

No `Content-Type` HTTP response header is included, since the response body lacks content to describe.

There are two result types within this category: Redirect and HTTP Status Code.

- **HTTP Status Code**

This type returns an HTTP status code. A couple of helper methods of this type are `BadRequest`, `NotFound`, and `Ok`. For example, `return BadRequest();` produces a 400 status code when executed. When methods such as `BadRequest`, `NotFound`, and `Ok` are overloaded, they no longer qualify as HTTP Status Code responders, since content negotiation is taking place.

- **Redirect**

  This type returns a redirect to an action or destination (using `Redirect`, `LocalRedirect`, `RedirectToAction`, or `RedirectToRoute`). For example, `return RedirectToAction("Complete", new {id = 123});` redirects to `Complete`, passing an anonymous object.

  The Redirect result type differs from the HTTP Status Code type primarily in the addition of a `Location` HTTP response header.

2. Methods resulting in a non-empty response body with a predefined content type

Most helper methods in this category include a `ContentType` property, allowing you to set the `Content-Type` response header to describe the response body.

There are two result types within this category: View and Formatted Response.

- **View**

  This type returns a view which uses a model to render HTML. For example, `return View(customer);` passes a model to the view for data-binding.

- **Formatted Response**

  This type returns JSON or a similar data exchange format to represent an object in a specific manner. For example, `return Json(customer);` serializes the provided object into JSON format.

  Other common methods of this type include `File` and `PhysicalFile`. For example, `return PhysicalFile(customerFilePath, "text/xml");` returns PhysicalFileResult.

3. Methods resulting in a non-empty response body formatted in a content type negotiated with the client

This category is better known as **Content Negotiation**. Content negotiation applies whenever an action returns an ObjectResult type or something other than an

IActionResult implementation. An action that returns a non-`IActionResult` implementation (for example, `object`) also returns a Formatted Response.

Some helper methods of this type include `BadRequest`, `CreatedAtRoute`, and `Ok`. Examples of these methods include `return BadRequest(modelState);`, `return CreatedAtRoute("routename", values, newobject);`, and `return Ok(value);`, respectively. Note that `BadRequest` and `Ok` perform content negotiation only when passed a value; without being passed a value, they instead serve as HTTP Status Code result types. The `CreatedAtRoute` method, on the other hand, always performs content negotiation since its overloads all require that a value be passed.

Cross-Cutting Concerns

Applications typically share parts of their workflow. Examples include an app that requires authentication to access the shopping cart, or an app that caches data on some pages. To perform logic before or after an action method, use a *filter*. Using Filters on cross-cutting concerns can reduce duplication.

Most filter attributes, such as `[Authorize]`, can be applied at the controller or action level depending upon the desired level of granularity.

Error handling and response caching are often cross-cutting concerns:

- Handle errors
- Response Caching

Many cross-cutting concerns can be handled using filters or custom middleware.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Routing to controller actions in ASP.NET Core

Article • 07/19/2023

By Ryan Nowak, Kirk Larkin, and Rick Anderson

ASP.NET Core controllers use the Routing middleware to match the URLs of incoming requests and map them to actions. Route templates:

- Are defined at startup in `Program.cs` or in attributes.
- Describe how URL paths are matched to actions.
- Are used to generate URLs for links. The generated links are typically returned in responses.

Actions are either conventionally-routed or attribute-routed. Placing a route on the controller or action makes it attribute-routed. See Mixed routing for more information.

This document:

- Explains the interactions between MVC and routing:
  - How typical MVC apps make use of routing features.
  - Covers both:
    - Conventional routing typically used with controllers and views.
    - *Attribute routing* used with REST APIs. If you're primarily interested in routing for REST APIs, jump to the Attribute routing for REST APIs section.
  - See Routing for advanced routing details.
- Refers to the default routing system called endpoint routing. It's possible to use controllers with the previous version of routing for compatibility purposes. See the 2.2-3.0 migration guide for instructions.

Set up conventional route

The ASP.NET Core MVC template generates conventional routing code similar to the following:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();
```

```
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

MapControllerRoute is used to create a single route. The single route is named `default` route. Most apps with controllers and views use a route template similar to the `default` route. REST APIs should use attribute routing.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

The route template `"{controller=Home}/{action=Index}/{id?}"`:

- Matches a URL path like `/Products/Details/5`
- Extracts the route values `{ controller = Products, action = Details, id = 5 }` by tokenizing the path. The extraction of route values results in a match if the app has a controller named `ProductsController` and a `Details` action:

  C#

  ```
  public class ProductsController : Controller
  {
      public IActionResult Details(int id)
      {
          return ControllerContext.MyDisplayRouteInfo(id);
      }
  }
  ```

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

- `/Products/Details/5` model binds the value of `id = 5` to set the `id` parameter to `5`. See Model Binding for more details.
- `{controller=Home}` defines `Home` as the default `controller`.
- `{action=Index}` defines `Index` as the default `action`.
- The `?` character in `{id?}` defines `id` as optional.
  - Default and optional route parameters don't need to be present in the URL path for a match. See Route Template Reference for a detailed description of route template syntax.
- Matches the URL path `/`.
- Produces the route values `{ controller = Home, action = Index }`.

The values for `controller` and `action` make use of the default values. `id` doesn't produce a value since there's no corresponding segment in the URL path. `/` only matches if there exists a `HomeController` and `Index` action:

C#

```
public class HomeController : Controller
{
    public IActionResult Index() { ... }
}
```

Using the preceding controller definition and route template, the `HomeController.Index` action is run for the following URL paths:

- `/Home/Index/17`
- `/Home/Index`
- `/Home`
- `/`

The URL path `/` uses the route template default `Home` controllers and `Index` action. The URL path `/Home` uses the route template default `Index` action.

The convenience method MapDefaultControllerRoute:

C#

```
app.MapDefaultControllerRoute();
```

Replaces:

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

> ⓘ **Important**
>
> Routing is configured using the UseRouting and UseEndpoints middleware. To use controllers:
>
> - Call MapControllers to map attribute routed controllers.
> - Call MapControllerRoute or MapAreaControllerRoute, to map both conventionally routed controllers and attribute routed controllers.
>
> Apps typically don't need to call `UseRouting` or `UseEndpoints`. WebApplicationBuilder configures a middleware pipeline that wraps middleware added in `Program.cs` with `UseRouting` and `UseEndpoints`. For more information, see Routing in ASP.NET Core.

Conventional routing

Conventional routing is used with controllers and views. The `default` route:

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

The preceding is an example of a *conventional route*. It's called *conventional routing* because it establishes a *convention* for URL paths:

- The first path segment, `{controller=Home}`, maps to the controller name.
- The second segment, `{action=Index}`, maps to the action name.

- The third segment, `{id?}` is used for an optional `id`. The `?` in `{id?}` makes it optional. `id` is used to map to a model entity.

Using this `default` route, the URL path:

- `/Products/List` maps to the `ProductsController.List` action.
- `/Blog/Article/17` maps to `BlogController.Article` and typically model binds the `id` parameter to 17.

This mapping:

- Is based on the controller and action names **only**.
- Isn't based on namespaces, source file locations, or method parameters.

Using conventional routing with the default route allows creating the app without having to come up with a new URL pattern for each action. For an app with CRUD style actions, having consistency for the URLs across controllers:

- Helps simplify the code.
- Makes the UI more predictable.

> ⚠ **Warning**
>
> The `id` in the preceding code is defined as optional by the route template. Actions can execute without the optional ID provided as part of the URL. Generally, when `id` is omitted from the URL:
>
> - `id` is set to `0` by model binding.
> - No entity is found in the database matching `id == 0`.
>
> Attribute routing provides fine-grained control to make the ID required for some actions and not for others. By convention, the documentation includes optional parameters like `id` when they're likely to appear in correct usage.

Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful. The default conventional route `{controller=Home}/{action=Index}/{id?}`:

- Supports a basic and descriptive routing scheme.
- Is a useful starting point for UI-based apps.
- Is the only route template needed for many web UI apps. For larger web UI apps, another route using Areas is frequently all that's needed.

MapControllerRoute and MapAreaRoute :

- Automatically assign an **order** value to their endpoints based on the order they are invoked.

Endpoint routing in ASP.NET Core:

- Doesn't have a concept of routes.
- Doesn't provide ordering guarantees for the execution of extensibility, all endpoints are processed at once.

Enable Logging to see how the built-in routing implementations, such as Route, match requests.

Attribute routing is explained later in this document.

Multiple conventional routes

Multiple conventional routes can be configured by adding more calls to MapControllerRoute and MapAreaControllerRoute. Doing so allows defining multiple conventions, or to adding conventional routes that are dedicated to a specific action, such as:

```
C#

app.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
app.MapControllerRoute(name: "default",
               pattern: "{controller=Home}/{action=Index}/{id?}");
```

The `blog` route in the preceding code is a **dedicated conventional route**. It's called a dedicated conventional route because:

- It uses conventional routing.
- It's dedicated to a specific action.

Because `controller` and `action` don't appear in the route template `"blog/{*article}"` as parameters:

- They can only have the default values `{ controller = "Blog", action = "Article" }`.
- This route always maps to the action `BlogController.Article`.

`/Blog`, `/Blog/Article`, and `/Blog/{any-string}` are the only URL paths that match the blog route.

The preceding example:

- `blog` route has a higher priority for matches than the `default` route because it is added first.
- Is an example of Slug style routing where it's typical to have an article name as part of the URL.

> ⚠ **Warning**
>
> In ASP.NET Core, routing doesn't:
>
> - Define a concept called a *route*. `UseRouting` adds route matching to the middleware pipeline. The `UseRouting` middleware looks at the set of endpoints defined in the app, and selects the best endpoint match based on the request.
> - Provide guarantees about the execution order of extensibility like **IRouteConstraint** or **IActionConstraint**.
>
> See **Routing** for reference material on routing.

Conventional routing order

Conventional routing only matches a combination of action and controller that are defined by the app. This is intended to simplify cases where conventional routes overlap. Adding routes using MapControllerRoute, MapDefaultControllerRoute, and MapAreaControllerRoute automatically assign an order value to their endpoints based on the order they are invoked. Matches from a route that appears earlier have a higher priority. Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area. Dedicated conventional routes with catch-all route parameters like `{*article}` can make a route too greedy, meaning that it matches URLs that you intended to be matched by other routes. Put the greedy routes later in the route table to prevent greedy matches.

Resolving ambiguous actions

When two endpoints match through routing, routing must do one of the following:

- Choose the best candidate.

- Throw an exception.

For example:

```
C#
public class Products33Controller : Controller
{
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpPost]
    public IActionResult Edit(int id, Product product)
    {
        return ControllerContext.MyDisplayRouteInfo(id, product.name);
    }
}
```

The preceding controller defines two actions that match:

- The URL path `/Products33/Edit/17`
- Route data `{ controller = Products33, action = Edit, id = 17 }`.

This is a typical pattern for MVC controllers:

- `Edit(int)` displays a form to edit a product.
- `Edit(int, Product)` processes the posted form.

To resolve the correct route:

- `Edit(int, Product)` is selected when the request is an HTTP `POST`.
- `Edit(int)` is selected when the HTTP verb is anything else. `Edit(int)` is generally called via `GET`.

The HttpPostAttribute, `[HttpPost]`, is provided to routing so that it can choose based on the HTTP method of the request. The `HttpPostAttribute` makes `Edit(int, Product)` a better match than `Edit(int)`.

It's important to understand the role of attributes like `HttpPostAttribute`. Similar attributes are defined for other HTTP verbs. In conventional routing, it's common for actions to use the same action name when they're part of a show form, submit form workflow. For example, see Examine the two Edit action methods.

If routing can't choose a best candidate, an AmbiguousMatchException is thrown, listing the multiple matched endpoints.

Conventional route names

The strings `"blog"` and `"default"` in the following examples are conventional route names:

```
C#

app.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
app.MapControllerRoute(name: "default",
               pattern: "{controller=Home}/{action=Index}/{id?}");
```

The route names give the route a logical name. The named route can be used for URL generation. Using a named route simplifies URL creation when the ordering of routes could make URL generation complicated. Route names must be unique application wide.

Route names:

- Have no impact on URL matching or handling of requests.
- Are used only for URL generation.

The route name concept is represented in routing as IEndpointNameMetadata. The terms **route name** and **endpoint name**:

- Are interchangeable.
- Which one is used in documentation and code depends on the API being described.

Attribute routing for REST APIs

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs.

Attribute routing uses a set of attributes to map actions directly to route templates. The following code is typical for a REST API and is used in the next sample:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();
```

```
app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

In the preceding code, MapControllers is called to map attribute routed controllers.

In the following example:

- `HomeController` matches a set of URLs similar to what the default conventional route `{controller=Home}/{action=Index}/{id?}` matches.

C#

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The `HomeController.Index` action is run for any of the URL paths `/`, `/Home`, `/Home/Index`, or `/Home/Index/3`.

This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route. The conventional default route handles routes more succinctly. However, attribute routing allows and requires precise control of which route templates apply to each action.

With attribute routing, the controller and action names play no part in which action is matched, unless token replacement is used. The following example matches the same URLs as the previous example:

C#

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The following code uses token replacement for `action` and `controller`:

C#

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("[controller]/[action]")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Route("[controller]/[action]")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The following code applies `[Route("[controller]/[action]")]` to the controller:

C#

```
[Route("[controller]/[action]")]
public class HomeController : Controller
{
    [Route("~/")]
    [Route("/Home")]
```

```
    [Route("~/Home/Index")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

In the preceding code, the `Index` method templates must prepend `/` or `~/` to the route templates. Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller.

See Route template precedence for information on route template selection.

Reserved routing names

The following keywords are reserved route parameter names when using Controllers or Razor Pages:

- `action`
- `area`
- `controller`
- `handler`
- `page`

Using `page` as a route parameter with attribute routing is a common error. Doing that results in inconsistent and confusing behavior with URL generation.

C#

```
public class MyDemo2Controller : Controller
{
    [Route("/articles/{page}")]
    public IActionResult ListArticles(int page)
    {
        return ControllerContext.MyDisplayRouteInfo(page);
    }
}
```

The special parameter names are used by the URL generation to determine if a URL generation operation refers to a Razor Page or to a Controller.

The following keywords are reserved in the context of a Razor view or a Razor Page:

- `page`
- `using`
- `namespace`
- `inject`
- `section`
- `inherits`
- `model`
- `addTagHelper`
- `removeTagHelper`

These keywords shouldn't be used for link generations, model bound parameters, or top level properties.

HTTP verb templates

ASP.NET Core has the following HTTP verb templates:

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]
- [HttpHead]
- [HttpPatch]

Route templates

ASP.NET Core has the following route templates:

- All the HTTP verb templates are route templates.
- [Route]

Attribute routing with Http verb attributes

Consider the following controller:

C#

```
[Route("api/[controller]")]
[ApiController]
public class Test2Controller : ControllerBase
```

```
{
    [HttpGet]   // GET /api/test2
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]   // GET /api/test2/xyz
    public IActionResult GetProduct(string id)
    {
       return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int/{id:int}")] // GET /api/test2/int/3
    public IActionResult GetIntProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int2/{id}")]  // GET /api/test2/int2/3
    public IActionResult GetInt2Product(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

- Each action contains the `[HttpGet]` attribute, which constrains matching to HTTP GET requests only.
- The `GetProduct` action includes the `"{id}"` template, therefore `id` is appended to the `"api/[controller]"` template on the controller. The methods template is `"api/[controller]/{id}"`. Therefore this action only matches GET requests for the form `/api/test2/xyz`, `/api/test2/123`, `/api/test2/{any string}`, etc.

  C#

  ```
  [HttpGet("{id}")]   // GET /api/test2/xyz
  public IActionResult GetProduct(string id)
  {
     return ControllerContext.MyDisplayRouteInfo(id);
  }
  ```

- The `GetIntProduct` action contains the `"int/{id:int}"` template. The `:int` portion of the template constrains the `id` route values to strings that can be converted to an integer. A GET request to `/api/test2/int/abc`:
  - Doesn't match this action.
  - Returns a 404 Not Found error.

C#

```
[HttpGet("int/{id:int}")] // GET /api/test2/int/3
public IActionResult GetIntProduct(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

- The `GetInt2Product` action contains `{id}` in the template, but doesn't constrain `id` to values that can be converted to an integer. A GET request to `/api/test2/int2/abc`:
  - Matches this route.
  - Model binding fails to convert `abc` to an integer. The `id` parameter of the method is integer.
  - Returns a 400 Bad Request because model binding failed to convert `abc` to an integer.

C#

```
[HttpGet("int2/{id}")]  // GET /api/test2/int2/3
public IActionResult GetInt2Product(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

Attribute routing can use HttpMethodAttribute attributes such as HttpPostAttribute, HttpPutAttribute, and HttpDeleteAttribute. All of the HTTP verb attributes accept a route template. The following example shows two actions that match the same route template:

C#

```
[ApiController]
public class MyProductsController : ControllerBase
{
    [HttpGet("/products3")]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpPost("/products3")]
    public IActionResult CreateProduct(MyProduct myProduct)
    {
        return ControllerContext.MyDisplayRouteInfo(myProduct.Name);
    }
}
```

Using the URL path `/products3`:

- The `MyProductsController.ListProducts` action runs when the HTTP verb is `GET`.
- The `MyProductsController.CreateProduct` action runs when the HTTP verb is `POST`.

When building a REST API, it's rare that you'll need to use `[Route(...)]` on an action method because the action accepts all HTTP methods. It's better to use the more specific HTTP verb attribute to be precise about what your API supports. Clients of REST APIs are expected to know what paths and HTTP verbs map to specific logical operations.

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs. This means that many operations, for example, GET and POST on the same logical resource use the same URL. Attribute routing provides a level of control that's needed to carefully design an API's public endpoint layout.

Since an attribute route applies to a specific action, it's easy to make parameters required as part of the route template definition. In the following example, `id` is required as part of the URL path:

C#

```
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The `Products2ApiController.GetProduct(int)` action:

- Is run with URL path like `/products2/3`
- Isn't run with the URL path `/products2`.

The [Consumes] attribute allows an action to limit the supported request content types. For more information, see Define supported request content types with the Consumes attribute.

See Routing for a full description of route templates and related options.

For more information on `[ApiController]`, see ApiController attribute.

Route name

The following code defines a route name of `Products_List`:

C#

```
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Route names can be used to generate a URL based on a specific route. Route names:

- Have no impact on the URL matching behavior of routing.
- Are only used for URL generation.

Route names must be unique application-wide.

Contrast the preceding code with the conventional default route, which defines the `id` parameter as optional (`{id?}`). The ability to precisely specify APIs has advantages, such as allowing `/products` and `/products/5` to be dispatched to different actions.

Combining attribute routes

To make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual actions. Any route templates defined on the controller are prepended to route templates on the actions. Placing a route attribute on the controller makes **all** actions in the controller use attribute routing.

C#

```
[ApiController]
[Route("products")]
public class ProductsApiController : ControllerBase
{
    [HttpGet]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]
```

```
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding example:

- The URL path `/products` can match `ProductsApi.ListProducts`
- The URL path `/products/5` can match `ProductsApi.GetProduct(int)`.

Both of these actions only match HTTP `GET` because they're marked with the `[HttpGet]` attribute.

Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller. The following example matches a set of URL paths similar to the default route.

C#

```
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]
    [Route("Index")]
    [Route("/")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Route("About")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The following table explains the `[Route]` attributes in the preceding code:

Expand table

Attribute	Combines with `[Route("Home")]`	Defines route template
`[Route("")]`	Yes	`"Home"`
`[Route("Index")]`	Yes	`"Home/Index"`

Attribute	Combines with `[Route("Home")]`	Defines route template
`[Route("/")]`	No	`""`
`[Route("About")]`	Yes	`"Home/About"`

Attribute route order

Routing builds a tree and matches all endpoints simultaneously:

- The route entries behave as if placed in an ideal ordering.
- The most specific routes have a chance to execute before the more general routes.

For example, an attribute route like `blog/search/{topic}` is more specific than an attribute route like `blog/{*article}`. The `blog/search/{topic}` route has higher priority, by default, because it's more specific. Using conventional routing, the developer is responsible for placing routes in the desired order.

Attribute routes can configure an order using the Order property. All of the framework provided route attributes include `Order` . Routes are processed according to an ascending sort of the `Order` property. The default order is `0`. Setting a route using `Order = -1` runs before routes that don't set an order. Setting a route using `Order = 1` runs after default route ordering.

Avoid depending on `Order`. If an app's URL-space requires explicit order values to route correctly, then it's likely confusing to clients as well. In general, attribute routing selects the correct route with URL matching. If the default order used for URL generation isn't working, using a route name as an override is usually simpler than applying the `Order` property.

Consider the following two controllers which both define the route matching `/home`:

C#

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
```

```
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

C#

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Requesting `/home` with the preceding code throws an exception similar to the following:

text

```
AmbiguousMatchException: The request matched multiple endpoints. Matches:

 WebMvcRouting.Controllers.HomeController.Index
 WebMvcRouting.Controllers.MyDemoController.MyIndex
```

Adding `Order` to one of the route attributes resolves the ambiguity:

C#

```
[Route("")]
[Route("Home", Order = 2)]
[Route("Home/MyIndex")]
public IActionResult MyIndex()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

With the preceding code, `/home` runs the `HomeController.Index` endpoint. To get to the `MyDemoController.MyIndex`, request `/home/MyIndex`. **Note**:

- The preceding code is an example or poor routing design. It was used to illustrate the `Order` property.
- The `Order` property only resolves the ambiguity, that template cannot be matched. It would be better to remove the `[Route("Home")]` template.

See Razor Pages route and app conventions: Route order for information on route order with Razor Pages.

In some cases, an HTTP 500 error is returned with ambiguous routes. Use logging to see which endpoints caused the `AmbiguousMatchException`.

Token replacement in route templates [controller], [action], [area]

For convenience, attribute routes support *token replacement* by enclosing a token in square-brackets (`[`, `]`). The tokens `[action]`, `[area]`, and `[controller]` are replaced with the values of the action name, area name, and controller name from the action where the route is defined:

C#

```
[Route("[controller]/[action]")]
public class Products0Controller : Controller
{
    [HttpGet]
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }


    [HttpGet("{id}")]
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

C#

```
[HttpGet]
public IActionResult List()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

- Matches `/Products0/List`

C#

```
[HttpGet("{id}")]
public IActionResult Edit(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

- Matches `/Products0/Edit/{id}`

Token replacement occurs as the last step of building the attribute routes. The preceding example behaves the same as the following code:

C#

```
public class Products20Controller : Controller
{
    [HttpGet("[controller]/[action]")]  // Matches '/Products20/List'
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("[controller]/[action]/{id}")]   // Matches
'/Products20/Edit/{id}'
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

Attribute routes can also be combined with inheritance. This is powerful combined with token replacement. Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` generates a unique route name for each action:

C#

```
[ApiController]
[Route("api/[controller]/[action]", Name = "[controller]_[action]")]
public abstract class MyBase2Controller : ControllerBase
{
}

public class Products11Controller : MyBase2Controller
{
    [HttpGet]                       // /api/products11/list
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]              //    /api/products11/edit/3
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

To match the literal token replacement delimiter `[` or `]`, escape it by repeating the character (`[[` or `]]`).

Use a parameter transformer to customize token replacement

Token replacement can be customized using a parameter transformer. A parameter transformer implements IOutboundParameterTransformer and transforms the value of parameters. For example, a custom `SlugifyParameterTransformer` parameter transformer changes the `SubscriptionManagement` route value to `subscription-management`:

C#

```
using System.Text.RegularExpressions;

public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string? TransformOutbound(object? value)
    {
        if (value == null) { return null; }

        return Regex.Replace(value.ToString()!,
                             "([a-z])([A-Z])",
                             "$1-$2",
                             RegexOptions.CultureInvariant,
```

```
TimeSpan.FromMilliseconds(100)).ToLowerInvariant();
    }
}
```

The RouteTokenTransformerConvention is an application model convention that:

- Applies a parameter transformer to all attribute routes in an application.
- Customizes the attribute route token values as they are replaced.

C#

```
public class SubscriptionManagementController : Controller
{
    [HttpGet("[controller]/[action]")]
    public IActionResult ListAll()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `ListAll` method matches `/subscription-management/list-all`.

The `RouteTokenTransformerConvention` is registered as an option:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews(options =>
{
    options.Conventions.Add(new RouteTokenTransformerConvention(
                                    new SlugifyParameterTransformer()));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();
```

```
app.MapControllerRoute(name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

See MDN web docs on Slug for the definition of Slug.

> ⚠ **Warning**
>
> When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Multiple attribute routes

Attribute routing supports defining multiple routes that reach the same action. The most common usage of this is to mimic the behavior of the default conventional route as shown in the following example:

C#

```
[Route("[controller]")]
public class Products13Controller : Controller
{
    [Route("")]     // Matches 'Products13'
    [Route("Index")] // Matches 'Products13/Index'
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

Putting multiple route attributes on the controller means that each one combines with each of the route attributes on the action methods:

C#

```
[Route("Store")]
[Route("[controller]")]
public class Products6Controller : Controller
{
    [HttpPost("Buy")]       // Matches 'Products6/Buy' and 'Store/Buy'
    [HttpPost("Checkout")]  // Matches 'Products6/Checkout' and
'Store/Checkout'
    public IActionResult Buy()
    {
```

```
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

All the HTTP verb route constraints implement `IActionConstraint`.

When multiple route attributes that implement IActionConstraint are placed on an action:

- Each action constraint combines with the route template applied to the controller.

C#

```
[Route("api/[controller]")]
public class Products7Controller : ControllerBase
{
    [HttpPut("Buy")]        // Matches PUT 'api/Products7/Buy'
    [HttpPost("Checkout")]  // Matches POST 'api/Products7/Checkout'
    public IActionResult Buy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

Using multiple routes on actions might seem useful and powerful, it's better to keep your app's URL space basic and well defined. Use multiple routes on actions **only** where needed, for example, to support existing clients.

Specifying attribute route optional parameters, default values, and constraints

Attribute routes support the same inline syntax as conventional routes to specify optional parameters, default values, and constraints.

C#

```
public class Products14Controller : Controller
{
    [HttpPost("product14/{id:int}")]
    public IActionResult ShowProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code, `[HttpPost("product14/{id:int}")]` applies a route constraint. The `Products14Controller.ShowProduct` action is matched only by URL paths like `/product14/3`. The route template portion `{id:int}` constrains that segment to only integers.

See Route Template Reference for a detailed description of route template syntax.

Custom route attributes using IRouteTemplateProvider

All of the route attributes implement IRouteTemplateProvider. The ASP.NET Core runtime:

- Looks for attributes on controller classes and action methods when the app starts.
- Uses the attributes that implement `IRouteTemplateProvider` to build the initial set of routes.

Implement `IRouteTemplateProvider` to define custom route attributes. Each `IRouteTemplateProvider` allows you to define a single route with a custom route template, order, and name:

C#

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
    public string Template => "api/[controller]";
    public int? Order => 2;
    public string Name { get; set; } = string.Empty;
}

[MyApiController]
[ApiController]
public class MyTestApiController : ControllerBase
{
    // GET /api/MyTestApi
    [HttpGet]
    public IActionResult Get()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `Get` method returns `Order = 2, Template = api/MyTestApi`.

Use application model to customize attribute routes

The application model:

- Is an object model created at startup in `Program.cs`.
- Contains all of the metadata used by ASP.NET Core to route and execute the actions in an app.

The application model includes all of the data gathered from route attributes. The data from route attributes is provided by the `IRouteTemplateProvider` implementation.

Conventions:

- Can be written to modify the application model to customize how routing behaves.
- Are read at app startup.

This section shows a basic example of customizing routing using application model. The following code makes routes roughly line up with the folder structure of the project.

C#

```
public class NamespaceRoutingConvention : Attribute,
IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
                                                 selector.AttributeRouteModel
!= null);
        if (hasRouteAttributes)
        {
            return;
        }

        var namespc = controller.ControllerType.Namespace;
        if (namespc == null)
            return;
        var template = new StringBuilder();
        template.Append(namespc, _baseNamespace.Length + 1,
                        namespc.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
        template.Append("/[controller]/[action]/{id?}");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
```

```
        }
    }
}
```

The following code prevents the `namespace` convention from being applied to controllers that are attribute routed:

C#

```
public void Apply(ControllerModel controller)
{
    var hasRouteAttributes = controller.Selectors.Any(selector =>
                                            selector.AttributeRouteModel !=
null);
    if (hasRouteAttributes)
    {
        return;
    }
```

For example, the following controller doesn't use `NamespaceRoutingConvention`:

C#

```
[Route("[controller]/[action]/{id?}")]
public class ManagersController : Controller
{
    // /managers/index
    public IActionResult Index()
    {
        var template =
ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        return Content($"Index- template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

The `NamespaceRoutingConvention.Apply` method:

- Does nothing if the controller is attribute routed.
- Sets the controllers template based on the `namespace`, with the base `namespace` removed.

The `NamespaceRoutingConvention` can be applied in `Program.cs`:

C#

```
using My.Application.Controllers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews(options =>
{
    options.Conventions.Add(
     new NamespaceRoutingConvention(typeof(HomeController).Namespace!));
});

var app = builder.Build();
```

For example, consider the following controller:

C#

```
using Microsoft.AspNetCore.Mvc;

namespace My.Application.Admin.Controllers
{
    public class UsersController : Controller
    {
        // GET /admin/controllers/users/index
        public IActionResult Index()
        {
            var fullname = typeof(UsersController).FullName;
            var template =

ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
            var path = Request.Path.Value;

            return Content($"Path: {path} fullname: {fullname}  template:
{template}");
        }

        public IActionResult List(int? id)
        {
            var path = Request.Path.Value;
            return Content($"Path: {path} ID:{id}");
        }
    }
}
```

In the preceding code:

- The base `namespace` is `My.Application`.
- The full name of the preceding controller is `My.Application.Admin.Controllers.UsersController`.

- The `NamespaceRoutingConvention` sets the controllers template to `Admin/Controllers/Users/[action]/{id?`.

The `NamespaceRoutingConvention` can also be applied as an attribute on a controller:

```
C#

[NamespaceRoutingConvention("My.Application")]
public class TestController : Controller
{
    // /admin/controllers/test/index
    public IActionResult Index()
    {
        var template =
ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        var actionname = ControllerContext.ActionDescriptor.ActionName;
        return Content($"Action- {actionname} template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

Mixed routing: Attribute routing vs conventional routing

ASP.NET Core apps can mix the use of conventional routing and attribute routing. It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. Actions that define attribute routes cannot be reached through the conventional routes and vice-versa. ***Any*** route attribute on the controller makes ***all*** actions in the controller attribute routed.

Attribute routing and conventional routing use the same routing engine.

Routing with special characters

Routing with special characters can lead to unexpected results. For example, consider a controller with the following action method:

C#

```
[HttpGet("{id?}/name")]
public async Task<ActionResult<string>> GetName(string id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null || todoItem.Name == null)
    {
        return NotFound();
    }

    return todoItem.Name;
}
```

When `string id` contains the following encoded values, unexpected results might occur:

Expand table

ASCII	Encoded
`/`	`%2F`
` `	`+`

Route parameters are not always URL decoded. This problem may be addressed in the future. For more information, see this GitHub issue;

URL Generation and ambient values

Apps can use routing URL generation features to generate URL links to actions. Generating URLs eliminates hard-coding URLs, making code more robust and maintainable. This section focuses on the URL generation features provided by MVC and only cover basics of how URL generation works. See Routing for a detailed description of URL generation.

The IUrlHelper interface is the underlying element of infrastructure between MVC and routing for URL generation. An instance of `IUrlHelper` is available through the `Url` property in controllers, views, and view components.

In the following example, the `IUrlHelper` interface is used through the `Controller.Url` property to generate a URL to another action.

C#

```
public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

If the app is using the default conventional route, the value of the `url` variable is the URL path string `/UrlGeneration/Destination`. This URL path is created by routing by combining:

- The route values from the current request, which are called **ambient values**.
- The values passed to `Url.Action` and substituting those values into the route template:

text

```
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action =
"Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

Each route parameter in the route template has its value substituted by matching names with the values and ambient values. A route parameter that doesn't have a value can:

- Use a default value if it has one.
- Be skipped if it's optional. For example, the `id` from the route template `{controller}/{action}/{id?}`.

URL generation fails if any required route parameter doesn't have a corresponding value. If URL generation fails for a route, the next route is tried until all routes have been tried or a match is found.

The preceding example of `Url.Action` assumes conventional routing. URL generation works similarly with attribute routing, though the concepts are different. With conventional routing:

- The route values are used to expand a template.
- The route values for `controller` and `action` usually appear in that template. This works because the URLs matched by routing adhere to a convention.

The following example uses attribute routing:

C#

```
public class UrlGenerationAttrController : Controller
{
    [HttpGet("custom")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination()
    {
       return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The `Source` action in the preceding code generates `custom/url/to/destination`.

LinkGenerator was added in ASP.NET Core 3.0 as an alternative to `IUrlHelper`. `LinkGenerator` offers similar but more flexible functionality. Each method on `IUrlHelper` has a corresponding family of methods on `LinkGenerator` as well.

Generating URLs by action name

Url.Action, LinkGenerator.GetPathByAction, and all related overloads all are designed to generate the target endpoint by specifying a controller name and action name.

When using `Url.Action`, the current route values for `controller` and `action` are provided by the runtime:

- The value of `controller` and `action` are part of both ambient values and values. The method `Url.Action` always uses the current values of `action` and `controller` and generates a URL path that routes to the current action.

Routing attempts to use the values in ambient values to fill in information that wasn't provided when generating a URL. Consider a route like `{a}/{b}/{c}/{d}` with ambient values `{ a = Alice, b = Bob, c = Carol, d = David }`:

- Routing has enough information to generate a URL without any additional values.
- Routing has enough information because all route parameters have a value.

If the value `{ d = Donovan }` is added:

- The value `{ d = David }` is ignored.
- The generated URL path is `Alice/Bob/Carol/Donovan`.

Warning: URL paths are hierarchical. In the preceding example, if the value `{ c = Cheryl }` is added:

- Both of the values `{ c = Carol, d = David }` are ignored.
- There is no longer a value for `d` and URL generation fails.
- The desired values of `c` and `d` must be specified to generate a URL.

You might expect to hit this problem with the default route `{controller}/{action}/{id?}`. This problem is rare in practice because `Url.Action` always explicitly specifies a `controller` and `action` value.

Several overloads of Url.Action take a route values object to provide values for route parameters other than `controller` and `action`. The route values object is frequently used with `id`. For example, `Url.Action("Buy", "Products", new { id = 17 })`. The route values object:

- By convention is usually an object of anonymous type.
- Can be an `IDictionary<>` or a POCO).

Any additional route values that don't match route parameters are put in the query string.

C#

```
public IActionResult Index()
{
    var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
    return Content(url!);
}
```

The preceding code generates `/Products/Buy/17?color=red`.

The following code generates an absolute URL:

C#

```
public IActionResult Index2()
{
    var url = Url.Action("Buy", "Products", new { id = 17 }, protocol:
Request.Scheme);
    // Returns https://localhost:5001/Products/Buy/17
    return Content(url!);
}
```

To create an absolute URL, use one of the following:

- An overload that accepts a `protocol`. For example, the preceding code.
- LinkGenerator.GetUriByAction, which generates absolute URIs by default.

Generate URLs by route

The preceding code demonstrated generating a URL by passing in the controller and action name. `IUrlHelper` also provides the Url.RouteUrl family of methods. These methods are similar to Url.Action, but they don't copy the current values of `action` and `controller` to the route values. The most common usage of `Url.RouteUrl`:

- Specifies a route name to generate the URL.
- Generally doesn't specify a controller or action name.

C#

```
public class UrlGeneration2Controller : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination2", Name = "Destination_Route")]
    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

The following Razor file generates an HTML link to the `Destination_Route`:

CSHTML

```
<h1>Test Links</h1>

<ul>
```

```
    <li><a href="@Url.RouteUrl("Destination_Route")">Test
Destination_Route</a></li>
</ul>
```

Generate URLs in HTML and Razor

IHtmlHelper provides the HtmlHelper methods Html.BeginForm and Html.ActionLink to generate `<form>` and `<a>` elements respectively. These methods use the Url.Action method to generate a URL and they accept similar arguments. The `Url.RouteUrl` companions for `HtmlHelper` are `Html.BeginRouteForm` and `Html.RouteLink` which have similar functionality.

TagHelpers generate URLs through the `form` TagHelper and the `<a>` TagHelper. Both of these use `IUrlHelper` for their implementation. See Tag Helpers in forms for more information.

Inside views, the `IUrlHelper` is available through the `Url` property for any ad-hoc URL generation not covered by the above.

URL generation in Action Results

The preceding examples showed using `IUrlHelper` in a controller. The most common usage in a controller is to generate a URL as part of an action result.

The ControllerBase and Controller base classes provide convenience methods for action results that reference another action. One typical usage is to redirect after accepting user input:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        ViewData["Message"] = $"Successful edit of customer {id}";
        return RedirectToAction("Index");
    }
    return View(customer);
}
```

The action results factory methods such as RedirectToAction and CreatedAtAction follow a similar pattern to the methods on `IUrlHelper`.

Special case for dedicated conventional routes

Conventional routing can use a special kind of route definition called a dedicated conventional route. In the following example, the route named `blog` is a dedicated conventional route:

```
C#

app.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
app.MapControllerRoute(name: "default",
               pattern: "{controller=Home}/{action=Index}/{id?}");
```

Using the preceding route definitions, `Url.Action("Index", "Home")` generates the URL path `/` using the `default` route, but why? You might guess the route values `{ controller = Home, action = Index }` would be enough to generate a URL using `blog`, and the result would be `/blog?action=Index&controller=Home`.

Dedicated conventional routes rely on a special behavior of default values that don't have a corresponding route parameter that prevents the route from being too greedy with URL generation. In this case the default values are `{ controller = Blog, action = Article }`, and neither `controller` nor `action` appears as a route parameter. When routing performs URL generation, the values provided must match the default values. URL generation using `blog` fails because the values `{ controller = Home, action = Index }` don't match `{ controller = Blog, action = Article }`. Routing then falls back to try `default`, which succeeds.

Areas

Areas are an MVC feature used to organize related functionality into a group as a separate:

- Routing namespace for controller actions.
- Folder structure for views.

Using areas allows an app to have multiple controllers with the same name, as long as they have different areas. Using areas creates a hierarchy for the purpose of routing by

adding another route parameter, `area` to `controller` and `action`. This section discusses how routing interacts with areas. See Areas for details about how areas are used with views.

The following example configures MVC to use the default conventional route and an `area` route for an `area` named `Blog`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapAreaControllerRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
app.MapControllerRoute("default_route", "{controller}/{action}/{id?}");

app.Run();
```

In the preceding code, MapAreaControllerRoute is called to create the `"blog_route"`. The second parameter, `"Blog"`, is the area name.

When matching a URL path like `/Manage/Users/AddUser`, the `"blog_route"` route generates the route values `{ area = Blog, controller = Users, action = AddUser }`. The `area` route value is produced by a default value for `area`. The route created by `MapAreaControllerRoute` is equivalent to the following:

C#

```
app.MapControllerRoute("blog_route", "Manage/{controller}/{action}/{id?}",
        defaults: new { area = "Blog" }, constraints: new { area = "Blog"
});
app.MapControllerRoute("default_route", "{controller}/{action}/{id?}");
```

`MapAreaControllerRoute` creates a route using both a default value and constraint for `area` using the provided area name, in this case `Blog`. The default value ensures that the route always produces `{ area = Blog, ... }`, the constraint requires the value `{ area = Blog, ... }` for URL generation.

Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area.

Using the preceding example, the route values `{ area = Blog, controller = Users, action = AddUser }` match the following action:

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The [Area] attribute is what denotes a controller as part of an area. This controller is in the `Blog` area. Controllers without an `[Area]` attribute are not members of any area, and do **not** match when the `area` route value is provided by routing. In the following example, only the first controller listed can match the route values `{ area = Blog, controller = Users, action = AddUser }`.

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
```

```
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        // GET /zebra/users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
```

```
    {
        // GET /users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The namespace of each controller is shown here for completeness. If the preceding controllers used the same namespace, a compiler error would be generated. Class namespaces have no effect on MVC's routing.

The first two controllers are members of areas, and only match when their respective area name is provided by the `area` route value. The third controller isn't a member of any area, and can only match when no value for `area` is provided by routing.

In terms of matching *no value*, the absence of the `area` value is the same as if the value for `area` were null or the empty string.

When executing an action inside an area, the route value for `area` is available as an ambient value for routing to use for URL generation. This means that by default areas act *sticky* for URL generation as demonstrated by the following sample.

C#

```
app.MapAreaControllerRoute(name: "duck_route",
                                    areaName: "Duck",
                                    pattern:
"Manage/{controller}/{action}/{id?}");
app.MapControllerRoute(name: "default",
                               pattern:
"Manage/{controller=Home}/{action=Index}/{id?}");
```

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
```

```
    public class UsersController : Controller
    {
        // GET /Manage/users/GenerateURLInArea
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area.
            var url = Url.Action("Index", "Home");
            // Returns /Manage/Home/Index
            return Content(url);
        }

        // GET /Manage/users/GenerateURLOutsideOfArea
        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area.
            var url = Url.Action("Index", "Home", new { area = "" });
            // Returns /Manage
            return Content(url);
        }
    }
}
```

The following code generates a URL to `/Zebra/Users/AddUser`:

C#

```
public class HomeController : Controller
{
    public IActionResult About()
    {
        var url = Url.Action("AddUser", "Users", new { Area = "Zebra" });
        return Content($"URL: {url}");
    }
```

Action definition

Public methods on a controller, except those with the NonAction attribute, are actions.

Sample code

- MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.
- View or download sample code (how to download)

Debug diagnostics

For detailed routing diagnostic output, set `Logging:LogLevel:Microsoft` to `Debug`. In the development environment, set the log level in `appsettings.Development.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Debug",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Dependency injection into controllers in ASP.NET Core

Article • 10/30/2023

By Shadi Alnamrouti and Rick Anderson

ASP.NET Core MVC controllers request dependencies explicitly via constructors. ASP.NET Core has built-in support for dependency injection (DI). DI makes apps easier to test and maintain.

View or download sample code (how to download)

Constructor injection

Services are added as a constructor parameter, and the runtime resolves the service from the service container. Services are typically defined using interfaces. For example, consider an app that requires the current time. The following interface exposes the `IDateTime` service:

C#

```
public interface IDateTime
{
    DateTime Now { get; }
}
```

The following code implements the `IDateTime` interface:

C#

```
public class SystemDateTime : IDateTime
{
    public DateTime Now
    {
        get { return DateTime.Now; }
    }
}
```

Add the service to the service container:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();

    services.AddControllersWithViews();
}
```

For more information on AddSingleton, see DI service lifetimes.

The following code displays a greeting to the user based on the time of day:

C#

```
public class HomeController : Controller
{
    private readonly IDateTime _dateTime;

    public HomeController(IDateTime dateTime)
    {
        _dateTime = dateTime;
    }

    public IActionResult Index()
    {
        var serverTime = _dateTime.Now;
        if (serverTime.Hour < 12)
        {
            ViewData["Message"] = "It's morning here - Good Morning!";
        }
        else if (serverTime.Hour < 17)
        {
            ViewData["Message"] = "It's afternoon here - Good Afternoon!";
        }
        else
        {
            ViewData["Message"] = "It's evening here - Good Evening!";
        }
        return View();
    }
```

Run the app and a message is displayed based on the time.

Action injection with `FromServices`

The FromServicesAttribute enables injecting a service directly into an action method without using constructor injection:

C#

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    return Content( $"Current server time: {dateTime.Now}");
}
```

Action injection with `FromKeyedServices`

The following code shows how to access keyed services from the DI container by using the [FromKeyedServices] attribute:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
builder.Services.AddControllers();

var app = builder.Build();

app.MapControllers();

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

[ApiController]
[Route("/cache")]
public class CustomServicesApiController : Controller
{
    [HttpGet("big")]
    public ActionResult<object> GetBigCache([FromKeyedServices("big")]
ICache cache)
    {
        return cache.Get("data-mvc");
    }
```

```
    [HttpGet("small")]
    public ActionResult<object> GetSmallCache([FromKeyedServices("small")]
ICache cache)
    {
        return cache.Get("data-mvc");
    }
}
```

Access settings from a controller

Accessing app or configuration settings from within a controller is a common pattern. The *options pattern* described in Options pattern in ASP.NET Core is the preferred approach to manage settings. Generally, don't directly inject IConfiguration into a controller.

Create a class that represents the options. For example:

C#

```
public class SampleWebSettings
{
    public string Title { get; set; }
    public int Updates { get; set; }
}
```

Add the configuration class to the services collection:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();
    services.Configure<SampleWebSettings>(Configuration);

    services.AddControllersWithViews();
}
```

Configure the app to read the settings from a JSON-formatted file:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
```

```
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("samplewebsettings.json",
                    optional: false,
                    reloadOnChange: true);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code requests the `IOptions<SampleWebSettings>` settings from the service container and uses them in the `Index` method:

C#

```
public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}
```

Additional resources

- See Test controller logic in ASP.NET Core to learn how to make code easier to test by explicitly requesting dependencies in controllers.
- Keyed service dependency injection container support
- Replace the default dependency injection container with a third party implementation.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Dependency injection into views in ASP.NET Core

Article • 06/08/2022

ASP.NET Core supports dependency injection into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. Most of the data views display should be passed in from the controller.

View or download sample code (how to download)

Configuration injection

The values in settings files, such as `appsettings.json` and `appsettings.Development.json`, can be injected into a view. Consider the `appsettings.Development.json` from the sample code:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "MyRoot": {
    "MyParent": {
      "MyChildName": "Joe"
    }
  }
}
```

The following markup displays the configuration value in a Razor Pages view:

CSHTML

```
@page
@model PrivacyModel
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration
@{
    ViewData["Title"] = "Privacy RP";
}
<h1>@ViewData["Title"]</h1>
```

```
<p>PR Privacy</p>

<h2>
   MyRoot:MyParent:MyChildName:
@Configuration["MyRoot:MyParent:MyChildName"]
</h2>
```

The following markup displays the configuration value in a MVC view:

CSHTML

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration
@{
    ViewData["Title"] = "Privacy MVC";
}
<h1>@ViewData["Title"]</h1>

<p>MVC Use this page to detail your site's privacy policy.</p>

<h2>
   MyRoot:MyParent:MyChildName:
@Configuration["MyRoot:MyParent:MyChildName"]
</h2>
```

For more information, see Configuration in ASP.NET Core

Service injection

A service can be injected into a view using the `@inject` directive.

CSHTML

```
@using System.Threading.Tasks
@using ViewInjectSample.Model
@using ViewInjectSample.Model.Services
@model IEnumerable<ToDoItem>
@inject StatisticsService StatsService
<!DOCTYPE html>
<html>
<head>
    <title>To Do Items</title>
</head>
<body>
    <div>
        <h1>To Do Items</h1>
        <ul>
            <li>Total Items: @StatsService.GetCount()</li>
            <li>Completed: @StatsService.GetCompletedCount()</li>
            <li>Avg. Priority: @StatsService.GetAveragePriority()</li>
```

```
        </ul>
        <table>
            <tr>
                <th>Name</th>
                <th>Priority</th>
                <th>Is Done?</th>
            </tr>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.Name</td>
                    <td>@item.Priority</td>
                    <td>@item.IsDone</td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

This view displays a list of `ToDoItem` instances, along with a summary showing overall statistics. The summary is populated from the injected `StatisticsService`. This service is registered for dependency injection in `ConfigureServices` in `Program.cs`:

C#

```
using ViewInjectSample.Helpers;
using ViewInjectSample.Infrastructure;
using ViewInjectSample.Interfaces;
using ViewInjectSample.Model.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
builder.Services.AddTransient<StatisticsService>();
builder.Services.AddTransient<ProfileOptionsService>();
builder.Services.AddTransient<MyHtmlHelper>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseRouting();

app.MapRazorPages();

app.MapDefaultControllerRoute();


app.Run();
```

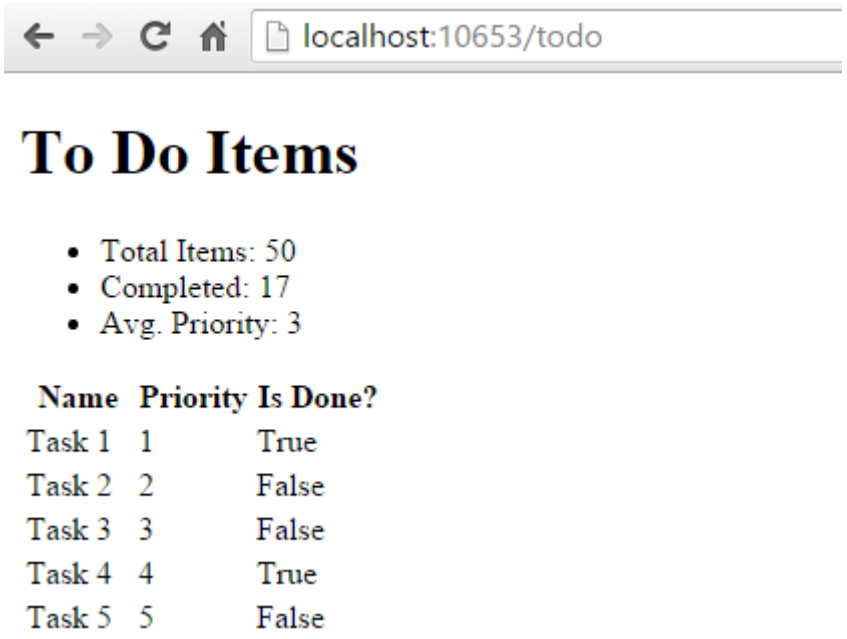
The `StatisticsService` performs some calculations on the set of `ToDoItem` instances, which it accesses via a repository:

C#

```
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _toDoItemRepository;

        public StatisticsService(IToDoItemRepository toDoItemRepository)
        {
            _toDoItemRepository = toDoItemRepository;
        }

        public int GetCount()
        {
            return _toDoItemRepository.List().Count();
        }

        public int GetCompletedCount()
        {
            return _toDoItemRepository.List().Count(x => x.IsDone);
        }

        public double GetAveragePriority()
        {
            if (_toDoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _toDoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

The sample repository uses an in-memory collection. An in-memory implementation shouldn't be used for large, remotely accessed data sets.

The sample displays data from the model bound to the view and the service injected into the view:



Populating Lookup Data

View injection can be useful to populate options in UI elements, such as dropdown lists. Consider a user profile form that includes options for specifying gender, state, and other preferences. Rendering such a form using a standard approach might require the controller or Razor Page to:

- Request data access services for each of the sets of options.
- Populate a model or `ViewBag` with each set of options to be bound.

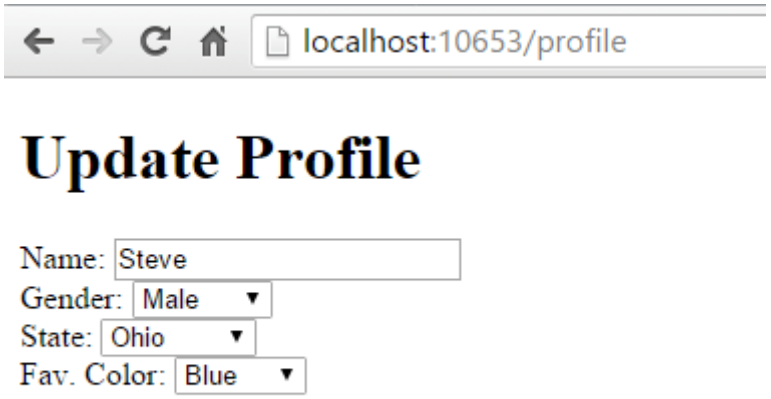
An alternative approach injects services directly into the view to obtain the options. This minimizes the amount of code required by the controller or razor Page, moving this view element construction logic into the view itself. The controller action or Razor Page to display a profile editing form only needs to pass the form the profile instance:

C#

```
using Microsoft.AspNetCore.Mvc;
using ViewInjectSample.Model;

namespace ViewInjectSample.Controllers;

public class ProfileController : Controller
{
```

```
    public IActionResult Index()
    {
        // A real app would up profile based on the user.
        var profile = new Profile()
        {
            Name = "Rick",
            FavColor = "Blue",
            Gender = "Male",
            State = new State("Ohio","OH")
        };
        return View(profile);
    }
}
```

The HTML form used to update the preferences includes dropdown lists for three of the properties:



These lists are populated by a service that has been injected into the view:

CSHTML

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
    <title>Update Profile</title>
</head>
<body>
<div>
    <h1>Update Profile</h1>
    Name: @Html.TextBoxFor(m => m.Name)
    <br/>
    Gender: @Html.DropDownList("Gender",
           Options.ListGenders().Select(g =>
                new SelectListItem() { Text = g, Value = g }))
    <br/>
```

```
    State: @Html.DropDownListFor(m => m.State!.Code,
           Options.ListStates().Select(s =>
                new SelectListItem() { Text = s.Name, Value = s.Code}))
    <br />

    Fav. Color: @Html.DropDownList("FavColor",
           Options.ListColors().Select(c =>
                new SelectListItem() { Text = c, Value = c }))
    </div>
</body>
</html>
```

The `ProfileOptionsService` is a UI-level service designed to provide just the data needed for this form:

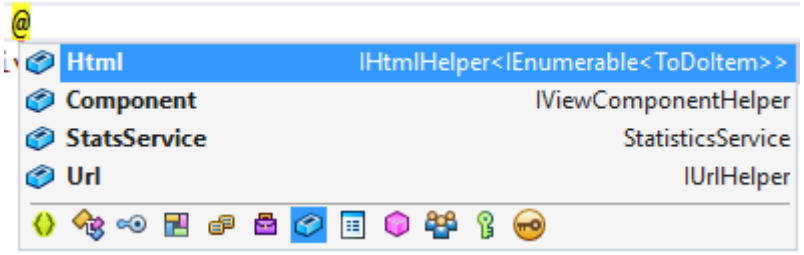
C#

```
namespace ViewInjectSample.Model.Services;

public class ProfileOptionsService
{
    public List<string> ListGenders()
    {
        // Basic sample
        return new List<string>() {"Female", "Male"};
    }

    public List<State> ListStates()
    {
        // Add a few states
        return new List<State>()
        {
            new State("Alabama", "AL"),
            new State("Alaska", "AK"),
            new State("Ohio", "OH")
        };
    }

    public List<string> ListColors()
    {
        return new List<string>() { "Blue","Green","Red","Yellow" };
    }
}
```

Note an unregistered type throws an exception at runtime because the service provider is internally queried via GetRequiredService.

Overriding Services

In addition to injecting new services, this technique can be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:



The default fields include `Html`, `Component`, and `Url`. To replace the default HTML Helpers with a custom version, use `@inject`:

CSHTML

```
@using System.Threading.Tasks
@using ViewInjectSample.Helpers
@inject MyHtmlHelper Html
<!DOCTYPE html>
<html>
<head>
    <title>My Helper</title>
</head>
<body>
    <div>
        Test: @Html.Value
    </div>
</body>
</html>
```

See Also

- Simon Timms Blog: Getting Lookup Data Into Your View

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Unit test controller logic in ASP.NET Core

Article • 09/27/2022

By Steve Smith

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Unit testing controllers

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app.

View or download sample code (how to download)

The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

C#

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new
StormSessionViewModel()
```

```
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.
- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

C#

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

C#

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
  - The `Add` method on the repository is called.
  - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

C#

```
[Fact]
public async Task
IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task
IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);
```

```
    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>
(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> **Note**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (`Controllers/SessionController.cs`):

C#

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }
```

```
    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

C#

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task
IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
```

```
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

C#

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
```

```
        }

        var result = session.Ideas.Select(idea => new IdeaDTO()
        {
            Id = idea.Id,
            Name = idea.Name,
            Description = idea.Description,
            DateCreated = idea.DateCreated
        }).ToList();

        return Ok(result);
    }

    [HttpPost("create")]
    public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        var session = await _sessionRepository.GetByIdAsync(model.SessionId);
        if (session == null)
        {
            return NotFound(model.SessionId);
        }

        var idea = new Idea()
        {
            DateCreated = DateTimeOffset.Now,
            Description = model.Description,
            Name = model.Name
        };
        session.AddIdea(idea);

        await _sessionRepository.UpdateAsync(session);

        return Ok(session);
    }
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).

- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

C#

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

C#

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
```

```
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState`:

C#

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null`, so the mock repository is configured to return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

C#

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
```

```
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

C#

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

ActionResult<T> (ActionResult<TValue>) can return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

C#

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int
sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

C#

```
[Fact]
public async Task
ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
```

```
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await
controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

C#

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

C#

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>>
CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id
}, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

C#

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
```

```
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

C#

```
[Fact]
public async Task
CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

C#

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>
(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>
(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnValue.Ideas.LastOrDefault().Description);
}
```

Additional resources

- Integration tests in ASP.NET Core
- Create and run unit tests with Visual Studio
- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)

- JustMockLite: A mocking framework for .NET developers. (*Not maintained or supported by Microsoft.*)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Welcome to Blazor!

Blazor is a .NET frontend web framework that supports both server-side rendering and client interactivity in a single programming model:

- Create rich interactive UIs using C#.
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.
- Build hybrid desktop and mobile apps with .NET and Blazor.

Using .NET for client-side web development offers the following advantages:

- Write code in C#, which can improve productivity in app development and maintenance.
- Leverage the existing .NET ecosystem of .NET libraries.
- Benefit from .NET's performance, reliability, and security.
- Stay productive on Windows, Linux, or macOS with a development environment, such as Visual Studio or Visual Studio Code. Integrate with modern hosting platforms, such as Docker.
- Build on a common set of languages, frameworks, and tools that are stable, feature-rich, and easy to use.

> **Note**
>
> For a Blazor quick start tutorial, see Build your first Blazor app.

Components

Blazor apps are based on *components*. A component in Blazor is an element of UI, such as a page, dialog, or data entry form.

Components are .NET C# classes built into .NET assemblies that:

- Define flexible UI rendering logic.
- Handle user events.
- Can be nested and reused.
- Can be shared and distributed as Razor class libraries or NuGet packages.

The component class is usually written in the form of a Razor markup page with a `.razor` file extension. Components in Blazor are formally referred to as *Razor components*, informally as *Blazor components*. Razor is a syntax for combining HTML markup with C# code designed for developer productivity. Razor allows you to switch between HTML markup and C# in the same file with IntelliSense programming support in Visual Studio.

Blazor uses natural HTML tags for UI composition. The following Razor markup demonstrates a component that increments a counter when the user selects a button.

razor

```razor
<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Components render into an in-memory representation of the browser's Document Object Model (DOM) called a *render tree*, which is used to update the UI in a flexible and efficient way.

Build a full-stack web app with Blazor

Blazor Web Apps provide a component-based architecture with server-side rendering and full client-side interactivity in a single solution, where you can switch between server-side and client-side rendering modes and even mix them in the same page.

Blazor Web Apps can quickly deliver UI to the browser by statically rendering HTML content from the server in response to requests. The page loads fast because UI rendering is performed quickly on the server without the need to download a large JavaScript bundle. Blazor can also further improve the user experience with various progressive enhancements to server rendering, such as enhanced navigation with form posts and streaming rendering of asynchronously-generated content.

Blazor supports *interactive* server-side rendering (interactive SSR), where UI interactions are handled from the server over a real-time connection with the browser. Interactive SSR enables a rich user experience like one would expect from a client app but without the need to create API endpoints to access server resources. Page content for interactive pages is prerendered, where content on the server is initially generated and sent to the client without enabling event handlers for rendered controls. The server outputs the HTML UI of the page as soon as possible in response to the initial request, which makes the app feel more responsive to users.

Blazor Web Apps support interactivity with client-side rendering (CSR) that relies on a .NET runtime built with WebAssembly that you can download with your app. When running Blazor on WebAssembly, your .NET code can access the full functionality of the browser and interop with JavaScript. Your .NET code runs in the browser's security sandbox with the protections that the sandbox provides against malicious actions on the client machine.

Blazor apps can entirely target running on WebAssembly in the browser without the involvement of a server. For a *standalone Blazor WebAssembly app*, assets are deployed as static files to a web server or service capable of serving static content to clients. Once downloaded, standalone Blazor WebAssembly apps can be cached and executed offline as a Progressive Web App (PWA).

Build a native client app with Blazor Hybrid

Blazor Hybrid enables using Razor components in a native client app with a blend of native and web technologies for web, mobile, and desktop platforms. Code runs natively in the .NET process and renders web UI to an embedded Web View control using a local interop channel. WebAssembly isn't used in Hybrid apps. Hybrid apps are built with .NET Multi-platform App UI (.NET MAUI), which is a cross-platform framework for creating native mobile and desktop apps with C# and XAML.

The Blazor Hybrid supports Windows Presentation Foundation (WPF) and Windows Forms to transition apps from earlier technology to .NET MAUI.

Next steps

Blazor Tutorial - Build your first Blazor app

ASP.NET Core Blazor supported platforms

ASP.NET Core Blazor supported platforms

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Blazor is supported in the browsers shown in the following table on both mobile and desktop platforms.

Expand table

Browser	Version
Apple Safari	Current†
Google Chrome	Current†
Microsoft Edge	Current†
Mozilla Firefox	Current†

†*Current* refers to the latest version of the browser.

For Blazor Hybrid apps, we test on and support the latest platform Web View control versions:

- Microsoft Edge WebView2 on Windows
- Chrome on Android
- Safari on iOS and macOS

Additional resources

- ASP.NET Core Blazor hosting models
- ASP.NET Core SignalR supported platforms

Tooling for ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes tools for building Blazor apps using several tools:

- Visual Studio (VS): The most comprehensive integrated development environment (IDE) for .NET developers on Windows. Includes an array of tools and features to elevate and enhance every stage of software development.
- Visual Studio Code (VS Code) is an open source, cross-platform code editor that can be used to develop Blazor apps.
- .NET CLI: The .NET command-line interface (CLI) is a cross-platform toolchain for developing, building, running, and publishing .NET applications. The .NET CLI is included with the .NET SDK and runs on any platform supported by the SDK.

Select the pivot of this article that matches your tooling choice.

To create a Blazor app with Visual Studio, use the following guidance:

- Install the latest version of Visual Studio with the **ASP.NET and web development** workload.
- Create a new project using one of the available Blazor templates:
  - **Blazor Web App**: Creates a Blazor web app that supports interactive server-side rendering (interactive SSR) and client-side rendering (CSR). The Blazor Web App template is recommended for getting started with Blazor to learn about server-side and client-side Blazor features.
  - **Blazor WebAssembly Standalone App**: Creates a standalone client web app that can be deployed as a static site.

Select **Next**.

- Provide a **Project name** and confirm that the **Location** is correct.
- For more information on the options in the **Additional information** dialog, see the Blazor project templates and template options section.

> **Note**

The hosted Blazor WebAssembly project template isn't available in ASP.NET Core 8.0 or later. To create a hosted Blazor WebAssembly app, a **Framework** option earlier than .NET 8.0 must be selected with the **ASP.NET Core Hosted** checkbox.

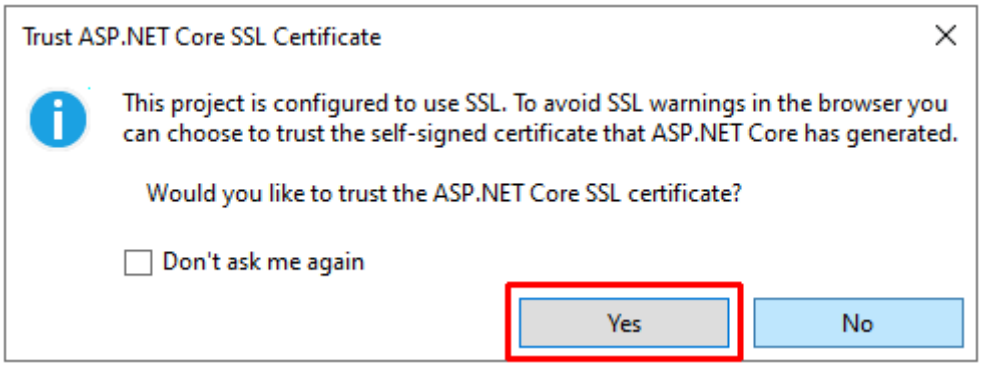
- Select **Create**.

Run the app

> **ⓘ Important**
>
> When executing a Blazor Web App, run the app from the solution's server project, which is the project with a name that doesn't end in `.Client`.

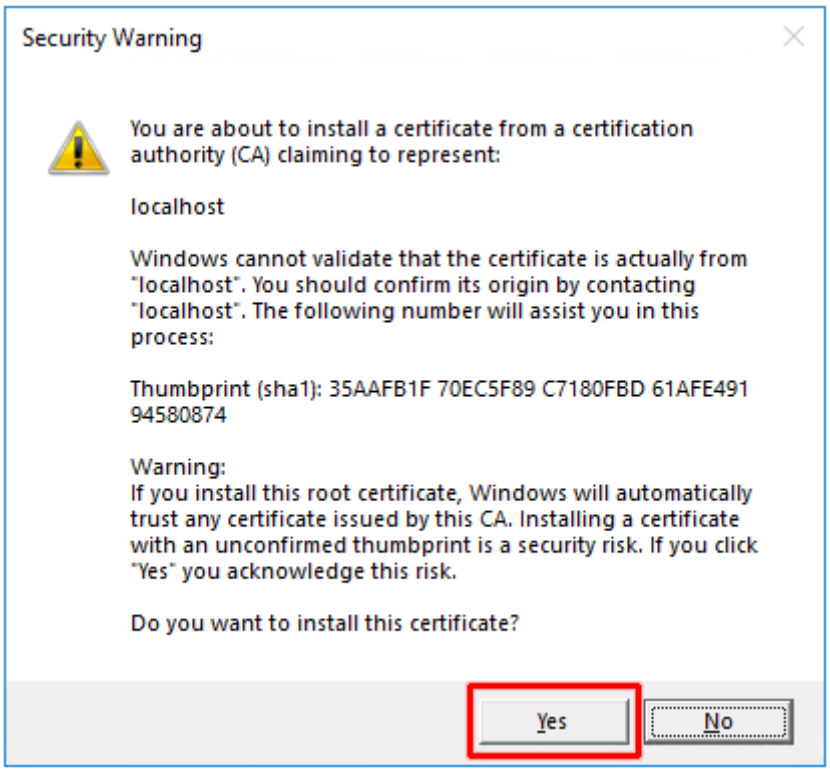
Press `Ctrl`+`F5` on the keyboard to run the app without the debugger.

Visual Studio displays the following dialog when a project isn't configured to use SSL:



Select **Yes** if you trust the ASP.NET Core SSL certificate.

The following dialog is displayed:



Select **Yes** to acknowledge the risk and install the certificate.

Visual Studio:

- Compiles and runs the app.
- Launches the default browser at `https://localhost:{PORT}`, which displays the app's UI. The `{PORT}` placeholder is the random port assigned at app creation. If you need to change the port due to a local port conflict, change the port in the project's `Properties/launchSettings.json` file.

Stop the app

Stop the app using either of the following approaches:

- Close the browser window.
- In Visual Studio, either:
  - Use the Stop button in Visual Studio's menu bar:

    
  - Press `Shift`+`F5` on the keyboard.

Visual Studio solution file (`.sln`)

A *solution* is a container to organize one or more related code projects. Solution files use a unique format and aren't intended to be edited directly.

Visual Studio and Visual Studio Code (VS Code) use a solution file (`.sln`) to store settings for a solution. The .NET CLI doesn't organize projects using a solution file, but it can create solution files and list/modify the projects in solution files via the dotnet sln command. Other .NET CLI commands use the path of the solution file for various publishing, testing, and packaging commands.

For more information, see the following resources:

- Introduction to projects and solutions (Visual Studio documentation)
- What are solutions and projects in Visual Studio? (Visual Studio documentation)
- Project management (VS Code documentation)

Blazor project templates and template options

The Blazor framework provides project templates for creating new apps. The templates are used to create new Blazor projects and solutions regardless of the tooling that you select for Blazor development (Visual Studio, Visual Studio Code, or the .NET command-line interface (CLI)):

- Blazor Web App project template: `blazor`
- Standalone Blazor WebAssembly app project template: `blazorwasm`

For more information on Blazor project templates, see ASP.NET Core Blazor project structure.

Rendering terms and concepts used in the following subsections are introduced in the following sections of the *Fundamentals* overview article:

- Client and server rendering concepts
- Static and interactive rendering concepts
- Render modes

Detailed guidance on render modes is provided by the ASP.NET Core Blazor render modes article.

Interactive render mode

- Interactive server-side rendering (interactive SSR) is enabled with the **Server** option.

- To only enable interactivity with client-side rendering (CSR), use the **WebAssembly** option.
- To enable both interactive rendering modes and the ability to automatically switch between them at runtime, use the **Auto (Server and WebAssembly)** (automatic) render mode option.
- If interactivity is set to `None`, the generated app has no interactivity. The app is only configured for static server-side rendering.

The Interactive Auto render mode initially uses interactive SSR while the .NET app bundle and runtime are download to the browser. After the .NET WebAssembly runtime is activated, the render mode switches to Interactive WebAssembly rendering.

The Blazor Web App template enables both static and interactive SSR using a single project. If you also enable CSR, the project includes an additional client project (`.Client`) for your WebAssembly-based components. The built output from the client project is downloaded to the browser and executed on the client. Any components using the WebAssembly or automatic render modes must be built from the client project.

> **ⓘ Important**
>
> When using a Blazor Web App, most of the Blazor documentation example components ***require*** interactivity to function and demonstrate the concepts covered by the articles. When you test an example component provided by an article, make sure that either the app adopts global interactivity or the component adopts an interactive render mode.

Interactivity location

Interactivity location options:

- **Per page/component**: The default sets up interactivity per page or per component.
- **Global**: Using this option sets up interactivity globally for the entire app.

Interactivity location can only be set if **Interactive render mode** isn't `None` and authentication isn't enabled.

Sample pages

To include sample pages and a layout based on Bootstrap styling, use the **Include sample pages** option. Disable this option for project without sample pages and

Bootstrap styling.

Additional guidance on template options

- ASP.NET Core Blazor render modes
- The *.NET default templates for dotnet new* article in the .NET Core documentation:
  - blazor
  - blazorwasm
- Passing the help option (`-h` or `--help`) to the dotnet new CLI command in a command shell:
  - `dotnet new blazor -h`
  - `dotnet new blazorwasm -h`

Additional resources

- Visual Studio
- Visual Studio Code
- ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation
- .NET command-line interface (CLI)
- .NET SDK
- .NET Hot Reload support for ASP.NET Core
- ASP.NET Core Blazor hosting models
- ASP.NET Core Blazor project structure
- ASP.NET Core Blazor Hybrid tutorials

ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes the build tools for standalone Blazor WebAssembly apps and how to compile an app ahead of deployment with ahead-of-time (AOT) compilation.

Although the article primarily focuses on standalone Blazor WebAssembly apps, the section on heap size for some mobile device browsers also applies to the client-side project (`.Client`) of a Blazor Web App.

.NET WebAssembly build tools

The .NET WebAssembly build tools are based on Emscripten, a compiler toolchain for the web platform. To install the build tools, use ***either*** of the following approaches:

- For the **ASP.NET and web development** workload in the Visual Studio installer, select the **.NET WebAssembly build tools** option from the list of optional components.
- Execute `dotnet workload install wasm-tools` in an administrative command shell.

> **Note**
>
> .NET WebAssembly build tools for .NET 6 projects
>
> The `wasm-tools` workload installs the build tools for the latest release. However, the current version of the build tools are incompatible with existing projects built with .NET 6. Projects using the build tools that must support both .NET 6 and a later release must use multi-targeting.
>
> Use the `wasm-tools-net6` workload for .NET 6 projects when developing apps with the .NET 7 SDK. To install the `wasm-tools-net6` workload, execute the following

command from an administrative command shell:

.NET CLI

```
dotnet workload install wasm-tools-net6
```

Ahead-of-time (AOT) compilation

Blazor WebAssembly supports ahead-of-time (AOT) compilation, where you can compile your .NET code directly into WebAssembly. AOT compilation results in runtime performance improvements at the expense of a larger app size.

Without enabling AOT compilation, Blazor WebAssembly apps run on the browser using a .NET Intermediate Language (IL) interpreter implemented in WebAssembly with partial just-in-time (JIT) runtime support, informally referred to as the *Jiterpreter*. Because the .NET IL code is interpreted, apps typically run slower than they would on a server-side .NET JIT runtime without any IL interpretation. AOT compilation addresses this performance issue by compiling an app's .NET code directly into WebAssembly for native WebAssembly execution by the browser. The AOT performance improvement can yield dramatic improvements for apps that execute CPU-intensive tasks. The drawback to using AOT compilation is that AOT-compiled apps are generally larger than their IL-interpreted counterparts, so they usually take longer to download to the client when first requested.

For guidance on installing the .NET WebAssembly build tools, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

To enable WebAssembly AOT compilation, add the `<RunAOTCompilation>` property set to `true` to the Blazor WebAssembly app's project file:

XML

```
<PropertyGroup>
  <RunAOTCompilation>true</RunAOTCompilation>
</PropertyGroup>
```

To compile the app to WebAssembly, publish the app. Publishing the `Release` configuration ensures the .NET Intermediate Language (IL) linking is also run to reduce the size of the published app:

.NET CLI

```
dotnet publish -c Release
```

WebAssembly AOT compilation is only performed when the project is published. AOT compilation isn't used when the project is run during development (`Development` environment) because AOT compilation usually takes several minutes on small projects and potentially much longer for larger projects. Reducing the build time for AOT compilation is under development for future releases of ASP.NET Core.

The size of an AOT-compiled Blazor WebAssembly app is generally larger than the size of the app if compiled into .NET IL:

- Although the size difference depends on the app, most AOT-compiled apps are about twice the size of their IL-compiled versions. This means that using AOT compilation trades off load-time performance for runtime performance. Whether this tradeoff is worth using AOT compilation depends on your app. Blazor WebAssembly apps that are CPU intensive generally benefit the most from AOT compilation.

- The larger size of an AOT-compiled app is due to two conditions:
  - More code is required to represent high-level .NET IL instructions in native WebAssembly.
  - AOT doesn't trim out managed DLLs when the app is published. Blazor requires the DLLs for reflection metadata and to support certain .NET runtime features. Requiring the DLLs on the client increases the download size but provides a more compatible .NET experience.

> **Note**
>
> For Mono /WebAssembly MSBuild properties and targets, see WasmApp.Common.targets (dotnet/runtime GitHub repository). Official documentation for common MSBuild properties is planned per Document blazor msbuild configuration options (dotnet/docs #27395).

Trim .NET IL after ahead-of-time (AOT) compilation

The `WasmStripILAfterAOT` MSBuild option enables removing the .NET Intermediate Language (IL) for compiled methods after performing AOT compilation to WebAssembly, which reduces the size of the `_framework` folder.

In the app's project file:

```xml
<PropertyGroup>
  <RunAOTCompilation>true</RunAOTCompilation>
  <WasmStripILAfterAOT>true</WasmStripILAfterAOT>
</PropertyGroup>
```

This setting trims away the IL code for most compiled methods, including methods from libraries and methods in the app. Not all compiled methods can be trimmed, as some are still required by the .NET interpreter at runtime.

To report a problem with the trimming option, open an issue on the dotnet/runtime GitHub repository.

Disable the trimming property if it prevents your app from running normally:

```xml
<WasmStripILAfterAOT>false</WasmStripILAfterAOT>
```

Heap size for some mobile device browsers

When building a Blazor app that runs on the client and targets mobile device browsers, especially Safari on iOS, decreasing the maximum memory for the app with the MSBuild property `EmccMaximumHeapSize` may be required. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

Runtime relinking

One of the largest parts of a Blazor WebAssembly app is the WebAssembly-based .NET runtime (`dotnet.wasm`) that the browser must download when the app is first accessed by a user's browser. Relinking the .NET WebAssembly runtime trims unused runtime code and thus improves download speed.

Runtime relinking requires installation of the .NET WebAssembly build tools. For more information, see Tooling for ASP.NET Core Blazor.

With the .NET WebAssembly build tools installed, runtime relinking is performed automatically when an app is **published** in the `Release` configuration. The size reduction

is particularly dramatic when disabling globalization. For more information, see ASP.NET Core Blazor globalization and localization.

> **ⓘ Important**
>
> Runtime relinking trims class instance JavaScript-invokable .NET methods unless they're protected. For more information, see **Call .NET methods from JavaScript functions in ASP.NET Core Blazor**.

Single Instruction, Multiple Data (SIMD)

Blazor uses WebAssembly Single Instruction, Multiple Data (SIMD) to improve the throughput of vectorized computations by performing an operation on multiple pieces of data in parallel using a single instruction.

To disable SIMD, for example when targeting old browsers or browsers on mobile devices that don't support SIMD, set the `<WasmEnableSIMD>` property to `false` in the app's project file (`.csproj`):

XML

```
<PropertyGroup>
  <WasmEnableSIMD>false</WasmEnableSIMD>
</PropertyGroup>
```

For more information, see Configuring and hosting .NET WebAssembly applications: SIMD - Single instruction, multiple data and note that the guidance isn't versioned and applies to the latest public release.

Exception handling

Exception handling is enabled by default. To disable exception handling, add the `<WasmEnableExceptionHandling>` property with a value of `false` in the app's project file (`.csproj`):

XML

```
<PropertyGroup>
  <WasmEnableExceptionHandling>false</WasmEnableExceptionHandling>
</PropertyGroup>
```

For more information, see the following resources:

- Configuring and hosting .NET WebAssembly applications: EH - Exception handling
- Exception handling

Additional resources

- ASP.NET Core Blazor WebAssembly native dependencies
- Webcil packaging format for .NET assemblies

ASP.NET Core Blazor hosting models

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains Blazor hosting models, primarily focused on Blazor Server and Blazor WebAssembly apps in versions of .NET earlier than .NET 8. The guidance in this article is relevant under all .NET releases for Blazor Hybrid apps that run on native mobile and desktop platforms. Blazor Web Apps in .NET 8 or later are better conceptualized by how Razor components are rendered, which is described as their *render mode*. Render modes are briefly touched on in the *Fundamentals* overview article and covered in detail in ASP.NET Core Blazor render modes of the *Components* node.

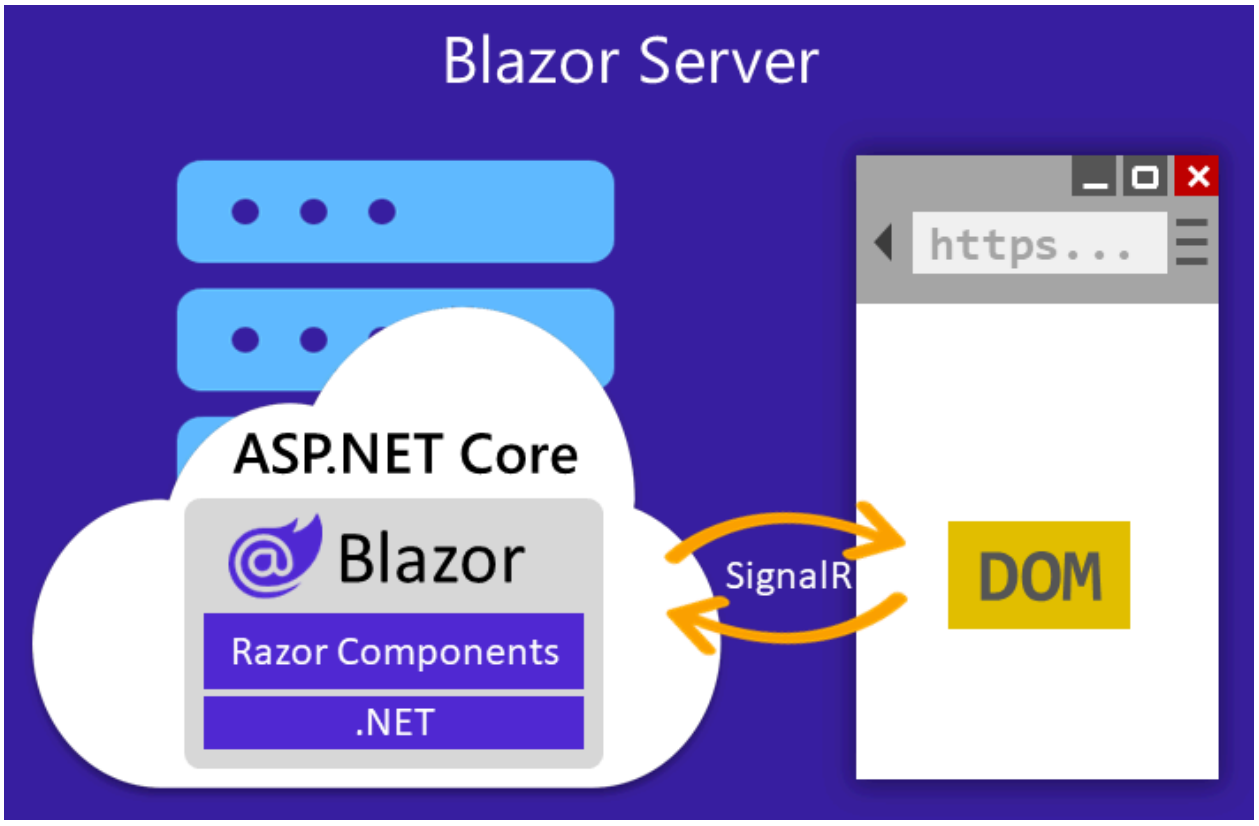
Blazor is a web framework for building web UI components (Razor components) that can be hosted in different ways. Razor components can run server-side in ASP.NET Core (*Blazor Server*) versus client-side in the browser on a WebAssembly -based .NET runtime (*Blazor WebAssembly*, *Blazor WASM*). You can also host Razor components in native mobile and desktop apps that render to an embedded Web View control (*Blazor Hybrid*). Regardless of the hosting model, the way you build Razor components *is the same*. The same Razor components can be used with any of the hosting models unchanged.

Blazor Server

With the Blazor Server hosting model, components are executed on the server from within an ASP.NET Core app. UI updates, event handling, and JavaScript calls are handled over a SignalR connection using the WebSockets protocol. The state on the server associated with each connected client is called a *circuit*. Circuits aren't tied to a specific network connection and can tolerate temporary network interruptions and attempts by the client to reconnect to the server when the connection is lost.

In a traditional server-rendered app, opening the same app in multiple browser screens (tabs or `iframes`) typically doesn't translate into additional resource demands on the server. For the Blazor Server hosting model, each browser screen requires a separate circuit and separate instances of server-managed component state. Blazor considers closing a browser tab or navigating to an external URL a *graceful* termination. In the

event of a graceful termination, the circuit and associated resources are immediately released. A client may also disconnect non-gracefully, for instance due to a network interruption. Blazor Server stores disconnected circuits for a configurable interval to allow the client to reconnect.



On the client, the Blazor script establishes the SignalR connection with the server. The script is served from an embedded resource in the ASP.NET Core shared framework.

The Blazor Server hosting model offers several benefits:

- Download size is significantly smaller than when the Blazor WebAssembly hosting model is used, and the app loads much faster.
- The app takes full advantage of server capabilities, including the use of .NET Core APIs.
- .NET Core on the server is used to run the app, so existing .NET tooling, such as debugging, works as expected.
- Thin clients are supported. For example, Blazor Server works with browsers that don't support WebAssembly and on resource-constrained devices.
- The app's .NET/C# code base, including the app's component code, isn't served to clients.

The Blazor Server hosting model has the following limitations:

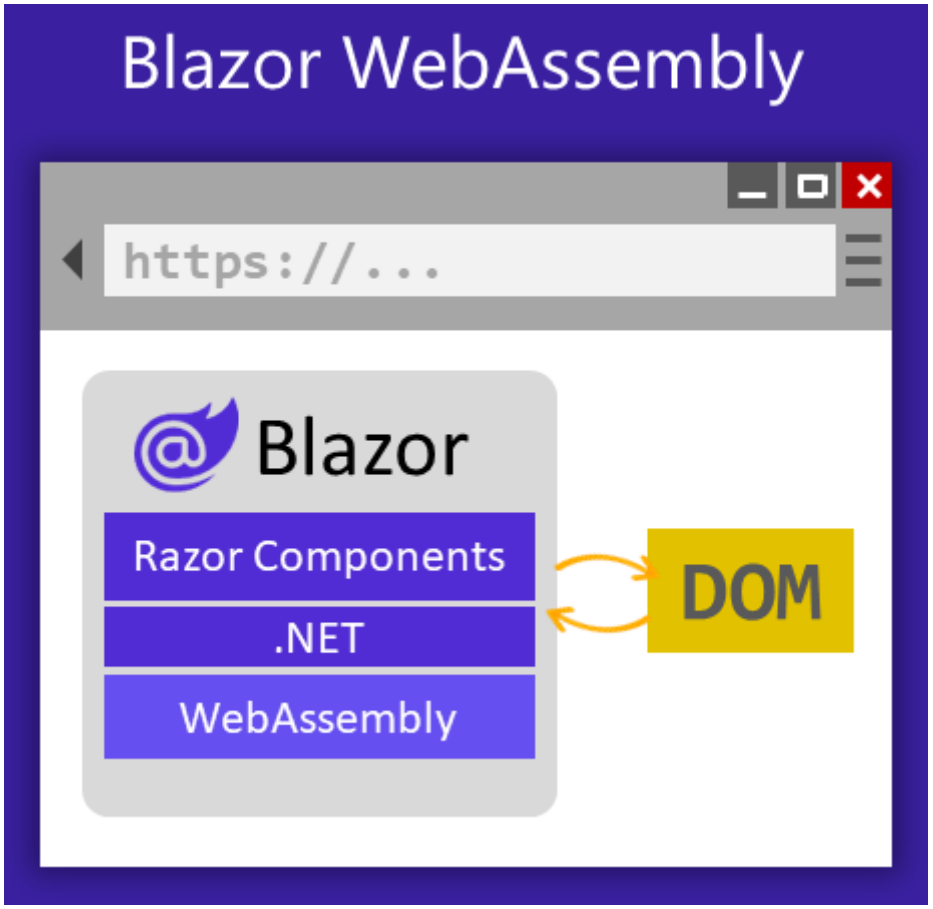
- Higher latency usually exists. Every user interaction involves a network hop.
- There's no offline support. If the client connection fails, interactivity fails.

- Scaling apps with many users requires server resources to handle multiple client connections and client state.
- An ASP.NET Core server is required to serve the app. Serverless deployment scenarios aren't possible, such as serving the app from a Content Delivery Network (CDN).

We recommend using the Azure SignalR Service for apps that adopt the Blazor Server hosting model. The service allows for scaling up a Blazor Server app to a large number of concurrent SignalR connections.

Blazor WebAssembly

The Blazor WebAssembly hosting model runs components client-side in the browser on a WebAssembly-based .NET runtime. Razor components, their dependencies, and the .NET runtime are downloaded to the browser. Components are executed directly on the browser UI thread. UI updates and event handling occur within the same process. Assets are deployed as static files to a web server or service capable of serving static content to clients.



Blazor web apps can use the Blazor WebAssembly hosting model to enable client-side interactivity. When an app is created that exclusively runs on the Blazor WebAssembly

hosting model without server-side rendering and interactivity, the app is called a *standalone* Blazor WebAssembly app.

When a standalone Blazor WebAssembly app uses a backend ASP.NET Core app to serve its files, the app is called a *hosted* Blazor WebAssembly app. Using hosted Blazor WebAssembly, you get a full-stack web development experience with .NET, including the ability to share code between the client and server apps, support for prerendering, and integration with MVC and Razor Pages. A hosted client app can interact with its backend server app over the network using a variety of messaging frameworks and protocols, such as web API, gRPC-web, and SignalR (Use ASP.NET Core SignalR with Blazor).

A Blazor WebAssembly app built as a Progressive Web App (PWA) uses modern browser APIs to enable many of the capabilities of a native client app, such as working offline, running in its own app window, launching from the host's operating system, receiving push notifications, and automatically updating in the background.

The Blazor script handles:

- Downloading the .NET runtime, Razor components, and the component's dependencies.
- Initialization of the runtime.

The size of the published app, its *payload size*, is a critical performance factor for an app's usability. A large app takes a relatively long time to download to a browser, which diminishes the user experience. Blazor WebAssembly optimizes payload size to reduce download times:

- Unused code is stripped out of the app when it's published by the Intermediate Language (IL) Trimmer.
- HTTP responses are compressed.
- The .NET runtime and assemblies are cached in the browser.

The Blazor WebAssembly hosting model offers several benefits:

- For standalone Blazor WebAssembly apps, there's no .NET server-side dependency after the app is downloaded from the server, so the app remains functional if the server goes offline.
- Client resources and capabilities are fully leveraged.
- Work is offloaded from the server to the client.
- For standalone Blazor WebAssembly apps, an ASP.NET Core web server isn't required to host the app. Serverless deployment scenarios are possible, such as serving the app from a Content Delivery Network (CDN).

The Blazor WebAssembly hosting model has the following limitations:

- Razor components are restricted to the capabilities of the browser.
- Capable client hardware and software (for example, WebAssembly support) is required.
- Download size is larger, and components take longer to load.
- Code sent to the client can't be protected from inspection and tampering by users.

The .NET Intermediate Language (IL) interpreter includes partial just-in-time (JIT) runtime support to achieve improved runtime performance. The JIT interpreter optimizes execution of interpreter bytecodes by replacing them with tiny blobs of WebAssembly code. The JIT interpreter is automatically enabled for Blazor WebAssembly apps except when debugging.

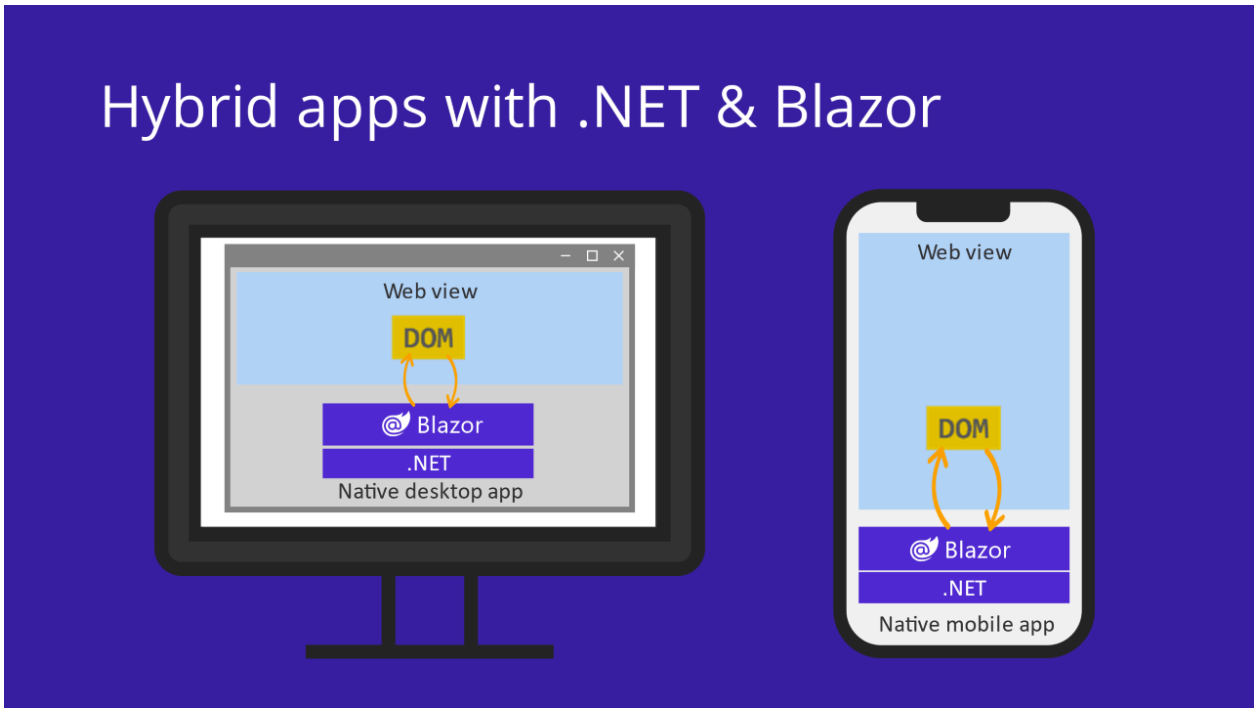
Blazor supports ahead-of-time (AOT) compilation, where you can compile your .NET code directly into WebAssembly. AOT compilation results in runtime performance improvements at the expense of a larger app size. For more information, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

The same .NET WebAssembly build tools used for AOT compilation also relink the .NET WebAssembly runtime to trim unused runtime code. Blazor also trims unused code from .NET framework libraries. The .NET compiler further precompresses a standalone Blazor WebAssembly app for a smaller app payload.

WebAssembly-rendered Razor components can use native dependencies built to run on WebAssembly.

Blazor Hybrid

Blazor can also be used to build native client apps using a hybrid approach. Hybrid apps are native apps that leverage web technologies for their functionality. In a Blazor Hybrid app, Razor components run directly in the native app (not on WebAssembly) along with any other .NET code and render web UI based on HTML and CSS to an embedded Web View control through a local interop channel.



Blazor Hybrid apps can be built using different .NET native app frameworks, including .NET MAUI, WPF, and Windows Forms. Blazor provides `BlazorWebView` controls for adding Razor components to apps built with these frameworks. Using Blazor with .NET MAUI offers a convenient way to build cross-platform Blazor Hybrid apps for mobile and desktop, while Blazor integration with WPF and Windows Forms can be a great way to modernize existing apps.

Because Blazor Hybrid apps are native apps, they can support functionality that isn't available with only the web platform. Blazor Hybrid apps have full access to native platform capabilities through normal .NET APIs. Blazor Hybrid apps can also share and reuse components with existing Blazor Server or Blazor WebAssembly apps. Blazor Hybrid apps combine the benefits of the web, native apps, and the .NET platform.

The Blazor Hybrid hosting model offers several benefits:

- Reuse existing components that can be shared across mobile, desktop, and web.
- Leverage web development skills, experience, and resources.
- Apps have full access to the native capabilities of the device.

The Blazor Hybrid hosting model has the following limitations:

- Separate native client apps must be built, deployed, and maintained for each target platform.
- Native client apps usually take longer to find, download, and install over accessing a web app in a browser.

For more information, see ASP.NET Core Blazor Hybrid.

For more information on Microsoft native client frameworks, see the following resources:

- .NET Multi-platform App UI (.NET MAUI)
- Windows Presentation Foundation (WPF)
- Windows Forms

Which Blazor hosting model should I choose?

A component's hosting model is set by its *render mode*, either at compile time or runtime, which is described with examples in ASP.NET Core Blazor render modes. The following table shows the primary considerations for setting the render mode to determine a component's hosting model. For standalone Blazor WebAssembly apps, all of the app's components are rendered on the client with the Blazor WebAssembly hosting model.

Blazor Hybrid apps include .NET MAUI, WPF, and Windows Forms framework apps.

Expand table

Feature	Blazor Server	Blazor WebAssembly (WASM)	Blazor Hybrid
Complete .NET API compatibility	✔	✘	✔
Direct access to server and network resources	✔	✘†	✘†
Small payload size with fast initial load time	✔	✘	✘
Near native execution speed	✔	✔‡	✔
App code secure and private on the server	✔	✘†	✘†
Run apps offline once downloaded	✘	✔	✔
Static site hosting	✘	✔	✘
Offloads processing to clients	✘	✔	✔
Full access to native client capabilities	✘	✘	✔
Web-based deployment	✔	✔	✘

†Blazor WebAssembly and Blazor Hybrid apps can use server-based APIs to access server/network resources and access private and secure app code.
‡Blazor WebAssembly only reaches near-native performance with ahead-of-time (AOT) compilation.

After you choose the app's hosting model, you can generate a Blazor Server or Blazor WebAssembly app from a Blazor project template. For more information, see Tooling for ASP.NET Core Blazor.

To create a Blazor Hybrid app, see the articles under ASP.NET Core Blazor Hybrid tutorials.

Complete .NET API compatibility

Components rendered for the Blazor Server hosting model and Blazor Hybrid apps have complete .NET API compatibility, while components rendered for Blazor WebAssembly are limited to a subset of .NET APIs. When an app's specification requires one or more .NET APIs that are unavailable to WebAssembly-rendered components, then choose to render components for Blazor Server or use Blazor Hybrid.

Direct access to server and network resources

Components rendered for the Blazor Server hosting model have direct access to server and network resources where the app is executing. Because components hosted using Blazor WebAssembly or Blazor Hybrid execute on a client, they don't have direct access to server and network resources. Components can access server and network resources *indirectly* via protected server-based APIs. Server-based APIs might be available via third-party libraries, packages, and services. Take into account the following considerations:

- Third-party libraries, packages, and services might be costly to implement and maintain, weakly supported, or introduce security risks.
- If one or more server-based APIs are developed internally by your organization, additional resources are required to build and maintain them.

Use the Blazor Server hosting model to avoid the need to expose APIs from the server environment.

Small payload size with fast initial load time

Rendering components from the server reduces the app payload size and improves initial load times. When a fast initial load time is desired, use the Blazor Server hosting

model or consider static server-side rendering.

Near native execution speed

Blazor Hybrid apps run using the .NET runtime natively on the target platform, which offers the best possible speed.

Components rendered for the Blazor WebAssembly hosting model, including Progressive Web Apps (PWAs), and standalone Blazor WebAssembly apps run using the .NET runtime for WebAssembly, which is slower than running directly on the platform. Consider using ahead-of-time (AOT) compiled to improve runtime performance when using Blazor WebAssembly.

App code secure and private on the server

Maintaining app code securely and privately on the server is a built-in feature of components rendered for the Blazor Server hosting model. Components rendered using the Blazor WebAssembly or Blazor Hybrid hosting models can use server-based APIs to access functionality that must be kept private and secure. The considerations for developing and maintaining server-based APIs described in the Direct access to server and network resources section apply. If the development and maintenance of server-based APIs isn't desirable for maintaining secure and private app code, render components for the Blazor Server hosting model.

Run apps offline once downloaded

Standalone Blazor WebAssembly apps built as Progressive Web Apps (PWAs) and Blazor Hybrid apps can run offline, which is particularly useful when clients aren't able to connect to the Internet. Components rendered for the Blazor Server hosting model fail to run when the connection to the server is lost. If an app must run offline, standalone Blazor WebAssembly and Blazor Hybrid are the best choices.

Static site hosting

Static site hosting is possible with standalone Blazor WebAssembly apps because they're downloaded to clients as a set of static files. Standalone Blazor WebAssembly apps don't require a server to execute server-side code in order to download and run and can be delivered via a Content Delivery Network (CDN) (for example, Azure CDN).

Although Blazor Hybrid apps are compiled into one or more self-contained deployment assets, the assets are usually provided to clients through a third-party app store. If static

hosting is an app requirement, select standalone Blazor WebAssembly.

Offloads processing to clients

Blazor WebAssembly and Blazor Hybrid apps execute on clients and thus offload processing to clients. Blazor Server apps execute on a server, so server resource demand typically increases with the number of users and the amount of processing required per user. When it's possible to offload most or all of an app's processing to clients and the app processes a significant amount of data, Blazor WebAssembly or Blazor Hybrid is the best choice.

Full access to native client capabilities

Blazor Hybrid apps have full access to native client API capabilities via .NET native app frameworks. In Blazor Hybrid apps, Razor components run directly in the native app, not on WebAssembly. When full client capabilities are a requirement, Blazor Hybrid is the best choice.

Web-based deployment

Blazor web apps are updated on the next app refresh from the browser.

Blazor Hybrid apps are native client apps that typically require an installer and platform-specific deployment mechanism.

Setting a component's hosting model

To set a component's hosting model to Blazor Server or Blazor WebAssembly at compile-time or dynamically at runtime, you set its *render mode*. Render modes are fully explained and demonstrated in the ASP.NET Core Blazor render modes article. We don't recommend that you jump from this article directly to the *Render modes* article without reading the content in the articles between these two articles. For example, render modes are more easily understood by looking at Razor component examples, but basic Razor component structure and function isn't covered until the ASP.NET Core Blazor fundamentals article is reached. It's also helpful to learn about Blazor's project templates and tooling before working with the component examples in the *Render modes* article.

ASP.NET Core Blazor tutorials

Article • 11/18/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

The following tutorials provide basic working experiences for building Blazor apps.

For an overview of Blazor, see ASP.NET Core Blazor.

- Build your first Blazor app
- Build a Blazor todo list app (Blazor Web App)
- Build a Blazor movie database app (Overview) (Blazor Web App)
- Use ASP.NET Core SignalR with Blazor (Blazor Web App)
- ASP.NET Core Blazor Hybrid tutorials
- Microsoft Learn
  - Blazor Learning Path
  - Blazor Learn Modules

Build a Blazor todo list app

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial provides a basic working experience for building and modifying a Blazor app. For detailed Blazor guidance, see the Blazor reference documentation.

Learn how to:

- ✓ Create a todo list Blazor app project
- ✓ Modify Razor components
- ✓ Use event handling and data binding in components
- ✓ Use routing in a Blazor app

At the end of this tutorial, you'll have a working todo list app.

Prerequisites

Download and install .NET if it isn't already installed on the system or if the system doesn't have the latest version installed.

Create a Blazor app

Create a new Blazor Web App named `TodoList` in a command shell:

.NET CLI

```
dotnet new blazor -o TodoList
```

The `-o|--output` option creates a folder for the project. If you've created a folder for the project and the command shell is open in that folder, omit the `-o|--output` option and value to create the project.

The preceding command creates a folder named `TodoList` with the `-o|--output` option to hold the app. The `TodoList` folder is the *root folder* of the project. Change directories to the `TodoList` folder with the following command:

.NET CLI

```
cd TodoList
```

Build a todo list Blazor app

Add a new `Todo` Razor component to the app using the following command:

.NET CLI

```
dotnet new razorcomponent -n Todo -o Components/Pages
```

The `-n|--name` option in the preceding command specifies the name of the new Razor component. The new component is created in the project's `Components/Pages` folder with the `-o|--output` option.

> **ⓘ Important**
>
> Razor component file names require a capitalized first letter. Open the `Pages` folder and confirm that the `Todo` component file name starts with a capital letter `T`. The file name should be `Todo.razor`.

Open the `Todo` component in any file editor and make the following changes at the top of the file:

- Add an `@page` Razor directive with a relative URL of `/todo`.
- Enable interactivity on the page so that it isn't just statically rendered. The Interactive Server render mode enables the component to handle UI events from the server.
- Add a page title with the `PageTitle` component, which enables adding an HTML `<title>` element to the page.

`Todo.razor`:

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>
```

```
@code {

}
```

Save the `Todo.razor` file.

Add the `Todo` component to the navigation bar.

The `NavMenu` component is used in the app's layout. Layouts are components that allow you to avoid duplication of content in an app. The `NavLink` component provides a cue in the app's UI when the component URL is loaded by the app.

In the navigation element (`<nav>`) content of the `NavMenu` component, add the following `<div>` element for the `Todo` component.

In `Components/Layout/NavMenu.razor`:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="todo">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Todo
    </NavLink>
</div>
```

Save the `NavMenu.razor` file.

Build and run the app by executing the dotnet watch run command in the command shell from the `TodoList` folder. After the app is running, visit the new Todo page by selecting the **`Todo`** link in the app's navigation bar, which loads the page at `/todo`.

Leave the app running the command shell. Each time a file is saved, the app is automatically rebuilt, and the page in the browser is automatically reloaded.

Add a `TodoItem.cs` file to the root of the project (the `TodoList` folder) to hold a class that represents a todo item. Use the following C# code for the `TodoItem` class.

`TodoItem.cs`:

C#

```
public class TodoItem
{
    public string? Title { get; set; }
```

```
    public bool IsDone { get; set; }
}
```

> **Note**
>
> If using Visual Studio to create the `TodoItem.cs` file and `TodoItem` class, use ***either*** of the following approaches:
>
> - Remove the namespace that Visual Studio generates for the class.
> - Use the **Copy** button in the preceding code block and replace the entire contents of the file that Visual Studio generates.

Return to the `Todo` component and perform the following tasks:

- Add a field for the todo items in the `@code` block. The `Todo` component uses this field to maintain the state of the todo list.
- Add unordered list markup and a `foreach` loop to render each todo item as a list item (`<li>`).

`Components/Pages/Todo.razor`:

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

@code {
    private List<TodoItem> todos = new();
}
```

The app requires UI elements for adding todo items to the list. Add a text input (`<input>`) and a button (`<button>`) below the unordered list (`<ul>...</ul>`):

razor

```razor
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" />
<button>Add todo</button>

@code {
    private List<TodoItem> todos = new();
}
```

Save the `TodoItem.cs` file and the updated `Todo.razor` file. In the command shell, the app is automatically rebuilt when the files are saved. The browser reloads the page.

When the **`Add todo`** button is selected, nothing happens because an event handler isn't attached to the button.

Add an `AddTodo` method to the `Todo` component and register the method for the button using the `@onclick` attribute. The `AddTodo` C# method is called when the button is selected:

razor

```razor
<input placeholder="Something todo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private List<TodoItem> todos = new();

    private void AddTodo()
    {
        // Todo: Add the todo
    }
}
```

To get the title of the new todo item, add a `newTodo` string field at the top of the `@code` block:

C#

```
private string? newTodo;
```

Modify the text `<input>` element to bind `newTodo` with the `@bind` attribute:

razor

```
<input placeholder="Something todo" @bind="newTodo" />
```

Update the `AddTodo` method to add the `TodoItem` with the specified title to the list. Clear the value of the text input by setting `newTodo` to an empty string:

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h3>Todo</h3>

<ul>
    @foreach (var todo in todos)
    {
        <li>@todo.Title</li>
    }
</ul>

<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>

@code {
    private List<TodoItem> todos = new();
    private string? newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

Save the `Todo.razor` file. The app is automatically rebuilt in the command shell, and the page reloads in the browser.

The title text for each todo item can be made editable, and a checkbox can help the user keep track of completed items. Add a checkbox input for each todo item and bind its value to the `IsDone` property. Change `@todo.Title` to an `<input>` element bound to `todo.Title` with `@bind`:

razor

```
<ul>
      @foreach (var todo in todos)
      {
         <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
         </li>
      }
</ul>
```

Update the `<h3>` header to show a count of the number of todo items that aren't complete (`IsDone` is `false`). The Razor expression in the following header evaluates each time Blazor rerenders the component.

razor

```
<h3>Todo (@todos.Count(todo => !todo.IsDone))</h3>
```

The completed `Todo` component:

razor

```
@page "/todo"
@rendermode InteractiveServer

<PageTitle>Todo</PageTitle>

<h1>Todo (@todos.Count(todo => !todo.IsDone))</h1>

<ul>
    @foreach (var todo in todos)
    {
        <li>
            <input type="checkbox" @bind="todo.IsDone" />
            <input @bind="todo.Title" />
        </li>
    }
</ul>

<input placeholder="Something todo" @bind="newTodo" />
<button @onclick="AddTodo">Add todo</button>
```

```
@code {
    private List<TodoItem> todos = new();
    private string? newTodo;

    private void AddTodo()
    {
        if (!string.IsNullOrWhiteSpace(newTodo))
        {
            todos.Add(new TodoItem { Title = newTodo });
            newTodo = string.Empty;
        }
    }
}
```

Save the `Todo.razor` file. The app is automatically rebuilt in the command shell, and the page reloads in the browser.

Add items, edit items, and mark todo items done to test the component.

When finished, shut down the app in the command shell. Many command shells accept the keyboard command `Ctrl`+`C` (Windows) or `⌘`+`C` (macOS) to stop an app.

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app.

Next steps

In this tutorial, you learned how to:

- ✓ Create a todo list Blazor app project
- ✓ Modify Razor components
- ✓ Use event handling and data binding in components
- ✓ Use routing in a Blazor app

Learn about tooling for ASP.NET Core Blazor:

ASP.NET Core Blazor

Build a Blazor movie database app (Overview)

Article • 11/15/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial explains the basics of building a Blazor Web App with a database, Entity Framework (EF) Core, and user interactivity.

Parts of this series include:

1. Create a Blazor Web App
2. Add and scaffold a model
3. Learn about Razor components
4. Work with a database
5. Add validation
6. Add search
7. Add a new field
8. Add interactivity

At the end of the tutorial, you'll have a Blazor Web App that can display and manage movies in a movie database.

Secure authentication flow required for production apps

This tutorial uses a local database that doesn't require user authentication. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production Blazor Web Apps, see the following resources:

- ASP.NET Core Blazor authentication and authorization
- ASP.NET Core Blazor authentication and authorization and the following articles in the *Server* security node
- Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)
- Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Sample app

If you don't intend to create the demonstration app while reading the article, you can refer to the completed sample app in the Blazor samples GitHub repository (dotnet/blazor-samples). Select the latest version folder in the repository. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Article code examples

The line breaks of code examples shown in the ASP.NET Core documentation often don't match line breaks in scaffolded code generated by tooling for an app. This is due to an article publishing limitation. Lines of code in articles are generally limited to 85 characters in length, and we manually adjust the line length using line breaks to satisfy our publishing guidelines.

As you work through this tutorial or use any other ASP.NET Core article's code examples, you never need to adjust scaffolded code in your app to match the line breaks displayed in article code examples.

Report a tutorial issue

To open a documentation GitHub issue for an article of the series, use the **Open a documentation issue** link at the bottom of the article. Using the link to create your issue adds important tracking metadata to the issue and automatically pings the author of the article.

Support requests

We welcome feedback on the tutorial's articles, such as bug reports and comments on the article's text, but we're often unable to provide product support. If you run into a problem while following the tutorial, don't immediately open a documentation issue.

Check the steps that you've taken against the article and compare your code to the sample app before opening an issue because many problems can be traced to missing a step or not following a step correctly.

For general questions about .NET and Blazor beyond the tutorial and reference documentation or to obtain assistance from the .NET community, converse with developers in public forums.

Next steps

Next: Create a Blazor Web App

Build a Blazor movie database app (Part 1 - Create a Blazor Web App)

Article • 11/15/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the first part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series covers how to create a Blazor Web App that adopts static server-side rendering (static SSR). Static SSR means that content is rendered on the server and sent to the client for display in response to individual requests.

Prerequisites

Visual Studio (latest release) with the **ASP.NET and web development** workload

Create a Blazor Web App

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorWebAppMovies` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.
- Confirm that the **Location** for the app is suitable. Set the **Place solution and project in the same directory** checkbox to match your preferred solution file location. Select the **Next** button.

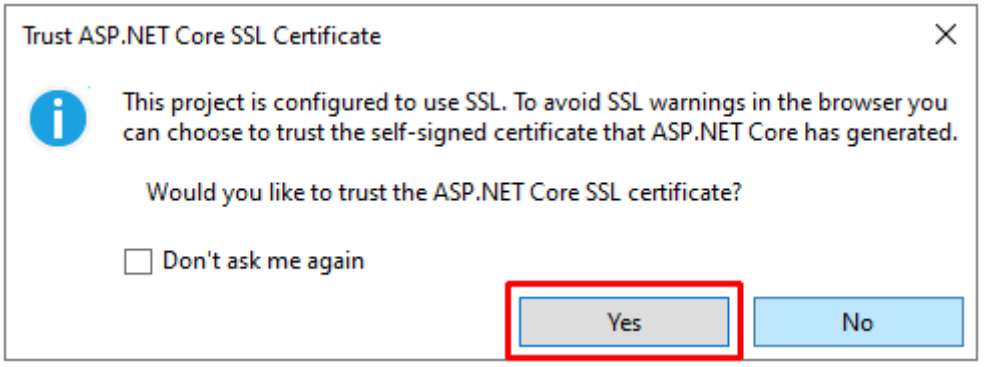
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Select **.NET 8.0 (Long Term Support)**.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **Server**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected
  - Select **Create**.

The Visual Studio instructions in parts of this tutorial series use EF Core commands to add database migrations and update the database. EF Core commands are issued using Visual Studio Connected Services. More information is provided later in this tutorial series.

Run the app

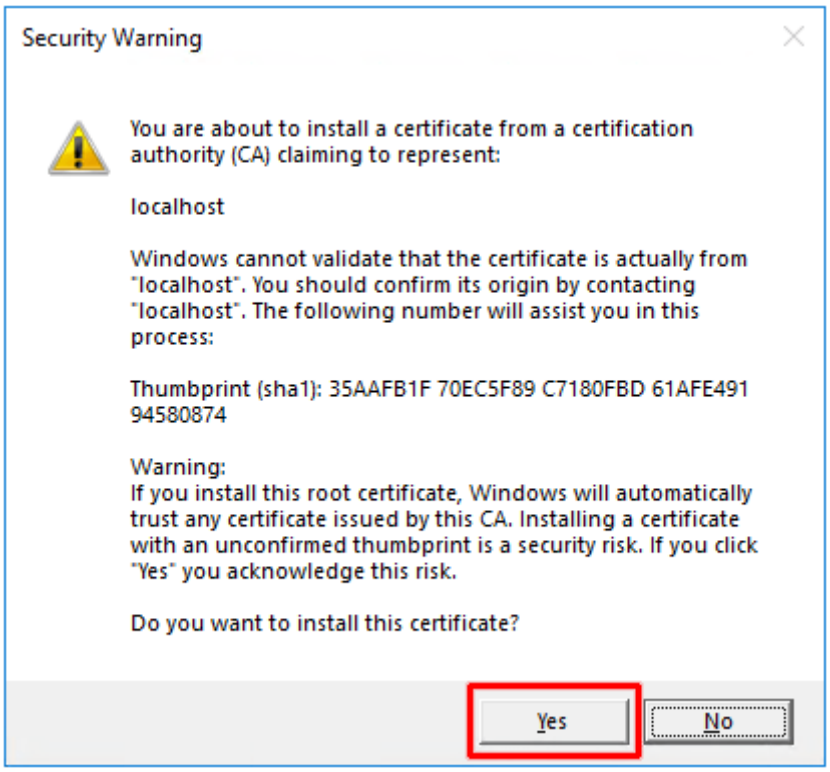
Press F5 on the keyboard to run the app.

Visual Studio displays the following dialog when a project isn't configured to use SSL:



Select **Yes** if you trust the ASP.NET Core SSL certificate.

The following dialog is displayed:



Select **Yes** to acknowledge the risk and install the certificate.

Visual Studio:

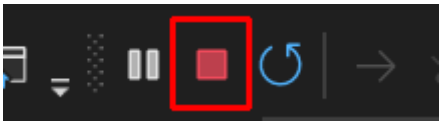
- Compiles and runs the app.
- Launches the default browser at `https://localhost:{PORT}`, which displays the app's UI. The `{PORT}` placeholder is the random port assigned to the app when the app is created. If you need to change the port due to a local port conflict, change the port in the project's `Properties/launchSettings.json` file.

Navigate the pages of the app to confirm that the app is working normally.

Stop the app

Stop the app using either of the following approaches:

- Close the browser window.
- In Visual Studio, either:
  - Use the Stop button in Visual Studio's menu bar:

    
  - Press `Shift`+`F5` on the keyboard.

Examine the project files

The following sections contain an overview of the project's folders and files.

If you're building the app, you don't need to make changes to the project files in the following sections. As you read the descriptions of the folders and files, examine them in the project.

If you're only reading the articles and not building the app, you can refer to the completed sample app in the Blazor samples GitHub repository (dotnet/blazor-samples). Select the latest version folder in the repository. The sample folder for this tutorial's project is named `BlazorWebAppMovies`. The sample app is the *finished version* of the app after following all of the steps of the tutorial series. Code in the sample doesn't always match steps of the tutorial before the end of the series.

`Properties` folder

The `Properties` folder holds development environment configuration in the `launchSettings.json` file.

`wwwroot` folder

The `wwwroot` folder contains static assets, such as image, JavaScript (`.js`), and stylesheet (`.css`) files.

`Components`, `Components/Pages`, and `Components/Layout` folders

These folders contain *Razor components*, often referred to as "components," and supporting files. A component is a self-contained portion of user interface (UI) with optional processing logic. Components can be nested, reused, and shared among projects.

Components are implemented using a combination of C# and HTML markup in Razor component files with the `.razor` file extension.

Typically, components that are nested within other components and not directly reachable ("routable") at a URL are placed in the `Components` folder. Components that are routable via a URL are usually placed in the `Components/Pages` folder.

The `Components/Layout` folder contains the following layout components and stylesheets:

- `MainLayout` component (`MainLayout.razor`): The app's main layout component.
- `MainLayout.razor.css`: Stylesheet for the app's main layout.
- `NavMenu` component (`NavMenu.razor`): Implements sidebar navigation. This component uses several `NavLink` components to render navigation links to other Razor components.
- `NavMenu.razor.css`: Stylesheet for the app's navigation menu.

`Components/_Imports.razor` file

The `_Imports` file (`_Imports.razor`) includes common *Razor directives* to include in the app's Razor components. Razor directives are reserved keywords prefixed with `@` that appear in Razor markup and change the way component markup or component elements are compiled or function.

`Components/App.razor` file

The `App` component (`App.razor`) is the root component of the app that includes:

- HTML markup.
- The `Routes` component.
- The Blazor script (`<script>` tag for `blazor.web.js`).

The root component is the first component that the app loads.

`Components/Routes.razor` file

The `Routes` component (`Routes.razor`) sets up routing for the app.

`appsettings.json` file

The `appsettings.json` file contains configuration data, such as connection strings.

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production

environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

`Program.cs` file

The `Program.cs` file contains code to create the app and configure the request processing pipeline of the app.

The order of the lines in the Blazor Web App project template changes across releases of .NET, so the order of the lines in the `Program.cs` file might not match the order of the lines covered in this section.

A WebApplicationBuilder creates the app with preconfigured defaults:

C#

```
var builder = WebApplication.CreateBuilder(args);
```

Razor component services are added to the app by calling AddRazorComponents, which enables Razor components to render and execute code on the server, and AddInteractiveServerComponents adds services to support rendering Interactive Server components:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

The WebApplication (held by the `app` variable in the following code) is built:

C#

```
var app = builder.Build();
```

Next, the HTTP request pipeline is configured.

In the development environment:

- Exception Handler Middleware (UseExceptionHandler) processes errors and displays a developer exception page during development app runs.
- HTTP Strict Transport Security Protocol (HSTS) Middleware (UseHsts) processes HSTS.

```
C#

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}
```

HTTPS Redirection Middleware (UseHttpsRedirection) enforces the HTTPS protocol by redirecting HTTP requests to HTTPS if an HTTPS port is available:

```
C#

app.UseHttpsRedirection();
```

Antiforgery Middleware (UseAntiforgery) enforces antiforgery protection for form processing:

```
C#

app.UseAntiforgery();
```

Static File Middleware (UseStaticFiles) serves static files, such as images, scripts, and stylesheets from the `wwwroot` folder:

```
C#

app.UseStaticFiles();
```

MapRazorComponents maps components defined in the root `App` component to the given .NET assembly and renders routable components, and AddInteractiveServerRenderMode configures interactive server-side rendering (interactive SSR) support for the app:

```
C#

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

> **Note**
>
> The extension methods AddInteractiveServerComponents on AddRazorComponents and AddInteractiveServerRenderMode on MapRazorComponents make the app capable of adopting interactive SSR, which isn't relevant until the last part of the tutorial series on interactivity. Over the next several articles, the app's components only adopt static SSR.

The app is run by calling Run on the WebApplication (`app`):

```
C#

app.Run();
```

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

When using VS Code or the .NET CLI, this tutorial series adopts insecure HTTP protocol to ease the transition of adopting SSL/HTTPS security for Linux and macOS users. For information on adopting SSL/HTTPS, see Enforce HTTPS in ASP.NET Core.

Next steps

Next: Add and scaffold a model

Build a Blazor movie database app (Part 2 - Add and scaffold a model)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the second part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

In this part of the tutorial series:

- A class is added to the app representing a movie in a database.
- Entity Framework Core (EF Core) services and tooling create a database context and database. EF Core is an object-relational mapper (O/RM) that simplifies data access. You write model classes first, and EF Core creates the database from the model classes, which is called *scaffolding*.
- Additional tooling scaffolds a Razor component-based UI for interacting with the movie records in the database via the database context.

Add a data model

In **Solution Explorer**, right-click the `BlazorWebAppMovies` project and select **Add** > **New Folder**. Name the folder `Models`.

Right-click the `Models` folder. Select **Add** > **Class**. Name the file `Movie.cs`. Use the following contents for the file.

`Models/Movie.cs`:

```
C#

using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace BlazorWebAppMovies.Models;

public class Movie
```

```
{
    public int Id { get; set; }

    public string? Title { get; set; }

    public DateOnly ReleaseDate { get; set; }

    public string? Genre { get; set; }

    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
}
```

The `Movie` class contains:

- A record identifier property (`Id`), which is required by EF Core and the database to track records. In the database, the `Id` property is the database record's primary key.
- Other properties that describe aspects of a movie:
  - Title (`Title`)
  - Release date (`ReleaseDate`)
  - Genre (`Genre`)
  - Price (`Price`)

The question mark on a `string` type indicates that the property is nullable (it can hold a `null` value).

The EF Core database provider selects data types based on the .NET types of the model's properties. The provider also takes into account other metadata provided by System.ComponentModel.DataAnnotations, which are a set of attribute classes placed above a model's property with the following format, where the `{ANNOTATION}` placeholder is the annotation name. You can place multiple annotations on the same line separated by commas inside the brackets, or you can place multiple annotations on separate lines, which is the approach adopted by this tutorial series.

C#

```
[{ANNOTATION}]
```

The `Price` property in the `Movie` class file has two data annotations:

C#

```
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

These annotations specify:

- That the property holds a currency data type.
- The database column is a decimal of 18 digits with two decimal places.

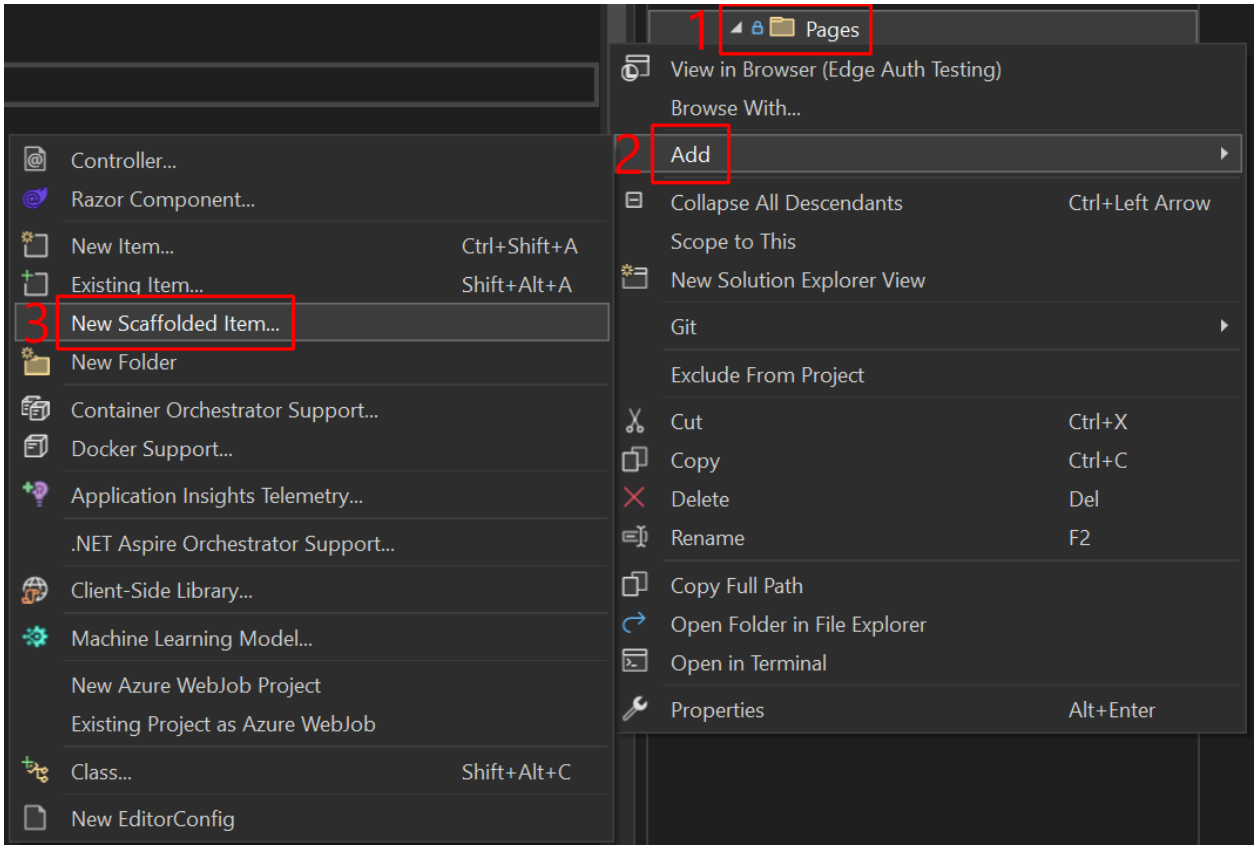
More information on data annotations, including adding data annotations for validation, is covered in a later part of the tutorial series.

Select **Build** > **Build Solution** from the menu bar or press F6 on the keyboard. Confirm in the **Output** panel that the build succeeded.

Scaffold the model

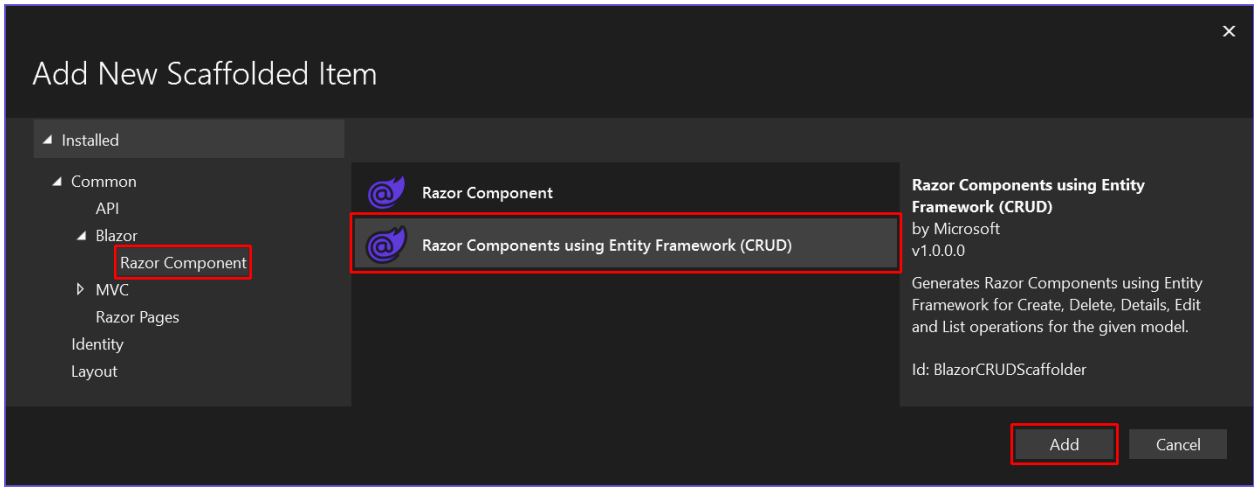
In this section, the `Movie` model is used to *scaffold* a database context and a UI for managing movies in the database. .NET scaffolding is a code generation framework for .NET applications. You use scaffolding to quickly add database and UI code that interacts with data models.

Right-click on the `Components/Pages` folder and select **Add** > **New Scaffolded Item**:



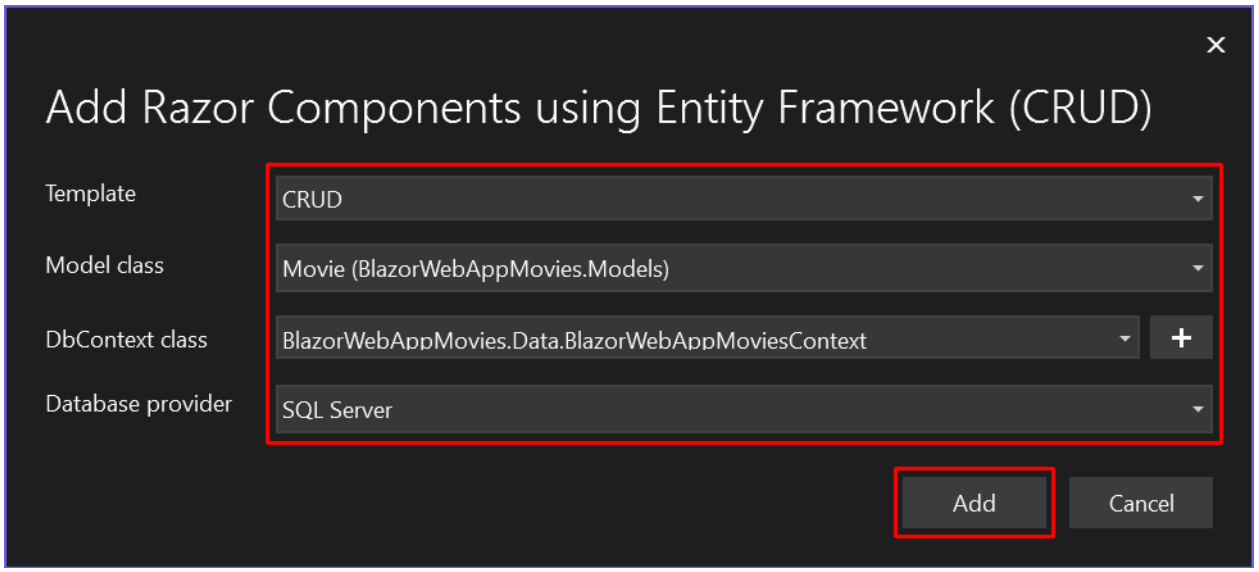
With the **Add New Scaffold Item** dialog open to **Installed** > **Common** > **Blazor** > **Razor Component**, select **Razor Components using Entity Framework (CRUD)**. Select the **Add** button.

CRUD is an acronym for Create, Read, Update, and Delete. The scaffolder produces create, edit, delete, details, and index components for the app.



Complete the **Add Razor Components using Entity Framework (CRUD)** dialog:

- The **Template** dropdown list includes other templates for specifically creating create, edit, delete, details, and list components. This dropdown list comes in handy when you only need to create a specific type of component scaffolded to a model class. Leave the **Template** dropdown list set to **CRUD** to scaffold a full set of components.
- In the **Model class** dropdown list, select **Movie (BlazorWebAppMovies.Models)**.
- For **DbContext class**, select the + (plus sign) button.
- In the **Add Data Context** modal dialog, the class name `BlazorWebAppMovies.Data.BlazorWebAppMoviesContext` is generated. Use the default generated value. Select the **Add** button.
- After the model dialog closes, the **Database provider** dropdown list defaults to **SQL Server**. When you're building apps in the future, you can select the appropriate provider for the database that you're using. The options include SQLite, PostgreSQL, and Azure Cosmos DB. Leave the **Database provider** dropdown list set to **SQL Server**.
- Select **Add**.



The `appsettings.json` file is updated with the connection string used to connect to a local database. In the following example, the `{CONNECTION STRING}` is the connection string automatically generated by the scaffolder:

JSON

```
"ConnectionStrings": {
  "BlazorWebAppMoviesContext": "{CONNECTION STRING}"
}
```

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

Files created and updated by scaffolding

The scaffolding process creates the following component files and movie database context class file:

- `Components/Pages/MoviePages`
  - `Create.razor`: Creates new movie entities.
  - `Delete.razor`: Deletes a movie entity.
  - `Details.razor`: Shows movie entity details.
  - `Edit.razor`: Updates a movie entity.
  - `Index.razor`: Lists movie entities (records) in the database.
- `Data/BlazorWebAppMoviesContext.cs`: Database context file (DbContext).

The component files in the `MoviePages` folder are described in greater detail in the next part of this tutorial. The database context is described later in this article.

ASP.NET Core is built with dependency injection, which is a software design pattern for achieving Inversion of Control (IoC) between classes and their dependencies. Services, such as the EF Core database context, are registered with dependency injection during application startup. These services are injected into Razor components for use by the components.

The QuickGrid component is a Razor component for efficiently displaying data in tabular form. The scaffolder places a `QuickGrid` component in the `Index` component (`Components/Pages/Index.razor`) to display movie entities. Calling AddQuickGridEntityFrameworkAdapter on the service collection adds an EF Core adapter for `QuickGrid` to recognize EF Core-supplied IQueryable<T> instances and to resolve database queries asynchronously for efficiency.

In combination with UseDeveloperExceptionPage, AddDatabaseDeveloperPageExceptionFilter captures database-related exceptions that can be resolved by using Entity Framework migrations. When these exceptions occur, an HTML response is generated with details about possible actions to resolve the issue.

The following code is added to the `Program` file by the scaffolder:

C#

```
builder.Services.AddDbContextFactory<BlazorWebAppMoviesContext>(options =>
    options.UseSqlServer(

builder.Configuration.GetConnectionString("BlazorWebAppMoviesContext") ??
        throw new InvalidOperationException(
            "Connection string 'BlazorWebAppMoviesContext' not found.")));

builder.Services.AddQuickGridEntityFrameworkAdapter();

builder.Services.AddDatabaseDeveloperPageExceptionFilter();
```

Create the initial database schema using EF Core's migration feature

The migrations feature in EF Core:

- Creates the initial database schema.
- Incrementally updates the database schema to keep it synchronized with the app's data model. Existing data in the database is preserved.

EF Core adopts the *code-first* approach for database design and maintenance:

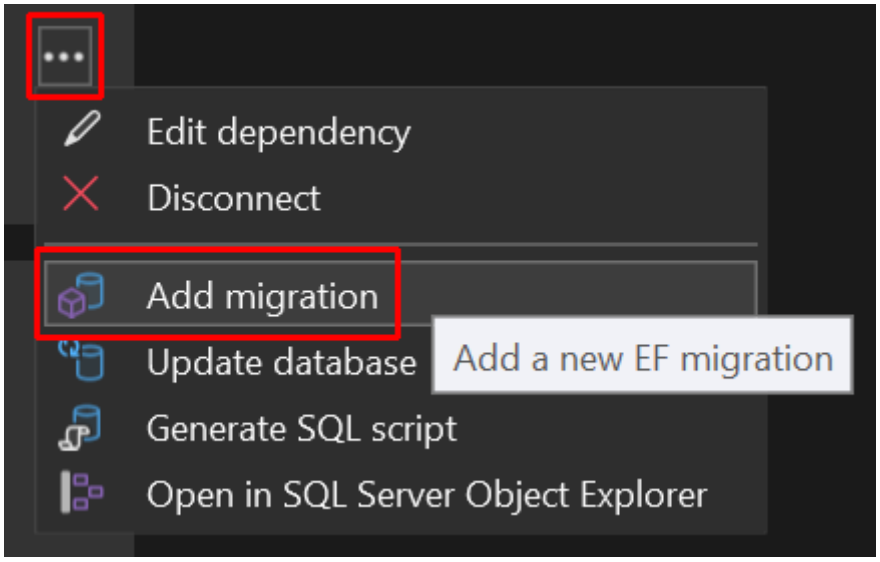
- Entity classes are created and updated first in the app.
- The database is created and updated from the app's entity classes.

This is the reverse procedure of *database-first* approaches, where the database is designed, built, and updated first. Adopting EF Core's code-first approach speeds up the process of app development because most of the difficult and time-consuming database creation and management procedures are handled transparently by the EF Core tooling, so you can focus on app development.

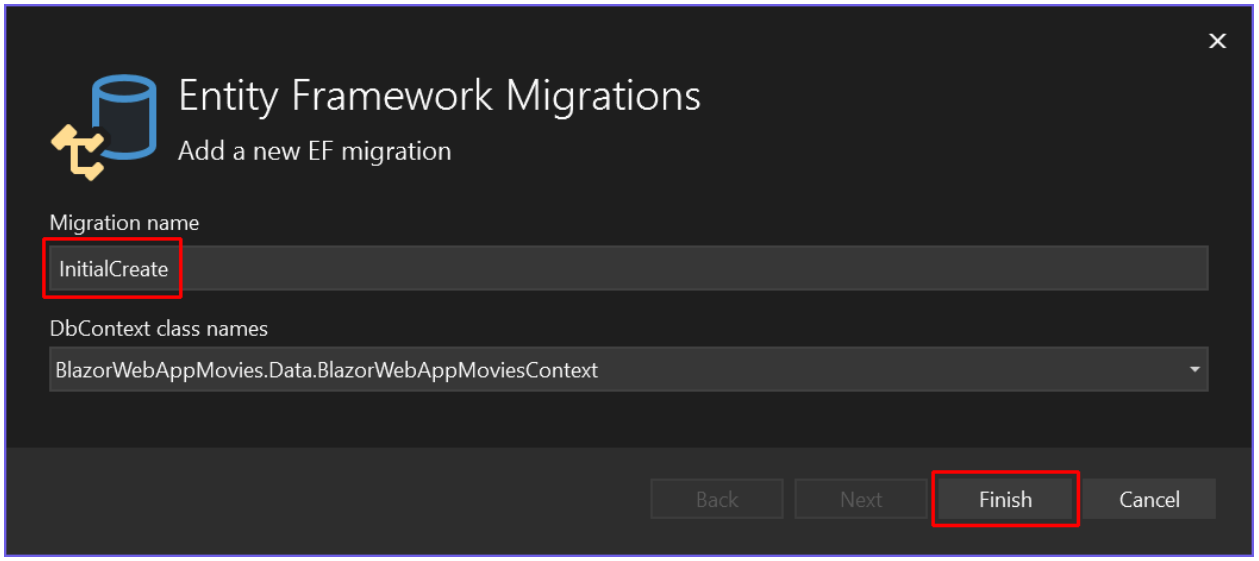
In this section, Visual Studio Connected Services are used to issue EF Core commands that:

- Add an initial migration.
- Update the database using the initial migration.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**:



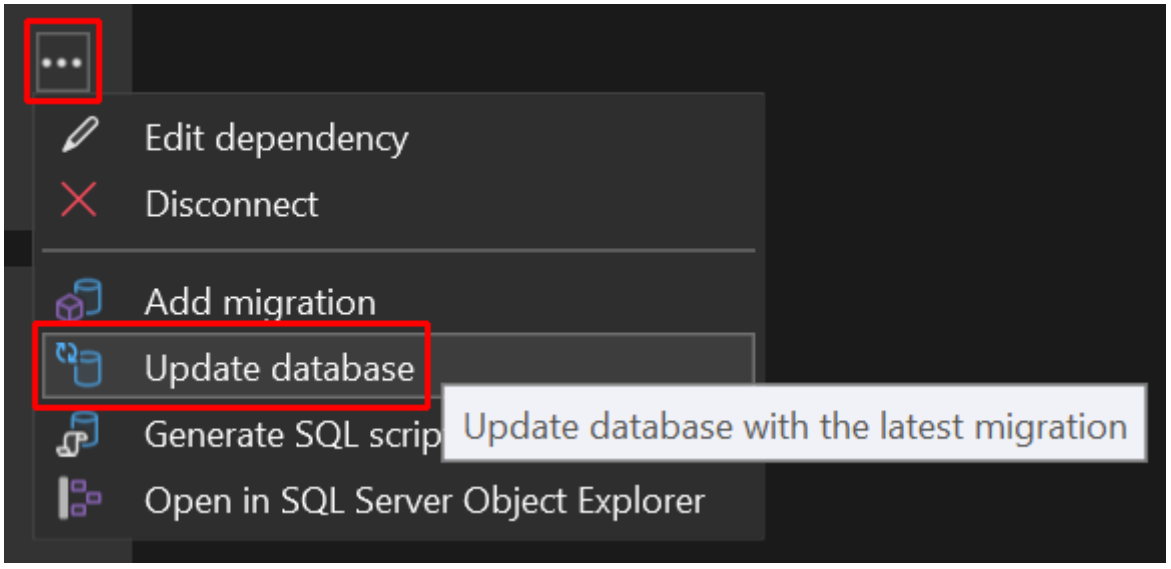
Give the migration a **Migration name** of `InitialCreate`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration:



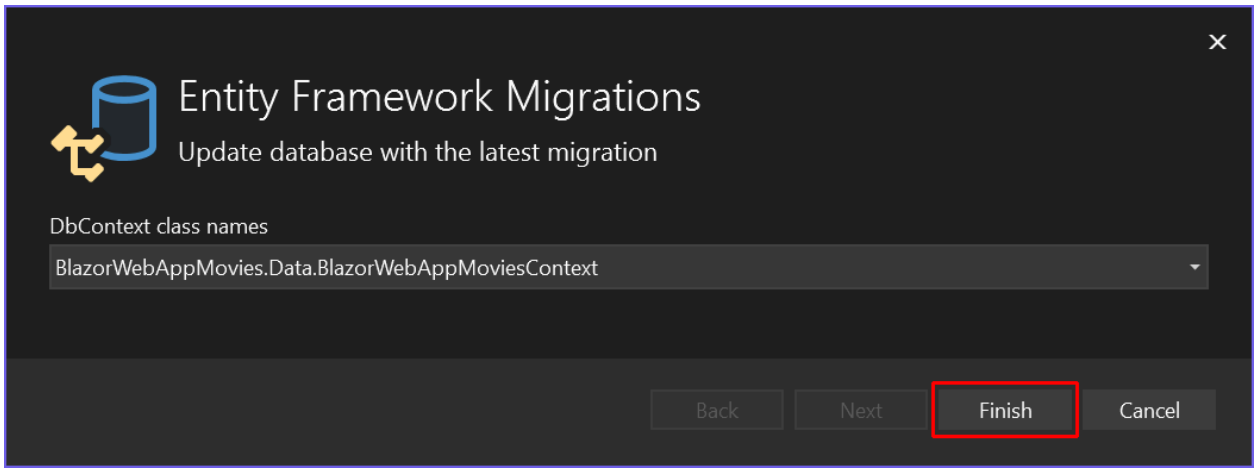
Select the **Close** button after the operation finishes.

Adding a migration generates code to create the initial database schema. The schema is based on the model specified in DbContext.

Select the ellipsis (`...`) again followed by the **Update database** command:



The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button:



The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_InitialCreate.cs` file, which creates the database. The `{TIME STAMP}` placeholder is a time stamp.

Select the **Close** button after the operation finishes.

The database context `BlazorWebAppMoviesContext` (`Data/BlazorWebAppMoviesContext.cs`):

- Derives from Microsoft.EntityFrameworkCore.DbContext.
- Specifies which entities are included in the data model.
- Coordinates EF Core functionality, such as CRUD operations, for the `Movie` model.
- Contains a DbSet<TEntity> property for the `Movie` entity set. In EF terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the connection string is read from the `appsettings.json` file.

> **⚠ Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret

Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

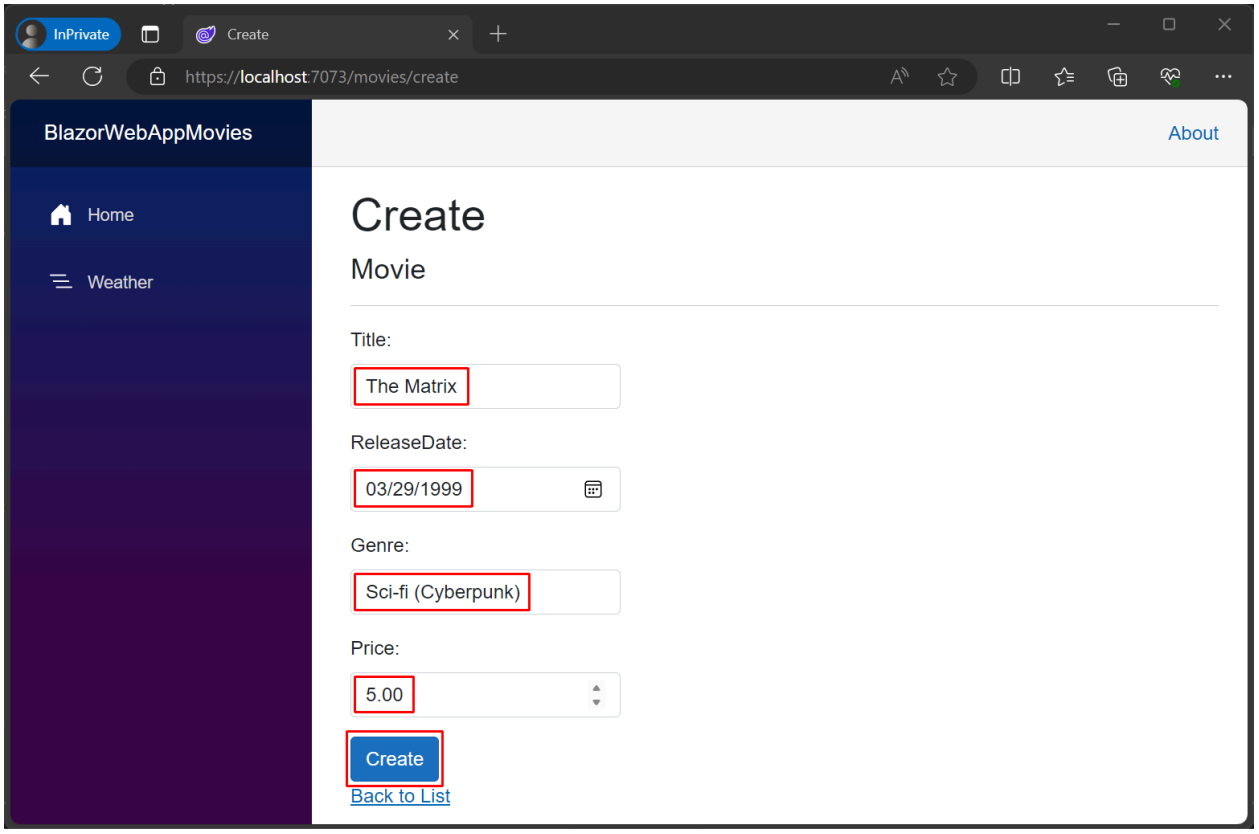
Test the app

Run the app.

Add `/movies` to the URL in the browser's address bar to navigate to the movies `Index` page.

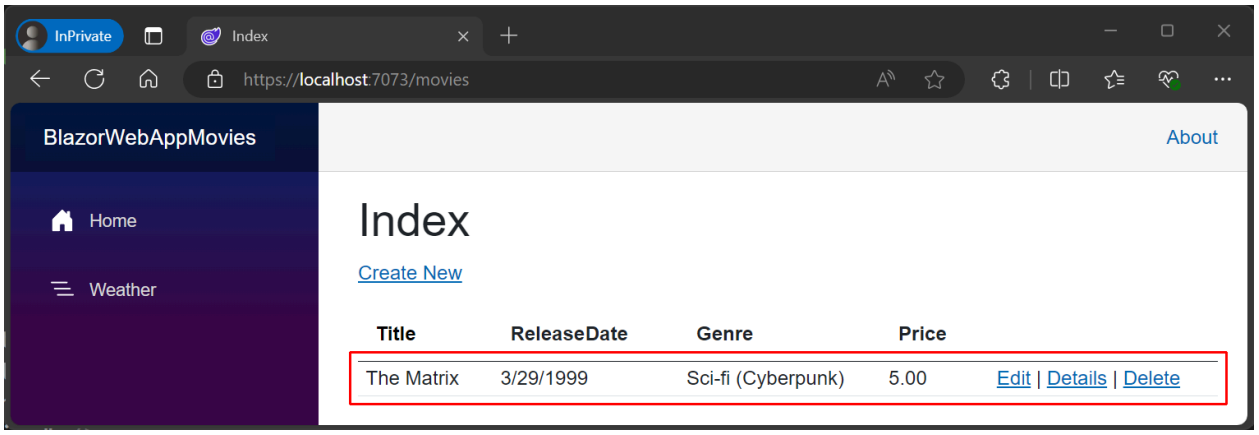
After the `Index` page loads, select the **Create New** link.

Add a movie to the database. In the following example, the classic sci-fi movie *The Matrix* (©1999 Warner Bros. Entertainment Inc.) is added as the first movie entry. Selecting the **Create** button adds the movie to the database:



When you select the **Create** button, the movie data is posted to the server and saved in the database.

When the app returns to the `Index` page, the added entity appears:



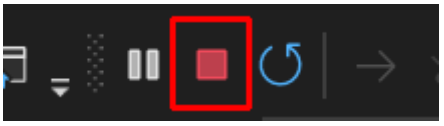
Open the `Edit` page. Edit the movie's record and save the changes.

Examine the `Delete` page, but don't delete *The Matrix* movie from the database. The presence of this movie record is valuable in the next step of the tutorial where rendered HTML is studied and some enhancements are made to the data displayed. If you already deleted the movie, add the movie to the database again using the `Create` page before proceeding to the next part of the tutorial series.

Stop the app

Stop the app using either of the following approaches:

- Close the browser window.
- In Visual Studio, either:
  - Use the Stop button in Visual Studio's menu bar:

  - Press `Shift`+`F5` on the keyboard.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

EF Core documentation:

- Entity Framework Core
- Entity Framework Core tools reference - .NET Core CLI
- Data Types
- SaveChangesAsync: The API document includes basic information on how entities are saved and change detection.
- Environment-based Startup class and methods
- dotnet aspnet-codegenerator
- Dependency injection in ASP.NET Core
- ASP.NET Core Blazor QuickGrid component

Legal

The Matrix is a copyright of Warner Bros. Entertainment Inc. .

Next steps

Previous: Create a Blazor Web App | Next: Learn about Razor components

Build a Blazor movie database app (Part 3 - Learn about Razor components)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the third part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series examines the Razor components in the project that were scaffolded into the app. Improvements are made for the display of movie data.

Razor components

Blazor apps are based on *Razor components*, often referred to as just *components*. A *component* is an element of UI, such as a page, dialog, or data entry form. Components are .NET C# classes built into .NET assemblies.

Razor refers to how components are usually written in the form of a Razor markup page (`.razor` file extension) for client-side UI logic and composition. Razor is a syntax for combining HTML markup with C# code designed for developer productivity.

Although developers and online resources use the term "Blazor components," the documentation uses the formal name "Razor components" (or just "components").

The anatomy of a Razor component has the following general pattern:

- At the top of the component definition (`.razor` file), various Razor directives specify how the component markup is compiled or functions.
- Next, Razor markup specifies how HTML is rendered, which includes ordinary HTML elements.
- Finally, an `@code` block contains C# code to define members for the component class, including component parameters and event handlers.

Consider the following `Welcome` component (`Welcome.razor`):

razor

```razor
@page "/welcome"

<PageTitle>Welcome!</PageTitle>

<h1>Welcome to Blazor!</h1>

<p>@welcomeMessage</p>

@code {
    private string welcomeMessage = "We ❤ Blazor!";
}
```

The first line represents an important Razor construct in Razor components, a *Razor directive*. A Razor directive is a reserved keyword prefixed with `@` that appears in Razor markup that changes the way component markup is compiled or functions. The `@page` Razor directive specifies the route template for the component. This component is reached in a browser at the relative URL `/welcome`. By convention, most of a component's directives are placed at the top of the component definition file.

The PageTitle component is a component built into the framework that specifies a page title.

"`Welcome to Blazor!`" is the first rendered body markup of the component per the content of the H1 heading element (`<h1>`).

Next, a welcome message is displayed using Razor syntax by prefixing the at symbol (`@`) to a C# variable (`welcomeMessage`).

The `@code` block contains the C# code of the component. `welcomeMessage` is a private string initialized with a value.

In the following sections of this article:

- Three components for webpage navigation and layout are described, the `NavMenu`, NavLink, and `MainLayout` components.
- The components created by scaffolding for CRUD operations on movie database entities are discussed.

`NavMenu` component for navigation

The `NavMenu` component (`Components/Layout/NavMenu.razor`) implements sidebar navigation using NavLink components, which render navigation links to other Razor

components.

A NavLink component behaves like an `<a>` element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed. NavLinkMatch.All assigned to the Match parameter configures the component to display an active CSS class when it matches the entire current URL.

The NavLink component built into the Blazor framework for any Blazor app to use, while the `NavMenu` component is only part of Blazor project templates.

`Components/Layout/NavMenu.razor`:

razor

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">BlazorWebAppMovies</a>
    </div>
</div>

<input type="checkbox" title="Navigation menu" class="navbar-toggler" />

<div class="nav-scrollable" onclick="document.querySelector('.navbar-
toggler').click()">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="bi bi-house-door-fill-nav-menu" aria-
hidden="true"></span> Home
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="weather">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true">
</span> Weather
            </NavLink>
        </div>
    </nav>
</div>
```

Notice in the `NavMenu` component's first `<div>` element the brand link text (`<a>` element content). Change the brand from `BlazorWebAppMovies` to `Sci-fi Movies`:

diff

```
- <a class="navbar-brand" href="">BlazorWebAppMovies</a>
```

```
+ <a class="navbar-brand" href="">Sci-fi Movies</a>
```

To allow users to reach the movies `Index` page, add a navigation menu entry to the `NavMenu` component. Immediately after the markup (`<div>`) for the `Weather` component's `NavLink`, add the following markup:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="movies">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Movies
    </NavLink>
</div>
```

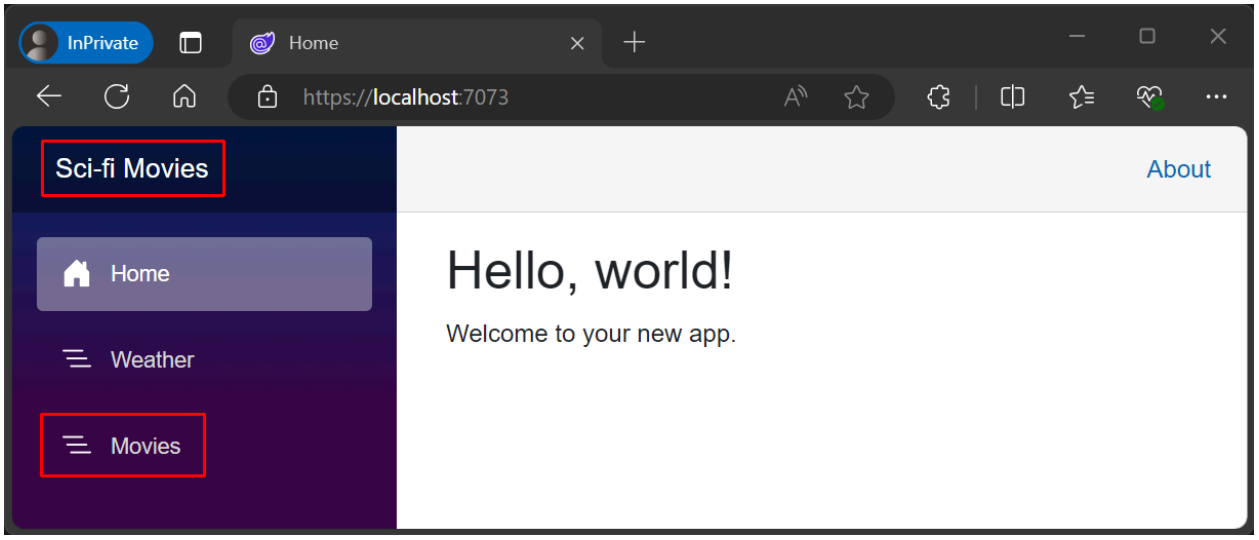
The final `NavMenu` component after making the preceding changes:

razor

```
<div class="top-row ps-3 navbar navbar-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href="">Sci-fi Movies</a>
    </div>
</div>

<input type="checkbox" title="Navigation menu" class="navbar-toggler" />

<div class="nav-scrollable" onclick="document.querySelector('.navbar-
toggler').click()">
    <nav class="nav flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="bi bi-house-door-fill-nav-menu" aria-
hidden="true"></span> Home
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="weather">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true">
</span> Weather
            </NavLink>
        </div>

        <div class="nav-item px-3">
            <NavLink class="nav-link" href="movies">
                <span class="bi bi-list-nested-nav-menu" aria-hidden="true">
</span> Movies
            </NavLink>
        </div>
```

```
    </nav>
</div>
```

Run the app to see the updated brand at the top of the sidebar navigation and a link to reach the movies page (**Movies**):



Stop the app by closing the browser's window.

`MainLayout` component for layout

The `MainLayout` component is the app's default layout. The `MainLayout` component inherits LayoutComponentBase, which is a base class for components that represent a layout. The app's components that use the layout are rendered where the Body (`@Body`) appears in the markup.

`Components/Layout/MainLayout.razor`:

razor

```
@inherits LayoutComponentBase

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <a href="https://learn.microsoft.com/aspnet/core/"
target="_blank">About</a>
        </div>

        <article class="content px-4">
            @Body
```

```
        </article>
    </main>
</div>
```

The `MainLayout` component adopts the following additional specifications:

- The `NavMenu` component is rendered in the sidebar. Notice that you only need to place an HTML tag with the component name to render a component at that location in Razor markup. This allows you to nest components within each other and within any HTML layout that you implement.
- The `<main>` element's content includes:
  - An **About** link that sends the user to the ASP.NET Core documentation landing page.
  - An `<article>` element with the Body (`@Body`) parameter, where components that use the layout are rendered.
  - The error UI (`<div id="blazor-error-ui" ...>`), where a notice about an unhandled error is displayed.

The default layout (`MainLayout` component) is specified in the `Routes` component (`Components/Pages/Routes.razor`):

razor

```
<RouteView RouteData="routeData" DefaultLayout="typeof(Layout.MainLayout)"
/>
```

Individual components are free to set their own non-default layout, and a layout can be applied to whole folder of components via an `_Imports.razor` file in the same folder. These features are covered in detail in the Blazor documentation.

Create, Read, Update, Delete (CRUD) components

The following sections explain the composition of the movie CRUD components and how they work.

`Index` component

Open the `Index` component definition file (`Components/Pages/Movies/Index.razor`) and examine the Razor directives at the top of the file.

The `@page` directive's route template indicates the URL for the page is `/movies`.

`@using` directives appear to access the following API:

- Microsoft.EntityFrameworkCore
- Microsoft.AspNetCore.Components.QuickGrid
- `BlazorWebAppMovies.Models`
- `BlazorWebAppMovies.Data`

The database context factory (`IDbContextFactory<T>`, where the type (`T`) is a `BlazorWebAppMoviesContext`) is injected into the component with the `@inject` directive. The factory approach requires that a database context be disposed, so the component implements the IAsyncDisposable interface with the `@implements` directive.

The page title is set via the Blazor framework's PageTitle component, and an H1 section heading is the first rendered element:

razor

```razor
<PageTitle>Index</PageTitle>

<h1>Index</h1>
```

A link is rendered to navigate to the `Create` page at `/movies/create`:

razor

```razor
<p>
    <a href="movies/create">Create New</a>
</p>
```

The QuickGrid component displays movie entities. The item provider is a `DbSet<Movie>` obtained from the created database context (CreateDbContext) of the injected database context factory (`DbFactory`). For each movie entity, the component displays the movie's title, release date, genre, and price. A column also holds links to edit, see details, and delete each movie entity.

razor

```razor
<QuickGrid Class="table" Items="context.Movie">
    <PropertyColumn Property="movie => movie.Title" />
    <PropertyColumn Property="movie => movie.ReleaseDate" />
    <PropertyColumn Property="movie => movie.Genre" />
    <PropertyColumn Property="movie => movie.Price" />
```

```
    <TemplateColumn Context="movie">
        <a href="@($"movies/edit?id={movie.Id}")">Edit</a> |
        <a href="@($"movies/details?id={movie.Id}")">Details</a> |
        <a href="@($"movies/delete?id={movie.Id}")">Delete</a>
    </TemplateColumn>
</QuickGrid>

@code {
    private BlazorWebAppMoviesContext context = default!;

    protected override void OnInitialized()
    {
        context = DbFactory.CreateDbContext();
    }

    public async ValueTask DisposeAsync() => await context.DisposeAsync();
}
```

In the code block (`@code`):

- The `context` field holds the database context, typed as a `BlazorWebAppMoviesContext`.
- The `OnInitialized` lifecycle method assigns a created database context (CreateDbContext) from the injected factory (`DbFactory`) to the `context` variable.
- The asynchronous `DisposeAsync` method disposes of the database context when the component is disposed.

Notice how the context (`Context`) parameter of the TemplateColumn<TGridItem> specifies a parameter name (`movie`) for the context instance of the column. Specifying a name for the context instance makes the markup more readable (the default name for the context is simply `context`). `Movie` class properties are read from the context instance. For example, the movie identifier (`Id`) is available in `movie.Id`.

The at symbol (`@`) with parentheses (`@(...)`), which is called an *explicit Razor expression*, allows the `href` of each link to include the movie entity's `Id` property in the link query string as an *interpolated string* (`$...{...}...`). For a movie identifier (`Id`) of 7, the string value provided to the `href` to edit that movie is `movies/edit?id=7`. When the link is followed, the `id` field is read from the query string by the `Edit` component to load the movie.

For the movie example from the last part of the tutorial series, *The Matrix*©, the QuickGrid component renders the following HTML markup (some elements and attributes aren't present to simplify display). See how the explicit Razor expressions and interpolated strings produced the `href` values for the links to other pages. The movie's identifier in the database happens to be `3` for this example, so the `id` is `3` in the query

strings for the `Edit`, `Details`, and `Delete` pages. You may see a different value when you run the app.

HTML

```
<table>
    <thead>
        <tr>
            <th>Title</th>
            <th>ReleaseDate</th>
            <th>Genre</th>
            <th>Price</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>The Matrix</td>
            <td>3/29/1999</td>
            <td>Sci-fi (Cyberpunk)</td>
            <td>5.00</td>
            <td>
                <a href="movies/edit?id=3">Edit</a> |
                <a href="movies/details?id=3">Details</a> |
                <a href="movies/delete?id=3">Delete</a>
            </td>
        </tr>
    </tbody>
</table>
```

The column names are taken from the `Movie` model properties, so the release date doesn't have a space between the words. Add a Title to the PropertyColumn<TGridItem,TProp> with a value that includes a space between the words:

diff

```
- <PropertyColumn Property="movie => movie.ReleaseDate" />
+ <PropertyColumn Property="movie => movie.ReleaseDate" Title="Release Date"
/>
```

Run the app to see that the column displays two words for the release date.

Stop the app by closing the browser's window.

`Details` component

Open the `Details` component definition file (`Components/Pages/Movies/Details.razor`).

The `@page` directive at the top of the file indicates the relative URL for the page is `/movies/details`. As before, the database context is injected, and namespaces are provided to access API (`BlazorWebAppMovies.Models` and `Microsoft.EntityFrameworkCore`). The `Details` component also injects the app's NavigationManager, which is used for a variety of navigation-related operations in components.

razor

```razor
@page "/movies/details"
@using Microsoft.EntityFrameworkCore
@using BlazorWebAppMovies.Models
@inject IDbContextFactory<BlazorWebAppMovies.Data.BlazorWebAppMoviesContext>
DbFactory
@inject NavigationManager NavigationManager
```

Details for a movie entity are only shown if the movie, located by its identifier (`Id`) from the query string, has been loaded for display. The presence of the movie in `movie` is checked with an `@if` Razor statement:

razor

```razor
@if (movie is null)
{
    <p><em>Loading...</em></p>
}
```

When the movie is loaded, it's displayed as a description list (MDN documentation) along with two links:

- The first link provides the user an opportunity to edit the entity.
- The second link allows the user to return to the movies `Index` page.

CSS classes aren't shown in the following example to simplify the Razor markup for display:

razor

```razor
<dl>
    <dt>Title</dt>
    <dd>@movie.Title</dd>
    <dt>ReleaseDate</dt>
    <dd>@movie.ReleaseDate</dd>
    <dt>Genre</dt>
    <dd>@movie.Genre</dd>
    <dt>Price</dt>
```

```
    <dd>@movie.Price</dd>
</dl>
<div>
    <a href="@($"/movies/edit?id={movie.Id}")">Edit</a> |
    <a href="@($"/movies")">Back to List</a>
</div>
</div>
```

Add a space to the content of the description term element (`<dt>`) for the movie's release date to separate the words:

diff

```
- <dt class="col-sm-2">ReleaseDate</dt>
+ <dt class="col-sm-2">Release Date</dt>
```

Examine the C# of the component's `@code` block:

C#

```
private Movie? movie;

[SupplyParameterFromQuery]
private int Id { get; set; }

protected override async Task OnInitializedAsync()
{
    using var context = DbFactory.CreateDbContext();
    movie = await context.Movie.FirstOrDefaultAsync(m => m.Id == Id);

    if (movie is null)
    {
        NavigationManager.NavigateTo("notfound");
    }
}
```

The `movie` variable is a private field of type `Movie`, which is a null-reference type (`?`), meaning that `movie` might be set to `null`.

The `Id` is a *component parameter* supplied from the component's query string due to the presence of the [SupplyParameterFromQuery] attribute. If the identifier is missing, `Id` defaults to zero (`0`).

`OnInitializedAsync` is the first component lifecycle method that we've seen. This method is executed when the component loads. FirstOrDefaultAsync is called on the database set (`DbSet<Movie>`) to retrieve the movie entity with and `Id` equal to the `Id`

parameter that was set by the query string. If `movie` is `null`, NavigationManager.NavigateTo is used to navigate to a `notfound` endpoint.

There isn't an actual `notfound` endpoint (Razor component) in the app. When adopting server-side rendering (SSR), Blazor doesn't have a mechanism to return a 404 (Not Found) status code. As a temporary workaround, a 404 is generated by navigating to a non-existent endpoint. This scaffolded code is for your further implementation of a suitable result when not finding an entity. For example, you could have the component direct the user to a page where they can file an inquiry with your support team, or you could remove the injected NavigationManager and NavigationManager.NavigateTo code and replace it with Razor markup and code that displays a message to the user that the entity wasn't found.

`Create` component

Open the `Create` component definition file (`Components/Pages/Movies/Create.razor`).

The component uses a built-in component called an EditForm, which renders a form for user input and includes validation features.

CSS classes aren't present in the following example to simplify the display:

razor

```
<EditForm method="post" Model="Movie" OnValidSubmit="AddMovie"
FormName="create" Enhance>
    <DataAnnotationsValidator />
    <ValidationSummary role="alert" />
    <div>
        <label for="title">Title:</label>
        <InputText id="title" @bind-Value="Movie.Title" />
        <ValidationMessage For="() => Movie.Title" />
    </div>
    <div>
        <label for="releasedate">ReleaseDate:</label>
        <InputDate id="releasedate" @bind-Value="Movie.ReleaseDate" />
        <ValidationMessage For="() => Movie.ReleaseDate" />
    </div>
    <div>
        <label for="genre">Genre:</label>
        <InputText id="genre" @bind-Value="Movie.Genre" />
        <ValidationMessage For="() => Movie.Genre" />
    </div>
    <div>
        <label for="price">Price:</label>
        <InputNumber id="price" @bind-Value="Movie.Price" />
        <ValidationMessage For="() => Movie.Price" />
    </div>
```

```
    <button type="submit">Create</button>
</EditForm>
```

Add a space to the content of the label element (`<label>`) for the movie's release date to separate the words:

diff

```
- <label for="releasedate" class="form-label">ReleaseDate:</label>
+ <label for="releasedate" class="form-label">Release Date:</label>
```

The Model parameter is assigned the model, in this case `Movie`. OnValidSubmit specifies a method to invoke (`AddMovie`) when the form is submitted and the data is valid. By convention, every form should assign a FormName to prevent form collisions when multiple forms are present on a page. The Enhance flag activates a Blazor feature for server-side rendering (SSR) that submits the form without performing a full-page reload.

For validation:

- The DataAnnotationsValidator adds data annotations validation support, which is covered later in this tutorial series.
- The ValidationSummary component displays a list of validation messages.
- The ValidationMessage<TValue> components hold validation messages for the form's fields.

Blazor includes several form element components to assist you with creating forms, including EditForm and various input components, such as InputText, InputDate<TValue>, and InputNumber<TValue>. Each input component is bound to a model property with `@bind-Value` Razor syntax, where `Value` is a property in each input component.

In the component's `@code` block, C# code includes a `Movie` component parameter tied to the form via the [SupplyParameterFromForm] attribute.

The `AddMovie` method:

- Is called when the form is submitted.
- Adds the movie data bound to the form's model (`Movie`) if form validation passes.
- SaveChangesAsync is called on the database context to save the movie.
- NavigationManager is used to return the user to the movies `Index` page.

C#

```
@code {
    [SupplyParameterFromForm]
    private Movie Movie { get; set; } = new();

    private async Task AddMovie()
    {
        using var context = DbFactory.CreateDbContext();
        context.Movie.Add(Movie);
        await context.SaveChangesAsync();
        NavigationManager.NavigateTo("/movies");
    }
}
```

> ⚠ **Warning**
>
> Although it isn't a concern for the app in this tutorial, binding form data to entity data models can be susceptible to overposting attacks. Additional information on this subject appears later in this article.

`Delete` component

Open the `Delete` component definition file (`Components/Pages/Movies/Delete.razor`).

Add a space to the content of the description term element (`<dt>`) for the movie's release date to separate the words:

diff

```
- <dt class="col-sm-2">ReleaseDate</dt>
+ <dt class="col-sm-2">Release Date</dt>
```

Examine the Razor markup for the submit button of the EditForm (CSS class removed for simplicity):

razor

```
<button type="submit" disabled="@(movie is null)">Delete</button>
```

The **Delete** button sets its disabled HTML attribute (MDN documentation) based on the presence of the movie (not `null`) using an explicit Razor expression (`@(...)`).

In the C# code of the `@code` block, the `DeleteMovie` method removes the movie, saves the changes to the database, and navigates the user to the movies `Index` page. The

exclamation point on the movie field (`movie!`) is the null-forgiving operator (C# Language Reference), which suppresses nullable warnings for `movie`.

C#

```
private async Task DeleteMovie()
{
    using var context = DbFactory.CreateDbContext();
    context.Movie.Remove(movie!);
    await context.SaveChangesAsync();
    NavigationManager.NavigateTo("/movies");
}
```

`Edit` component

Open the `Edit` component definition file (`Components/Pages/Movies/Edit.razor`).

Add a space to the content of the label element (`<label>`) for the movie's release date to separate the words:

diff

```
- <label for="releasedate" class="form-label">ReleaseDate:</label>
+ <label for="releasedate" class="form-label">Release Date:</label>
```

The component uses an EditForm similar to the `Create` component.

The movie entity's identifier `Id` is stored in a hidden field of the form:

razor

```
<input type="hidden" name="Movie.Id" value="@Movie.Id" />
```

Examine the C# code of the `@code` block:

C#

```
private async Task UpdateMovie()
{
    using var context = DbFactory.CreateDbContext();
    context.Attach(Movie!).State = EntityState.Modified;

    try
    {
        await context.SaveChangesAsync();
    }
```

```
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie!.Id))
        {
            NavigationManager.NavigateTo("notfound");
        }
        else
        {
            throw;
        }
    }

    NavigationManager.NavigateTo("/movies");
}

private bool MovieExists(int id)
{
    using var context = DbFactory.CreateDbContext();
    return context.Movie.Any(e => e.Id == id);
}
```

The movie entity's EntityState is set to Modified, which signifies that the entity is tracked by the context, exists in the database, and that some or all of its property values are modified.

If there's a concurrency exception and the movie entity no longer exists at the time that changes are saved, the component redirects to the non-existent endpoint (`notfound`), which results in returning a 404 (Not Found) status code. You could change this code to notify the user that the movie no longer exists in the database or create a dedicated *Not Found* component and navigate the user to that endpoint. If the movie exists and a concurrency exception is thrown, for example when another user has already modified the entity, the exception is rethrown by the component with the throw statement (C# Language Reference). Additional guidance on handling concurrency with EF Core in Blazor apps is provided by the Blazor documentation.

> ⚠ **Warning**
>
> Although it isn't a concern for the app in this tutorial, binding form data to entity data models can be susceptible to overposting attacks. Additional information on this subject appears in the next section.

Mitigate overposting attacks

Statically-rendered server-side forms, such as those in the `Create` and `Edit` components, can be vulnerable to an *overposting* attack, also known as a *mass*

assignment attack. An overposting attack occurs when a malicious user issues an HTML form POST to the server that processes data for properties that aren't part of the rendered form and that the developer doesn't wish to allow users to modify. The term "overposting" literally means that the malicious user has *over*-POSTed with the form.

In the example `Create` and `Edit` components of this tutorial, the `Movie` model doesn't include restricted properties for create and update operations, so overposting isn't a concern. However, it's important to keep overposting in mind when working with static SSR-based Blazor forms that you create and modify in the future.

To mitigate overposting, we recommend using a separate view model/data transfer object (DTO) for the form and database with create (insert) and update operations. When the form is submitted, only properties of the view model/DTO are used by the component and C# code to modify the database. Any extra data included by a malicious user is discarded, so the malicious user is prevented from conducting an overposting attack.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- NavLink component
- ASP.NET Core Blazor layouts
- Razor directives (Razor syntax article) / Razor directives (Blazor documentation)
- ASP.NET Core Blazor QuickGrid component
- ASP.NET Core Blazor forms overview
- Concurrency with EF Core in Blazor apps
- ASP.NET Core Blazor globalization and localization: Explains how to render globalized and localized content to users in different cultures and languages, including how to accept comma number separators.

Next steps

Previous: Add and scaffold a model
Next: Work with a database

Build a Blazor movie database app (Part 4 - Work with a database)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the fourth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series focuses on the database context and directly working with the database's schema and data. Seeding the database with data is also covered.

Secure authentication flow required for production apps

This tutorial uses a local database that doesn't require user authentication. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production Blazor Web Apps, see the following resources:

- ASP.NET Core Blazor authentication and authorization
- ASP.NET Core Blazor authentication and authorization and the following articles in the *Server* security node
- Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)
- Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Database context

The database context, `BlazorWebAppMoviesContext`, connects to the database and maps model objects to database records. The database context was created in the second part of this series. The scaffolded database context code appears in the `Program` file:

C#

```
builder.Services.AddDbContextFactory<BlazorWebAppMoviesContext>(options =>
    options.UseSqlServer(

builder.Configuration.GetConnectionString("BlazorWebAppMoviesContext") ??
        throw new InvalidOperationException(
            "Connection string 'BlazorWebAppMoviesContext' not found.")));
```

AddDbContextFactory registers a factory for the given context as a service in the app's service collection.

UseSqlServer or UseSqlite configures the context to connect to either a Microsoft SQL Server or SQLite database. Other providers are available to connect to additional types of databases.

GetConnectionString uses the ASP.NET Core Configuration system to read the `ConnectionStrings` key for the connection string name provided, which in the preceding example is `BlazorWebAppMoviesContext`.

For local development, configuration obtains the database connection string from the app settings file (`appsettings.json`). The `{CONNECTION STRING}` placeholder in the following example is the connection string:

JSON

```
"ConnectionStrings": {
  "BlazorWebAppMoviesContext": "{CONNECTION STRING}"
}
```

The following is an example connection string:

> Server=(localdb)\mssqllocaldb;Database=BlazorWebAppMoviesContext-00001111-aaaa-2222-bbbb-3333cccc4444;Trusted_Connection=True;MultipleActiveResultSets=true

When the app is deployed to a test/staging or production server, securely store the connection string outside of the project's configuration files.

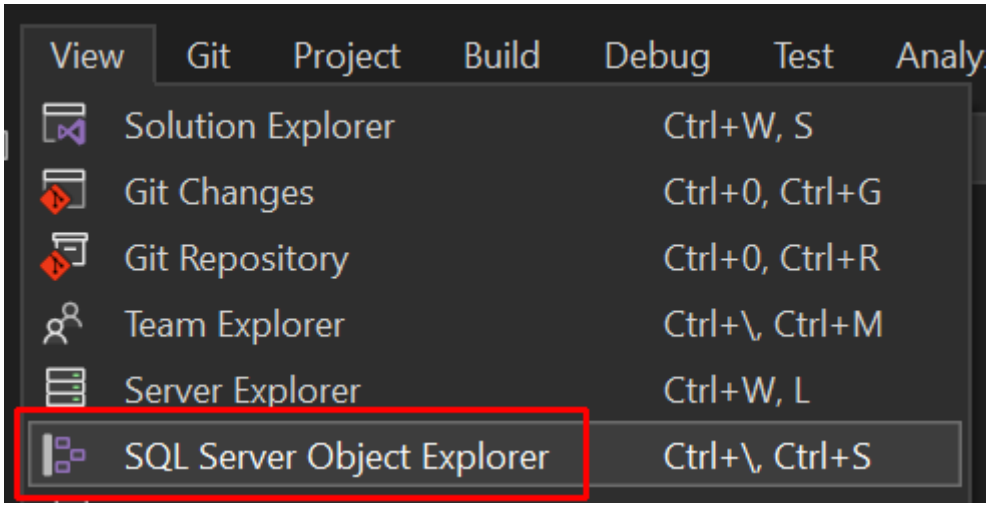
> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

Database technology

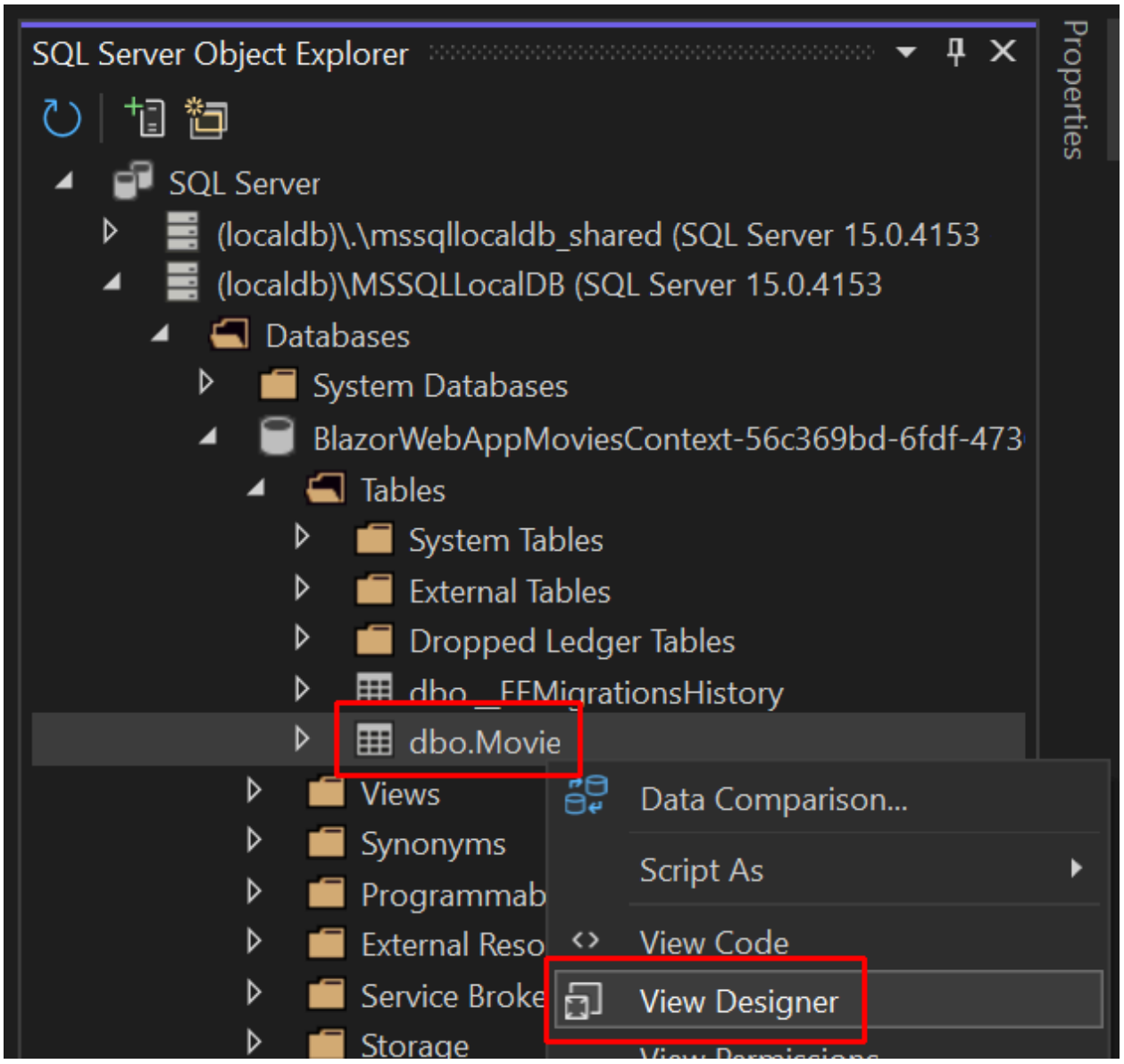
The Visual Studio version of this tutorial uses SQL Server.

SQL Server Express LocalDB is a lightweight version of the SQL Server Express database engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. Master database files (`*.mdf`) are placed in the `C:/Users/{USER}` directory, where the `{USER}` placeholder is the system's user ID.

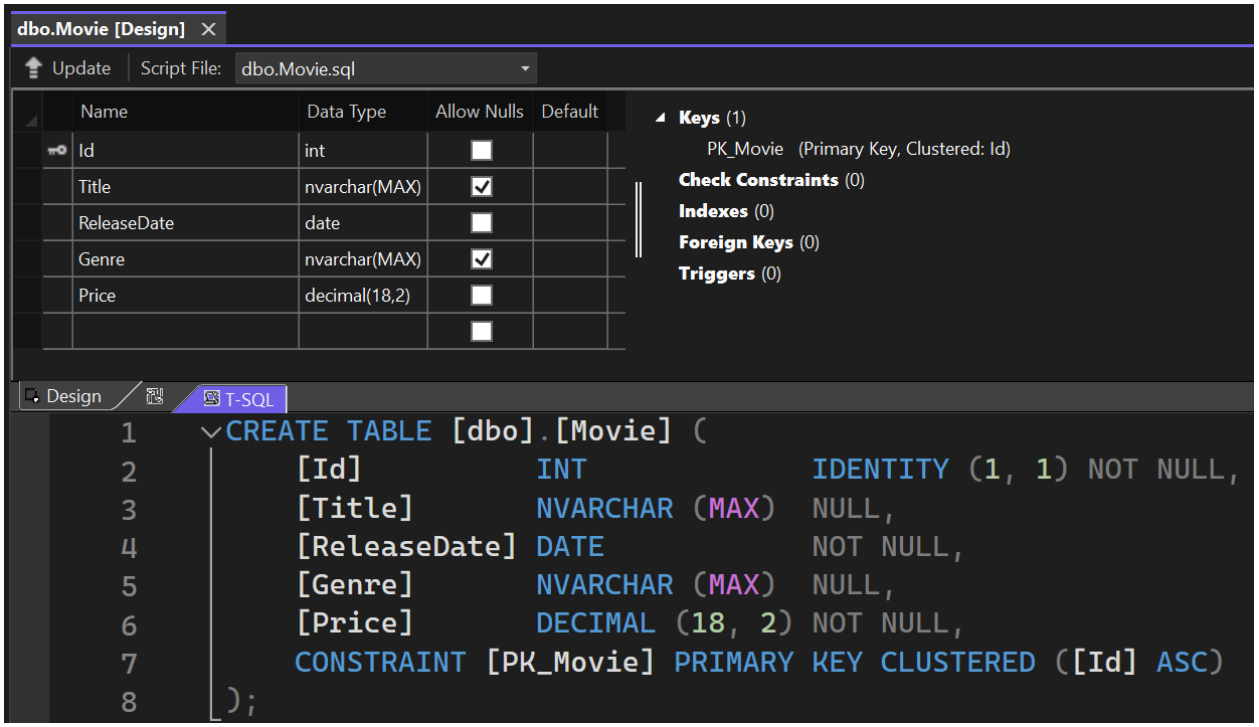
From the **View** menu, open **SQL Server Object Explorer** (SSOX).



Right-click on the `Movie` table and select **View Designer**:

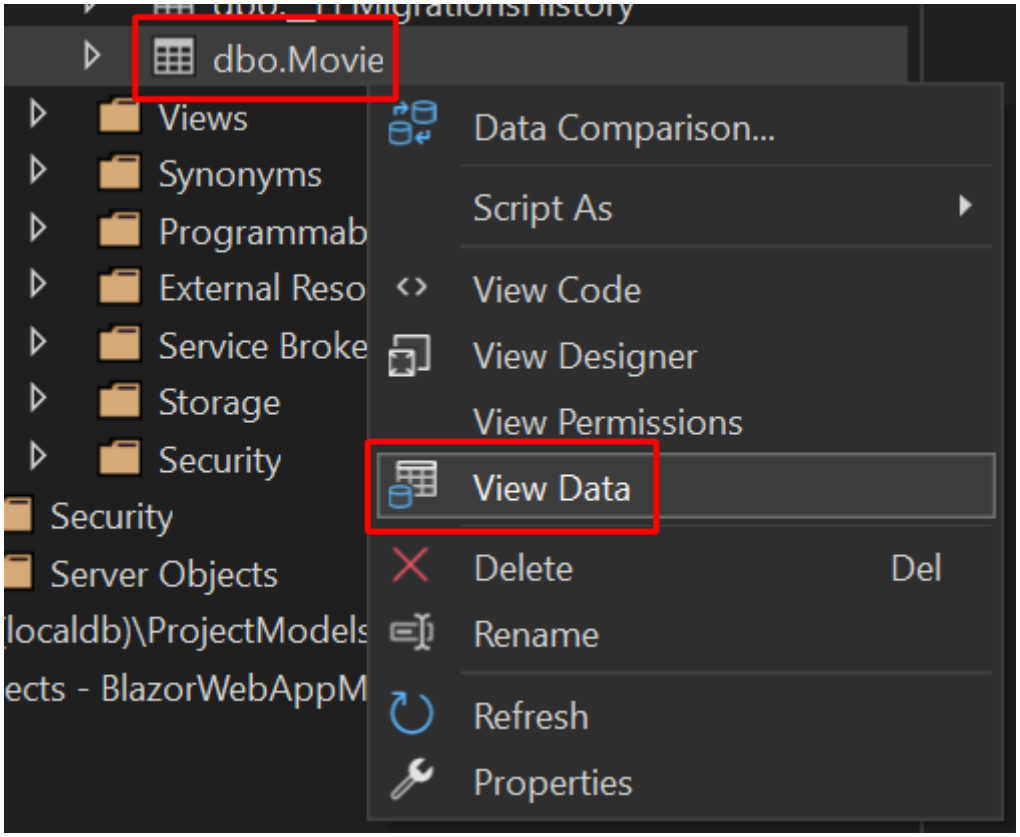


The **View Designer** opens:

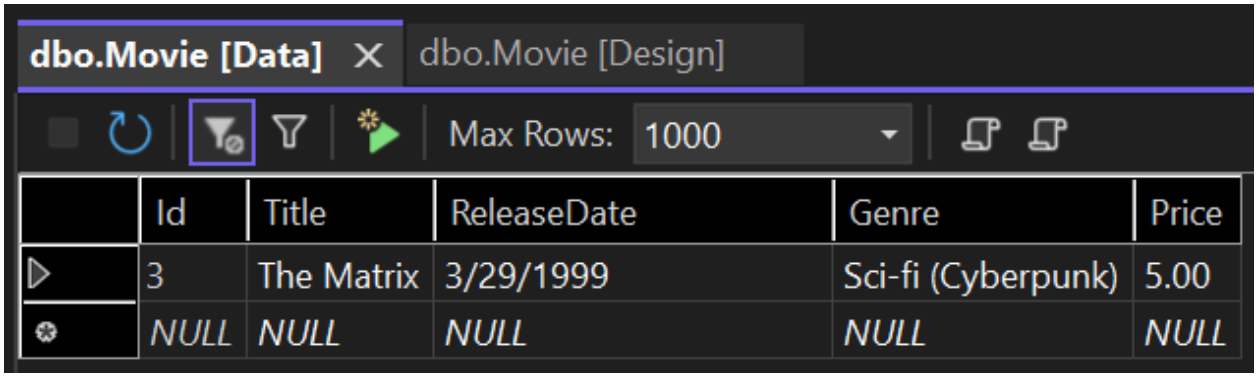


Note the key icon next to `ID`. EF creates a property named `ID` for the primary key.

Right-click on the `Movie` table and select **View Data**:



The table's data opens in a new tab in Visual Studio:



Seed the database

Seeding code can create a set of records for development testing or even be used to create the initial data for a new production database.

Create a new class named `SeedData` in the `Data` folder with the following code.

`Data/SeedData.cs`:

```
C#
```

```
using Microsoft.EntityFrameworkCore;
using BlazorWebAppMovies.Models;

namespace BlazorWebAppMovies.Data;

public class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using var context = new BlazorWebAppMoviesContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<BlazorWebAppMoviesContext>>());

        if (context == null || context.Movie == null)
        {
            throw new NullReferenceException(
                "Null BlazorWebAppMoviesContext or Movie DbSet");
        }

        if (context.Movie.Any())
        {
            return;
        }

        context.Movie.AddRange(
            new Movie
            {
                Title = "Mad Max",
                ReleaseDate = new DateOnly(1979, 4, 12),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 2.51M,
            },
            new Movie
            {
                Title = "The Road Warrior",
                ReleaseDate = new DateOnly(1981, 12, 24),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 2.78M,
            },
            new Movie
            {
                Title = "Mad Max: Beyond Thunderdome",
                ReleaseDate = new DateOnly(1985, 7, 10),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 3.55M,
            },
            new Movie
            {
                Title = "Mad Max: Fury Road",
                ReleaseDate = new DateOnly(2015, 5, 15),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 8.43M,
            },
            new Movie
```

```
            {
                Title = "Furiosa: A Mad Max Saga",
                ReleaseDate = new DateOnly(2024, 5, 24),
                Genre = "Sci-fi (Cyberpunk)",
                Price = 13.49M,
            });

        context.SaveChanges();
    }
}
```

A database context instance is obtained from the dependency injection (DI) container. If movies are present, `return` is called to avoid seeding the database. When the database is empty, the *Mad Max* franchise (©Warner Bros. Entertainment) movies are seeded.

To execute the seed initializer, add the following code to the `Program` file immediately after the line that builds the app (`var app = builder.Build();`). The using statement ensures that the database context is disposed after the seeding operation completes.

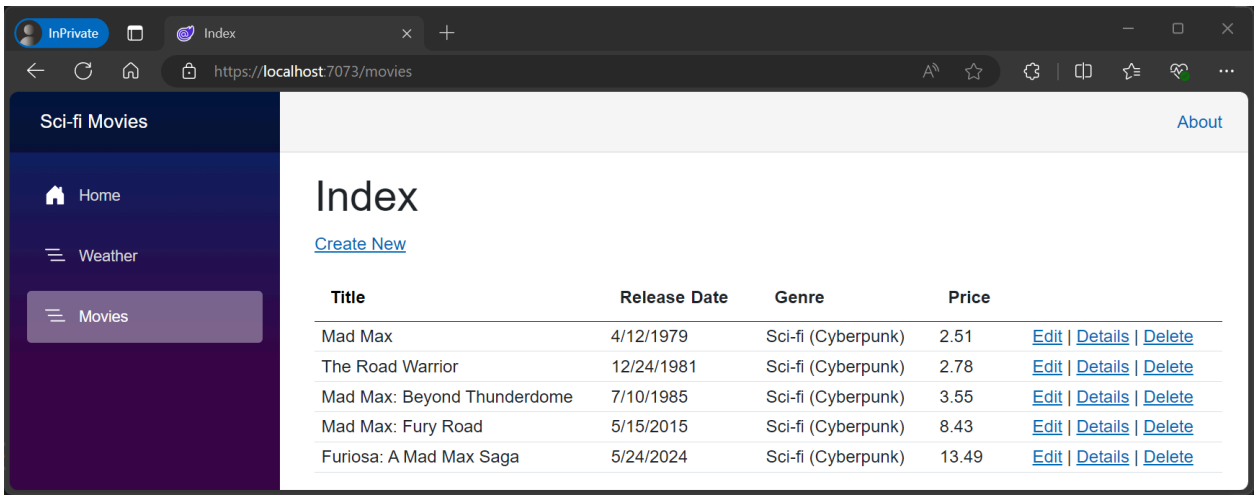
C#

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    SeedData.Initialize(services);
}
```

If the database contains records from earlier testing, run the app and delete the entities that you created in the database. Stop the app by closing the browser's window.

When the database is empty, run the app.

Navigate to the movies `Index` page to see the seeded movies:



Bind a form to a model

Review the the `Edit` component (`Components/Pages/MoviePages/Edit.razor`).

When an HTTP GET request is made for the `Edit` component page (for example at the relative URL: `/movies/edit?id=6`):

- The OnInitializedAsync method fetches the movie with an `Id` of `6` from the database and assigns it to the `Movie` property.
- The EditForm.Model parameter specifies the top-level model object for the form. An edit context is constructed for the form using the assigned model.
- The form is displayed with the values from the movie.

When the `Edit` page is posted to the server, the form values on the page are bound to the `Movie` property because the [SupplyParameterFromForm] attribute is annotated on the `Movie` property:

C#

```
[SupplyParameterFromForm]
private Movie? Movie { get; set; }
```

If the model state has errors when the form is posted, for example if `ReleaseDate` can't be converted into a date, the form is redisplayed with the submitted values. If no model errors exist, the movie is saved using the form's posted values.

Concurrency exception handling

Review the `UpdateMovie` method of the `Edit` component (`Components/Pages/MoviePages/Edit.razor`):

C#

```
private async Task UpdateMovie()
{
    using var context = DbFactory.CreateDbContext();
    context.Attach(Movie!).State = EntityState.Modified;

    try
    {
        await context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!MovieExists(Movie!.Id))
```

```
            {
                NavigationManager.NavigateTo("notfound");
            }
            else
            {
                throw;
            }
        }

        NavigationManager.NavigateTo("/movies");
    }
```

Concurrency exceptions are detected when one client deletes the movie and a different client posts changes to the movie.

To test how concurrency is handled by the preceding code:

1. Select **Edit** for a movie, make changes, but don't select **Save**.
2. In a different browser window, open the app to the movie `Index` page and select the **Delete** link for the same movie to delete the movie.
3. In the previous browser window, post changes to the movie by selecting the **Save** button.
4. The browser is navigated to the `notfound` endpoint, which doesn't exist and yields a 404 (Not Found) result.

Additional guidance on handling concurrency with EF Core in Blazor apps is available in the Blazor documentation.

Stop the app

If the app is running, shut the app down by closing the browser's window.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- Configuration articles:
  - Configuration in ASP.NET Core (ASP.NET Core Configuration system)
  - ASP.NET Core Blazor configuration (Blazor documentation)
  - Data seeding (EF Core documentation)
- Concurrency with EF Core in Blazor apps
- Database provider resources:
  - EF Core documentation
    - SQLite EF Core Database Provider Limitations
    - Customize migration code
  - SQLite ALTER TABLE statement (SQLite documentation)
- Blazor Web App security
  - ASP.NET Core Blazor authentication and authorization
  - ASP.NET Core Blazor authentication and authorization and the following articles in the *Server* security node
  - Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)
  - Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID

Legal

Mad Max, *The Road Warrior*, *Mad Max: Beyond Thunderdome*, *Mad Max: Fury Road*, and *Furiosa: A Mad Max Saga* are trademarks and copyrights of Warner Bros. Entertainment.

Next steps

Previous: Learn about Razor components | Next: Add Validation

Build a Blazor movie database app (Part 5 - Add validation)

Article • 11/14/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the fifth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series explains how metadata of the `Movie` model is used to validate user input in the forms that create and edit movies.

Validation using data annotations

Validation rules are specified on a model class using *data annotations*. The following list shows some of the System.ComponentModel.DataAnnotations attributes for user input validation of public properties in a form's model:

- [Required]: Require that the user provide a value.
- [StringLength]: Specifies the minimum and maximum length of characters. Note that a `MinimumLength` passed to the attribute doesn't make the string required (apply the [Required] attribute).
- [RegularExpression]: Specify a pattern to match for the user's input.
- [Range]: Specify the minimum and maximum values.

Value types, such as `decimal`, `int`, `float`, `DateOnly`, `TimeOnly`, and `DateTime`, are inherently required. Placing a [Required] attribute on value types isn't necessary.

Additional data annotations that you can use in your forms are covered by the Blazor reference documentation.

Add validation to the `Movie` model

Add the following data annotations to the `Movie` class properties. To update all of the properties at once, you can copy and paste the entire `Models/Movie.cs` file, which

appears after the following code example.

diff

```
+ [Required]
+ [StringLength(60, MinimumLength = 3)]
  public string? Title { get; set; }

+ [Required]
* [StringLength(30)]
+ [RegularExpression(@"^[A-Z]+[a-zA-Z()\s-]*$")]
  public string? Genre { get; set; }

+ [Range(0, 100)]
  [DataType(DataType.Currency)]
  [Column(TypeName = "decimal(18, 2)")]
  public decimal Price { get; set; }
```

`Models/Movie.cs` file after the preceding data annotations are applied to the properties:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace BlazorWebAppMovies.Models;

public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(60, MinimumLength = 3)]
    public string? Title { get; set; }

    public DateOnly ReleaseDate { get; set; }

    [Required]
    [StringLength(30)]
    [RegularExpression(@"^[A-Z]+[a-zA-Z()\s-]*$")]
    public string? Genre { get; set; }

    [Range(0, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
}
```

The preceding validation rules are merely for demonstration and aren't optimal for a production system. For example, the preceding validation prevents entering a movie

with only one or two characters and doesn't allow additional special characters for a movie's genre.

Create an EF Core migration and update the database

A data model schema defines how data is organized and connected within a relational database.

Adding the data annotations to the `Movie` class in the preceding section doesn't automatically result in matching changes to the database's schema.

Review the annotations applied to the `Title` property:

C#

```
[Required]
[StringLength(60, MinimumLength = 3)]
public string? Title { get; set; }
```

The difference between the model property and the database's schema are summarized in the following table. Neither constraint matches after the data annotation is applied to the `Movie` model.

Expand table

Constraint	Model `Title` property	Database `Title` column
Maximum length	60 characters	Byte pairs up to ~2 GB† NVARCHAR (MAX)
Required	✔ [Required]	✘ `NULL` is permitted in the column.

†Database character columns are defined by *size* (byte pairs). One byte-pair per character is used for characters defined in the Unicode range 0 to 65,535. However, individual characters outside of that Unicode range take multiple byte-pairs to store, so the actual number of characters that a column can store is arbitrary. The important concept for our purposes in comparing the `Title` property of the `Movie` model and the database schema for the `Title` column is that ~2 GB of stored byte-pairs in the database far exceeds the 60 character limit set for the property. The database schema *should be adjusted downward* to match the app's constraint.

To match the `Movie` model's `Title` property length in the app, the database should indicate `NVARCHAR (60)` for the size of the `Title` column. The schema difference doesn't cause EF Core to throw an exception when the app is used because a user posting a 60 character movie title fits within the database's ~2 GB byte-pair limit for a movie title. However, consider the reverse situation where a model property is given a constraint larger than what the database permits and the user posts a string too long for a database character column: An exception is thrown by the database or data is truncated when the user posts the value. You should always keep the app's models aligned with the database's schema because a misaligned schema can cause exceptions and the storage of incorrect data.

Although the `Title` property is a nullable reference type (NRT), as indicated by the `?` on the `string` type (`string?`), the database shouldn't store a `NULL` value in its `Title` column due to the model's `Required` constraint. When the database's schema is updated in the next step, the database's `Title` column should reflect `NOT NULL` for the `Title` column to match the property. The important concept is that just because a model property is an NRT and can hold a `null` value in code doesn't mean that the database column's schema should be nullable (`NULL` permitted). These are independent conditions used for different purposes: NRTs are used to prevent coding errors with nullable types, while the database schema reflects the exact type and size of stored data.

To align the model and the database schema, create and apply an EF Core *database migration* with a migration name that identifies the migration changes. The migration name is similar to a commit message in a version control system. In the following command examples, the migration name "`NewMovieDataAnnotations`" reflects that new data annotations are added to the `Movie` model.

> **ⓘ Important**
>
> Make sure that the app isn't running for the next steps.
>
> Stopping the app when using Visual Studio only requires you to close the browser's window.
>
> When using VS Code, close the browser's window and stop the app in VS Code with **Run** > **Stop Debugging** or by pressing `Shift`+`F5` on the keyboard.
>
> When using the .NET CLI, close the browser's window and stop the app in the command shell with `Ctrl`+`C` (Windows) or `⌘`+`C` (macOS).

In Visual Studio **Solution Explorer**, double-click **Connected Services**. In the **Service Dependencies** area, select the ellipsis (`...`) followed by **Add migration** in the **SQL Server Express LocalDB** area.

Give the migration a **Migration name** of `NewMovieDataAnnotations` to describe the migration. Wait for the database context to load in the **DbContext class names** field. Select **Finish** to create the migration. Select the **Close** button when the operation completes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load. Select the **Finish** button. Select the **Close** button when the operation completes.

After applying the migration, the model property and the database's schema match, as summarized in the following table.

Expand table

Constraint	**Model `Title` property**	**Database `Title` column**
Maximum length	60 characters	Sixty (60) byte pairs are permitted†. NVARCHAR (60)
Required	✔ [Required]	✔ `NOT NULL` is indicated for the column.

†Sixty (60) byte pairs is 60 characters if one byte-pair per character is used to store the movie title, which is true when using characters defined in the Unicode range 0 to 65,535.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- Tag Helpers in forms in ASP.NET Core
- Globalization and localization in ASP.NET Core
- Tag Helpers in ASP.NET Core
- Author Tag Helpers in ASP.NET Core
- Migrations overview (EF Core documentation)
- nchar and nvarchar (Transact-SQL) (SQL Server documentation)
- Blazor enhanced forms

Next steps

Previous: Work with a database | Next: Add search

Build a Blazor movie database app (Part 6 - Add search)

Article • 11/14/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the sixth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series covers adding a search feature to the movies `Index` component to filter movies by title.

Implement a filter feature for the `QuickGrid` component

The QuickGrid component is used by the movie `Index` component (`Components/MoviePages/Index.razor`) to display movies from the database:

razor

```
<QuickGrid Class="table" Items="context.Movie">
    ...
</QuickGrid>
```

The Items parameter receives an `IQueryable<TGridItem>`, where `TGridItem` is the type of data represented by each row in the grid (`Movie`). Items is assigned a collection of movie entities (`DbSet<Movie>`) obtained from the created database context (CreateDbContext) of the injected database context factory (`DbFactory`).

To make the `QuickGrid` component filter on the movie title, the `Index` component should:

- Set a filter string as a *component parameter* from the query string.
- If the parameter has a value, filter the movies returned from the database.

- Provide an input for the user to provide the filter string and a button to trigger a reload using the filter.

Start by adding the following code to the `@code` block of the `Index` component (`MoviePages/Index.razor`):

C#

```
[SupplyParameterFromQuery]
private string? TitleFilter { get; set; }

private IQueryable<Movie> FilteredMovies =>
    context.Movie.Where(m => m.Title!.Contains(TitleFilter ??
string.Empty));
```

`TitleFilter` is the filter string. The property is provided the [SupplyParameterFromQuery] attribute, which lets Blazor know that the value of `TitleFilter` should be assigned from the query string when the query string contains a field of the same name (for example, `?titleFilter=road+warrior` yields a `TitleFilter` value of `road warrior`). Note that query string field names, such as `titleFilter`, aren't case sensitive.

The `FilteredMovies` property is an `IQueryable<Movie>`, which is the type for assignment to the `QuickGrid`'s Items parameter. The property filters the list of movies based on the supplied `TitleFilter`. If a `TitleFilter` isn't assigned a value from the query string (`TitleFilter` is `null`), an empty string (`string.Empty`) is used for the Contains clause. Therefore, no movies are filtered for display.

Change the `QuickGrid` component's Items parameter to use the `movies` collection:

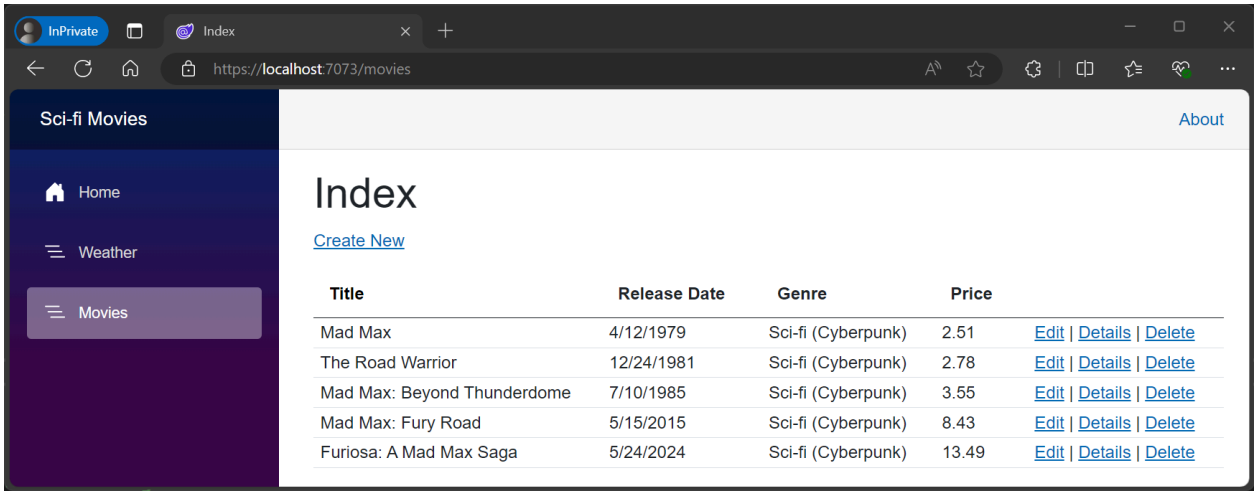
diff

```
- <QuickGrid Class="table" Items="context.Movie">
+ <QuickGrid Class="table" Items="FilteredMovies">
```

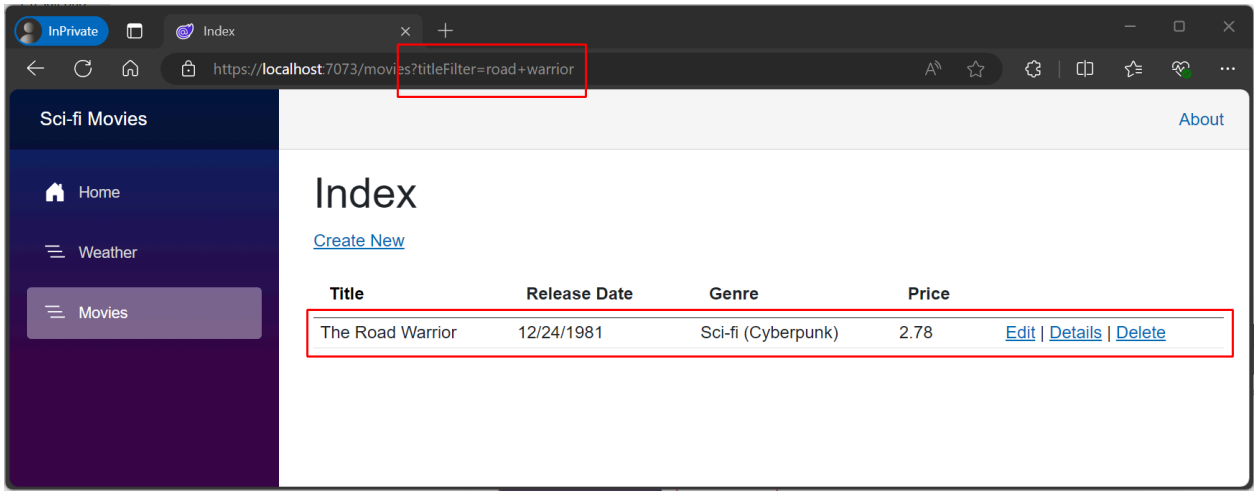
The `movie => movie.Title!.Contains(...)` code is a *lambda expression*. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where or Contains methods. LINQ queries aren't executed when they're defined or when they're modified by calling a method, such as Where, Contains, or OrderBy. Rather, query execution is deferred. The evaluation of an expression is delayed until its realized value is iterated.

The Contains method is run on the database, not in the C# code. The case sensitivity of the query depends on the database and the collation. For SQL Server, Contains maps to SQL LIKE, which is case insensitive. SQLite with default collation provides a mixture of case sensitive and case insensitive filtering, depending on the query. For information on making case insensitive SQLite queries, see the Additional resources section of this article.

Run the app and navigate to the movies `Index` page at `/movies`. The movies in the database load:



Append a query string to the URL in the address bar: `?titleFilter=road+warrior`. For example, the full URL appears as `https://localhost:7073/movies?titleFilter=road+warrior`, assuming the port number is `7073`. The filtered movie is displayed:



Next, give users a way to provide the `titleFilter` filter string via the component's UI. Add the following HTML under the H1 heading (`<h1>Index</h1>`). The following HTML reloads the page with the contents of the textbox as a query string value:

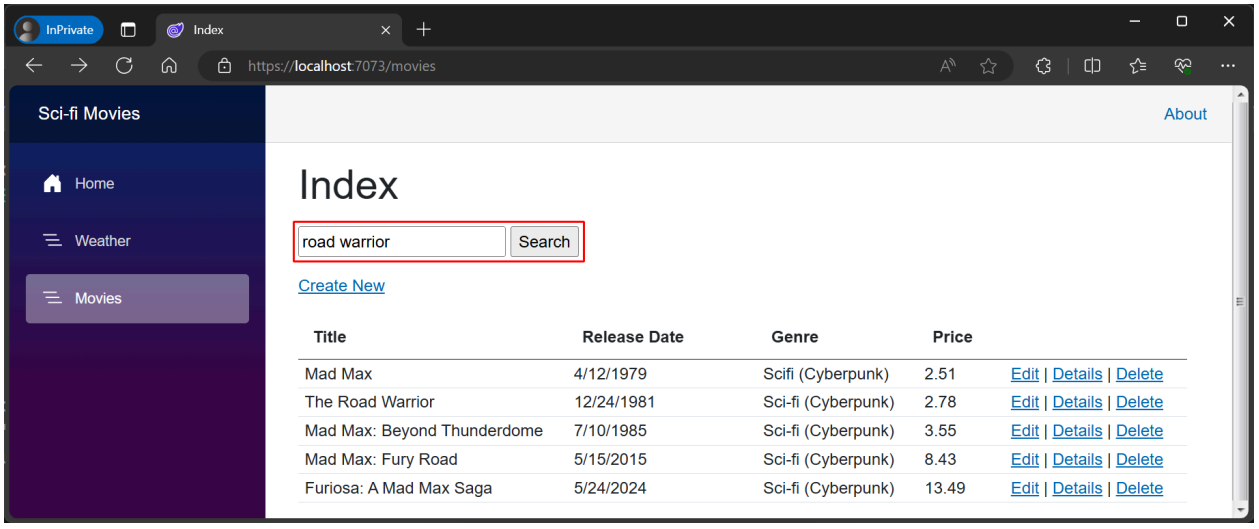
HTML

```
<div>
    <form action="/movies" data-enhance>
        <input type="search" name="titleFilter" />
        <input type="submit" value="Search" />
    </form>
</div>
```

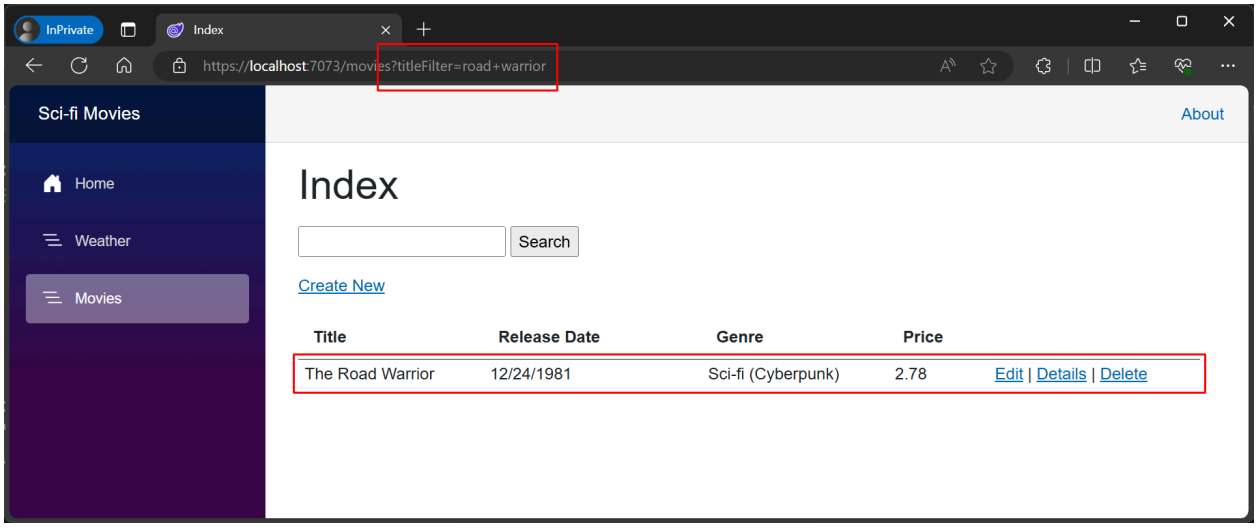
The `data-enhance` attribute applies *enhanced navigation* to the component, where Blazor intercepts the GET request and performs a fetch request instead. Blazor then patches the response content into the page, which avoids a full-page reload and preserves more of the page state. The page loads faster, usually without losing the user's scroll position.

Save the file that you're working on. Apply the change by either restarting the app or using Hot Reload to apply the change to the running app.

Type "`road warrior`" into the search box and select the **Search** button to filter the movies:



The result after searching on `road warrior`:



Notice that the search box loses the search value ("`road warrior`") when the movies are filtered. If you want to preserve the searched value, add the `data-permanent` attribute:

diff

```
- <form action="/movies" data-enhance>
+ <form action="/movies" data-enhance data-permanent>
```

Stop the app

Stop the app by closing the browser's window.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- LINQ documentation
- Write C# LINQ queries to query data (C# documentation)
- Lambda Expression (C# documentation
- Case insensitive SQLite queries
  - How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
  - How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
  - Collations and Case Sensitivity

Legal

Mad Max, *The Road Warrior*, *Mad Max: Beyond Thunderdome*, *Mad Max: Fury Road*, and *Furiosa: A Mad Max Saga* are trademarks and copyrights of Warner Bros. Entertainment.

Next steps

Previous: Add validation | Next: Add a new field

Build a Blazor movie database app (Part 7 - Add a new field)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the seventh part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

This part of the tutorial series covers adding a new field to the movie class, CRUD pages, and database.

The database update is handled by EF Core migrations. EF Core transparently tracks changes to the database in a migration history table and automatically throws an exception if the app's model classes aren't synchronized with the database's tables and columns. EF Core migrations make it possible to quickly troubleshoot database consistency problems.

> **Important**
>
> Confirm that the app isn't running for the next steps.

Add a movie rating to the app's model

Open the `Models/Movie.cs` file and add a `Rating` property with a regular expression that limits the value of `Rating` to the exact Motion Picture Association film rating designations:

C#

```
[Required]
[RegularExpression(@"^(G|PG|PG-13|R|NC-17)$")]
public string? Rating { get; set; }
```

Add the movie rating to the app's CRUD components

Open the `Create` component definition file (`Components/Pages/MoviePages/Create.razor`).

Add the following `<div>` block between the `<div>` block for `Price` and the create button (`<button>`):

razor

```
<div class="mb-3">
    <label for="rating" class="form-label">Rating:</label>
    <InputText id="rating" @bind-Value="Movie.Rating" class="form-control"
/>
    <ValidationMessage For="() => Movie.Rating" class="text-danger" />
</div>
```

Open the `Delete` component definition file (`Components/Pages/MoviePages/Delete.razor`).

Add the following description list (`<dl>`) block between the description list block for `Price` and the `EditForm` component:

razor

```
<dl class="row">
    <dt class="col-sm-2">Rating</dt>
    <dd class="col-sm-10">@movie.Rating</dd>
</dl>
```

Open the `Details` component definition file (`Components/Pages/MoviePages/Details.razor`).

Add the following description list term (`<dt>`) and description list element (`<dl>`) after the term and element for `Price` just inside the closing `</dl>` tag:

razor

```
<dt class="col-sm-2">Rating</dt>
<dd class="col-sm-10">@movie.Rating</dd>
```

Open the `Edit` component definition file (`Components/Pages/MoviePages/Edit.razor`).

Add the following `<div>` block between the `<div>` block for `Price` and the save button (`<button>`):

razor

```
<div class="mb-3">
    <label for="rating" class="form-label">Rating:</label>
    <InputText id="rating" @bind-Value="Movie.Rating" class="form-control"
/>
    <ValidationMessage For="() => Movie.Rating" class="text-danger" />
</div>
```

Open the `Index` component definition file (`Components/Pages/MoviePages/Index.razor`).

Update the `QuickGrid` component to include the movie rating. Add the following PropertyColumn<TGridItem,TProp> immediately after the column for `Price`:

razor

```
<PropertyColumn Property="movie => movie.Rating" />
```

Update the `SeedData` class (`Data/SeedData.cs`) to provide a default value for the new `Rating` property for reseeding operations.

The following change is for the *Mad Max* `new Movie` block. `Rating = "R",` is added to the block:

diff

```
new Movie
{
    Title = "Mad Max",
    ReleaseDate = DateOnly.Parse("1979-4-12"),
    Genre = "Sci-fi (Cyberpunk)",
    Price = 2.51M,
+   Rating = "R",
},
```

Add the `Rating` property to each of the other `new Movie` blocks in the same fashion. Here are the ratings of the remaining *Mad Max* movies when you add the new `Rating` lines:

- *The Road Warrior*: R
- *Mad Max: Beyond Thunderdome*: PG-13
- *Mad Max: Fury Road*: R

- *Furiosa: A Mad Max Saga*: R

Save all of the updated files.

Don't run the app yet. Build the app to confirm that there are no errors.

Select **Build** > **Rebuild Solution** from the menu bar.

Fix any errors in the app that were introduced by pasting the preceding markup and code before proceeding to the next step.

Update the database

If you were to try and run the app at this point, the app would fail with a SQL exception because the database doesn't include a `Rating` column in its `Movie` table. There are three approaches that you can take to resolve the discrepancy between the database's schema and the model's schema:

- Modify the schema of the database so that it matches the model class. The advantage of this approach is that it maintains the database's data. Adopt this approach using either database tooling or by creating a database change script, which are approaches not covered by this tutorial but can be learned using other articles. The downside to this approach is that it requires more time and is more prone to errors due to its increased complexity. *This tutorial doesn't adopt this approach.*
- Use EF Core to automatically drop and recreate the database using the new model class schema, losing the data stored in the database in the process. The database is reseeded with fresh data when the app is run. This approach allows you to quickly evolve the model and database schema together. Don't use this approach on a production database with data that must be preserved. *This tutorial doesn't adopt this approach.*
- Use an EF Core migration to update the database schema after the model is changed in the app. This approach is efficient and preserves the database's data. ***This tutorial adopts this approach.***

Create a migration to update the database's schema. The movie rating, as a `Rating` column, is added to the database's `Movie` table.

In Visual Studio **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name** of `AddRatingField` to describe the migration. Wait for the database context to load in the **DbContext class names** field. Select **Finish** to create the migration. Select **Close** when the operation is complete.

The migration:

- Compares the `Movie` model with the `Movie` database table schema.
- Creates code to migrate the database schema to match the model.

Creating the migration doesn't automatically provision a default value for the rating when the database is updated. However, you can manually make a change to the migration file to apply a default movie rating value, which can be helpful when there are many records that require a default value. In this case, all but one of the *Mad Max* movies is rated *R*, so a default value of "`R`" for the `Rating` column is an appropriate choice. The one movie that doesn't have an *R* rating can be updated later in the running app.

Examine the files in the `Migrations` folder of the project in Visual Studio's **Solution Explorer**. Open the migration file that adds the movie rating field, which has a file name of `{TIME STAMP}_AddRatingField.cs`, where the `{TIME STAMP}` placeholder is a time stamp (for example, `20240530123755_AddRatingField.cs`).

Find the `AddColumn` block that adds the rating column to the `Movie` table in the database. Modify the last line that applies a default value (`defaultValue`). Change it from an empty string (`""`) to an *R* movie rating (`"R"`):

diff

```
migrationBuilder.AddColumn<string>(
    name: "Rating",
    table: "Movie",
    type: "nvarchar(max)",
    nullable: false,
-   defaultValue: "");
+   defaultValue: "R");
```

Save the migration file.

In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load. Select the **Finish** button. Select the **Close** button when the operation completes.

Modify the one movie that isn't rated *R*:

1. Run the app.
2. Edit the *Mad Max: Beyond Thunderdome* movie.
3. Update the movie rating from `R` to `PG-13`. Save the change.

> ⓘ **Note**
>
> An alternative to modifying the migration file is to delete the records in the database and rerun the app to reseed the database. The seeding code was modified earlier to supply default values. This approach is useful in cases where the assignment of default values to fields is better controlled or faster for you to implement with C# code during seeding. For more information see the Delete all database records and reseed the database section at the end of this article.

Run the app and verify you can create, edit, and display movies with the new movie rating field.

Troubleshoot

In the event that the database becomes corrupted, delete the database and use migrations to recreate the database:

1. Select the database in **SQL Server Object Explorer** (SSOX).
2. Right-click on the database, and select **Delete**. *Make sure that you select the correct database in the list.*
3. Check **Close existing connections**.
4. Select **OK**.
5. In **Solution Explorer**, double-click **Connected Services**. In the **Service Dependencies** area, select the ellipsis (`...`) followed by **Update database** in the **SQL Server Express LocalDB** area. Updating the database executes the existing migrations that recreate the database.

Delete all database records and reseed the database

This section describes an alternative process for updating the database with a default value for a new model property without modifying the migration file. Following the guidance in this section isn't necessary if you followed all of the steps in the Update the database section earlier in this article.

To delete all of the records in the database, use one of the following approaches:

- Run the app and use the delete links in the browser. This approach is reasonably fast when there are only a few records to delete.
- In Visual Studio from **SQL Server Object Explorer** (SSOX), delete the database records. With the database table visible, right-click the table and select **View Data**. When the table opens to show the movie records, select the first record. Hold the `Shift` key and select the last record to select all of the records in the table. With all of the records selected, press the `Delete` key to delete the records. This approach is reasonably fast when there are many records to delete.
- In Visual Studio from SSOX, right-click the database and select **New Query**. A blank SQL file (`.sql`) opens. Paste the following command into the file: `DELETE FROM dbo.Movie;`. Select the green **Execute** triangle to execute the query or press `Ctrl`+`Shift`+`E` on the keyboard. In the **Message** tab, SSOX reports the number of rows affected, deleted in this case, by the query. You can save the query for later use if you wish. Otherwise, close the file without saving it. This approach is especially useful when the table is large and contains hundreds to thousands of records.

> **⊗ Caution**
>
> Use extreme caution when deleting records from a database. Deleting records is permanent without taking additional data loss mitigation steps. Production databases often provision automatic backup copies of data, either instantaneously as the database is modified or periodically, including with off-site copies and permanent physical storage of data.

After deleting all of the records, run the app. The initializer reseeds the database and includes the correct movie ratings for the `Rating` field based on the seeding code.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Additional resources

- Migrations (EF Core documentation)
- Customize migration code

Legal

Mad Max, *The Road Warrior*, *Mad Max: Beyond Thunderdome*, *Mad Max: Fury Road*, and *Furiosa: A Mad Max Saga* are trademarks and copyrights of Warner Bros. Entertainment.

Next steps

Previous: Add Search | Next: Add interactivity

Build a Blazor movie database app (Part 8 - Add interactivity)

Article • 12/05/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article is the eighth part of the Blazor movie database app tutorial that teaches you the basics of building an ASP.NET Core Blazor Web App with features to manage a movie database.

Up to this point in the tutorial, the entire app has been *enabled* for interactivity, but the app has only *adopted* interactivity in its `Counter` sample component. This part of the tutorial series explains how to adopt interactivity in the movie `Index` component.

> **Important**
>
> Confirm that the app isn't running for the next steps.

Adopt interactivity

Interactivity means that a component has the capacity to process UI events via C# code, such as a button click. The events are either processed on the server by the ASP.NET Core runtime or in the browser on the client by the WebAssembly-based Blazor runtime. This tutorial adopts interactive server-side rendering (interactive SSR), also known generally as Interactive Server (`InteractiveServer`) rendering. Client-side rendering (CSR), which is inherently interactive, is covered in the Blazor reference documentation.

Interactive SSR enables a rich user experience like one would expect from a client app but without the need to create API endpoints to access server resources. UI interactions are handled by the server over a real-time SignalR connection with the browser. Page content for interactive pages is prerendered, where content on the server is initially generated and sent to the client without enabling event handlers for rendered controls. With prerendering, the server outputs the HTML UI of the page as soon as possible in response to the initial request, which makes the app feel more responsive to users.

Review the API in the `Program` file (`Program.cs`) that enables interactive SSR. Razor component services are added to the app to enable Razor components to render statically from the server (AddRazorComponents) and execute code with interactive SSR (AddInteractiveServerComponents):

```
C#

builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

MapRazorComponents maps components defined in the root `App` component to the given .NET assembly and renders routable components, and AddInteractiveServerRenderMode configures the app's SignalR hub to support interactive SSR:

```
C#

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

Up to this point in the tutorial series, the calls to AddInteractiveServerComponents and AddInteractiveServerRenderMode weren't required for the movie pages because the app only adopted static SSR features for the movie components. This article explains how to adopt interactive features in the movie `Index` component.

When Blazor sets the type of rendering for a component, the rendering is referred to as the component's *render mode*. The following table shows the available render modes for rendering Razor components in a Blazor Web App.

Expand table

Name	Description	Render location	Interactive
Static Server	Static server-side rendering (static SSR)	Server	✗
Interactive Server	Interactive server-side rendering (interactive SSR) using Blazor Server.	Server	✓
Interactive WebAssembly	Client-side rendering (CSR) using Blazor WebAssembly.	Client	✓
Interactive Auto	Interactive SSR using Blazor Server initially and then CSR on subsequent visits after the Blazor bundle is downloaded.	Server, then client	✓

To apply a render mode to a component, the developer either uses the `@rendermode` directive or directive attribute on the component instance or on the component definition:

- The following example shows how to set the render mode on a component instance with the `@rendermode` directive attribute. The following example uses a hypothetical dialog (`Dialog`) component in a parent chat (`Chat`) component.

  Within the `Components/Pages/Chat.razor` file:

  ```razor
  <Dialog @rendermode="InteractiveServer" />
  ```

- The following example shows how to set the render mode on a component definition with the `@rendermode` directive. The following example shows setting the render mode in a hypothetical sales forecast (`SalesForecast`) component definition file (`.razor`).

  At the top of `Components/Pages/SalesForecast.razor` file:

  ```razor
  @page "/sales-forecast"
  @rendermode InteractiveServer
  ```

Using the preceding approaches, you can apply a render mode on a per-page/component basis. However, an entire app can adopt a single render mode via a root component that then by inheritance sets the render mode of every other component loaded. This is termed *global interactivity*, as opposed to *per-page/component interactivity*. Global interactivity is useful if most of the app requires interactive features. Global interactivity is usually applied via the `App` component, which is the root component of an app created from the Blazor Web App project template.

> **Note**
>
> More information on render modes and global interactivity is provided by Blazor's reference documentation. For the purposes of this tutorial, the app only adopts interactive SSR on a per-page/component basis. After the tutorial, you're free to use this app to study the other component render modes and the global interactivity location.

Open the movie `Index` component file (`Components/Pages/MoviePages/Index.razor`), and add the following `@rendermode` directive immediately after the `@page` directive to make the component interactive:

diff

```
@rendermode InteractiveServer
```

To see how making a component interactive enhances the user experience, let's provide three enhancements to the app in the following sections:

- Add pagination to the movie `QuickGrid` component.
- Make the `QuickGrid` component *sortable*.
- Replace the HTML form for filtering movies by title with C# code that:
  - Runs on the server.
  - Renders content interactively over the underlying SignalR connection.

Add pagination to the `QuickGrid`

The `QuickGrid` component can page data from the database.

Open the `Index` component (`Components/Pages/Movies/Index.razor`). Add a PaginationState instance to the `@code` block. Because the tutorial only uses five movie records for demonstration, set the ItemsPerPage to `2` items in order to demonstrate pagination. Normally, the number of items to display would be set to a higher value or set dynamically via a dropdown list.

C#

```
private PaginationState pagination = new PaginationState { ItemsPerPage = 2
};
```

Set the `QuickGrid` component's Pagination property to `pagination`:

diff

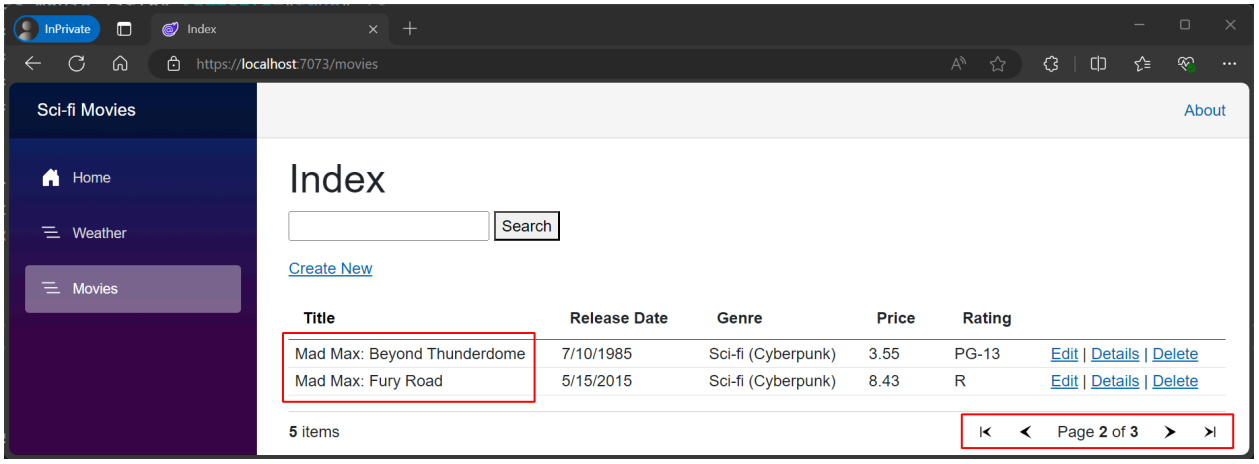
```
- <QuickGrid Class="table" Items="FilteredMovies">
+ <QuickGrid Class="table" Items="FilteredMovies" Pagination="pagination">
```

To provide a UI for pagination below the `QuickGrid` component, add a Paginator component below the `QuickGrid` component. Set the Paginator.State to `pagination`:

razor

```
<Paginator State="pagination" />
```

Run the app and navigate to the movies `Index` page. You can page through the movie items at two movies per page:



The component is *interactive*. The page doesn't reload for paging to occur. The paging is performed live over the SignalR connection between the browser and the server, where the paging operation is performed on the server with the rendered result sent back to the client for the browser to display.

Change ItemsPerPage to a more reasonable value, such as five items per page:

diff

```
- private PaginationState pagination = new PaginationState { ItemsPerPage =
2 };
+ private PaginationState pagination = new PaginationState { ItemsPerPage =
5 };
```

Sortable `QuickGrid`

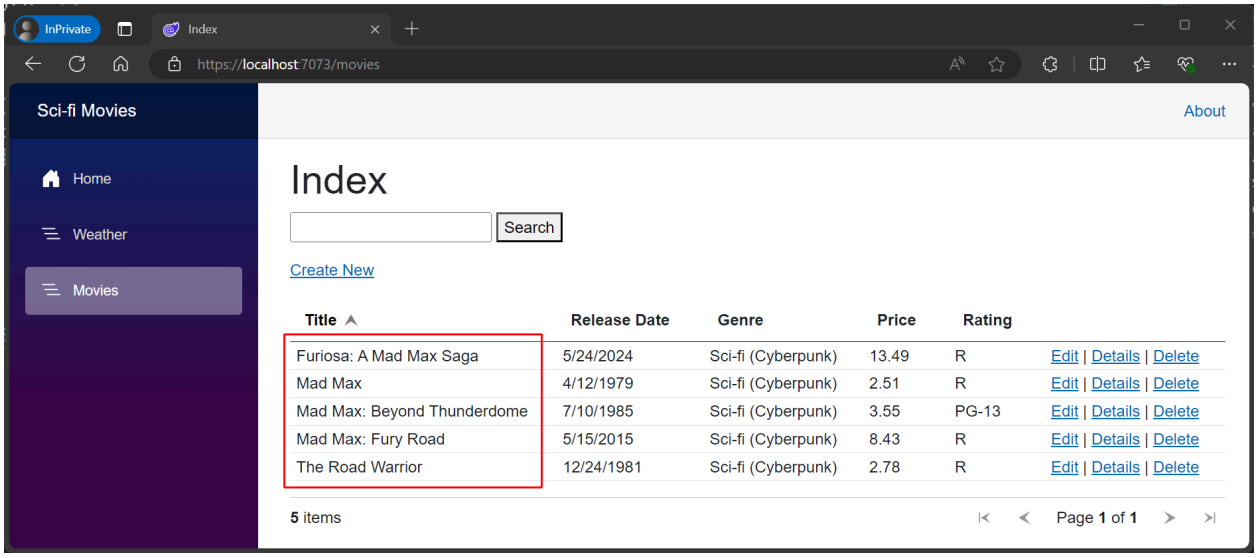
Open the `Index` component (`Components/Pages/Movies/Index.razor`).

Add `Sortable="true"` (Sortable) to the title PropertyColumn<TGridItem,TProp>:

diff

```
- <PropertyColumn Property="movie => movie.Title" />
+ <PropertyColumn Property="movie => movie.Title" Sortable="true" />
```

You can sort the `QuickGrid` by movie title by selecting the **Title** column. The page doesn't reload for sorting to occur. The sorting is performed live over the SignalR connection, where the sorting operation is performed on the server with the rendered result sent back to the client:



Use C# code and interactivity to search by title

In an earlier part of the tutorial series, the `Index` component was modified to allow the user to filter movies by title. This was accomplished by:

- Adding an HTML form that issues a GET request to the server with the user's title search string as a query string field-value pair (for example, `?titleFilter=road+warrior` if the user searches for "`road warrior`").
- Adding code to the component that obtains the title search string from the query string and uses it to filter the database's records.

The preceding approach is effective for a component that adopts static SSR, where the only interaction between the client and server is via HTTP requests. There was no live SignalR connection between the client and the server, and there was no way for the app on the server to process C# code *interactively* based on the user's actions in the component's UI and return content.

Now that the component is interactive, it can provide an improved user experience with Blazor features for binding and event handling.

Add a delegate event handler that the user can trigger to filter the database's movie records. The method uses the value of the `TitleFilter` property to perform the operation. If the user clears `TitleFilter` and searches, the method loads the entire movie list for display.

Delete the following lines from the `@code` block:

```diff
- [SupplyParameterFromQuery]
- private string? TitleFilter { get; set; }

- private IQueryable<Movie> FilteredMovies =>
-     context.Movie.Where(m => m.Title!.Contains(TitleFilter ??
string.Empty));
```

Replace the deleted code with the following code:

```C#
private string titleFilter = string.Empty;

private IQueryable<Movie> FilteredMovies =>
    context.Movie.Where(m => m.Title!.Contains(titleFilter));
```

Next, the component should bind the `titleFilter` field to an `<input>` element, so user input is stored in the `titleFilter` variable. Binding is achieved in Blazor with the `@bind` directive attribute.

Remove the HTML form from the component:

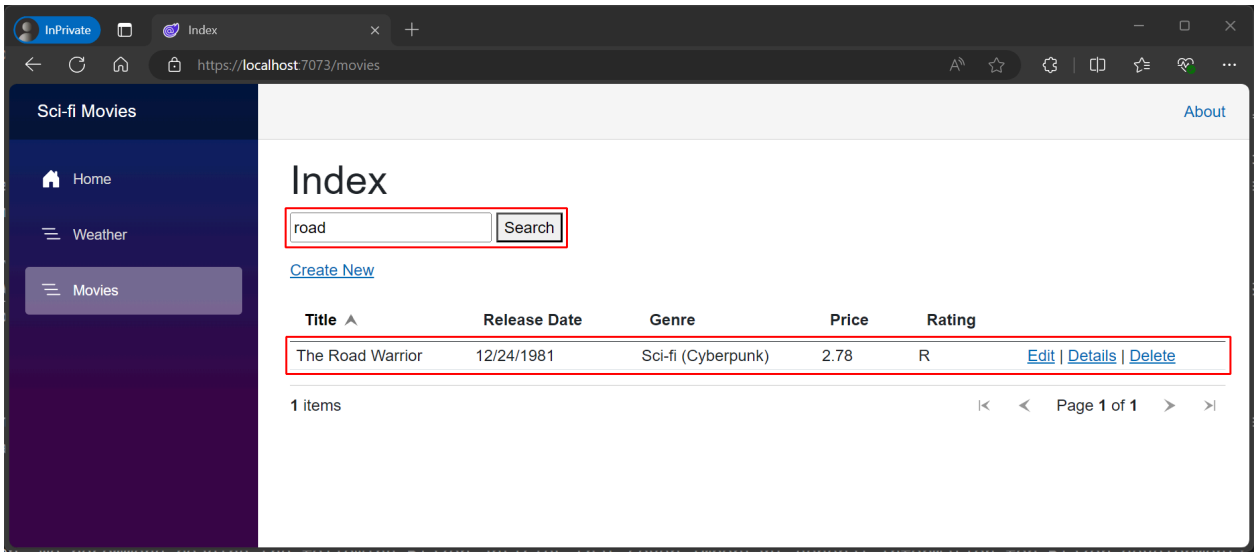
```diff
- <form action="/movies" data-enhance>
-     <input type="search" name="titleFilter" />
-     <input type="submit" value="Search" />
- </form>
```

In its place, add the following Razor markup:

```razor
<input type="search" @bind="titleFilter" @bind:event="oninput" />
```

`@bind:event="oninput"` performs binding for the HTML's `oninput` event, which fires when the `<input>` element's value is changed as a direct result of a user typing in the search box. The `QuickGrid` is bound to `FilteredMovies`. As `titleFilter` changes with the value of the search box, rerendering the `QuickGrid` bound to the `FilteredMovies` method filters movie entities based on the updated value of `titleFilter`.

Run the app, type "`road warrior`" into the search field and notice how the `QuickGrid` is filtered for each character entered until *The Road Warrior* movie is left when the search field reaches "`road `" ("road" followed by a space).



Filtering database records is performed on the server, and the server interactively sends back the HTML to display over the same SignalR connection. The page doesn't reload. The user feels like their interactions with the page are running code on the client. Actually, the code is running the server.

Instead of an HTML form, submitting a GET request in this scenario could've also used JavaScript to submit the request to the server, either using the Fetch API` or XMLHttpRequest API. In most cases, JavaScript can be replaced by using Blazor and C# in an interactive component.

Style the `QuickGrid` component

You can apply styles to the rendered `QuickGrid` component with a stylesheet isolated to the `Index` component using *CSS isolation*.

CSS isolation is applied by adding a stylesheet file using the file name format `{COMPONENT NAME}.razor.css`, where the `{COMPONENT NAME}` placeholder is the component name.

To apply styles to a child component, such as the `QuickGrid` component of the `Index` component, use the `::deep` pseudo-element.

In the `MoviePages` folder, add the following stylesheet for the `Index` component. Use `::deep` pseudo-elements to make the row height `3em` and vertically center the table cell content.

`Components/Pages/MoviePages/Index.razor.css`:

css

```
::deep tr {
    height: 3em;
}

    ::deep tr > td {
        vertical-align: middle;
    }
```

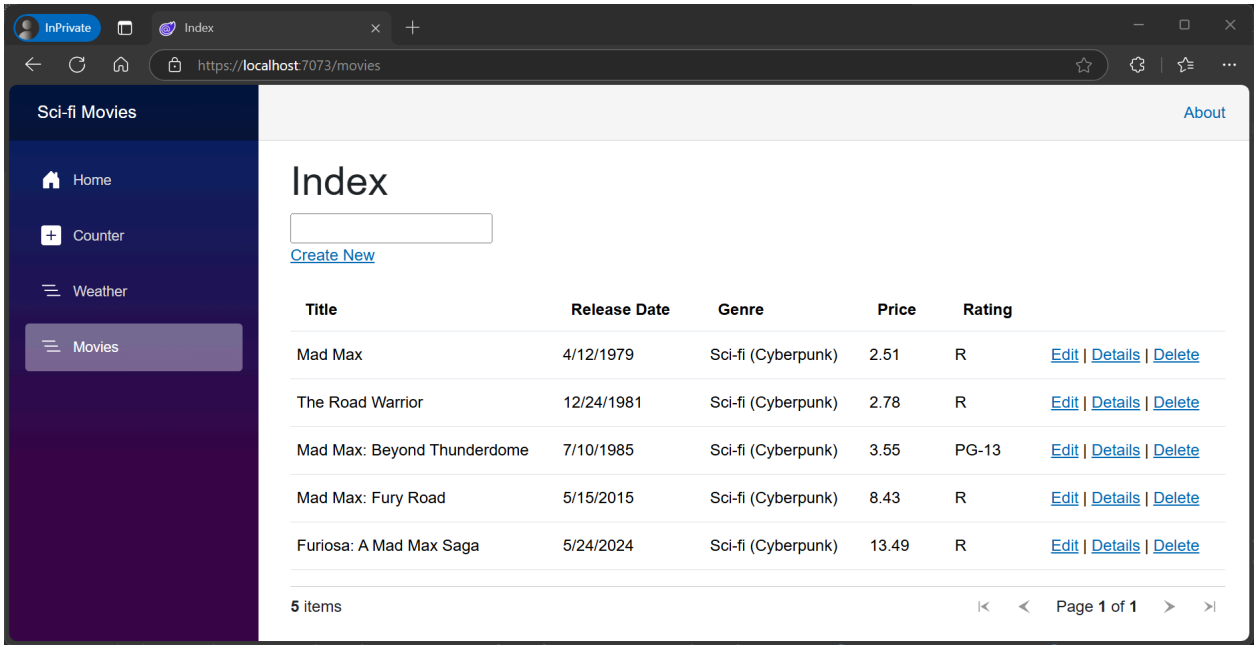
The `::deep` pseudo-element only works with descendant elements, so the `QuickGrid` component must be wrapped with a `<div>` or some other block-level element in order to apply the styles to it.

In `Components/Pages/MoviePages/Index.razor`, place `<div>` tags around the `QuickGrid` component:

diff

```
+ <div>
    <QuickGrid ...>
        ...
    </QuickGrid>
+ </div>
```

Blazor rewrites CSS selectors to match the markup rendered by the component. The rewritten CSS styles are bundled and produced as a static asset, so you don't need to take further action to apply the styles to the rendered `QuickGrid` component.



Clean up

When you're finished with the tutorial and delete the sample app from your local system, you can also delete the `BlazorWebAppMovies` database in Visual Studio's SQL Server Object Explorer (SSOX):

1. Access SSOX by selecting **View** > **SQL Server Object Explorer** from the menu bar.
2. Display the database list by selecting the triangles next to `SQL Server` > `(localdb)\MSSQLLocalDB` > `Databases`.
3. Right-click the `BlazorWebAppMovies` database in the list and select **Delete**.
4. Select the checkbox to close existing collections before selecting **OK**.

When deleting the database in SSOX, the database's physical files are removed from your Windows user folder.

Congratulations!

Congratulations on completing the tutorial series! We hope you enjoyed this tutorial on Blazor. Blazor offers many more features than we were able to cover in this series, and we invite you to explore the Blazor documentation, examples, and sample apps to learn more. *Happy coding with Blazor!*

Next steps

If you're new to Blazor, we recommend reading the following Blazor articles that cover important general information for Blazor development:

- ASP.NET Core Blazor
- ASP.NET Core Blazor supported platforms
- Tooling for ASP.NET Core Blazor
- ASP.NET Core Blazor hosting models
- ASP.NET Core Blazor project structure
- ASP.NET Core Blazor fundamentals and the other *Fundamentals* node articles.
- ASP.NET Core Razor components and the other *Components* node articles.
- ASP.NET Core Blazor forms overview
- ASP.NET Core Blazor JavaScript interoperability (JS interop) and the other *JavaScript interop* articles.
- ASP.NET Core Blazor authentication and authorization
- Host and deploy ASP.NET Core Blazor
- ASP.NET Core Blazor with Entity Framework Core (EF Core) covers concurrency with EF Core in Blazor apps.

In the documentation website's sidebar navigation, articles are organized by subject matter and laid out in roughly in a general-to-specific or basic-to-complex order. The best approach when starting to learn about Blazor is to read down the table of contents from top to bottom.

Troubleshoot with the completed sample

If you run into a problem while following the tutorial that you can't resolve from the text, compare your code to the completed project in the Blazor samples repository:

Blazor samples GitHub repository (dotnet/blazor-samples)

Select the latest version folder. The sample folder for this tutorial's project is named `BlazorWebAppMovies`.

Previous: Add a new field

Use ASP.NET Core SignalR with Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial provides a basic working experience for building a real-time app using SignalR with Blazor. This article is useful for developers who are already familiar with SignalR and are seeking to understand how to use SignalR in a Blazor app. For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets and the API documentation:

- Overview of ASP.NET Core SignalR
- ASP.NET Core Blazor
- .NET API browser

Learn how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

Prerequisites

Visual Studio

Visual Studio (latest release) with the **ASP.NET and web development** workload

Sample app

Downloading the tutorial's sample chat app isn't required for this tutorial. The sample app is the final, working app produced by following the steps of this tutorial. When you

open the samples repository, open the version folder that you plan to target and find the sample named `BlazorSignalRApp`.

View or download sample code (how to download)

Create a Blazor Web App

Follow the guidance for your choice of tooling:

Visual Studio

> **Note**
>
> Visual Studio 2022 or later and .NET Core SDK 8.0.0 or later are required.

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorSignalRApp` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.
- Confirm that the **Location** for the app is suitable. Leave the **Place solution and project in the same directory** checkbox selected. Select the **Next** button.
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Confirm that the latest framework is selected. If Visual Studio's **Framework** dropdown list doesn't include the latest available .NET framework, update Visual Studio and restart the tutorial.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **WebAssembly**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected
  - Select **Create**.

The guidance in this article uses a WebAssembly component for the SignalR client because it doesn't make sense to use SignalR to connect to a hub from an Interactive Server component in the same app, as that can lead to server port exhaustion.

Add the SignalR client library

Visual Studio

In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.

In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.

With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.

In the search results, select the latest release of the Microsoft.AspNetCore.SignalR.Client package. Select **Install**.

If the **Preview Changes** dialog appears, select **OK**.

If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the server `BlazorSignalRApp` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

C#

```
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Hubs;

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

Add services and an endpoint for the SignalR hub

Open the `Program` file of the server `BlazorSignalRApp` project.

Add the namespaces for Microsoft.AspNetCore.ResponseCompression and the `ChatHub` class to the top of the file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using BlazorSignalRApp.Hubs;
```

Add SignalR and Response Compression Middleware services:

C#

```
builder.Services.AddSignalR();

builder.Services.AddResponseCompression(opts =>
{
   opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
       ["application/octet-stream"]);
});
```

Use Response Compression Middleware at the top of the processing pipeline's configuration. Place the following line of code immediately after the line that builds the app (`var app = builder.Build();`):

C#

```
app.UseResponseCompression();
```

Add an endpoint for the hub immediately before the line that runs the app (`app.Run();`):

C#

```
app.MapHub<ChatHub>("/chathub");
```

Add Razor component code for chat

Add the following `Pages/Chat.razor` file to the `BlazorSignalRApp.Client` project:

razor

```razor
@page "/chat"
@rendermode InteractiveWebAssembly
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager Navigation
@implements IAsyncDisposable

<PageTitle>Chat</PageTitle>

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection? hubConnection;
    private List<string> messages = [];
    private string? userInput;
    private string? messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message)
=>
        {
            var encodedMsg = $"{user}: {message}";
            messages.Add(encodedMsg);
            InvokeAsync(StateHasChanged);
        });

        await hubConnection.StartAsync();
```

```
    }

    private async Task Send()
    {
        if (hubConnection is not null)
        {
            await hubConnection.SendAsync("SendMessage", userInput,
messageInput);
        }
    }

    public bool IsConnected =>
        hubConnection?.State == HubConnectionState.Connected;

    public async ValueTask DisposeAsync()
    {
        if (hubConnection is not null)
        {
            await hubConnection.DisposeAsync();
        }
    }
}
```

Add an entry to the `NavMenu` component to reach the chat page. In `Components/Layout/NavMenu.razor` immediately after the `<div>` block for the `Weather` component, add the following `<div>` block:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="chat">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Chat
    </NavLink>
</div>
```

> **Note**
>
> Disable Response Compression Middleware in the `Development` environment when using Hot Reload. For more information, see ASP.NET Core Blazor SignalR guidance.

Run the app

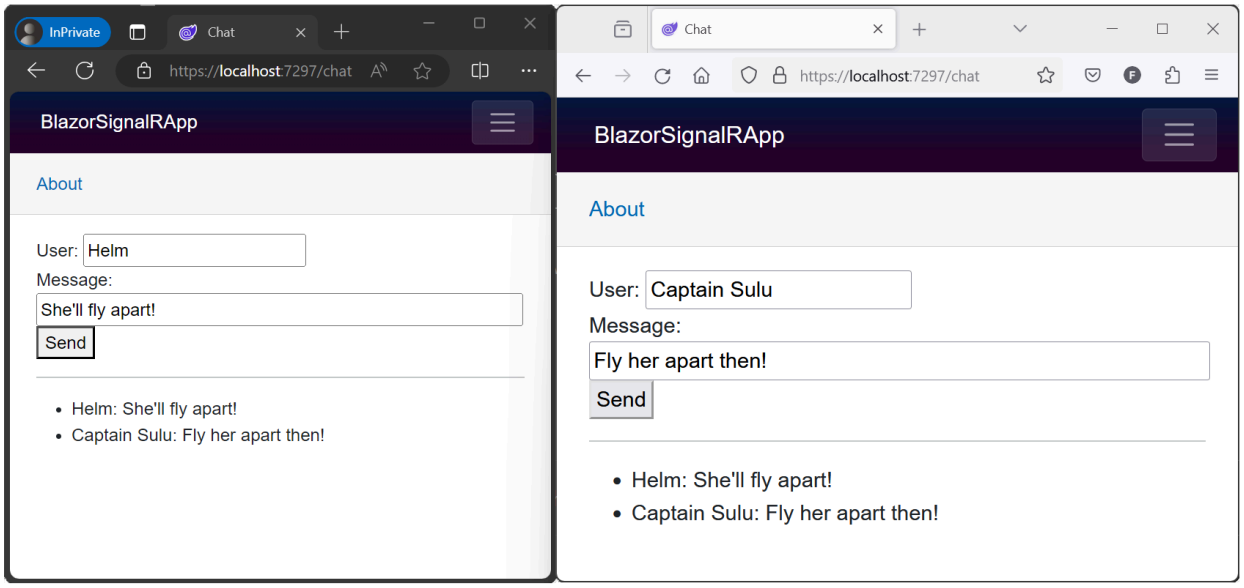
Follow the guidance for your tooling:

Visual Studio

With the server `BlazorSignalRApp` project selected in **Solution Explorer**, press `F5` to run the app with debugging or `Ctrl`+`F5` (Windows)/`⌘`+`F5` (macOS) to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* ©1991 Paramount

Next steps

In this tutorial, you learned how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets:

Overview of ASP.NET Core SignalR | ASP.NET Core Blazor

Additional resources

- Bearer token authentication with Identity Server, WebSockets, and Server-Sent Events
- Secure a SignalR hub in Blazor WebAssembly apps
- SignalR cross-origin negotiation for authentication
- SignalR configuration
- Debug ASP.NET Core Blazor apps
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor Hybrid

Article • 12/11/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains ASP.NET Core Blazor Hybrid, a way to build interactive client-side web UI with .NET in an ASP.NET Core app.

Use *Blazor Hybrid* to blend desktop and mobile native client frameworks with .NET and Blazor.

In a Blazor Hybrid app, Razor components run natively on the device. Components render to an embedded Web View control through a local interop channel. Components don't run in the browser, and WebAssembly isn't involved. Razor components load and execute code quickly, and components have full access to the native capabilities of the device through the .NET platform. Component styles rendered in a Web View are platform dependent and may require you to account for rendering differences across platforms using custom stylesheets.

Blazor Hybrid articles cover subjects pertaining to integrating Razor components into native client frameworks.

Blazor Hybrid apps with .NET MAUI

Blazor Hybrid support is built into the .NET Multi-platform App UI (.NET MAUI) framework. .NET MAUI includes the BlazorWebView control that permits rendering Razor components into an embedded Web View. By using .NET MAUI and Blazor together, you can reuse one set of web UI components across mobile, desktop, and web.

Blazor Hybrid apps with WPF and Windows Forms

Blazor Hybrid apps can be built with Windows Presentation Foundation (WPF) and Windows Forms. Blazor provides `BlazorWebView` controls for both of these frameworks (WPF BlazorWebView, Windows Forms BlazorWebView). Razor components run natively in the Windows desktop and render to an embedded Web View. Using Blazor in WPF

and Windows Forms enables you to add new UI to your existing Windows desktop apps that can be reused across platforms with .NET MAUI or on the web.

Web View configuration

Blazor Hybrid exposes the underlying Web View configuration for different platforms through events of the `BlazorWebView` control:

- `BlazorWebViewInitializing` provides access to the settings used to create the Web View on each platform, if settings are available.
- `BlazorWebViewInitialized` provides access to the Web View to allow further configuration of the settings.

Use the preferred patterns on each platform to attach event handlers to the events to execute your custom code.

API documentation:

- .NET MAUI
  - BlazorWebViewInitializing
  - BlazorWebViewInitialized
- WPF
  - BlazorWebViewInitializing
  - BlazorWebViewInitialized
- Windows Forms
  - BlazorWebViewInitializing
  - BlazorWebViewInitialized

Unhandled exceptions in Windows Forms and WPF apps

This section only applies to Windows Forms and WPF Blazor Hybrid apps.

Create a callback for `UnhandledException` on the System.AppDomain.CurrentDomain property. The following example uses a compiler directive to display a MessageBox that either alerts the user that an error has occurred or shows the error information to the developer. Log the error information in `error.ExceptionObject`.

C#

```
AppDomain.CurrentDomain.UnhandledException += (sender, error) =>
{
```

```
#if DEBUG
    MessageBox.Show(text: error.ExceptionObject.ToString(), caption:
"Error");
#else
    MessageBox.Show(text: "An error has occurred.", caption: "Error");
#endif

    // Log the error information (error.ExceptionObject)
};
```

Globalization and localization

This section only applies to .NET MAUI Blazor Hybrid apps.

.NET MAUI configures the CurrentCulture and CurrentUICulture based on the device's ambient information.

IStringLocalizer and other API in the Microsoft.Extensions.Localization namespace generally work as expected, along with globalization formatting, parsing, and binding that relies on the user's culture.

When dynamically changing the app culture at runtime, the app must be reloaded to reflect the change in culture, which takes care of rerendering the root component and passing the new culture to rerendered child components.

.NET's resource system supports embedding localized images (as blobs) into an app, but Blazor Hybrid can't display the embedded images in Razor components at this time. Even if a user reads an image's bytes into a Stream using ResourceManager, the framework doesn't currently support rendering the retrieved image in a Razor component.

For more information, see the following resources:

- Localization (.NET MAUI documentation)
- Blazor Image component to display images that are not accessible through HTTP endpoints (dotnet/aspnetcore #25274)

Access scoped services from native UI

BlazorWebView has a TryDispatchAsync method that calls a specified `Action<ServiceProvider>` asynchronously and passes in the scoped services available in Razor components. This enables code from the native UI to access scoped services such as NavigationManager:

C#

```
private async void MyMauiButtonHandler(object sender, EventArgs e)
{
    var wasDispatchCalled = await _blazorWebView.TryDispatchAsync(sp =>
    {
        var navMan = sp.GetRequiredService<NavigationManager>();
        navMan.CallSomeNavigationApi(...);
    });

    if (!wasDispatchCalled)
    {
        ...
    }
}
```

When `wasDispatchCalled` is `false`, consider what to do if the call wasn't dispatched. Generally, the dispatch shouldn't fail. If it fails, OS resources might be exhausted. If resources are exhausted, consider logging a message, throwing an exception, and perhaps alerting the user.

Additional resources

- ASP.NET Core Blazor Hybrid tutorials
- .NET Multi-platform App UI (.NET MAUI)
- Windows Presentation Foundation (WPF)
- Windows Forms

ASP.NET Core Blazor Hybrid tutorials

Article • 11/18/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

The following tutorials provide a basic working experience for building a Blazor Hybrid app:

- Build a .NET MAUI Blazor Hybrid app
- Build a .NET MAUI Blazor Hybrid app with a Blazor Web App
- Build a Windows Forms Blazor app
- Build a Windows Presentation Foundation (WPF) Blazor app

For an overview of Blazor and reference articles, see ASP.NET Core Blazor and the articles that follow it in the table of contents.

Build a .NET MAUI Blazor Hybrid app

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial shows you how to build and run a .NET MAUI Blazor Hybrid app. You learn how to:

- ✓ Create a .NET MAUI Blazor Hybrid app project in Visual Studio
- ✓ Run the app on Windows
- ✓ Run the app on an emulated mobile device in the Android Emulator

Prerequisites

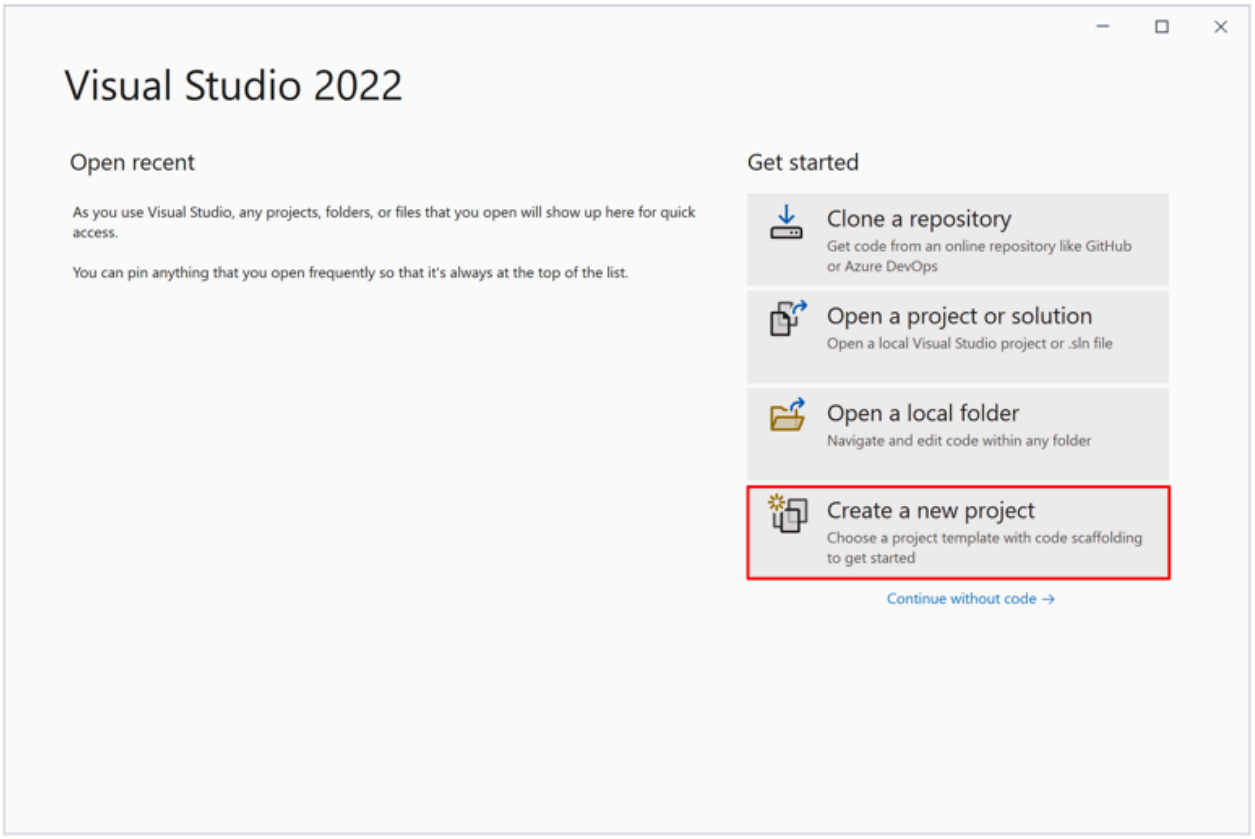
- Supported platforms (.NET MAUI documentation)
- Visual Studio with the **.NET Multi-platform App UI development** workload.
- Microsoft Edge WebView2: WebView2 is required on Windows when running a native app. When developing .NET MAUI Blazor Hybrid apps and only running them in Visual Studio's emulators, WebView2 isn't required.
- Enable hardware acceleration to improve the performance of the Android emulator.

For more information on prerequisites and installing software for this tutorial, see the following resources in the .NET MAUI documentation:

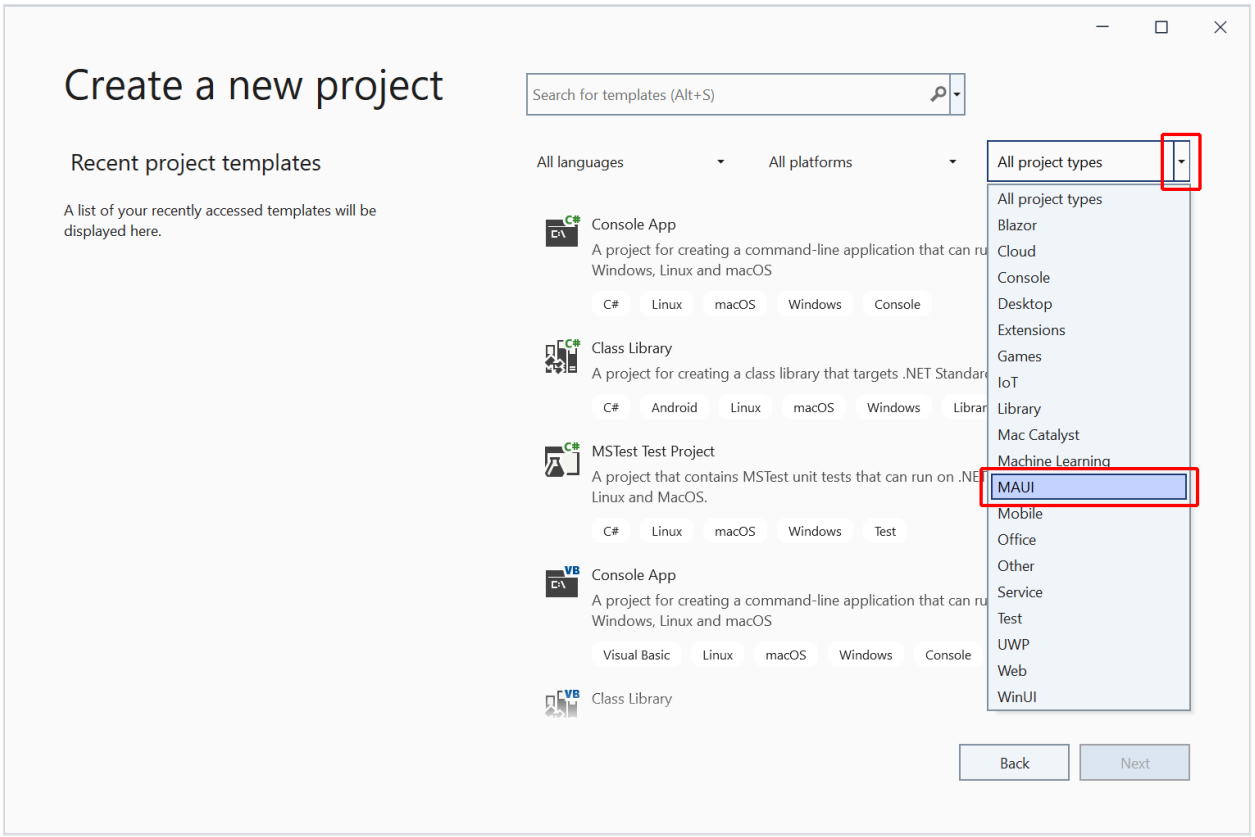
- Supported platforms for .NET MAUI apps
- Installation (Visual Studio)

Create a .NET MAUI Blazor Hybrid app

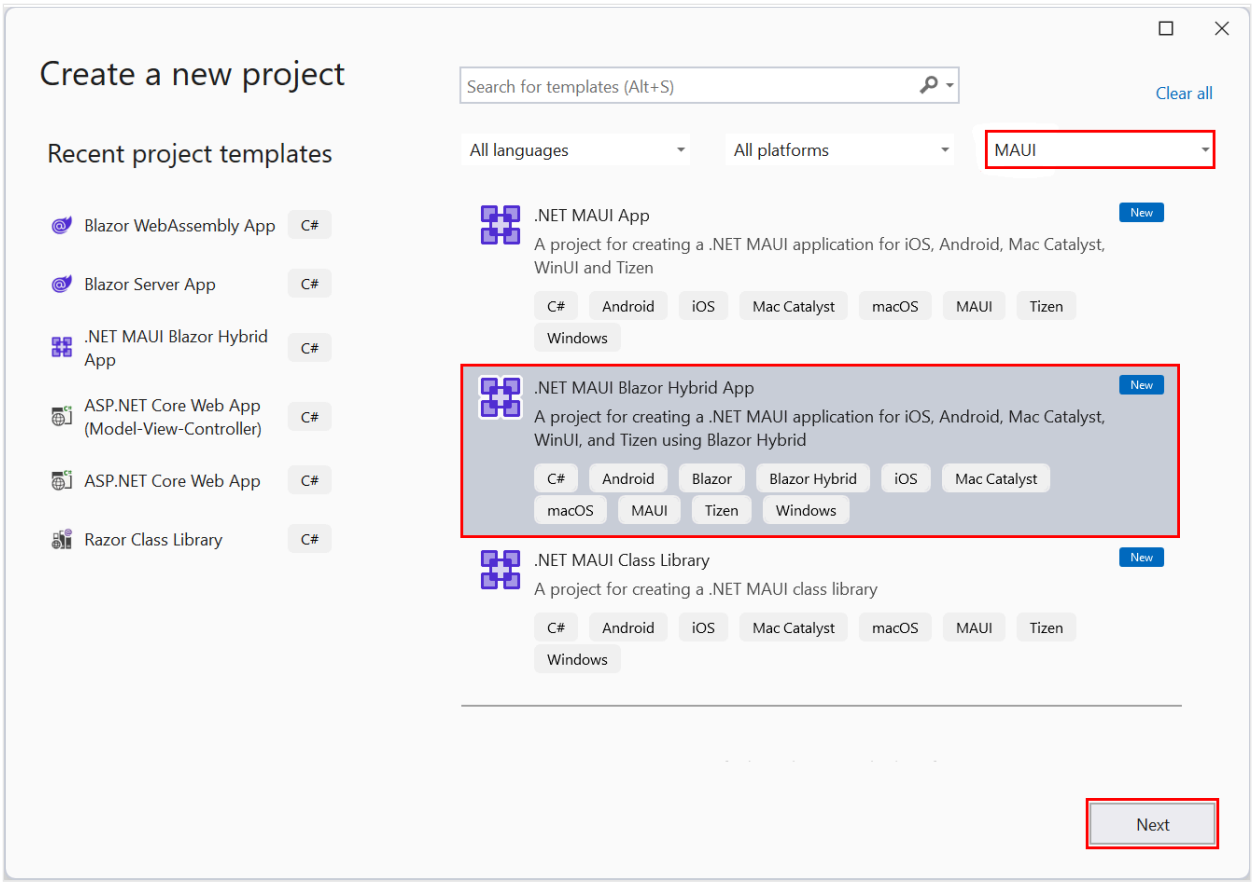
Launch Visual Studio. In the **Start Window**, select **Create a new project**:



In the **Create a new project** window, use the **Project type** dropdown to filter **MAUI** templates:



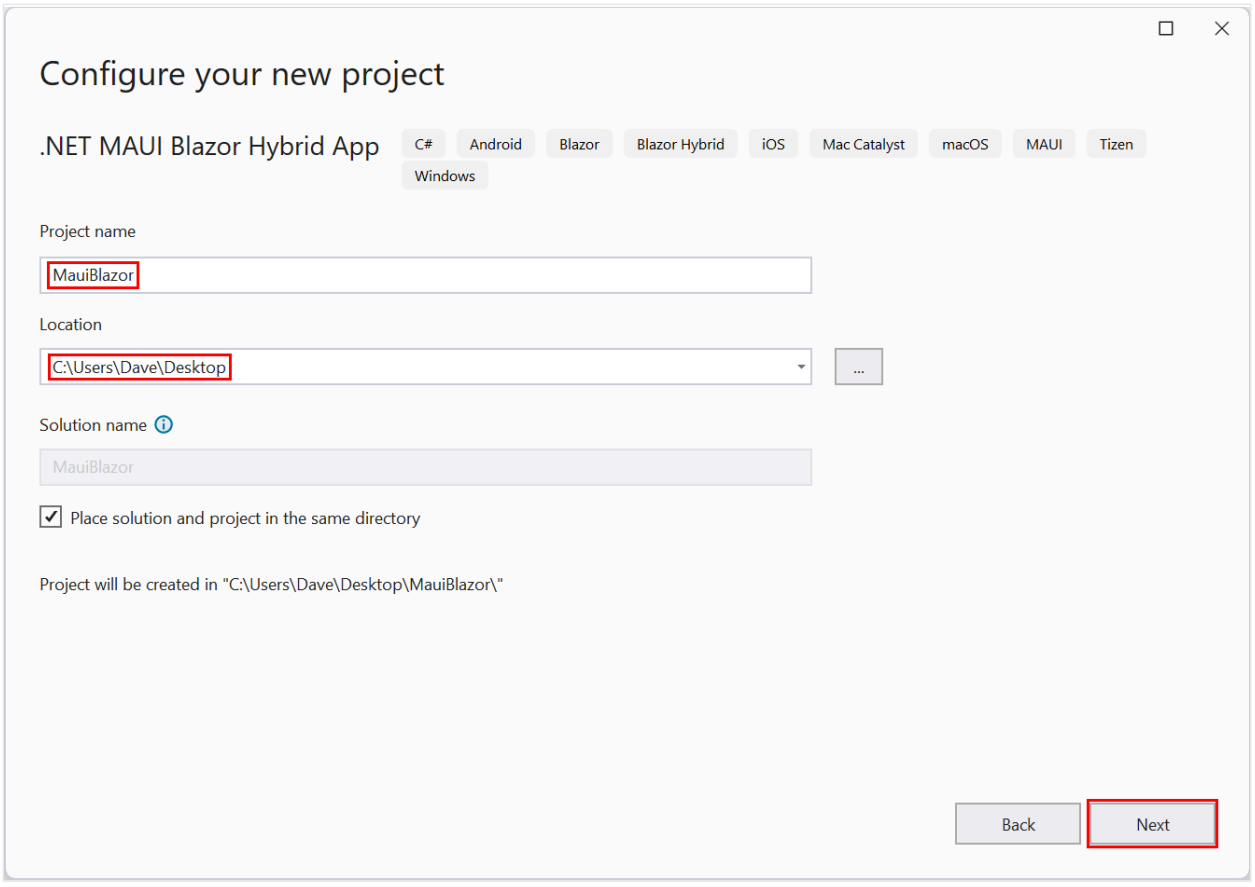
Select the **.NET MAUI Blazor Hybrid App** template and then select the **Next** button:



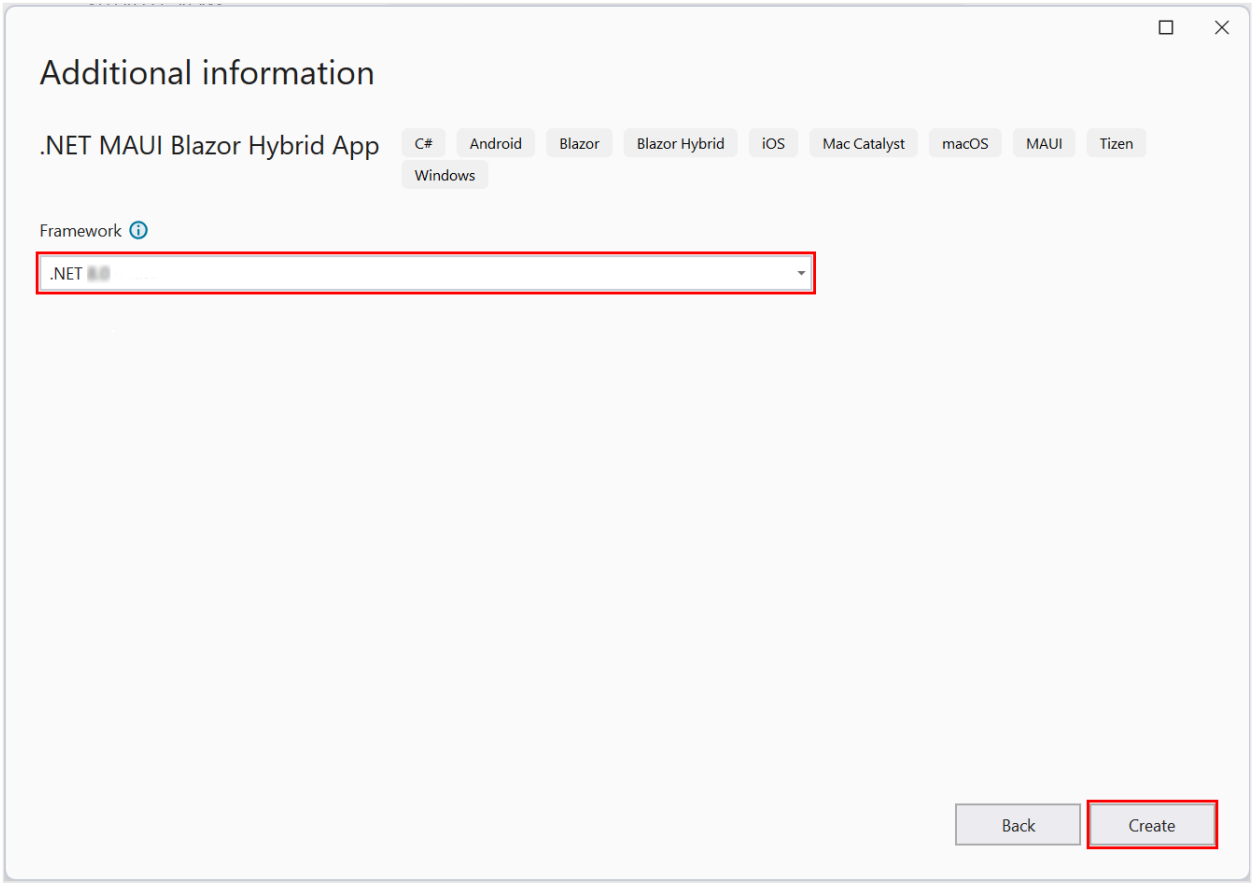
> **Note**
>
> In .NET 7 or earlier, the template is named **.NET MAUI Blazor App**.

In the **Configure your new project** dialog:

- Set the **Project name** to **MauiBlazor**.
- Choose a suitable location for the project.
- Select the **Next** button.

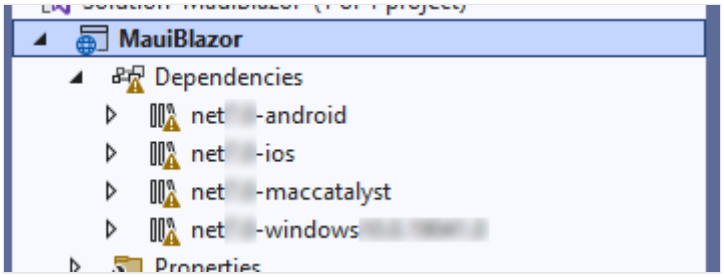


In the **Additional information** dialog, select the framework version with the **Framework** dropdown list. Select the **Create** button:

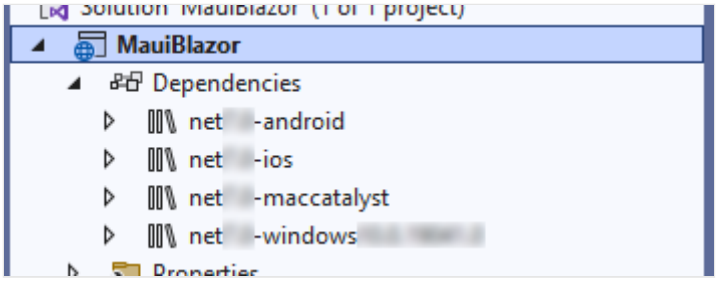


Wait for Visual Studio to create the project and restore the project's dependencies. Monitor the progress in **Solution Explorer** by opening the **Dependencies** entry.

Dependencies restoring:

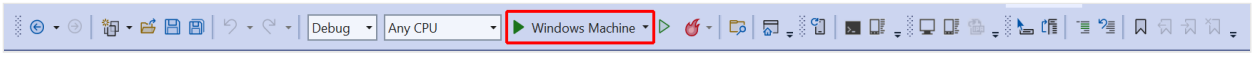


Dependencies restored:



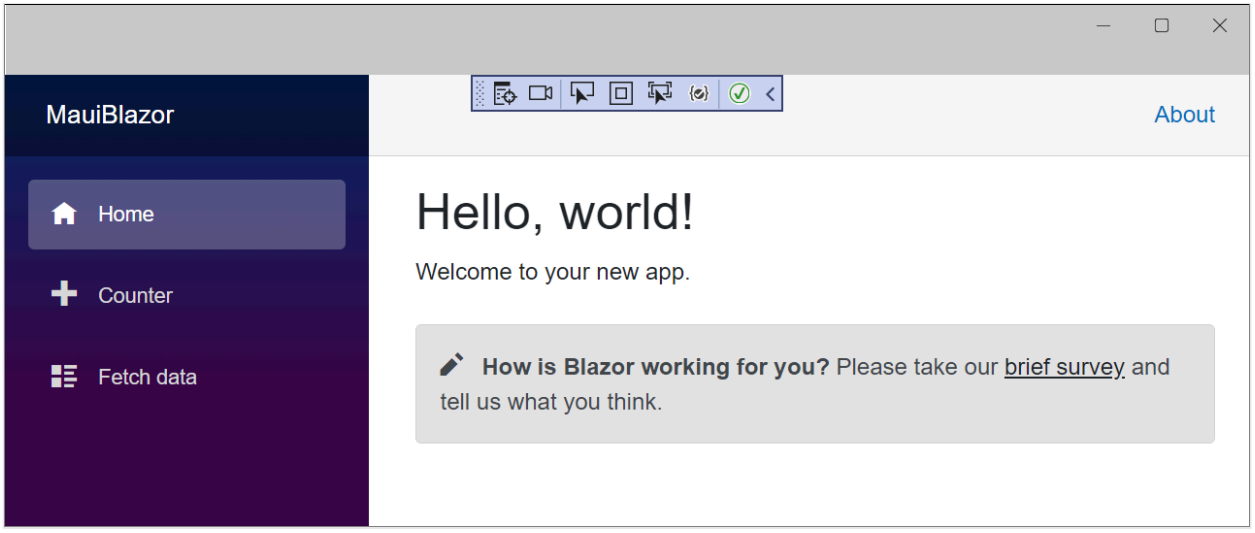
Run the app on Windows

In the Visual Studio toolbar, select the **Windows Machine** button to build and start the project:



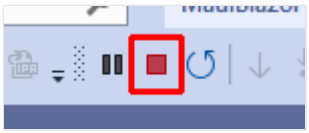
If Developer Mode isn't enabled, you're prompted to enable it in **Settings** > **For developers** > **Developer Mode** (Windows 10) or **Settings** > **Privacy & security** > **For developers** > **Developer Mode** (Windows 11). Set the switch to **On**.

The app running as a Windows desktop app:

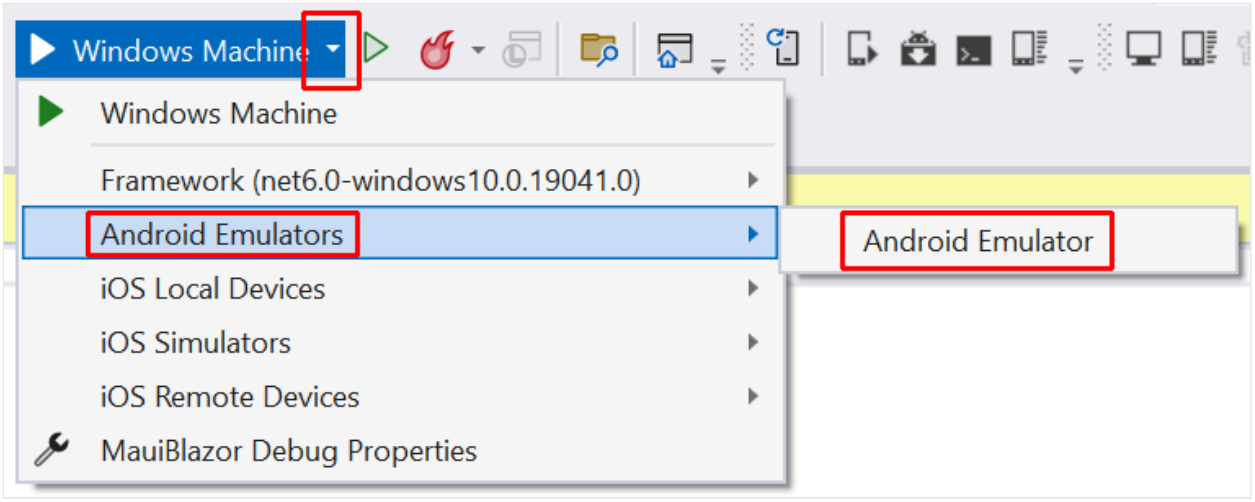


Run the app in the Android Emulator

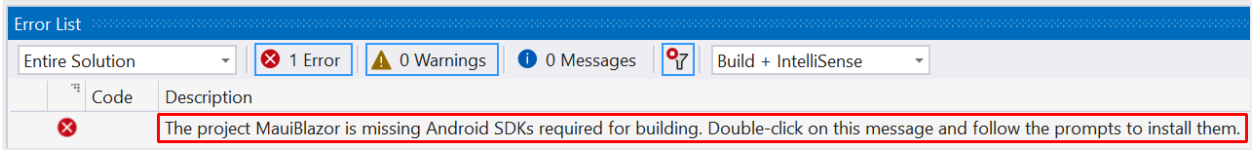
If you followed the guidance in the Run the app on Windows section, select the **Stop Debugging** button in the toolbar to stop the running Windows app:



In the Visual Studio toolbar, select the start configuration dropdown button. Select **Android Emulators** > **Android Emulator**:

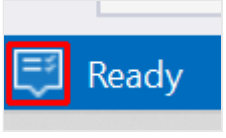


Android SDKs are required to build apps for Android. In the **Error List** panel, a message appears asking you to double-click the message to install the required Android SDKs:

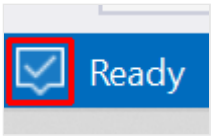


The **Android SDK License Acceptance** window appears, select the **Accept** button for each license that appears. An additional window appears for the **Android Emulator** and **SDK Patch Applier** licenses. Select the **Accept** button.

Wait for Visual Studio to download the Android SDK and Android Emulator. You can track the progress by selecting the background tasks indicator in the lower-left corner of the Visual Studio UI:



The indicator shows a checkmark when the background tasks are complete:



In the toolbar, select the **Android Emulator** button:



In the **Create a Default Android Device** window, select the **Create** button:



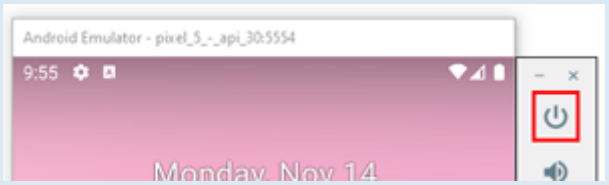
Wait for Visual Studio to download, unzip, and create an Android Emulator. When the Android phone emulator is ready, select the **Start** button.

> ⓘ **Note**
>
> Enable hardware acceleration to improve the performance of the Android emulator.

Close the **Android Device Manager** window. Wait until the emulated phone window appears, the Android OS loads, and the home screen appears.

> ⓘ **Important**
>
> The emulated phone must be powered on with the Android OS loaded in order to load and run the app in the debugger. If the emulated phone isn't running, turn on

the phone using either the `Ctrl`+`P` keyboard shortcut or by selecting the **Power** button in the UI:



In the Visual Studio toolbar, select the **Pixel 5 - {VERSION}** button to build and run the project, where the `{VERSION}` placeholder is the Android version. In the following example, the Android version is **API 30 (Android 11.0 - API 30)**, and a later version appears depending on the Android SDK installed:



Visual Studio builds the project and deploys the app to the emulator.

Starting the emulator, loading the emulated phone and OS, and deploying and running the app can take several minutes depending on the speed of the system and whether or not hardware acceleration is enabled. You can monitor the progress of the deployment by inspecting Visual Studio's status bar at the bottom of the UI. The **Ready** indicator receives a checkmark and the emulator's deployment and app loading indicators disappear when the app is running:

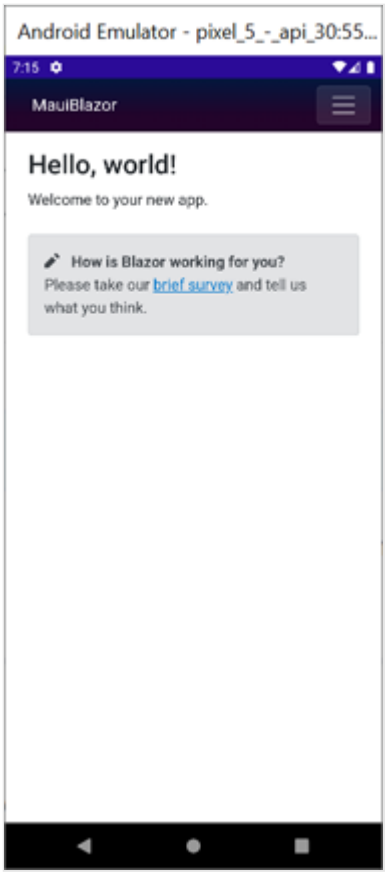
During deployment:



During app startup:



The app running in the Android Emulator:



Next steps

In this tutorial, you learned how to:

- ✓ Create a .NET MAUI Blazor Hybrid app project in Visual Studio
- ✓ Run the app on Windows
- ✓ Run the app on an emulated mobile device in the Android Emulator

Learn more about Blazor Hybrid apps:

ASP.NET Core Blazor Hybrid

Build a .NET MAUI Blazor Hybrid app with a Blazor Web App

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article shows you how to build a .NET MAUI Blazor Hybrid app with a Blazor Web App that uses a shared user interface via a Razor class library (RCL).

Prerequisites and preliminary steps

For prerequisites and preliminary steps, see Build a .NET MAUI Blazor Hybrid app. We recommend using the .NET MAUI Blazor Hybrid tutorial to set up your local system for .NET MAUI development before using the guidance in this article.

.NET MAUI Blazor Hybrid and Web App sample app

Obtain the sample app named `MauiBlazorWeb` from the Blazor samples GitHub repository (dotnet/blazor-samples) (.NET 8 or later).

The sample app is a starter solution that contains a .NET MAUI Blazor Hybrid (native, cross-platform) app, a Blazor Web App, and a Razor class library (RCL) that contains the shared UI (Razor components) used by the native and web apps.

Migrating a .NET MAUI Blazor Hybrid solution

Instead of using the sample app, you can migrate an existing .NET MAUI Blazor Hybrid app with the guidance in this section using Visual Studio.

Add new project to the solution with the **Blazor Web App** project template. Select the following options:

- **Project name**: Use the solution's name with `.Web` appended. The examples in this article assume the following naming:

    - Solution: `MauiBlazorWeb`
    - MAUI project: `MauiBlazorWeb.Maui`
    - Blazor Web App: `MauiBlazorWeb.Web`
    - Razor class library (RCL) (added in a later step): `MauiBlazorWeb.Shared`
- **Authentication type**: **None**
- **Configure for HTTPS**: Selected (enabled)
- **Interactive render mode**: **Server**
- **Interactive location**: **Global**
- **Sample pages**: Unselected (disabled)

The **Interactivity location** setting to **Global** is important because MAUI apps always run interactively and throw errors on Razor component pages that explicitly specify a render mode. If you don't use a global render mode, you must implement the approach described in the Use Blazor render modes section after following the guidance in this section. For more information, see BlazorWebView needs a way to enable overriding ResolveComponentForRenderMode (dotnet/aspnetcore #51235).

Add new **Razor class library** (RCL) project to the solution. The examples in this article assume that the project is named `MauiBlazorWeb.Shared`. Don't select **Support pages and views** when you add the project to the solution.

Add project references to the RCL from both MAUI project and the Blazor Web App project.

Move the `Components` folder and all of its contents from the MAUI project to the RCL. Confirm that the `Components` folder is deleted from the MAUI project.

> **Tip**
>
> When moving a folder or file in Visual Studio, either use keyboard commands or the shortcut menu by right-clicking for a cut and paste operation. Dragging the folder in Visual Studio only copies from one location to another, which requires an extra step to delete the original.

Move the `css` folder from the `wwwroot` folder of the MAUI project to the RCL's `wwwroot` folder.

Delete the following files from the RCL's `wwwroot` folder:

- `background.png`
- `exampleJsInterop.js`

In the RCL, replace the root `_Imports.razor` file with the one in the RCL's `Components` folder, overwriting the existing file in the RCL and deleting the original in the `Components` folder. After moving the file, open it and rename the last two `@using` statements to match the RCL's namespace. In the following example, the RCL's namespace is `MauiBlazorWeb.Shared`:

razor

```razor
@using MauiBlazorWeb.Shared
@using MauiBlazorWeb.Shared.Components
```

In the root of the RCL project, delete the following files:

- `Component1.razor`
- `ExampleJsInterop.cs`

In the RCL, open the `Components/Routes.razor` file and change `MauiProgram` to `Routes`:

diff

```diff
- <Router AppAssembly="@typeof(MauiProgram).Assembly">
+ <Router AppAssembly="@typeof(Routes).Assembly">
```

Open the `MainPage.xaml` file in the MAUI project. Add an `xmlns:shared` reference to the RCL in the ContentPage attributes. In the following example, the RCL's namespace is `MauiBlazorWeb.Shared`. Set the correct value for both the `clr-namespace` and the `assembly`:

XML

```
xmlns:shared="clr-
namespace:MauiBlazorWeb.Shared;assembly=MauiBlazorWeb.Shared"
```

Also in the `MainPage.xaml` file, update the BlazorWebView root component ComponentType from `local` to `shared`:

diff

```diff
- <RootComponent Selector="#app" ComponentType="{x:Type
local:Components.Routes}" />
+ <RootComponent Selector="#app" ComponentType="{x:Type
shared:Components.Routes}" />
```

In the MAUI project, open the `wwwroot/index.html` file and change stylesheets to point to the RCL's static asset path.

Remove the following lines:

```diff
- <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
- <link rel="stylesheet" href="css/app.css" />
```

Replace the preceding lines with the following markup. In the following example, the RCL's static asset path is `_content/MauiBlazorWeb.Shared/`:

```razor
<link rel="stylesheet"
href="_content/MauiBlazorWeb.Shared/css/bootstrap/bootstrap.min.css" />
<link rel="stylesheet" href="_content/MauiBlazorWeb.Shared/css/app.css" />
```

In the Blazor Web App, open the `_Imports.razor` file and add the following two `@using` statements for the RCL. In the following example, the RCL's namespace is `MauiBlazorWeb.Shared`:

```razor
@using MauiBlazorWeb.Shared
@using MauiBlazorWeb.Shared.Components
```

In the Blazor Web App project, open the `App` component (`Components/App.razor`). Remove the `app.css` stylesheet:

```diff
- <link rel="stylesheet" href="app.css" />
```

Replace the preceding line with the RCL's static asset stylesheet references. In the following example, the RCL's static asset path is `_content/MauiBlazorWeb.Shared/`:

```
<link rel="stylesheet"
href="_content/MauiBlazorWeb.Shared/css/bootstrap/bootstrap.min.css" />
<link rel="stylesheet" href="_content/MauiBlazorWeb.Shared/css/app.css" />
```

In the Blazor Web App project, delete the following folder and files:

- `Components/Layout` folder
- `Components/Routes.razor`
- `Components/Pages/Home.razor`
- `wwwroot/app.css`

Open the Blazor Web App's `Program.cs` file and add an additional assembly for the RCL to the app. In the following example, the RCL's namespace is `MauiBlazorWeb.Shared`:

```
C#

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode()

.AddAdditionalAssemblies(typeof(MauiBlazorWeb.Shared._Imports).Assembly);
```

Run the MAUI project by selecting the project in **Solution Explorer** and using Visual Studio's start button.

Run the Blazor Web App project by selecting the Blazor Web App project in **Solution Explorer** and using Visual Studio's start button with the `https` build configuration.

If you receive a build error that the RCL's assembly can't be resolved, build the RCL project first. If any MAUI project resource errors occur on build, rebuild the MAUI project to clear the errors.

Use Blazor render modes

Use the guidance in one of the following subsections that matches your app's specifications for applying Blazor render modes for a given interactivity location in the Blazor Web App but ignore the render mode assignments in the MAUI project.

Render mode and interactivity specification subsections:

- Global Server interactivity
- Global Auto or WebAssembly interactivity
- Per-page/component Server interactivity
- Per-page/component Auto interactivity
- Per-page/component WebAssembly interactivity

Global Server interactivity

- Interactive render mode: **Server**
- Interactivity location: **Global**
- Solution projects
  - MAUI (`MauiBlazorWeb.Maui`)
  - Blazor Web App (`MauiBlazorWeb.Web`)
  - RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components without setting render modes in each component.

Project references: `MauiBlazorWeb.Maui` and `MauiBlazorWeb.Web` have a project reference to `MauiBlazorWeb.Shared`.

Global Auto or WebAssembly interactivity

- Interactive render mode: **Auto** or **WebAssembly**
- Interactivity location: **Global**
- Solution projects
  - MAUI (`MauiBlazorWeb.Maui`)
  - Blazor Web App
    - Server project: `MauiBlazorWeb.Web`
    - Client project: `MauiBlazorWeb.Web.Client`
  - RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components without setting render modes in each component.

Project references:

- `MauiBlazorWeb.Maui`, `MauiBlazorWeb.Web`, and `MauiBlazorWeb.Web.Client` projects have a project reference to `MauiBlazorWeb.Shared`.
- `MauiBlazorWeb.Web` has a project reference to `MauiBlazorWeb.Web.Client`.

Per-page/component Server interactivity

- Interactive render mode: **Server**
- Interactivity location: **Per-page/component**
- Solution projects
  - MAUI (`MauiBlazorWeb.Maui`): Calls `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes` in `MauiProgram.cs`.
  - Blazor Web App (`MauiBlazorWeb.Web`): Doesn't set an `@rendermode` directive attribute on the `HeadOutlet` and `Routes` components of the `App` component (`Components/App.razor`).

- RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components that set the `InteractiveServer` render mode in each component.

`MauiBlazorWeb.Maui` and `MauiBlazorWeb.Web` have a project reference to `MauiBlazorWeb.Shared`.

Add the following `InteractiveRenderSettings` class to the RCL. The class properties are used to set component render modes.

The MAUI project is interactive by default, so no action is taken at the project level in the MAUI project other than calling `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes`.

For the Blazor Web App on the web client, the property values are assigned from RenderMode. When the components are loaded into a BlazorWebView for the MAUI project's native client, the render modes are unassigned (`null`) because the MAUI project explicitly sets the render mode properties to `null` when `ConfigureBlazorHybridRenderModes` is called.

`InteractiveRenderSettings.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace MauiBlazorWeb.Shared;

public static class InteractiveRenderSettings
{
    public static IComponentRenderMode? InteractiveServer { get; set; } =
        RenderMode.InteractiveServer;
    public static IComponentRenderMode? InteractiveAuto { get; set; } =
        RenderMode.InteractiveAuto;
    public static IComponentRenderMode? InteractiveWebAssembly { get; set; }
=
        RenderMode.InteractiveWebAssembly;

    public static void ConfigureBlazorHybridRenderModes()
    {
        InteractiveServer = null;
        InteractiveAuto = null;
        InteractiveWebAssembly = null;
    }
}
```

In `MauiProgram.CreateMauiApp` of `MauiProgram.cs`, call `ConfigureBlazorHybridRenderModes`:

```
C#

InteractiveRenderSettings.ConfigureBlazorHybridRenderModes();
```

In the `_Imports.razor` file of the `.Shared` RCL, replace the `@using` statement for Microsoft.AspNetCore.Components.Web.RenderMode with an `@using` statement for `InteractiveRenderSettings` to make the properties of the `InteractiveRenderSettings` class available to components:

```
diff

- @using static Microsoft.AspNetCore.Components.Web.RenderMode
+ @using static InteractiveRenderSettings
```

> **Note**
>
> The assignment of render modes via the RCL's `InteractiveRenderSettings` class properties differs from a typical standalone Blazor Web App. In a Blazor Web App, the render modes are normally provided by RenderMode in the Blazor Web App's `_Import` file.

Per-page/component Auto interactivity

- Interactive render mode: **Auto**
- Interactivity location: **Per-page/component**
- Solution projects
  - MAUI (`MauiBlazorWeb.Maui`): Calls `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes` in `MauiProgram.cs`.
  - Blazor Web App
    - Server project: `MauiBlazorWeb.Web`: Doesn't set an `@rendermode` directive attribute on the `HeadOutlet` and `Routes` components of the `App` component (`Components/App.razor`).
    - Client project: `MauiBlazorWeb.Web.Client`
  - RCL (`MauiBlazorWeb.Shared`): Contains the shared Razor components that set the `InteractiveAuto` render mode in each component.

Project references:

- `MauiBlazorWeb.Maui`, `MauiBlazorWeb.Web`, and `MauiBlazorWeb.Web.Client` have a project reference to `MauiBlazorWeb.Shared`.
- `MauiBlazorWeb.Web` has a project reference to `MauiBlazorWeb.Web.Client`.

Add the following `InteractiveRenderSettings` class is added to the RCL. The class properties are used to set component render modes.

The MAUI project is interactive by default, so no action is taken at the project level in the MAUI project other than calling `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes`.

For the Blazor Web App on the web client, the property values are assigned from RenderMode. When the components are loaded into a BlazorWebView for the MAUI project's native client, the render modes are unassigned (`null`) because the MAUI project explicitly sets the render mode properties to `null` when `ConfigureBlazorHybridRenderModes` is called.

`InteractiveRenderSettings.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace MauiBlazorWeb.Shared;

public static class InteractiveRenderSettings
{
    public static IComponentRenderMode? InteractiveServer { get; set; } =
        RenderMode.InteractiveServer;
    public static IComponentRenderMode? InteractiveAuto { get; set; } =
        RenderMode.InteractiveAuto;
    public static IComponentRenderMode? InteractiveWebAssembly { get; set; }
=
        RenderMode.InteractiveWebAssembly;

    public static void ConfigureBlazorHybridRenderModes()
    {
        InteractiveServer = null;
        InteractiveAuto = null;
        InteractiveWebAssembly = null;
    }
}
```

In `MauiProgram.CreateMauiApp` of `MauiProgram.cs`, call `ConfigureBlazorHybridRenderModes`:

```
C#

InteractiveRenderSettings.ConfigureBlazorHybridRenderModes();
```

In the `_Imports.razor` file of the `.Shared` RCL, replace the `@using` statement for Microsoft.AspNetCore.Components.Web.RenderMode with an `@using` statement for `InteractiveRenderSettings` to make the properties of the `InteractiveRenderSettings` class available to components:

```
diff

- @using static Microsoft.AspNetCore.Components.Web.RenderMode
+ @using static InteractiveRenderSettings
```

> **Note**
>
> The assignment of render modes via the RCL's `InteractiveRenderSettings` class properties differs from a typical standalone Blazor Web App. In a Blazor Web App, the render modes are normally provided by RenderMode in the Blazor Web App's `_Import` file.

Per-page/component WebAssembly interactivity

- Interactive render mode: **WebAssembly**
- Interactivity location: **Per-page/component**
- Solution projects
  - MAUI (`MauiBlazorWeb.Maui`)
  - Blazor Web App
    - Server project: `MauiBlazorWeb.Web`: Doesn't set an `@rendermode` directive attribute on the `HeadOutlet` and `Routes` components of the `App` component (`Components/App.razor`).
    - Client project: `MauiBlazorWeb.Web.Client`
  - RCLs
    - `MauiBlazorWeb.Shared`
    - `MauiBlazorWeb.Shared.Client`: Contains the shared Razor components that set the `InteractiveWebAssembly` render mode in each component. The

`.Shared.Client` RCL is maintained separately from the `.Shared` RCL because the app should maintain the components that are required to run on WebAssembly separately from the components that run on server and that stay on the server.

Project references:

- `MauiBlazorWeb.Maui` and `MauiBlazorWeb.Web` have project references to `MauiBlazorWeb.Shared`.
- `MauiBlazorWeb.Web` has a project reference to `MauiBlazorWeb.Web.Client`.
- `MauiBlazorWeb.Web.Client` and `MauiBlazorWeb.Shared` have a project reference to `MauiBlazorWeb.Shared.Client`.

Add the following AdditionalAssemblies parameter to the `Router` component instance for the `MauiBlazorWeb.Shared.Client` project assembly (via its `_Imports` file) in the `MauiBlazorWeb.Shared` project's `Routes.razor` file:

razor

```
<Router AppAssembly="@typeof(Routes).Assembly"
        AdditionalAssemblies="new [] {
typeof(MauiBlazorWeb.Shared.Client._Imports).Assembly }">
    <Found Context="routeData">
        <RouteView RouteData="@routeData"
DefaultLayout="@typeof(Components.Layout.MainLayout)" />
        <FocusOnNavigate RouteData="@routeData" Selector="h1" />
    </Found>
</Router>
```

Add the `MauiBlazorWeb.Shared.Client` project assembly (via its `_Imports` file) with the following AddAdditionalAssemblies call in the `MauiBlazorWeb.Web` project's `Program.cs` file:

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof(MauiBlazorWeb.Shared._Imports).Assembly)

.AddAdditionalAssemblies(typeof(MauiBlazorWeb.Shared.Client._Imports).Assemb
ly);
```

Add the following `InteractiveRenderSettings` class is added to the `.Shared.Client` RCL. The class properties are used to set component render modes for server-based components.

The MAUI project is interactive by default, so no action is taken at the project level in the MAUI project other than calling `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes`.

For the Blazor Web App on the web client, the property values are assigned from RenderMode. When the components are loaded into a BlazorWebView for the MAUI project's native client, the render modes are unassigned (`null`) because the MAUI project explicitly sets the render mode properties to `null` when `ConfigureBlazorHybridRenderModes` is called.

`InteractiveRenderSettings.cs` (`.Shared.Client` RCL):

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace MauiBlazorWeb.Shared;

public static class InteractiveRenderSettings
{
    public static IComponentRenderMode? InteractiveServer { get; set; } =
        RenderMode.InteractiveServer;
    public static IComponentRenderMode? InteractiveAuto { get; set; } =
        RenderMode.InteractiveAuto;
    public static IComponentRenderMode? InteractiveWebAssembly { get; set; }
=
        RenderMode.InteractiveWebAssembly;

    public static void ConfigureBlazorHybridRenderModes()
    {
        InteractiveServer = null;
        InteractiveAuto = null;
        InteractiveWebAssembly = null;
    }
}
```

A slightly different version of the `InteractiveRenderSettings` class is added to the `.Shared` RCL. In the class added to the `.Shared` RCL, `InteractiveRenderSettings.ConfigureBlazorHybridRenderModes` of the `.Shared.Client` RCL is called. This ensures that the render mode of WebAssembly components rendered on the MAUI client are unassigned (`null`) because they're interactive by default on the native client.

`InteractiveRenderSettings.cs` (`.Shared` RCL):

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace MauiBlazorWeb.Shared
{
    public static class InteractiveRenderSettings
    {
        public static IComponentRenderMode? InteractiveServer { get; set; }
=
            RenderMode.InteractiveServer;
        public static IComponentRenderMode? InteractiveAuto { get; set; } =
            RenderMode.InteractiveAuto;
        public static IComponentRenderMode? InteractiveWebAssembly { get;
set; } =
            RenderMode.InteractiveWebAssembly;

        public static void ConfigureBlazorHybridRenderModes()
        {
            InteractiveServer = null;
            InteractiveAuto = null;
            InteractiveWebAssembly = null;
            MauiBlazorWeb.Shared.Client.InteractiveRenderSettings
                .ConfigureBlazorHybridRenderModes();
        }
    }
}
```

In `MauiProgram.CreateMauiApp` of `MauiProgram.cs`, call `ConfigureBlazorHybridRenderModes`:

C#

```
InteractiveRenderSettings.ConfigureBlazorHybridRenderModes();
```

In the `_Imports.razor` file of the `.Shared.Client` RCL, replace the `@using` statement for Microsoft.AspNetCore.Components.Web.RenderMode with an `@using` statement for `InteractiveRenderSettings` to make the properties of the `InteractiveRenderSettings` class available to components:

diff

```
- @using static Microsoft.AspNetCore.Components.Web.RenderMode
+ @using static InteractiveRenderSettings
```

ⓘ Note

The assignment of render modes via the RCL's `InteractiveRenderSettings` class properties differs from a typical standalone Blazor Web App. In a Blazor Web App, the render modes are normally provided by RenderMode via in the Blazor Web App's `_Import` file.

Using interfaces to support different device implementations

The following example demonstrates how to use an interface to call into different implementations across the web app and the native (MAUI) app. The following example creates a component that displays the device form factor. Use the MAUI abstraction layer for native apps and provide an implementation for the web app.

In the Razor class library (RCL), an `Interfaces` folder contains an `IFormFactor` interface.

`Interfaces/IFormFactor.cs`:

C#

```
namespace MauiBlazorWeb.Shared.Interfaces;

public interface IFormFactor
{
    public string GetFormFactor();
    public string GetPlatform();
}
```

The following `DeviceFormFactor` component is present in the RCL's `Components` folder.

`Components/Pages/DeviceFormFactor.razor`:

razor

```
@page "/device-form-factor"
@using MauiBlazorWeb.Shared.Interfaces
@inject IFormFactor FormFactor

<PageTitle>Form Factor</PageTitle>

<h1>Device Form Factor</h1>

<p>You are running on:</p>

<ul>
    <li>Form Factor: @factor</li>
```

```
    <li>Platform: @platform</li>
</ul>

<p>
    <em>This component is defined in the MauiBlazorWeb.Shared library.</em>
</p>

@code {
    private string factor => FormFactor.GetFormFactor();
    private string platform => FormFactor.GetPlatform();
}
```

In the RCL, an entry for the `DeviceFormFactor` component is part of the navigation menu in the `NavMenu` component.

In `Components/Layout/NavMenu.razor`:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="device-form-factor">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Form Factor
    </NavLink>
</div>
```

The web and native apps contain the implementations for `IFormFactor`.

In the Blazor Web App, a folder named `Services` contains the following `FormFactor.cs` file with the `FormFactor` implementation for web app use.

`Services/FormFactor.cs` (`MauiBlazorWeb.Web` project):

C#

```
using MauiBlazorWeb.Shared.Interfaces;

namespace MauiBlazorWeb.Web.Services;

public class FormFactor : IFormFactor
{
    public string GetFormFactor() => "Web";

    public string GetPlatform() => Environment.OSVersion.ToString();
}
```

In the MAUI project, a folder named `Services` contains the following `FormFactor.cs` file with the `FormFactor` implementation for native use. The MAUI abstractions layer is used

to write code that works on all native device platforms.

`Services/FormFactor.cs` (`MauiBlazorWeb.Maui` project):

```
C#

using MauiBlazorWeb.Shared.Interfaces;

namespace MauiBlazorWeb.Maui.Services;

public class FormFactor : IFormFactor
{
    public string GetFormFactor() => DeviceInfo.Idiom.ToString();

    public string GetPlatform() =>
        DeviceInfo.Platform.ToString() + " - " + DeviceInfo.VersionString;
}
```

Dependency injection is used to obtain the implementations of these services.

In the MAUI project, the `MauiProgram.cs` file has following `using` statements at the top of the file:

```
C#

using MauiBlazorWeb.Maui.Services;
using MauiBlazorWeb.Shared.Interfaces;
```

Immediately before the call to `builder.Build()`, `FormFactor` is registered to add device-specific services used by the RCL:

```
C#

builder.Services.AddSingleton<IFormFactor, FormFactor>();
```

In the Blazor Web App, the `Program` file has the following `using` statements at the top of the file:

```
C#

using MauiBlazorWeb.Shared.Interfaces;
using MauiBlazorWeb.Web.Services;
```

Immediately before the call to `builder.Build()`, `FormFactor` is registered to add device-specific services used by the Blazor Web App:

C#

```
builder.Services.AddScoped<IFormFactor, FormFactor>();
```

If the solution also targets WebAssembly via a `.Web.Client` project, an implementation of the preceding API is also required in the `.Web.Client` project.

You can also use compiler preprocessor directives in your RCL to implement different UI depending on the device the app is running on. For this scenario, the app must multi-target the RCL just like the MAUI app does. For an example, see the BethMassi/BethTimeUntil GitHub repository.

Additional resources

- ASP.NET Core Blazor Hybrid authentication and authorization
- ASP.NET Core Blazor render modes
- Reuse Razor components in ASP.NET Core Blazor Hybrid apps

Build a Windows Forms Blazor app

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial shows you how to build and run a Windows Forms Blazor app. You learn how to:

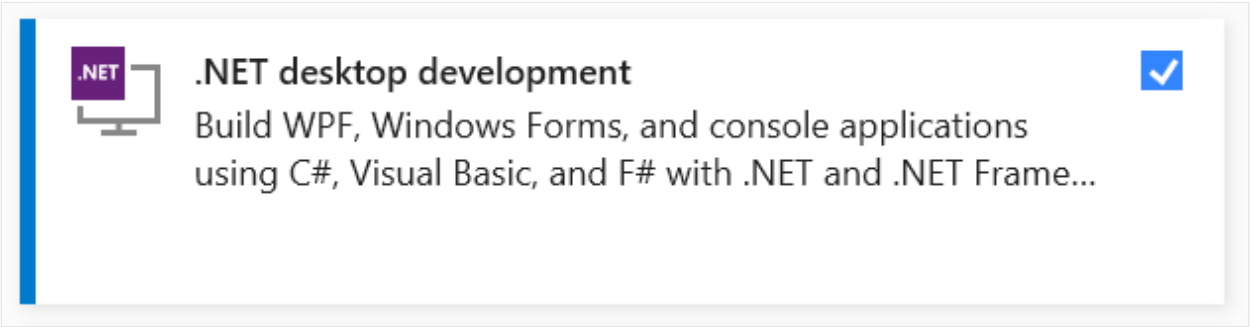
- ✓ Create a Windows Forms Blazor app project
- ✓ Run the app on Windows

Prerequisites

- Supported platforms (Windows Forms documentation)
- Visual Studio 2022 with the **.NET desktop development** workload

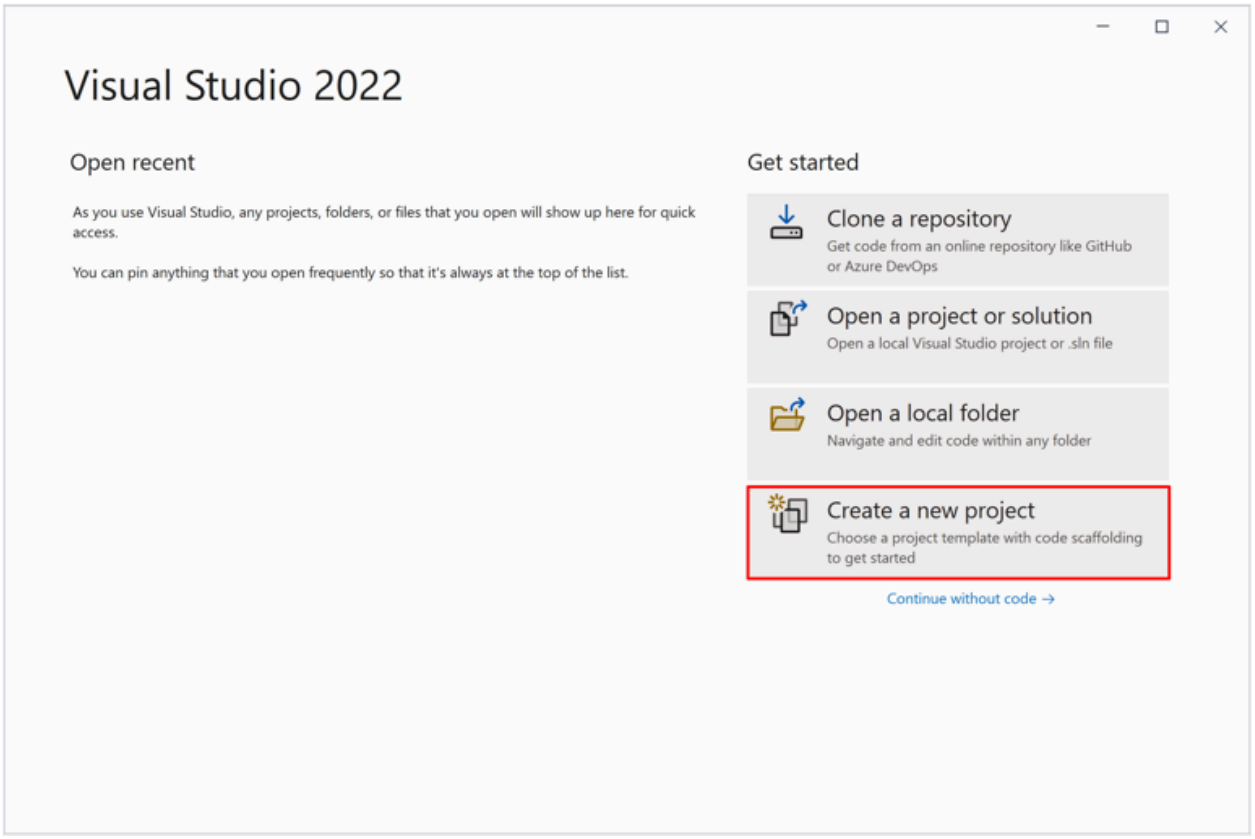
Visual Studio workload

If the **.NET desktop development** workload isn't installed, use the Visual Studio installer to install the workload. For more information, see Modify Visual Studio workloads, components, and language packs.

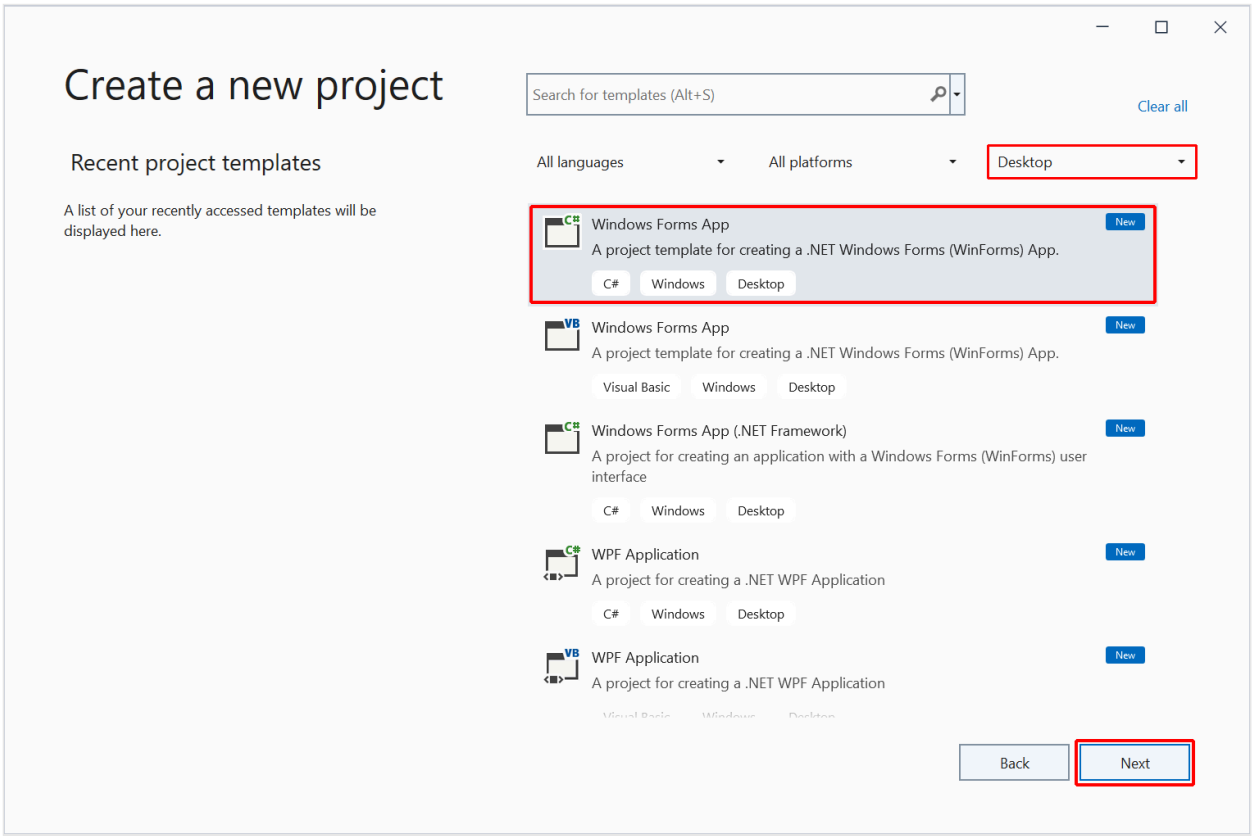


Create a Windows Forms Blazor project

Launch Visual Studio. In the **Start Window**, select **Create a new project**:



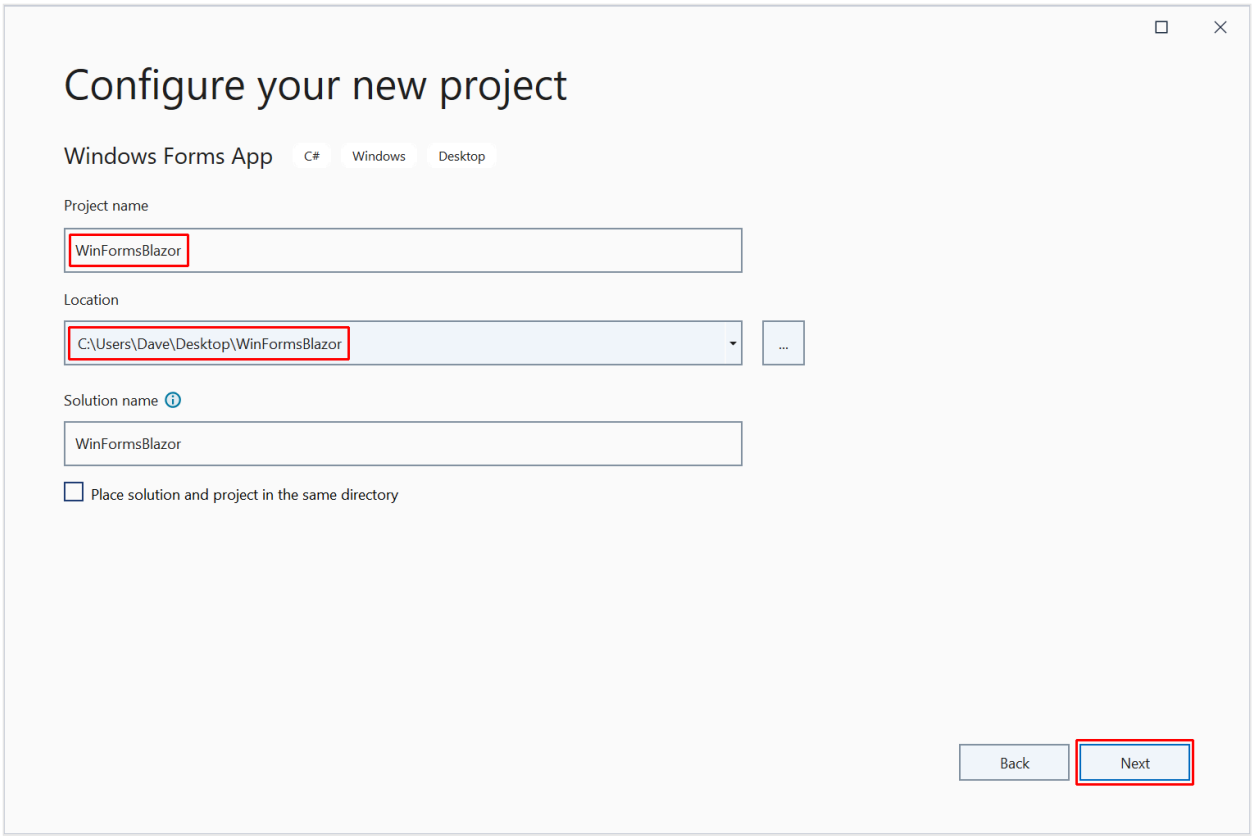
In the **Create a new project** dialog, filter the **Project type** dropdown to **Desktop**. Select the C# project template for **Windows Forms App** and select the **Next** button:



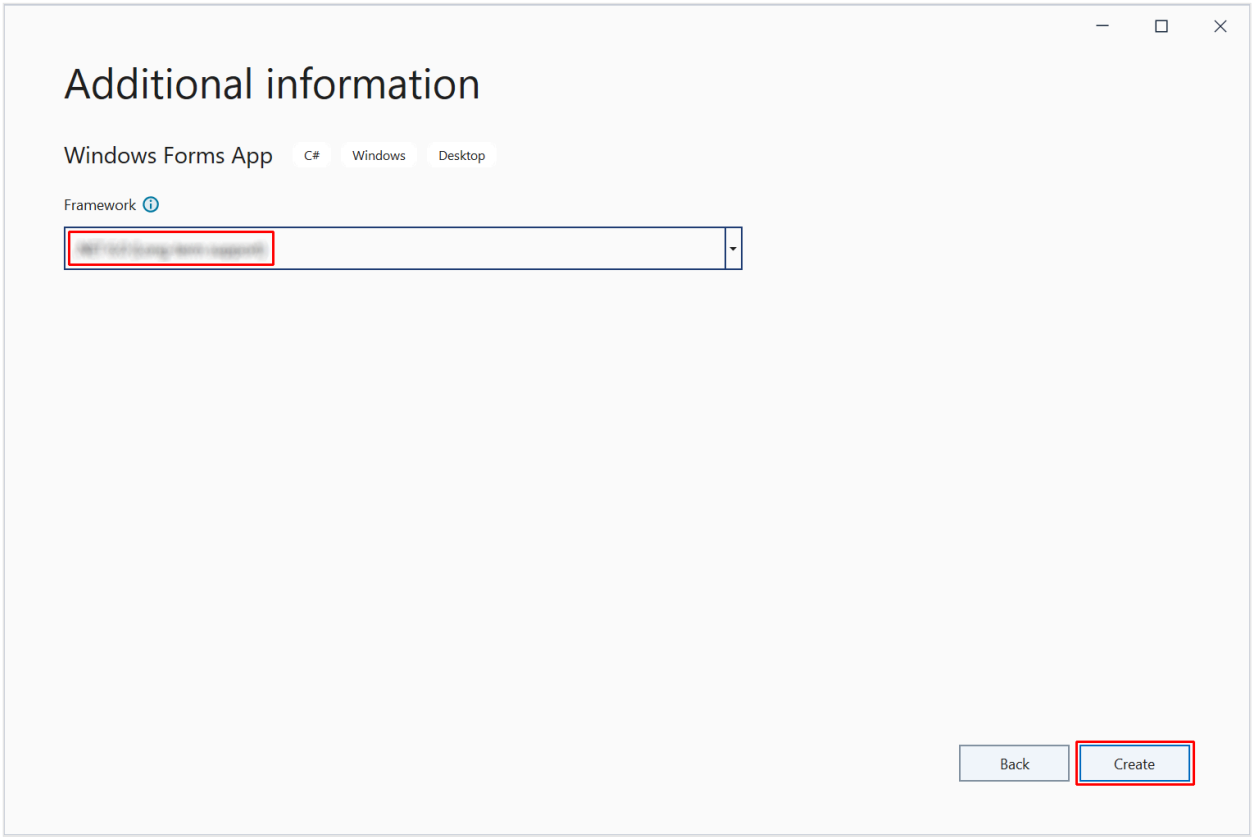
In the **Configure your new project** dialog:

- Set the **Project name** to **WinFormsBlazor**.
- Choose a suitable location for the project.

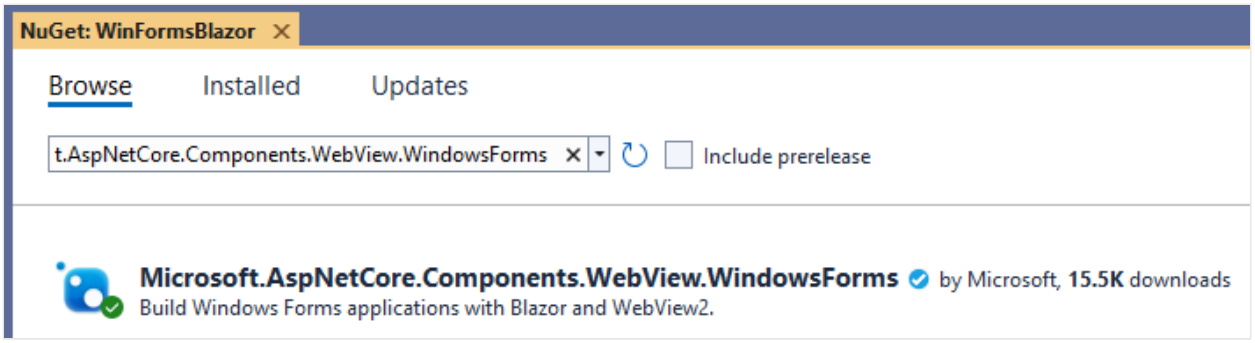
- Select the **Next** button.



In the **Additional information** dialog, select the framework version with the **Framework** dropdown list. Select the **Create** button:



Use NuGet Package Manager to install the Microsoft.AspNetCore.Components.WebView.WindowsForms NuGet package:



In **Solution Explorer**, right-click the project's name, **WinFormsBlazor**, and select **Edit Project File** to open the project file (`WinFormsBlazor.csproj`).

At the top of the project file, change the SDK to `Microsoft.NET.Sdk.Razor`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

Save the changes to the project file (`WinFormsBlazor.csproj`).

Add an `_Imports.razor` file to the root of the project with an @using directive for Microsoft.AspNetCore.Components.Web.

`_Imports.razor`:

razor

```razor
@using Microsoft.AspNetCore.Components.Web
```

Save the `_Imports.razor` file.

Add a `wwwroot` folder to the project.

Add an `index.html` file to the `wwwroot` folder with the following markup.

`wwwroot/index.html`:

HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>WinFormsBlazor</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
```

```
    <link href="css/app.css" rel="stylesheet" />
    <link href="WinFormsBlazor.styles.css" rel="stylesheet" />
</head>

<body>

    <div id="app">Loading...</div>

    <div id="blazor-error-ui" data-nosnippet>
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_framework/blazor.webview.js"></script>

</body>

</html>
```

Inside the `wwwroot` folder, create a `css` folder to hold stylesheets.

Add an `app.css` stylesheet to the `wwwroot/css` folder with the following content.

`wwwroot/css/app.css`:

```css
html, body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

h1:focus {
    outline: none;
}

a, .btn-link {
    color: #0071c1;
}

.btn-primary {
    color: #fff;
    background-color: #1b6ec2;
    border-color: #1861ac;
}

.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}

.invalid {
    outline: 1px solid red;
}
```

```
.validation-message {
    color: red;
}

#blazor-error-ui {
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}

    #blazor-error-ui .dismiss {
        cursor: pointer;
        position: absolute;
        right: 0.75rem;
        top: 0.5rem;
    }
```

Inside the `wwwroot/css` folder, create a `bootstrap` folder. Inside the `bootstrap` folder, place a copy of `bootstrap.min.css`. You can obtain the latest version of `bootstrap.min.css` from the Bootstrap website. Because all of the content at the site is versioned in the URL, a direct link can't be provided here. Therefore, follow navigation bar links to **Docs** > **Download** to obtain `bootstrap.min.css`.

Add the following `Counter` component to the root of the project, which is the default `Counter` component found in Blazor project templates.

`Counter.razor`:

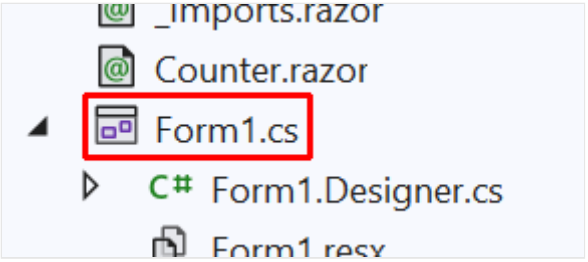
razor

```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
```

```
    }
}
```

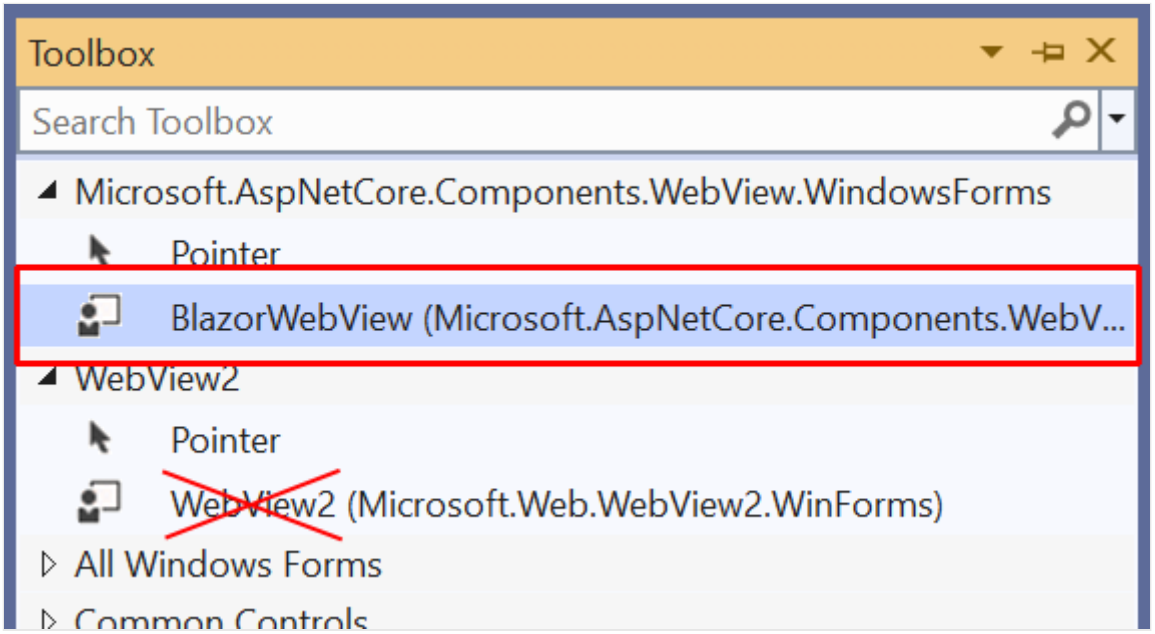
Save the `Counter` component (`Counter.razor`).

In **Solution Explorer**, double-click on the `Form1.cs` file to open the designer:

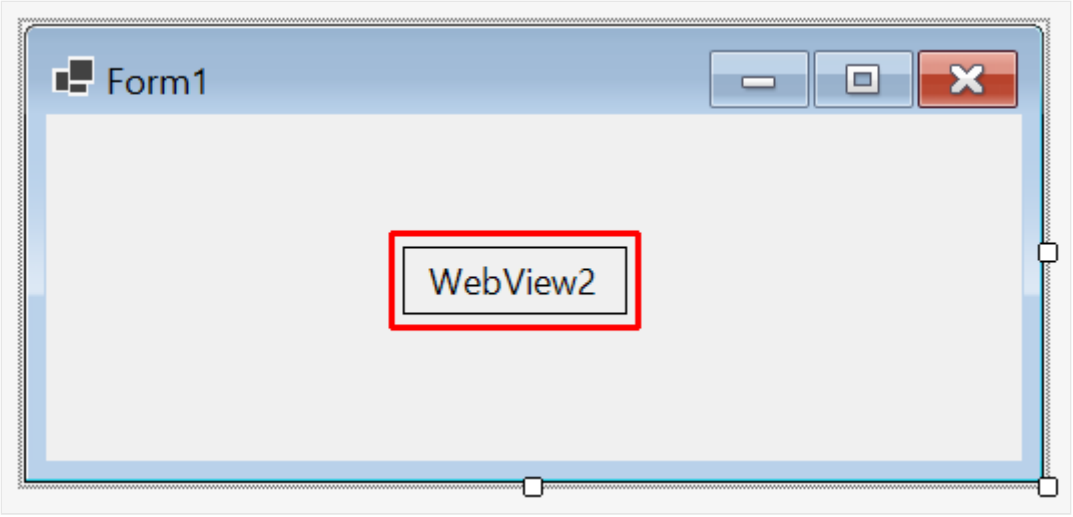


Open the **Toolbox** by either selecting the **Toolbox** button along the left edge of the Visual Studio window or selecting the **View** > **Toolbox** menu command.

Locate the **`BlazorWebView`** control under **`Microsoft.AspNetCore.Components.WebView.WindowsForms`**. Drag the BlazorWebView from the **Toolbox** into the `Form1` designer. Be careful not to accidentally drag a **`WebView2`** control into the form.

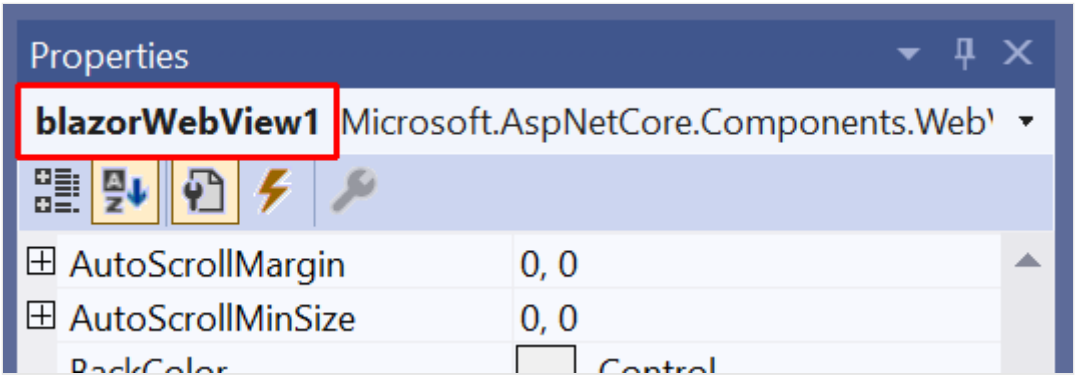


Visual Studio shows the BlazorWebView control in the form designer as `WebView2` and automatically names the control `blazorWebView1`:

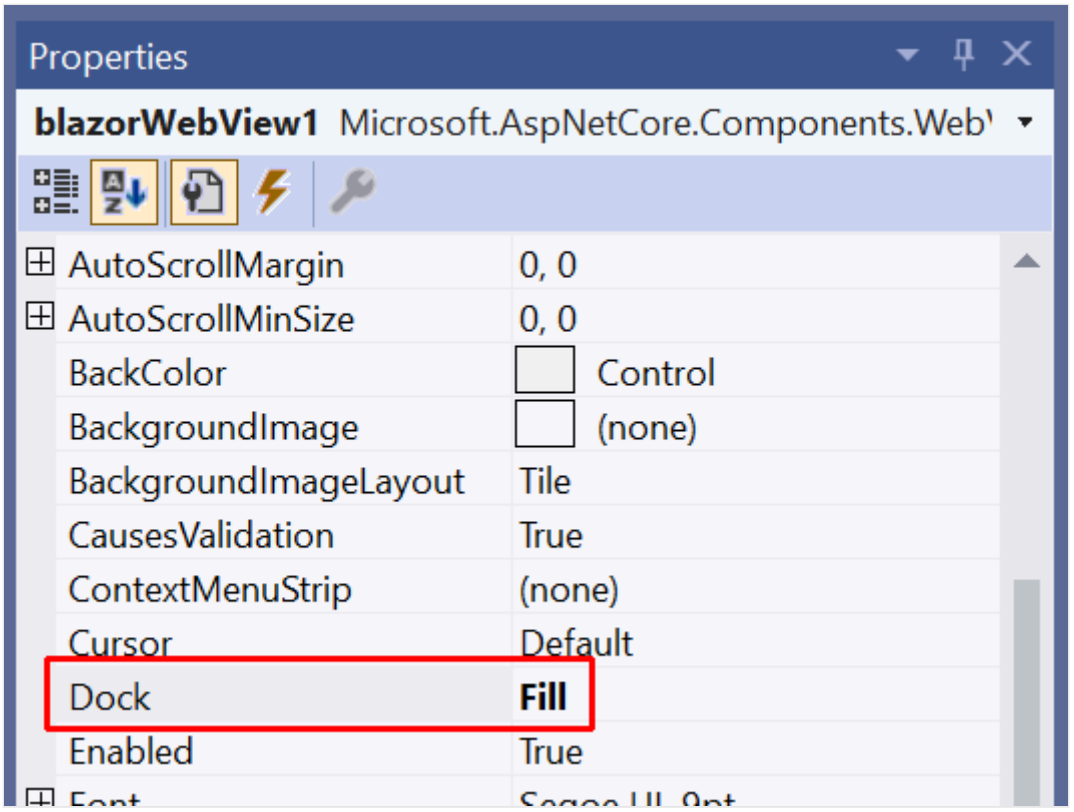


In `Form1`, select the BlazorWebView (`WebView2`) with a single click.

In the BlazorWebView's **Properties**, confirm that the control is named `blazorWebView1`. If the name isn't `blazorWebView1`, the wrong control was dragged from the **Toolbox**. Delete the `WebView2` control in `Form1` and drag the **`BlazorWebView` control** into the form.



In the control's properties, change the BlazorWebView's **Dock** value to **Fill**:



In the `Form1` designer, right-click `Form1` and select **View Code**.

Add namespaces for Microsoft.AspNetCore.Components.WebView.WindowsForms and Microsoft.Extensions.DependencyInjection to the top of the `Form1.cs` file:

C#

```
using Microsoft.AspNetCore.Components.WebView.WindowsForms;
using Microsoft.Extensions.DependencyInjection;
```

Inside the `Form1` constructor, after the `InitializeComponent` method call, add the following code:

C#

```
var services = new ServiceCollection();
services.AddWindowsFormsBlazorWebView();
blazorWebView1.HostPage = "wwwroot\\index.html";
blazorWebView1.Services = services.BuildServiceProvider();
blazorWebView1.RootComponents.Add<Counter>("#app");
```

> ⓘ Note
>
> The `InitializeComponent` method is automatically generated at app build time and added to the compilation object for the calling class.

The final, complete C# code of `Form1.cs` with a file-scoped namespace:

```C#
using Microsoft.AspNetCore.Components.WebView.WindowsForms;
using Microsoft.Extensions.DependencyInjection;

namespace WinFormsBlazor;

public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();

        var services = new ServiceCollection();
        services.AddWindowsFormsBlazorWebView();
        blazorWebView1.HostPage = "wwwroot\\index.html";
        blazorWebView1.Services = services.BuildServiceProvider();
        blazorWebView1.RootComponents.Add<Counter>("#app");
    }
}
```

Run the app

Select the start button in the Visual Studio toolbar:



The app running on Windows:



Next steps

In this tutorial, you learned how to:

- ✓ Create a Windows Forms Blazor app project
- ✓ Run the app on Windows

Learn more about Blazor Hybrid apps:

ASP.NET Core Blazor Hybrid

Build a Windows Presentation Foundation (WPF) Blazor app

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial shows you how to build and run a WPF Blazor app. You learn how to:

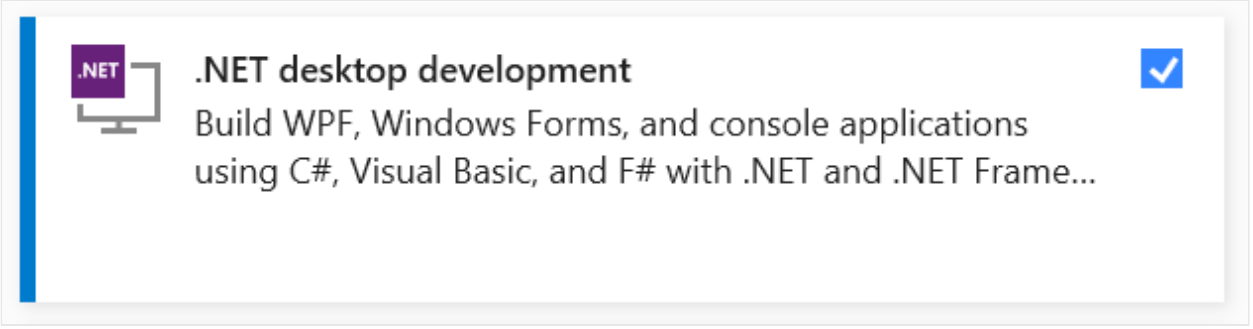
- ✓ Create a WPF Blazor app project
- ✓ Add a Razor component to the project
- ✓ Run the app on Windows

Prerequisites

- Supported platforms (WPF documentation)
- Visual Studio 2022 with the **.NET desktop development** workload

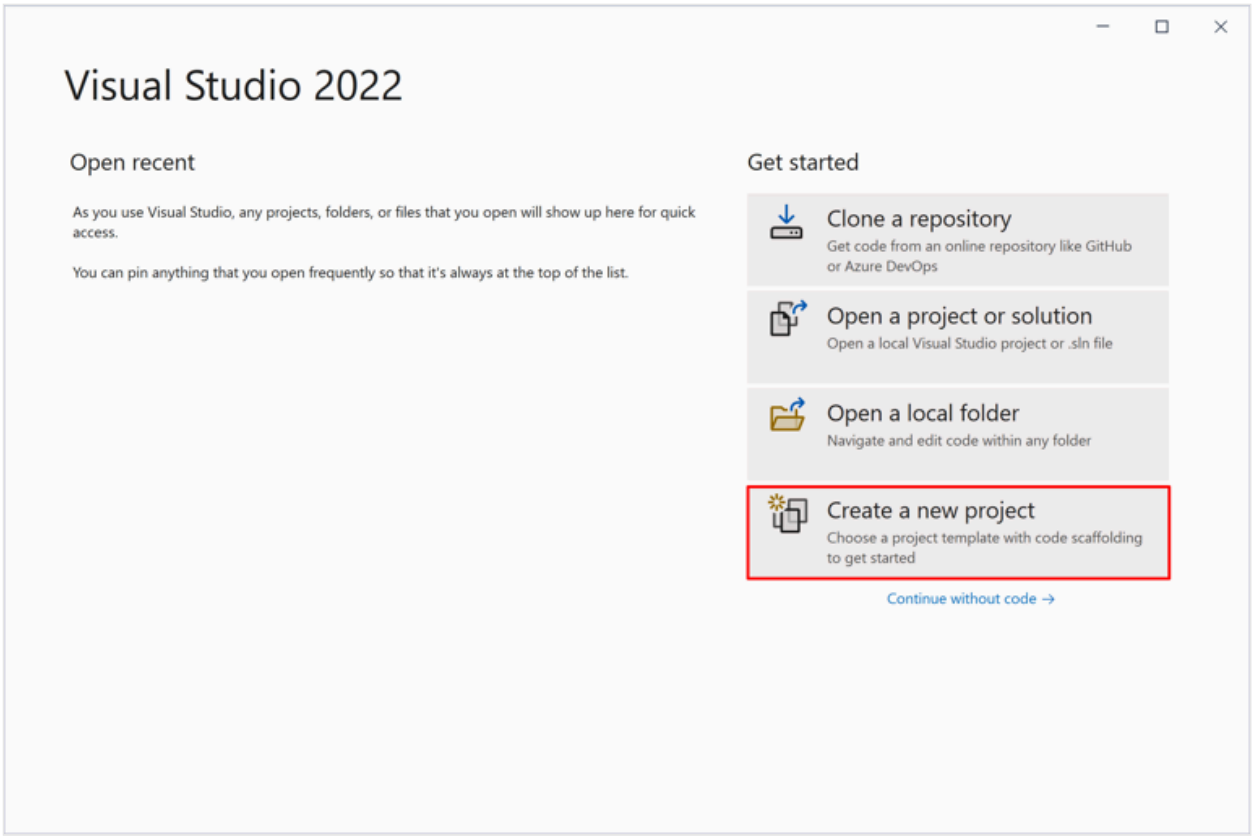
Visual Studio workload

If the **.NET desktop development** workload isn't installed, use the Visual Studio installer to install the workload. For more information, see Modify Visual Studio workloads, components, and language packs.

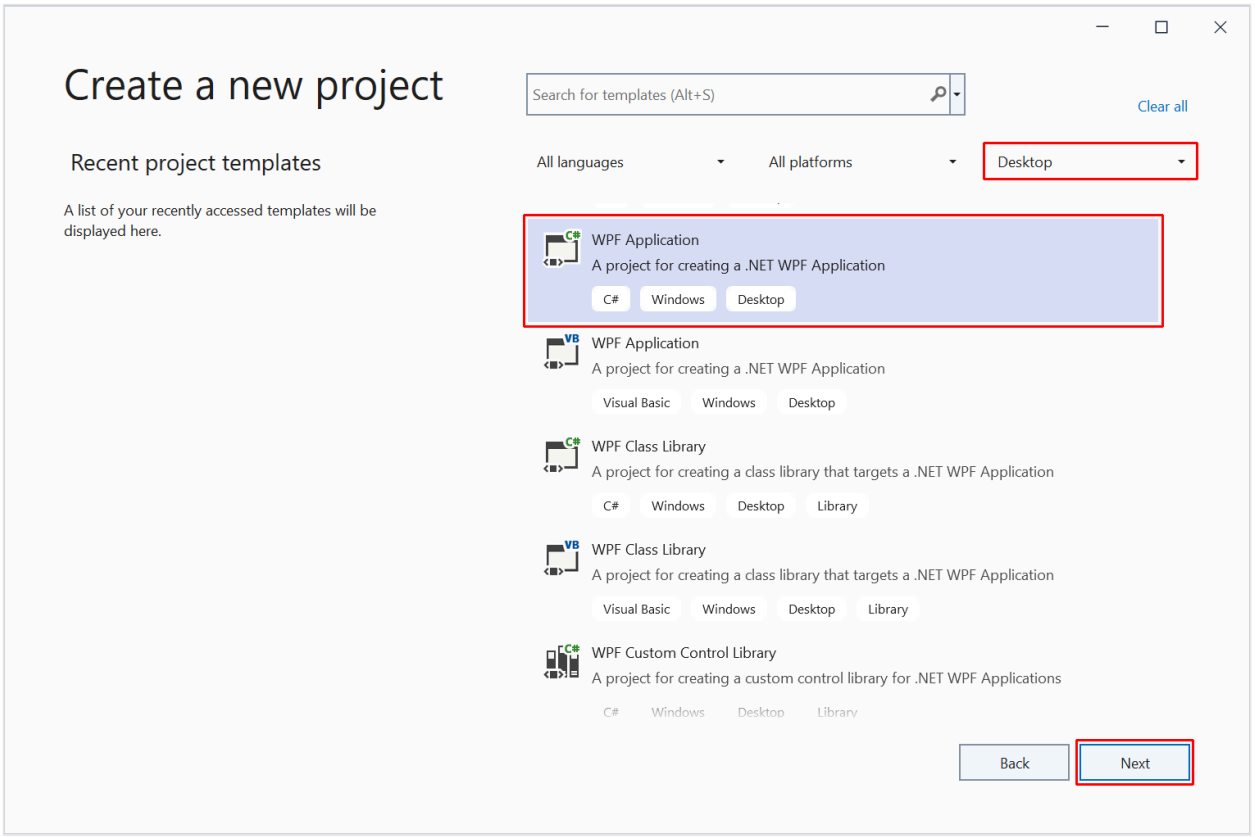


Create a WPF Blazor project

Launch Visual Studio. In the **Start Window**, select **Create a new project**:



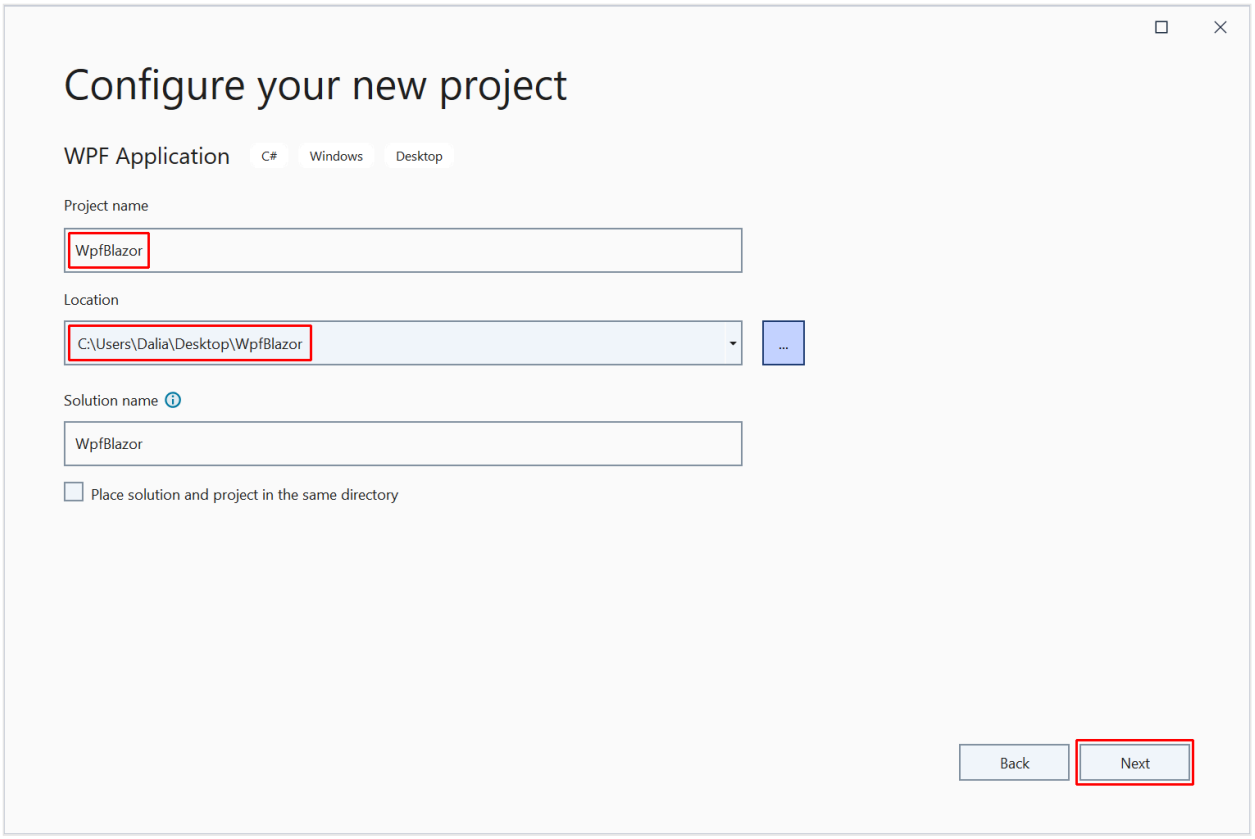
In the **Create a new project** dialog, filter the **Project type** dropdown to **Desktop**. Select the C# project template for **WPF Application** and select the **Next** button:



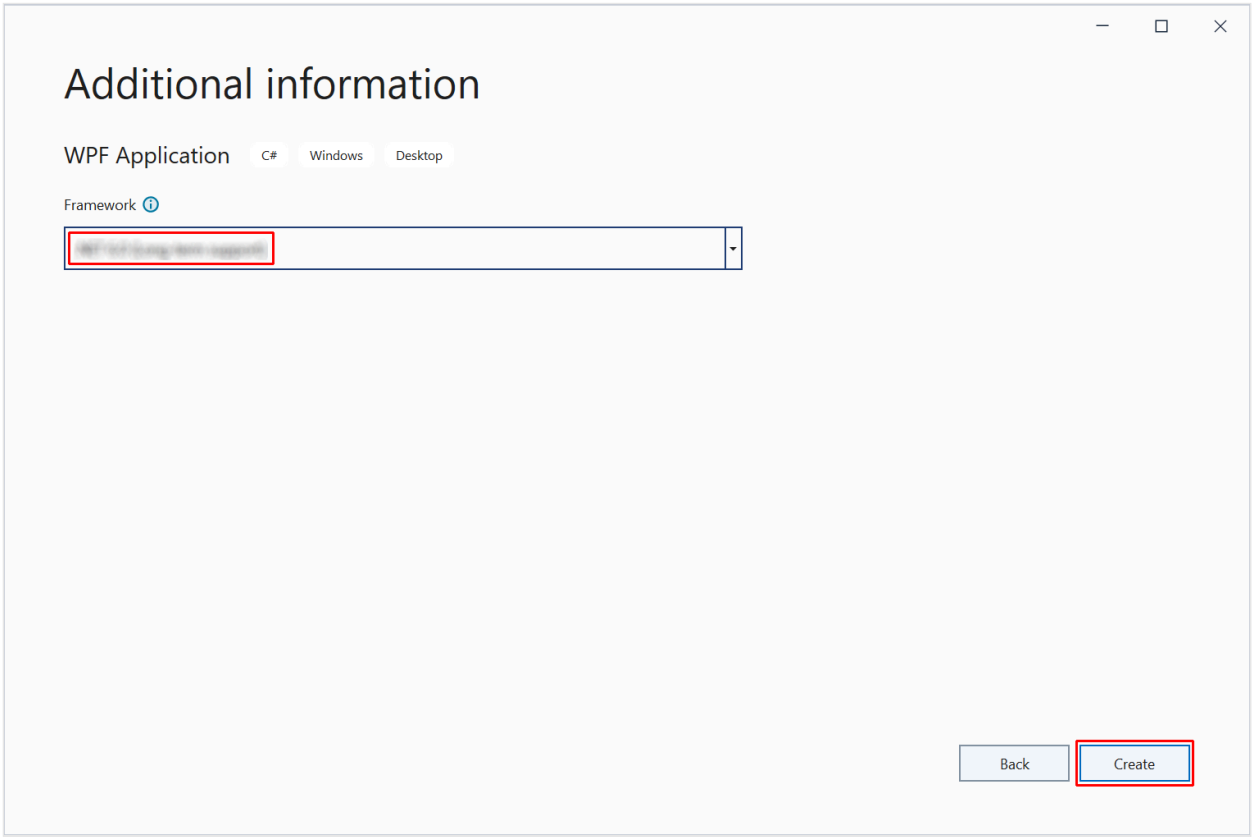
In the **Configure your new project** dialog:

- Set the **Project name** to **WpfBlazor**.
- Choose a suitable location for the project.

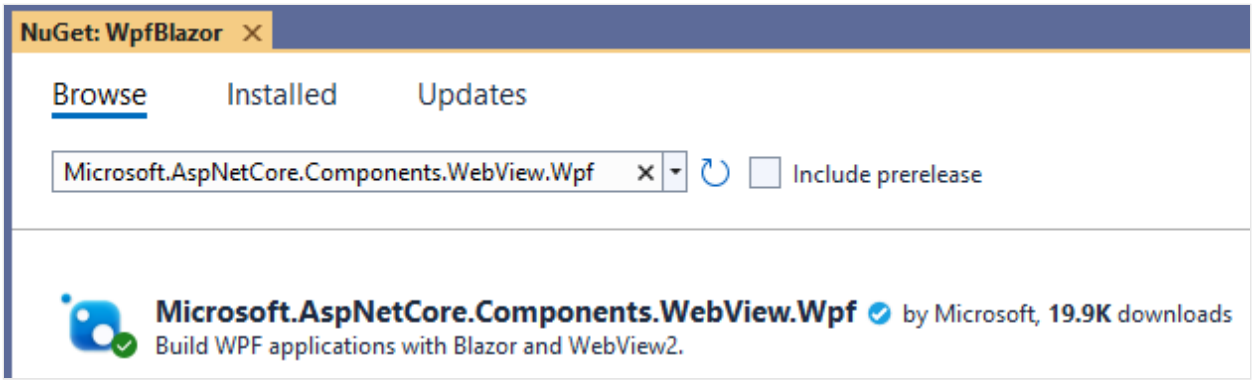
- Select the **Next** button.



In the **Additional information** dialog, select the framework version with the **Framework** dropdown list. Select the **Create** button:



Use NuGet Package Manager to install the Microsoft.AspNetCore.Components.WebView.Wpf NuGet package:



In **Solution Explorer**, right-click the project's name, **WpfBlazor**, and select **Edit Project File** to open the project file (`WpfBlazor.csproj`).

At the top of the project file, change the SDK to `Microsoft.NET.Sdk.Razor`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Razor">
```

In the project file's existing `<PropertyGroup>` add the following markup to set the app's root namespace, which is `WpfBlazor` in this tutorial:

XML

```xml
<RootNamespace>WpfBlazor</RootNamespace>
```

> **Note**
>
> The preceding guidance on setting the project's root namespace is a temporary workaround. For more information, see [Blazor][Wpf] Root namespace related issue (dotnet/maui #5861).

Save the changes to the project file (`WpfBlazor.csproj`).

Add an `_Imports.razor` file to the root of the project with an @using directive for Microsoft.AspNetCore.Components.Web.

`_Imports.razor`:

razor

```razor
@using Microsoft.AspNetCore.Components.Web
```

Save the `_Imports.razor` file.

Add a `wwwroot` folder to the project.

Add an `index.html` file to the `wwwroot` folder with the following markup.

`wwwroot/index.html`:

HTML

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>WpfBlazor</title>
    <base href="/" />
    <link href="css/bootstrap/bootstrap.min.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
    <link href="WpfBlazor.styles.css" rel="stylesheet" />
</head>

<body>
    <div id="app">Loading...</div>

    <div id="blazor-error-ui" data-nosnippet>
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>
    <script src="_framework/blazor.webview.js"></script>
</body>

</html>
```

Inside the `wwwroot` folder, create a `css` folder.

Add an `app.css` stylesheet to the `wwwroot/css` folder with the following content.

`wwwroot/css/app.css`:

css

```
html, body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

h1:focus {
    outline: none;
}
```

```
a, .btn-link {
    color: #0071c1;
}

.btn-primary {
    color: #fff;
    background-color: #1b6ec2;
    border-color: #1861ac;
}

.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}

.invalid {
    outline: 1px solid red;
}

.validation-message {
    color: red;
}

#blazor-error-ui {
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}

    #blazor-error-ui .dismiss {
        cursor: pointer;
        position: absolute;
        right: 0.75rem;
        top: 0.5rem;
    }
```

Inside the `wwwroot/css` folder, create a `bootstrap` folder. Inside the `bootstrap` folder, place a copy of `bootstrap.min.css`. You can obtain the latest version of `bootstrap.min.css` from the Bootstrap website. Follow navigation bar links to **Docs** > **Download**. A direct link can't be provided here because all of the content at the site is versioned in the URL.

Add the following `Counter` component to the root of the project, which is the default `Counter` component found in Blazor project templates.

`Counter.razor`:

razor

```razor
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Save the `Counter` component (`Counter.razor`).

If the `MainWindow` designer isn't open, open it by double-clicking the `MainWindow.xaml` file in **Solution Explorer**. In the `MainWindow` designer, replace the XAML code with the following:

XAML

```xaml
<Window x:Class="WpfBlazor.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:blazor="clr-
namespace:Microsoft.AspNetCore.Components.WebView.Wpf;assembly=Microsoft.Asp
NetCore.Components.WebView.Wpf"
        xmlns:local="clr-namespace:WpfBlazor"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <blazor:BlazorWebView HostPage="wwwroot\index.html" Services="
{DynamicResource services}">
            <blazor:BlazorWebView.RootComponents>
                <blazor:RootComponent Selector="#app" ComponentType="{x:Type
local:Counter}" />
            </blazor:BlazorWebView.RootComponents>
        </blazor:BlazorWebView>
    </Grid>
</Window>
```

In **Solution Explorer**, right-click `MainWindow.xaml` and select **View Code**:



Add the namespace Microsoft.Extensions.DependencyInjection to the top of the `MainWindow.xaml.cs` file:

C#

```
using Microsoft.Extensions.DependencyInjection;
```

Inside the `MainWindow` constructor, after the `InitializeComponent` method call, add the following code:

C#

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddWpfBlazorWebView();
Resources.Add("services", serviceCollection.BuildServiceProvider());
```

> ⓘ **Note**
>
> The `InitializeComponent` method is automatically generated at app build time and added to the compilation object for the calling class.

The final, complete C# code of `MainWindow.xaml.cs` with a file-scoped namespace and comments removed:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
```

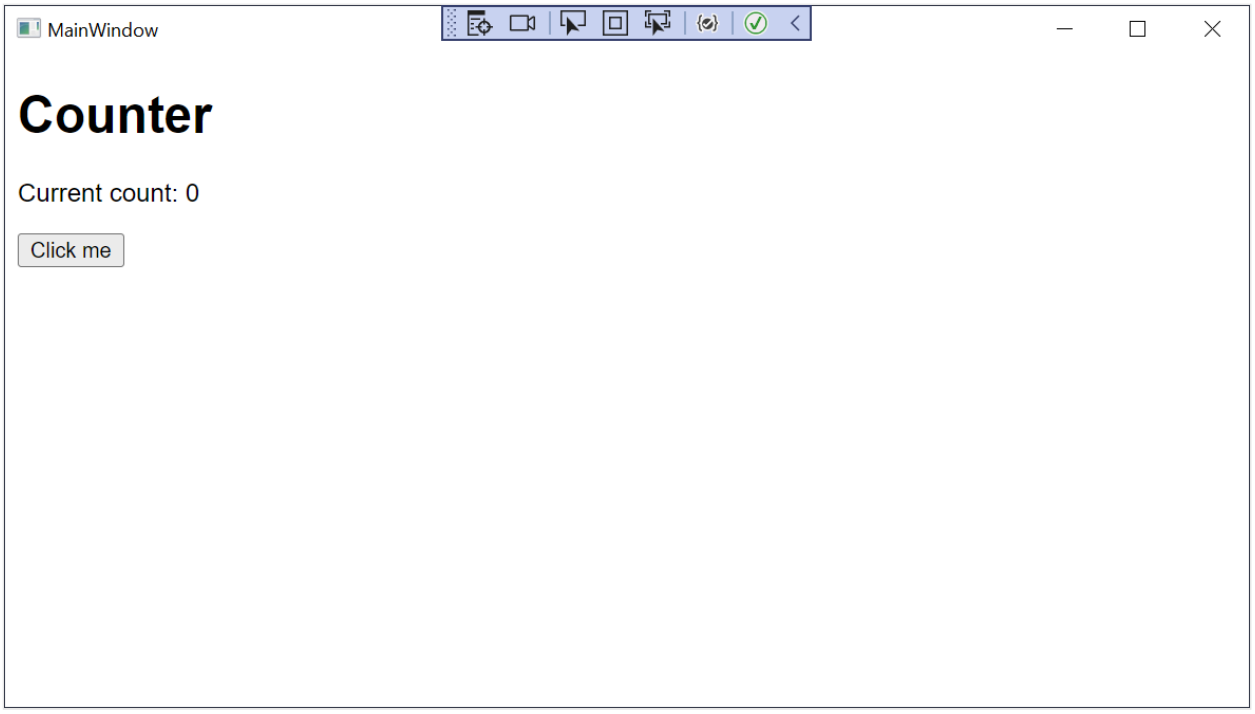
```
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Microsoft.Extensions.DependencyInjection;

namespace WpfBlazor;

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        var serviceCollection = new ServiceCollection();
        serviceCollection.AddWpfBlazorWebView();
        Resources.Add("services", serviceCollection.BuildServiceProvider());
    }
}
```

Run the app

Select the start button in the Visual Studio toolbar:



The app running on Windows:



Next steps

In this tutorial, you learned how to:

- ✓ Create a WPF Blazor app project
- ✓ Add a Razor component to the project
- ✓ Run the app on Windows

Learn more about Blazor Hybrid apps:

ASP.NET Core Blazor Hybrid

ASP.NET Core Blazor Hybrid routing and navigation

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to manage request routing and navigation in Blazor Hybrid apps.

URI request routing behavior

Default URI request routing behavior:

- A link is *internal* if the host name and scheme match between the app's origin URI and the request URI. When the host names and schemes don't match or if the link sets `target="_blank"`, the link is considered *external*.
- If the link is internal, the link is opened in the `BlazorWebView` by the app.
- If the link is external, the link is opened by an app determined by the device based on the device's registered handler for the link's scheme.
- For internal links that appear to request a file because the last segment of the URI uses dot notation (for example, `/file.x`, `/Maryia.Melnyk`, `/image.gif`) but don't point to any static content:
  - WPF and Windows Forms: The host page content is returned.
  - .NET MAUI: A 404 response is returned.

To change the link handling behavior for links that don't set `target="_blank"`, register the `UrlLoading` event and set the UrlLoadingEventArgs.UrlLoadingStrategy property. The UrlLoadingStrategy enumeration allows setting link handling behavior to any of the following values:

- OpenExternally: Load the URL using an app determined by the device. This is the default strategy for URIs with an external host.
- OpenInWebView: Load the URL within the `BlazorWebView`. This is the default strategy for URLs with a host matching the app origin. ***Don't use this strategy for external links unless you can ensure the destination URI is fully trusted.***
- CancelLoad: Cancels the current URL loading attempt.

The UrlLoadingEventArgs.Url property is used to get or dynamically set the URL.

> **⚠ Warning**
>
> External links are opened in an app determined by the device. Opening external links within a `BlazorWebView` can introduce security vulnerabilities and shouldn't be enabled unless you can ensure that the external links are fully trusted.

API documentation:

- .NET MAUI: BlazorWebView.UrlLoading
- WPF: BlazorWebView.UrlLoading
- Windows Forms: BlazorWebView.UrlLoading

The Microsoft.AspNetCore.Components.WebView namespace is required for the following examples:

C#

```
using Microsoft.AspNetCore.Components.WebView;
```

Add the following event handler to the constructor of the `Page` where the `BlazorWebView` is created, which is `MainPage.xaml.cs` in an app created from the .NET MAUI project template.

C#

```
blazorWebView.UrlLoading +=
    (sender, urlLoadingEventArgs) =>
    {
        if (urlLoadingEventArgs.Url.Host != "0.0.0.0")
        {
            urlLoadingEventArgs.UrlLoadingStrategy =
                UrlLoadingStrategy.OpenInWebView;
        }
    };
```

Get or set a path for initial navigation

Use the `BlazorWebView.StartPath` property to get or set the path for initial navigation within the Blazor navigation context when the Razor component is finished loading. The default start path is the relative root URL path (`/`).

In the `MainPage` XAML markup (`MainPage.xaml`), specify the start path. The following example sets the path to a welcome page at `/welcome`:

XAML

```
<BlazorWebView ... StartPath="/welcome" ...>
    ...
<BlazorWebView>
```

Alternatively, the start path can be set in the `MainPage` constructor (`MainPage.xaml.cs`):

C#

```
blazorWebView.StartPath = "/welcome";
```

Navigation among pages and Razor components

This section explains how to navigate among .NET MAUI content pages and Razor components.

The .NET MAUI Blazor hybrid project template isn't a Shell-based app, so the URI-based navigation for Shell-based apps isn't suitable for a project based on the project template. The examples in this section use a NavigationPage to perform modeless or modal navigation.

In the following example:

- The namespace of the app is `MauiBlazor`, which matches the suggested project name of the Build a .NET MAUI Blazor Hybrid app tutorial.
- A ContentPage is placed in a new folder added to the app named `Views`.

In `App.xaml.cs`, create the `MainPage` as a NavigationPage by making the following change:

diff

```
- MainPage = new MainPage();
+ MainPage = new NavigationPage(new MainPage());
```

`Views/NavigationExample.xaml`:

XAML

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MauiBlazor"
             x:Class="MauiBlazor.Views.NavigationExample"
             Title="Navigation Example"
             BackgroundColor="{DynamicResource PageBackgroundColor}">
    <StackLayout>
        <Label Text="Navigation Example"
               VerticalOptions="Center"
               HorizontalOptions="Center"
               FontSize="24" />
        <Button x:Name="CloseButton"
                Clicked="CloseButton_Clicked"
                Text="Close" />
    </StackLayout>
</ContentPage>
```

In the following `NavigationExample` code file, the `CloseButton_Clicked` event handler for the close button calls PopAsync to pop the ContentPage off of the navigation stack.

`Views/NavigationExample.xaml.cs`:

C#

```
namespace MauiBlazor.Views;

public partial class NavigationExample : ContentPage
{
    public NavigationExample()
    {
        InitializeComponent();
    }

    private async void CloseButton_Clicked(object sender, EventArgs e)
    {
        await Navigation.PopAsync();
    }
}
```

In a Razor component:

- Add the namespace for the app's content pages. In the following example, the namespace is `MauiBlazor.Views`.
- Add an HTML `button` element with an @onclick event handler to open the content page. The event handler method is named `OpenPage`.
- In the event handler, call PushAsync to push the ContentPage, `NavigationExample`, onto the navigation stack.

The following example is based on the `Index` component in the .NET MAUI Blazor project template.

`Pages/Index.razor`:

razor

```razor
@page "/"
@using MauiBlazor.Views

<h1>Hello, world!</h1>

Welcome to your new app.

<SurveyPrompt Title="How is Blazor working for you?" />

<button class="btn btn-primary" @onclick="OpenPage">Open</button>

@code {
    private async void OpenPage()
    {
        await App.Current.MainPage.Navigation.PushAsync(new
NavigationExample());
    }
}
```

To change the preceding example to modal navigation:

- In the `CloseButton_Clicked` method (`Views/NavigationExample.xaml.cs`), change PopAsync to PopModalAsync:

  diff

  ```
  - await Navigation.PopAsync();
  + await Navigation.PopModalAsync();
  ```

- In the `OpenPage` method (`Pages/Index.razor`), change PushAsync to PushModalAsync:

  diff

  ```
  - await App.Current.MainPage.Navigation.PushAsync(new
  NavigationExample());
  + await App.Current.MainPage.Navigation.PushModalAsync(new
  NavigationExample());
  ```

For more information, see the following resources:

- NavigationPage article (.NET MAUI documentation)
- NavigationPage (API documentation)

App linking (deep linking)

It's often desirable to connect a website and a mobile app so that links on a website launch the mobile app and display content in the mobile app. App linking, also known as *deep linking*, is a technique that enables a mobile device to respond to a URI and launch content in a mobile app that's represented by the URI.

For more information, see the following articles in the .NET MAUI documentation:

- Android app links
- Apple universal links

ASP.NET Core Blazor Hybrid static files

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes how to consume static asset files in Blazor Hybrid apps.

In a Blazor Hybrid app, static files are *app resources*, accessed by Razor components using the following approaches:

- .NET MAUI: .NET MAUI file system helpers
- WPF and Windows Forms: ResourceManager

When static assets are only used in the Razor components, static assets can be consumed from the web root (`wwwroot` folder) in a similar way to Blazor WebAssembly and Blazor Server apps. For more information, see the Static assets limited to Razor components section.

.NET MAUI

In .NET MAUI apps, *raw assets* using the `MauiAsset` build action and .NET MAUI file system helpers are used for static assets.

> **Note**
>
> Interfaces, classes, and supporting types to work with storage on devices across all supported platforms for features such as choosing a file, saving preferences, and using secure storage are in the Microsoft.Maui.Storage namespace. The namespace is available throughout a MAUI Blazor Hybrid app, so there's no need to specify a `using` statement in a class file or an `@using` Razor directive in a Razor component for the namespace.

Place raw assets into the `Resources/Raw` folder of the app. The example in this section uses a static text file.

`Resources/Raw/Data.txt`:

text

```
This is text from a static text file resource.
```

The following Razor component:

- Calls OpenAppPackageFileAsync to obtain a Stream for the resource.
- Reads the Stream with a StreamReader.
- Calls StreamReader.ReadToEndAsync to read the file.

`Pages/StaticAssetExample.razor`:

razor

```
@page "/static-asset-example"
@using System.IO
@using Microsoft.Extensions.Logging
@inject ILogger<StaticAssetExample> Logger

<h1>Static Asset Example</h1>

<p>@dataResourceText</p>

@code {
    public string dataResourceText = "Loading resource ...";

    protected override async Task OnInitializedAsync()
    {
        try
        {
            using var stream =
                await FileSystem.OpenAppPackageFileAsync("Data.txt");
            using var reader = new StreamReader(stream);

            dataResourceText = await reader.ReadToEndAsync();
        }
        catch (FileNotFoundException ex)
        {
            dataResourceText = "Data file not found.";
            Logger.LogError(ex, "'Resource/Raw/Data.txt' not found.");
        }
    }
}
```

For more information, see the following resources:

- Target multiple platforms from .NET MAUI single project (.NET MAUI documentation)
- Improve consistency with resizetizer (dotnet/maui #4367)

WPF

Place the asset into a folder of the app, typically at the project's root, such as a `Resources` folder. The example in this section uses a static text file.

`Resources/Data.txt`:

```text
This is text from a static text file resource.
```

If a `Properties` folder doesn't exist in the app, create a `Properties` folder in the root of the app.

If the `Properties` folder doesn't contain a resources file (`Resources.resx`), create the file in **Solution Explorer** with the **Add** > **New Item** contextual menu command.

Double-click the `Resource.resx` file.

Select **Strings** > **Files** from the dropdown list.

Select **Add Resource** > **Add Existing File**. If prompted by Visual Studio to confirm editing the file, select **Yes**. Navigate to the `Resources` folder, select the `Data.txt` file, and select **Open**.

In the following example component, ResourceManager.GetString obtains the string resource's text for display.

> ⚠ **Warning**
>
> Never use ResourceManager methods with untrusted data.

`StaticAssetExample.razor`:

```razor
@page "/static-asset-example"
@using System.Resources

<h1>Static Asset Example</h1>

<p>@dataResourceText</p>

@code {
    public string dataResourceText = "Loading resource ...";
```

```
    protected override void OnInitialized()
    {
        var resources =
            new ResourceManager(typeof(WpfBlazor.Properties.Resources));

        dataResourceText = resources.GetString("Data") ?? "'Data' not
found.";
    }
}
```

Windows Forms

Place the asset into a folder of the app, typically at the project's root, such as a `Resources` folder. The example in this section uses a static text file.

`Resources/Data.txt`:

text

```
This is text from a static text file resource.
```

Examine the files associated with `Form1` in **Solution Explorer**. If `Form1` doesn't have a resource file (`.resx`), add a `Form1.resx` file with the **Add** > **New Item** contextual menu command.

Double-click the `Form1.resx` file.

Select **Strings** > **Files** from the dropdown list.

Select **Add Resource** > **Add Existing File**. If prompted by Visual Studio to confirm editing the file, select **Yes**. Navigate to the `Resources` folder, select the `Data.txt` file, and select **Open**.

In the following example component:

- The app's assembly name is `WinFormsBlazor`. The ResourceManager's base name is set to the assembly name of `Form1` (`WinFormsBlazor.Form1`).
- ResourceManager.GetString obtains the string resource's text for display.

> ⚠ **Warning**
>
> Never use ResourceManager methods with untrusted data.

`StaticAssetExample.razor`:

```razor
@page "/static-asset-example"
@using System.Resources

<h1>Static Asset Example</h1>

<p>@dataResourceText</p>

@code {
    public string dataResourceText = "Loading resource ...";

    protected override async Task OnInitializedAsync()
    {
        var resources =
            new ResourceManager("WinFormsBlazor.Form1",
this.GetType().Assembly);

        dataResourceText = resources.GetString("Data") ?? "'Data' not
found.";
    }
}
```

Static assets limited to Razor components

A `BlazorWebView` control has a configured host file (HostPage), typically `wwwroot/index.html`. The HostPage path is relative to the project. All static web assets (scripts, CSS files, images, and other files) that are referenced from a `BlazorWebView` are relative to its configured HostPage.

Static web assets from a Razor class library (RCL) use special paths: `_content/{PACKAGE ID}/{PATH AND FILE NAME}`. The `{PACKAGE ID}` placeholder is the library's package ID. The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file. The `{PATH AND FILE NAME}` placeholder is path and file name under `wwwroot`. These paths are logically subpaths of the app's `wwwroot` folder, although they're actually coming from other packages or projects. Component-specific CSS style bundles are also built at the root of the `wwwroot` folder.

The web root of the HostPage determines which subset of static assets are available:

- `wwwroot/index.html` (*recommended*): All assets in the app's `wwwroot` folder are available (for example: `wwwroot/image.png` is available from `/image.png`), including subfolders (for example: `wwwroot/subfolder/image.png` is available from

`/subfolder/image.png`). RCL static assets in the RCL's `wwwroot` folder are available (for example: `wwwroot/image.png` is available from the path `_content/{PACKAGE ID}/image.png`), including subfolders (for example: `wwwroot/subfolder/image.png` is available from the path `_content/{PACKAGE ID}/subfolder/image.png`).

- `wwwroot/{PATH}/index.html`: All assets in the app's `wwwroot/{PATH}` folder are available using app web root relative paths. RCL static assets in `wwwroot/{PATH}` aren't because they would be in a non-existent theoretical location, such as `../../_content/{PACKAGE ID}/{PATH}`, which isn't a supported relative path.
- `wwwroot/_content/{PACKAGE ID}/index.html`: All assets in the RCL's `wwwroot/{PATH}` folder are available using RCL web root relative paths. The app's static assets in `wwwroot/{PATH}` are aren't because they would be in a non-existent theoretical location, such as `../../{PATH}`, which isn't a supported relative path.

For most apps, we recommend placing the HostPage at the root of the `wwwroot` folder of the app, which provides the greatest flexibility for supplying static assets from the app, RCLs, and via subfolders of the app and RCLs.

The following examples demonstrate referencing static assets from the app's web root (`wwwroot` folder) with a HostPage rooted in the `wwwroot` folder.

`wwwroot/data.txt`:

text

```
This is text from a static text file resource.
```

`wwwroot/scripts.js`:

JavaScript

```
export function showPrompt(message) {
  return prompt(message, 'Type anything here');
}
```

The following Jeep® image is also used in this section's example. You can right-click the following image to save it locally for use in a local test app.

`wwwroot/jeep-yj.png`:



In a Razor component:

- The static text file contents can be read using the following techniques:
  - .NET MAUI: .NET MAUI file system helpers (OpenAppPackageFileAsync)
  - WPF and Windows Forms: StreamReader.ReadToEndAsync
- JavaScript files are available at logical subpaths of `wwwroot` using `./` paths.
- The image can be the source attribute (`src`) of an image tag (`<img>`).

`StaticAssetExample2.razor`:

razor

```
@page "/static-asset-example-2"
@using Microsoft.Extensions.Logging
@implements IAsyncDisposable
@inject IJSRuntime JS
@inject ILogger<StaticAssetExample2> Logger

<h1>Static Asset Example 2</h1>

<h2>Read a file</h2>

<p>@dataResourceText</p>

<h2>Call JavaScript</h2>

<p>
    <button @onclick="TriggerPrompt">Trigger browser window prompt</button>
</p>

<p>@result</p>

<h2>Show an image</h2>

<p><img alt="1991 Jeep YJ" src="/jeep-yj.png" /></p>

<p>
    <em>Jeep</em> and <em>Jeep YJ</em> are registered trademarks of
```

```
    <a href="https://www.stellantis.com">FCA US LLC (Stellantis NV)</a>.
</p>

@code {
    private string dataResourceText = "Loading resource ...";
    private IJSObjectReference? module;
    private string result;

    protected override async Task OnInitializedAsync()
    {
        try
        {
            dataResourceText = await ReadData();
        }
        catch (FileNotFoundException ex)
        {
            dataResourceText = "Data file not found.";
            Logger.LogError(ex, "'wwwroot/data.txt' not found.");
        }
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./scripts.js");
        }
    }

    private async Task TriggerPrompt()
    {
        result = await Prompt("Provide some text");
    }

    public async ValueTask<string> Prompt(string message) =>
        module is not null ?
            await module.InvokeAsync<string>("showPrompt", message) : null;

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

In .NET MAUI apps, add the following `ReadData` method to the `@code` block of the preceding component:

C#

```
private async Task<string> ReadData()
{
    using var stream = await
FileSystem.OpenAppPackageFileAsync("wwwroot/data.txt");
    using var reader = new StreamReader(stream);

    return await reader.ReadToEndAsync();
}
```

In WPF and Windows Forms apps, add the following `ReadData` method to the `@code` block of the preceding component:

C#

```
private async Task<string> ReadData()
{
    using var reader = new StreamReader("wwwroot/data.txt");

    return await reader.ReadToEndAsync();
}
```

Collocated JavaScript files are also accessible at logical subpaths of `wwwroot`. Instead of using the script described earlier for the `showPrompt` function in `wwwroot/scripts.js`, the following collocated JavaScript file for the `StaticAssetExample2` component also makes the function available.

`Pages/StaticAssetExample2.razor.js`:

JavaScript

```
export function showPrompt(message) {
  return prompt(message, 'Type anything here');
}
```

Modify the module object reference in the `StaticAssetExample2` component to use the collocated JavaScript file path (`./Pages/StaticAssetExample2.razor.js`):

C#

```
module = await JS.InvokeAsync<IJSObjectReference>("import",
```

```
    "./Pages/StaticAssetExample2.razor.js");
```

Trademarks

Jeep and *Jeep YJ* are registered trademarks of FCA US LLC (Stellantis NV).

Additional resources

- ResourceManager
- Create resource files for .NET apps (.NET Fundamentals documentation)
- How to: Use resources in localizable apps (WPF documentation)

Use browser developer tools with ASP.NET Core Blazor Hybrid

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use browser developer tools with Blazor Hybrid apps.

Browser developer tools with .NET MAUI Blazor

Ensure the Blazor Hybrid project is configured to support browser developer tools. You can confirm developer tools support by searching the app for `AddBlazorWebViewDeveloperTools`.

If the project isn't already configured for browser developer tools, add support by:

1. Locating where the call to AddMauiBlazorWebView is made, likely within the app's `MauiProgram.cs` file.
2. At the top of the `MauiProgram.cs` file, confirm the presence of a `using` statement for Microsoft.Extensions.Logging. If the `using` statement isn't present, add it to the top of the file:

   C#

   ```
   using Microsoft.Extensions.Logging;
   ```

3. After the call to AddMauiBlazorWebView, add the following code:

   C#

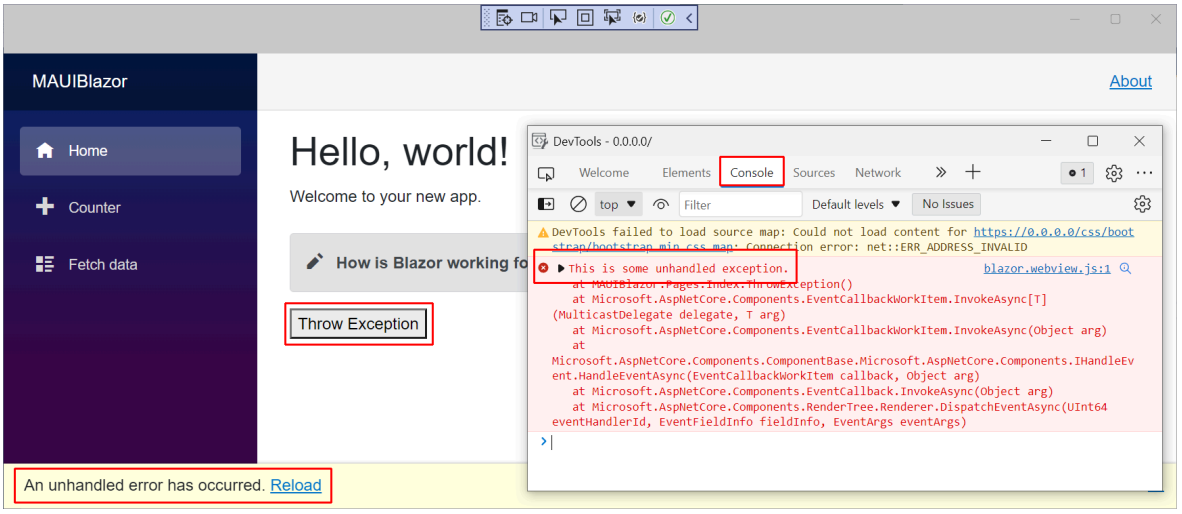
   ```
   #if DEBUG
       builder.Services.AddBlazorWebViewDeveloperTools();
       builder.Logging.AddDebug();
   #endif
   ```

To use browser developer tools with a Windows app:

1. Run the .NET MAUI Blazor Hybrid app for Windows and navigate to an app page that uses a BlazorWebView. The developer tools console is unavailable from ContentPages without a Blazor Web View.

2. Use the keyboard shortcut `Ctrl`+`Shift`+`I` to open browser developer tools.

3. Developer tools provide a variety of features for working with apps, including which assets the page requested, how long assets took to load, and the content of loaded assets. The following example shows the **Console** tab to see the console messages, which includes any exception messages generated by the framework or developer code:



Additional resources

- Chrome DevTools
- Microsoft Edge Developer Tools overview
- Safari Developer Help

Reuse Razor components in ASP.NET Core Blazor Hybrid

Article • 12/05/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to author and organize Razor components for the web and Web Views in Blazor Hybrid apps.

Razor components work across hosting models (Blazor WebAssembly, Blazor Server, and in the Web View of Blazor Hybrid) and across platforms (Android, iOS, and Windows). Hosting models and platforms have unique capabilities that components can leverage, but components executing across hosting models and platforms must leverage unique capabilities separately, which the following examples demonstrate:

- Blazor WebAssembly supports synchronous JavaScript (JS) interop, which isn't supported by the strictly asynchronous JS interop communication channel in Blazor Server and Web Views of Blazor Hybrid apps.
- Components in a Blazor Server app can access services that are only available on the server, such as an Entity Framework database context.
- Components in a `BlazorWebView` can directly access native desktop and mobile device features, such as geolocation services. Blazor Server and Blazor WebAssembly apps must rely upon web API interfaces of apps on external servers to provide similar features.

Design principles

In order to author Razor components that can seamlessly work across hosting models and platforms, adhere to the following design principles:

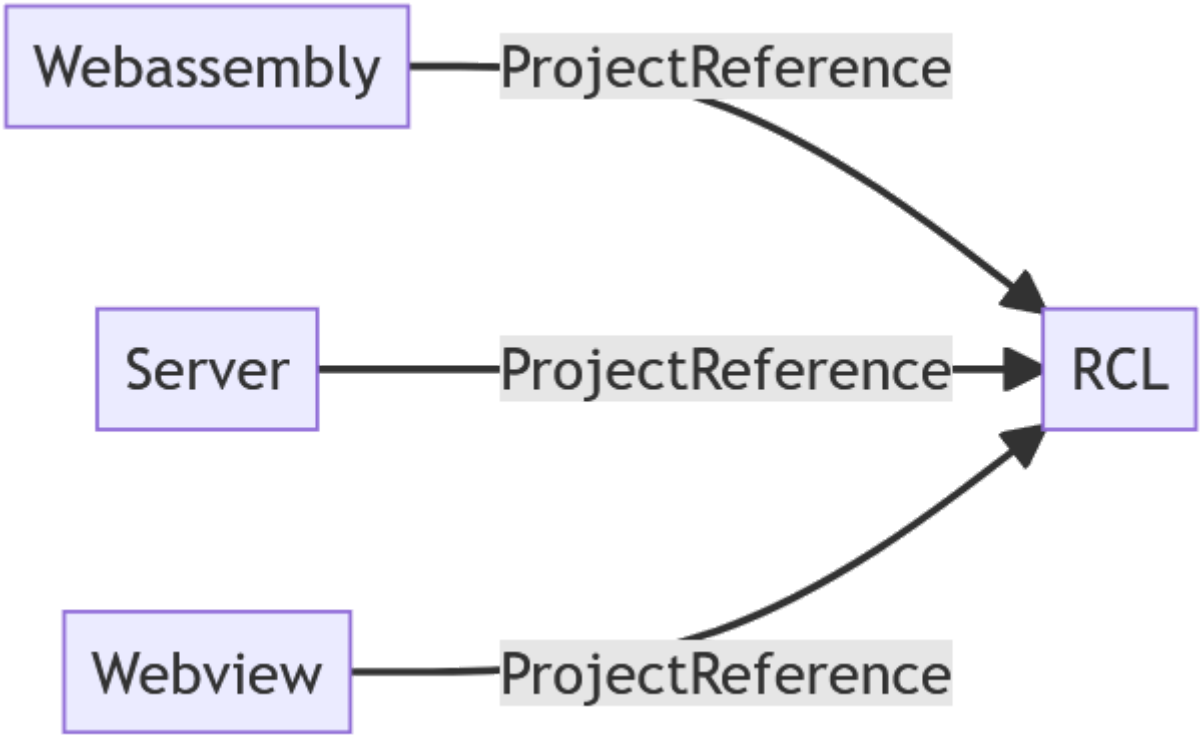
- Place shared UI code in Razor class libraries (RCLs), which are containers designed to maintain reusable pieces of UI for use across different hosting models and platforms.
- Implementations of unique features shouldn't exist in RCLs. Instead, the RCL should define abstractions (interfaces and base classes) that hosting models and platforms implement.

- Only opt-in to unique features by hosting model or platform. For example, Blazor WebAssembly supports the use of IJSInProcessRuntime and IJSInProcessObjectReference in a component as an optimization, but only use them with conditional casts and fallback implementations that rely on the universal IJSRuntime and IJSObjectReference abstractions that all hosting models and platforms support. For more information on IJSInProcessRuntime, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor. For more information on IJSInProcessObjectReference, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor.
- As a general rule, use CSS for HTML styling in components. The most common case is for consistency in the look and feel of an app. In places where UI styles must differ across hosting models or platforms, use CSS to style the differences.
- If some part of the UI requires additional or different content for a target hosting model or platform, the content can be encapsulated inside a component and rendered inside the RCL using DynamicComponent. Additional UI can also be provided to components via RenderFragment instances. For more information on RenderFragment, see Child content render fragments and Render fragments for reusable rendering logic.

Project code organization

As much as possible, place code and static content in Razor class libraries (RCLs). Each hosting model or platform references the RCL and registers individual implementations in the app's service collection that a Razor component might require.

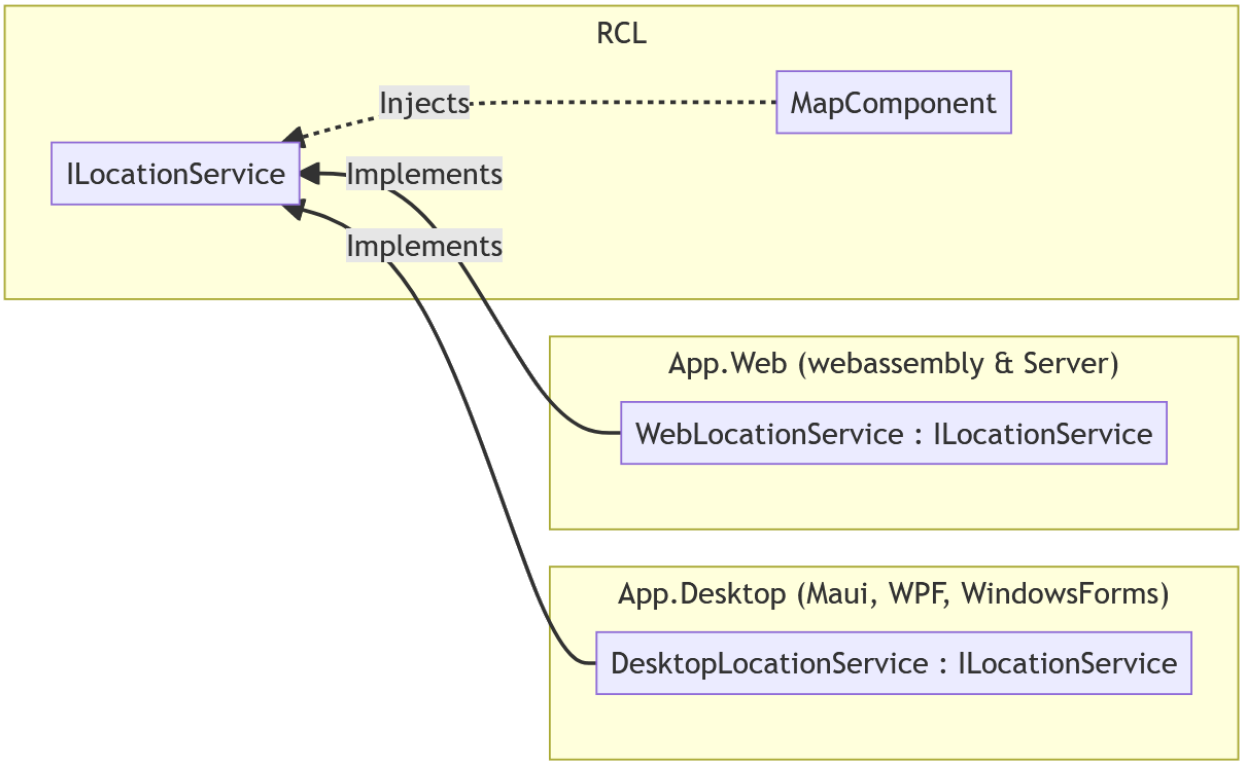
Each target assembly should contain only the code that is specific to that hosting model or platform along with the code that helps bootstrap the app.



Use abstractions for unique features

The following example demonstrates how to use an abstraction for a geolocation service by hosting model and platform.

- In a Razor class library (RCL) used by the app to obtain geolocation data for the user's location on a map, the `MapComponent` Razor component injects an `ILocationService` service abstraction.
- `App.Web` for Blazor WebAssembly and Blazor Server projects implement `ILocationService` as `WebLocationService`, which uses web API calls to obtain geolocation data.
- `App.Desktop` for .NET MAUI, WPF, and Windows Forms, implement `ILocationService` as `DesktopLocationService`. `DesktopLocationService` uses platform-specific device features to obtain geolocation data.



.NET MAUI Blazor platform-specific code

A common pattern in .NET MAUI is to create separate implementations for different platforms, such as defining partial classes with platform-specific implementations. For example, see the following diagram, where partial classes for `CameraService` are implemented in each of `CameraService.Windows.cs`, `CameraService.iOS.cs`, `CameraService.Android.cs`, and `CameraService.cs`:



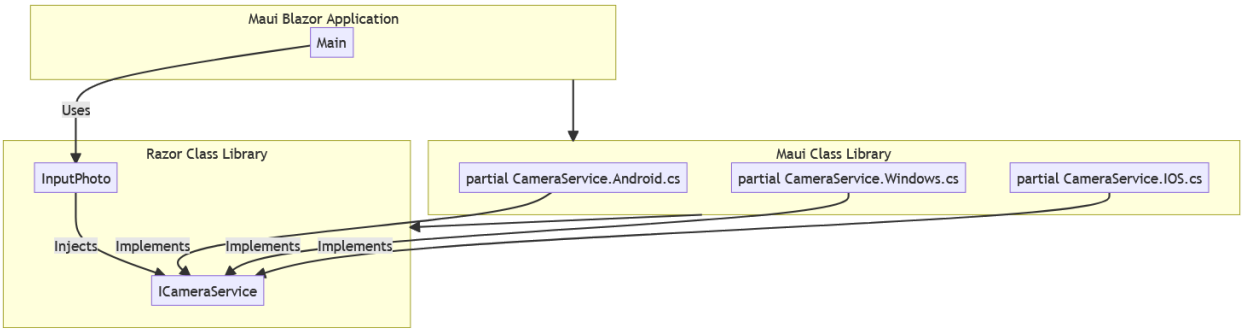
Where you want to pack platform-specific features into a class library that can be consumed by other apps, we recommend that you follow a similar approach to the one described in the preceding example and create an abstraction for the Razor component:

- Place the component in a Razor class library (RCL).
- From a .NET MAUI class library, reference the RCL and create the platform-specific implementations.
- Within the consuming app, reference the .NET MAUI class library.

The following example demonstrates the concepts for images in an app that organizes photographs:

- A .NET MAUI Blazor Hybrid app uses `InputPhoto` from an RCL that it references.
- The .NET MAUI app also references a .NET MAUI class library.

- `InputPhoto` in the RCL injects an `ICameraService` interface, which is defined in the RCL.
- `CameraService` partial class implementations for `ICameraService` are in the .NET MAUI class library (`CameraService.Windows.cs`, `CameraService.iOS.cs`, `CameraService.Android.cs`), which references the RCL.



For an example, see Build a .NET MAUI Blazor Hybrid app with a Blazor Web App.

Additional resources

- eShop Reference Application (AdventureWorks): The .NET MAUI Blazor Hybrid app is in the `src/HybridApp` folder.
  - For Azure hosting: Azure-Samples/eShopOnAzure GitHub repository
  - For general non-Azure hosting: dotnet/eShop GitHub repository.

Share assets across web and native clients using a Razor class library (RCL)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Use a Razor class library (RCL) to share Razor components, C# code, and static assets across web and native client projects.

This article builds on the general concepts found in the following articles:

- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- Reusable Razor UI in class libraries with ASP.NET Core

The examples in this article share assets between a server-side Blazor app and a .NET MAUI Blazor Hybrid app in the same solution:

- Although a server-side Blazor app is used, the guidance applies equally to Blazor WebAssembly apps sharing assets with a Blazor Hybrid app.
- Projects are in the same solution, but an RCL can supply shared assets to projects outside of a solution.
- The RCL is added as a project to the solution, but any RCL can be published as a NuGet package. A NuGet package can supply shared assets to web and native client projects.
- The order that the projects are created isn't important. However, projects that rely on an RCL for assets must create a project reference to the RCL *after* the RCL is created.

For guidance on creating an RCL, see Consume ASP.NET Core Razor components from a Razor class library (RCL). Optionally, access the additional guidance on RCLs that apply broadly to ASP.NET Core apps in Reusable Razor UI in class libraries with ASP.NET Core.

Sample app

For an example of the scenarios described in this article, see the eShop Reference Application (AdventureWorks) (dotnet/eShop GitHub repository). The .NET MAUI Blazor Hybrid app is in the `src/HybridApp` folder.

For a version of the sample app tailored for Azure hosting, see the Azure-Samples/eShopOnAzure GitHub repository.

The sample app showcases the following technologies:

- .NET
- ASP.NET Core
- Blazor
- .NET MAUI
- .NET Aspire
- Docker

Share web UI Razor components, code, and static assets

Components from an RCL can be simultaneously shared by web and native client apps built using Blazor. The guidance in Consume ASP.NET Core Razor components from a Razor class library (RCL) explains how to share Razor components using a Razor class library (RCL). The same guidance applies to reusing Razor components from an RCL in a Blazor Hybrid app.

Component namespaces are derived from the RCL's package ID or assembly name and the component's folder path within the RCL. For more information, see ASP.NET Core Razor components. @using directives can be placed in `_Imports.razor` files for components and code, as the following example demonstrates for an RCL named `SharedLibrary` with a `Shared` folder of shared Razor components and a `Data` folder of shared data classes:

razor

```
@using SharedLibrary
@using SharedLibrary.Shared
@using SharedLibrary.Data
```

Place shared static assets in the RCL's `wwwroot` folder and update static asset paths in the app to use the following path format:

> _content/{PACKAGE ID/ASSEMBLY NAME}/{PATH}/{FILE NAME}

Placeholders:

- `{PACKAGE ID/ASSEMBLY NAME}`: The package ID or assembly name of the RCL.

- `{PATH}`: Optional path within the RCL's `wwwroot` folder.
- `{FILE NAME}`: The file name of the static asset.

The preceding path format is also used in the app for static assets supplied by a NuGet package added to the RCL.

For an RCL named `SharedLibrary` and using the minified Bootstrap stylesheet as an example:

> _content/SharedLibrary/css/bootstrap/bootstrap.min.css

For additional information on how to share static assets across projects, see the following articles:

- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- Reusable Razor UI in class libraries with ASP.NET Core

The root `index.html` file is usually specific to the app and should remain in the Blazor Hybrid app or the Blazor WebAssembly app. The `index.html` file typically isn't shared.

The root Razor Component (`App.razor` or `Main.razor`) can be shared, but often might need to be specific to the hosting app. For example, `App.razor` is different in the server-side Blazor and Blazor WebAssembly project templates when authentication is enabled. You can add the AdditionalAssemblies parameter to specify the location of any shared routable components, and you can specify a shared default layout component for the router by type name.

Provide code and services independent of hosting model

When code must differ across hosting models or target platforms, abstract the code as interfaces and inject the service implementations in each project.

The following weather data example abstracts different weather forecast service implementations:

- Using an HTTP request for Blazor Hybrid and Blazor WebAssembly.
- Requesting data directly for a server-side Blazor app.

The example uses the following specifications and conventions:

- The RCL is named `SharedLibrary` and contains the following folders and namespaces:
  - `Data`: Contains the `WeatherForecast` class, which serves as a model for weather data.
  - `Interfaces`: Contains the service interface for the service implementations, named `IWeatherForecastService`.
- The `FetchData` component is maintained in the `Pages` folder of the RCL, which is routable by any of the apps consuming the RCL.
- Each Blazor app maintains a service implementation that implements the `IWeatherForecastService` interface.

`Data/WeatherForecast.cs` in the RCL:

C#

```
namespace SharedLibrary.Data;

public class WeatherForecast
{
    public DateTime Date { get; set; }
    public int TemperatureC { get; set; }
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    public string? Summary { get; set; }
}
```

`Interfaces/IWeatherForecastService.cs` in the RCL:

C#

```
using SharedLibrary.Data;

namespace SharedLibrary.Interfaces;

public interface IWeatherForecastService
{
    Task<WeatherForecast[]?> GetForecastAsync(DateTime startDate);
}
```

The `_Imports.razor` file in the RCL includes the following added namespaces:

razor

```
@using SharedLibrary.Data
@using SharedLibrary.Interfaces
```

`Services/WeatherForecastService.cs` in the Blazor Hybrid and Blazor WebAssembly apps:

C#

```
using System.Net.Http.Json;
using SharedLibrary.Data;
using SharedLibrary.Interfaces;

namespace {APP NAMESPACE}.Services;

public class WeatherForecastService : IWeatherForecastService
{
    private readonly HttpClient http;

    public WeatherForecastService(HttpClient http)
    {
        this.http = http;
    }

    public async Task<WeatherForecast[]?> GetForecastAsync(DateTime
startDate) =>
        await http.GetFromJsonAsync<WeatherForecast[]?>("WeatherForecast");
}
```

In the preceding example, the `{APP NAMESPACE}` placeholder is the app's namespace.

`Services/WeatherForecastService.cs` in the server-side Blazor app:

C#

```
using SharedLibrary.Data;
using SharedLibrary.Interfaces;

namespace {APP NAMESPACE}.Services;

public class WeatherForecastService : IWeatherForecastService
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
"Hot"
    };

    public async Task<WeatherForecast[]?> GetForecastAsync(DateTime
startDate) =>
        await Task.FromResult(Enumerable.Range(1, 5)
            .Select(index => new WeatherForecast
            {
                Date = startDate.AddDays(index),
                TemperatureC = Random.Shared.Next(-20, 55),
                Summary = Summaries[Random.Shared.Next(Summaries.Length)]
```

```
            }).ToArray());
}
```

In the preceding example, the `{APP NAMESPACE}` placeholder is the app's namespace.

The Blazor Hybrid, Blazor WebAssembly, and server-side Blazor apps register their weather forecast service implementations (`Services.WeatherForecastService`) for `IWeatherForecastService`.

The Blazor WebAssembly project also registers an HttpClient. The HttpClient registered in an app created from the Blazor WebAssembly project template is sufficient for this purpose. For more information, see Call a web API from an ASP.NET Core Blazor app.

`Pages/FetchData.razor` in the RCL:

razor

```razor
@page "/fetchdata"
@inject IWeatherForecastService ForecastService

<PageTitle>Weather forecast</PageTitle>

<h1>Weather forecast</h1>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
```

```
}

@code {
    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await ForecastService.GetForecastAsync(DateTime.Now);
    }
}
```

Additional resources

- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- Reusable Razor UI in class libraries with ASP.NET Core
- CSS isolation support with Razor class libraries

Pass root component parameters in ASP.NET Core Blazor Hybrid

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to pass root component parameters in a Blazor Hybrid app.

The `RootComponent` class of a `BlazorWebView` defines a `Parameters` property of type `IDictionary<string, object?>?`, which represents an optional dictionary of parameters to pass to the root component:

- .NET MAUI: Microsoft.AspNetCore.Components.WebView.Maui.RootComponent
- WPF: Microsoft.AspNetCore.Components.WebView.Wpf.RootComponent
- Windows Forms: Microsoft.AspNetCore.Components.WebView.WindowsForms.RootComponent

The following example passes a view model to the root component, which further passes the view model as a cascading type to a Razor component in the Blazor portion of the app. The example is based on the keypad example in the .NET MAUI documentation:

- Data binding and MVVM: Commanding (.NET MAUI documentation): Explains data binding with MVVM using a keypad example.
- .NET MAUI Samples

Although the keypad example focuses on implementing the MVVM pattern in .NET MAUI Blazor Hybrid apps:

- The dictionary of objects passed to root components can include any type for any purpose where you need to pass one or more parameters to the root component for use by Razor components in the app.
- The concepts demonstrated by the following .NET MAUI Blazor example are the same for Windows Forms Blazor apps and WPF Blazor apps.

Place the following view model into your .NET MAUI Blazor Hybrid app.

`KeypadViewModel.cs`:

C#

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

namespace MauiBlazor;

public class KeypadViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private string _inputString = "";
    private string _displayText = "";
    private char[] _specialChars = { '*', '#' };

    public ICommand AddCharCommand { get; private set; }
    public ICommand DeleteCharCommand { get; private set; }

    public string InputString
    {
        get => _inputString;
        private set
        {
            if (_inputString != value)
            {
                _inputString = value;
                OnPropertyChanged();
                DisplayText = FormatText(_inputString);

                // Perhaps the delete button must be enabled/disabled.
                ((Command)DeleteCharCommand).ChangeCanExecute();
            }
        }
    }

    public string DisplayText
    {
        get => _displayText;
        set
        {
            if (_displayText != value)
            {
                _displayText = value;
                OnPropertyChanged();
            }
        }
    }

    public KeypadViewModel()
    {
        // Command to add the key to the input string
        AddCharCommand = new Command<string>((key) => InputString += key);
```

```
        // Command to delete a character from the input string when allowed
        DeleteCharCommand =
            new Command(
                // Command strips a character from the input string
                () => InputString = InputString.Substring(0,
InputString.Length - 1),

                // CanExecute is processed here to return true when there's
something to delete
                () => InputString.Length > 0
            );
    }

    string FormatText(string str)
    {
        bool hasNonNumbers = str.IndexOfAny(_specialChars) != -1;
        string formatted = str;

        // Format the string based on the type of data and the length
        if (hasNonNumbers || str.Length < 4 || str.Length > 10)
        {
            // Special characters exist, or the string is too small or large
for special formatting
            // Do nothing
        }

        else if (str.Length < 8)
            formatted = string.Format("{0}-{1}", str.Substring(0, 3),
str.Substring(3));

        else
            formatted = string.Format("({0}) {1}-{2}", str.Substring(0, 3),
str.Substring(3, 3), str.Substring(6));

        return formatted;
    }

    public void OnPropertyChanged([CallerMemberName] string name = "") =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

In this article's example, the app's root namespace is `MauiBlazor`. Change the namespace of `KeypadViewModel` to match the app's root namespace:

C#

```
namespace MauiBlazor;
```

> **Note**

At the time the `KeypadViewModel` view model was created for the .NET MAUI sample app and the .NET MAUI documentation, view models were placed in a folder named `ViewModels`, but the namespace was set to the root of the app and didn't include the folder name. If you wish to update the namespace to include the folder in the `KeypadViewModel.cs` file, modify the example code in this article to match. Add `using` (C#) and `@using` (Razor) statements to the following files or fully-qualify the references to the view model type as `{APP NAMESPACE}.ViewModels.KeypadViewModel`, where the `{APP NAMESPACE}` placeholder is the app's root namespace.

Although you can set `Parameters` directly in XAML, the following example names the root component (`rootComponent`) in the XAML file and sets the parameter dictionary in the code-behind file.

In `MainPage.xaml`:

XAML

```
<RootComponent x:Name="rootComponent"
               Selector="#app"
               ComponentType="{x:Type local:Main}" />
```

In the code-behind file (`MainPage.xaml.cs`), assign the view model in the constructor:

C#

```
public MainPage()
{
    InitializeComponent();

    rootComponent.Parameters =
        new Dictionary<string, object>
        {
            { "KeypadViewModel", new KeypadViewModel() }
        };
}
```

The following example cascades the object (`KeypadViewModel`) down component hierarchies in the Blazor portion of the app as a CascadingValue.

In the `Main` component (`Main.razor`):

- Add a parameter matching the type of the object passed to the root component:

razor

```razor
@code {
    [Parameter]
    public KeypadViewModel KeypadViewModel { get; set; }
}
```

- Cascade the `KeypadViewModel` with the CascadingValue component. Update the `<Found>` XAML content to the following markup:

XAML

```xaml
<Found Context="routeData">
    <CascadingValue Value="KeypadViewModel">
        <RouteView RouteData="routeData"
DefaultLayout="typeof(MainLayout)" />
        <FocusOnNavigate RouteData="routeData" Selector="h1"/>
    </CascadingValue>
</Found>
```

At this point, the cascaded type is available to Razor components throughout the app as a CascadingParameter.

The following `Keypad` component example:

- Displays the current value of `KeypadViewModel.DisplayText`.
- Permits character deletion by calling the `KeypadViewModel.DeleteCharCommand` command if the display string length is greater than 0 (zero), which is checked by the call to ICommand.CanExecute.
- Permits adding characters by calling `KeypadViewModel.AddCharCommand` with the key pressed in the UI.

`Pages/Keypad.razor`:

razor

```razor
@page "/keypad"

<h1>Keypad</h1>

<table id="keypad">
    <thead>
        <tr>
            <th colspan="2">@KeypadViewModel.DisplayText</th>
            <th><button @onclick="DeleteChar">&#x21E6;</button></th>
        </tr>
    </thead>
    <tbody>
```

```
        <tr>
            <td><button @onclick="@(e => AddChar("1"))">1</button></td>
            <td><button @onclick="@(e => AddChar("2"))">2</button></td>
            <td><button @onclick="@(e => AddChar("3"))">3</button></td>
        </tr>
        <tr>
            <td><button @onclick="@(e => AddChar("4"))">4</button></td>
            <td><button @onclick="@(e => AddChar("5"))">5</button></td>
            <td><button @onclick="@(e => AddChar("6"))">6</button></td>
        </tr>
        <tr>
            <td><button @onclick="@(e => AddChar("7"))">7</button></td>
            <td><button @onclick="@(e => AddChar("8"))">8</button></td>
            <td><button @onclick="@(e => AddChar("9"))">9</button></td>
        </tr>
        <tr>
            <td><button @onclick="@(e => AddChar("*"))">*</button></td>
            <td><button @onclick="@(e => AddChar("0"))">0</button></td>
            <td><button @onclick="@(e => AddChar("#"))">#</button></td>
        </tr>
    </tbody>
</table>

@code {
    [CascadingParameter]
    protected KeypadViewModel KeypadViewModel { get; set; }

    private void DeleteChar()
    {
        if (KeypadViewModel.DeleteCharCommand.CanExecute(null))
        {
            KeypadViewModel.DeleteCharCommand.Execute(null);
        }
    }

    private void AddChar(string key)
    {
        KeypadViewModel.AddCharCommand.Execute(key);
    }
}
```

Purely for demonstration purposes, style the buttons by placing the following CSS styles in the `wwwroot/index.html` file's `<head>` content:

HTML

```
<style>
    #keypad button {
        border: 1px solid black;
        border-radius:6px;
        height: 35px;
        width:80px;
```

```
    }
</style>
```

Create a sidebar navigation entry in the `NavMenu` component (`Shared/NavMenu.razor`) with the following markup:

razor

```razor
<div class="nav-item px-3">
    <NavLink class="nav-link" href="keypad">
        <span class="oi oi-list-rich" aria-hidden="true"></span> Keypad
    </NavLink>
</div>
```

Additional resources

- Host a Blazor web app in a .NET MAUI app using BlazorWebView
- Data binding and MVVM: Commanding (.NET MAUI documentation)
- ASP.NET Core Blazor cascading values and parameters

ASP.NET Core Blazor Hybrid authentication and authorization

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes ASP.NET Core's support for the configuration and management of security and ASP.NET Core Identity in Blazor Hybrid apps.

Authentication in Blazor Hybrid apps is handled by native platform libraries, as they offer enhanced security guarantees that the browser sandbox can't offer. Authentication of native apps uses an OS-specific mechanism or via a federated protocol, such as OpenID Connect (OIDC). Follow the guidance for the identity provider that you've selected for the app and then further integrate identity with Blazor using the guidance in this article.

Integrating authentication must achieve the following goals for Razor components and services:

- Use the abstractions in the Microsoft.AspNetCore.Components.Authorization package, such as AuthorizeView.
- React to changes in the authentication context.
- Access credentials provisioned by the app from the identity provider, such as access tokens to perform authorized API calls.

After authentication is added to a .NET MAUI, WPF, or Windows Forms app and users are able to log in and log out successfully, integrate authentication with Blazor to make the authenticated user available to Razor components and services. Perform the following steps:

- Reference the Microsoft.AspNetCore.Components.Authorization package.

  > **Note**
  >
  > For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

- Implement a custom AuthenticationStateProvider, which is the abstraction that Razor components use to access information about the authenticated user and to receive updates when the authentication state changes.
- Register the custom authentication state provider in the dependency injection container.

.NET MAUI apps use the `WebAuthenticator` class to initiate browser-based authentication flows that listen for a callback to a specific URL registered with the app.

For additional guidance, see the following resources:

- Web authenticator (.NET MAUI documentation)
- Sample.Server.WebAuthenticator sample app

Create a custom `AuthenticationStateProvider` without user change updates

If the app authenticates the user immediately after the app launches and the authenticated user remains the same for the entirety of the app lifetime, user change notifications aren't required, and the app only provides information about the authenticated user. In this scenario, the user logs into the app when the app is opened, and the app displays the login screen again after the user logs out. The following `ExternalAuthStateProvider` is an example implementation of a custom AuthenticationStateProvider for this authentication scenario.

> **Note**
>
> The following custom AuthenticationStateProvider doesn't declare a namespace in order to make the code example applicable to any Blazor Hybrid app. However, a best practice is to provide your app's namespace when you implement the example in a production app.

`ExternalAuthStateProvider.cs`:

C#

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class ExternalAuthStateProvider : AuthenticationStateProvider
{
```

```
    private readonly Task<AuthenticationState> authenticationState;

    public ExternalAuthStateProvider(AuthenticatedUser user) =>
        authenticationState = Task.FromResult(new
AuthenticationState(user.Principal));

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        authenticationState;
}

public class AuthenticatedUser
{
    public ClaimsPrincipal Principal { get; set; } = new();
}
```

The following steps describe how to:

- Add required namespaces.
- Add the authorization services and Blazor abstractions to the service collection.
- Build the service collection.
- Resolve the `AuthenticatedUser` service to set the authenticated user's claims principal. See your identity provider's documentation for details.
- Return the built host.

In the `MauiProgram.CreateMauiApp` method of `MauiProgram.cs`, add namespaces for Microsoft.AspNetCore.Components.Authorization and System.Security.Claims:

C#

```
using Microsoft.AspNetCore.Components.Authorization;
using System.Security.Claims;
```

Remove the following line of code that returns a built Microsoft.Maui.Hosting.MauiApp:

diff

```
- return builder.Build();
```

Replace the preceding line of code with the following code. Add OpenID/MSAL code to authenticate the user. See your identity provider's documentation for details.

C#

```
builder.Services.AddAuthorizationCore();
builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
```

```
builder.Services.AddSingleton<AuthenticatedUser>();
var host = builder.Build();

var authenticatedUser = host.Services.GetRequiredService<AuthenticatedUser>
();

/*
Provide OpenID/MSAL code to authenticate the user. See your identity
provider's
documentation for details.

The user is represented by a new ClaimsPrincipal based on a new
ClaimsIdentity.
*/
var user = new ClaimsPrincipal(new ClaimsIdentity());

authenticatedUser.Principal = user;

return host;
```

Create a custom `AuthenticationStateProvider` with user change updates

To update the user while the Blazor app is running, call NotifyAuthenticationStateChanged within the AuthenticationStateProvider implementation using ***either*** of the following approaches:

- Signal an authentication update from outside of the BlazorWebView)
- Handle authentication within the BlazorWebView

Signal an authentication update from outside of the `BlazorWebView` (Option 1)

A custom AuthenticationStateProvider can use a global service to signal an authentication update. We recommend that the service offer an event that the AuthenticationStateProvider can subscribe to, where the event invokes NotifyAuthenticationStateChanged.

> ⓘ Note
>
> The following custom AuthenticationStateProvider doesn't declare a namespace in order to make the code example applicable to any Blazor Hybrid app. However, a

best practice is to provide your app's namespace when you implement the example in a production app.

`ExternalAuthStateProvider.cs`:

C#

```
using System;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class ExternalAuthStateProvider : AuthenticationStateProvider
{
    private AuthenticationState currentUser;

    public ExternalAuthStateProvider(ExternalAuthService service)
    {
        currentUser = new AuthenticationState(service.CurrentUser);

        service.UserChanged += (newUser) =>
        {
            currentUser = new AuthenticationState(newUser);
            NotifyAuthenticationStateChanged(Task.FromResult(currentUser));
        };
    }

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        Task.FromResult(currentUser);
}

public class ExternalAuthService
{
    public event Action<ClaimsPrincipal>? UserChanged;
    private ClaimsPrincipal? currentUser;

    public ClaimsPrincipal CurrentUser
    {
        get { return currentUser ?? new(); }
        set
        {
            currentUser = value;

            if (UserChanged is not null)
            {
                UserChanged(currentUser);
            }
        }
    }
}
```

In the `MauiProgram.CreateMauiApp` method of `MauiProgram.cs`, add a namespace for Microsoft.AspNetCore.Components.Authorization:

```
C#

using Microsoft.AspNetCore.Components.Authorization;
```

Add the authorization services and Blazor abstractions to the service collection:

```
C#

builder.Services.AddAuthorizationCore();
builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
builder.Services.AddSingleton<ExternalAuthService>();
```

Wherever the app authenticates a user, resolve the `ExternalAuthService` service:

```
C#

var authService = host.Services.GetRequiredService<ExternalAuthService>();
```

Execute your custom OpenID/MSAL code to authenticate the user. See your identity provider's documentation for details. The authenticated user (`authenticatedUser` in the following example) is a new ClaimsPrincipal based on a new ClaimsIdentity.

Set the current user to the authenticated user:

```
C#

authService.CurrentUser = authenticatedUser;
```

An alternative to the preceding approach is to set the user's principal on System.Threading.Thread.CurrentPrincipal instead of setting it via a service, which avoids use of the dependency injection container:

```
C#

public class CurrentThreadUserAuthenticationStateProvider :
AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        Task.FromResult(
            new AuthenticationState(Thread.CurrentPrincipal as
ClaimsPrincipal ??
```

```
                new ClaimsPrincipal(new ClaimsIdentity())));
}
```

Using the alternative approach, only authorization services (AddAuthorizationCore) and `CurrentThreadUserAuthenticationStateProvider` (`.TryAddScoped<AuthenticationStateProvider, CurrentThreadUserAuthenticationStateProvider>()`) are added to the service collection.

Handle authentication within the `BlazorWebView` (Option 2)

A custom AuthenticationStateProvider can include additional methods to trigger log in and log out and update the user.

> **Note**
>
> The following custom AuthenticationStateProvider doesn't declare a namespace in order to make the code example applicable to any Blazor Hybrid app. However, a best practice is to provide your app's namespace when you implement the example in a production app.

`ExternalAuthStateProvider.cs`:

C#

```
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.Authorization;

public class ExternalAuthStateProvider : AuthenticationStateProvider
{
    private ClaimsPrincipal currentUser = new ClaimsPrincipal(new
ClaimsIdentity());

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        Task.FromResult(new AuthenticationState(currentUser));

    public Task LogInAsync()
    {
        var loginTask = LogInAsyncCore();
        NotifyAuthenticationStateChanged(loginTask);

        return loginTask;

        async Task<AuthenticationState> LogInAsyncCore()
```

```
        {
            var user = await LoginWithExternalProviderAsync();
            currentUser = user;

            return new AuthenticationState(currentUser);
        }
    }

    private Task<ClaimsPrincipal> LoginWithExternalProviderAsync()
    {
        /*
            Provide OpenID/MSAL code to authenticate the user. See your
identity
            provider's documentation for details.

            Return a new ClaimsPrincipal based on a new ClaimsIdentity.
        */
        var authenticatedUser = new ClaimsPrincipal(new ClaimsIdentity());

        return Task.FromResult(authenticatedUser);
    }

    public void Logout()
    {
        currentUser = new ClaimsPrincipal(new ClaimsIdentity());
        NotifyAuthenticationStateChanged(
            Task.FromResult(new AuthenticationState(currentUser)));
    }
}
```

In the preceding example:

- The call to `LogInAsyncCore` triggers the login process.
- The call to NotifyAuthenticationStateChanged notifies that an update is in progress, which allows the app to provide a temporary UI during the login or logout process.
- Returning `loginTask` returns the task so that the component that triggered the login can await and react after the task is complete.
- The `LoginWithExternalProviderAsync` method is implemented by the developer to log in the user with the identity provider's SDK. For more information, see your identity provider's documentation. The authenticated user (`authenticatedUser`) is a new ClaimsPrincipal based on a new ClaimsIdentity.

In the `MauiProgram.CreateMauiApp` method of `MauiProgram.cs`, add the authorization services and the Blazor abstraction to the service collection:

```
C#
```

```
builder.Services.AddAuthorizationCore();
builder.Services.TryAddScoped<AuthenticationStateProvider,
ExternalAuthStateProvider>();
```

The following `LoginComponent` component demonstrates how to log in a user. In a typical app, the `LoginComponent` component is only shown in a parent component if the user isn't logged into the app.

`Shared/LoginComponent.razor`:

razor

```
@inject AuthenticationStateProvider AuthenticationStateProvider

<button @onclick="Login">Log in</button>

@code
{
    public async Task Login()
    {
        await ((ExternalAuthStateProvider)AuthenticationStateProvider)
            .LogInAsync();
    }
}
```

The following `LogoutComponent` component demonstrates how to log out a user. In a typical app, the `LogoutComponent` component is only shown in a parent component if the user is logged into the app.

`Shared/LogoutComponent.razor`:

razor

```
@inject AuthenticationStateProvider AuthenticationStateProvider

<button @onclick="Logout">Log out</button>

@code
{
    public async Task Logout()
    {
        await ((ExternalAuthStateProvider)AuthenticationStateProvider)
            .Logout();
    }
}
```

Accessing other authentication information

Blazor doesn't define an abstraction to deal with other credentials, such as access tokens to use for HTTP requests to web APIs. We recommend following the identity provider's guidance to manage the user's credentials with the primitives that the identity provider's SDK provides.

It's common for identity provider SDKs to use a token store for user credentials stored in the device. If the SDK's token store primitive is added to the service container, consume the SDK's primitive within the app.

The Blazor framework isn't aware of a user's authentication credentials and doesn't interact with credentials in any way, so the app's code is free to follow whatever approach you deem most convenient. However, follow the general security guidance in the next section, Other authentication security considerations, when implementing authentication code in an app.

Other authentication security considerations

The authentication process is external to Blazor, and we recommend that developers access the identity provider's guidance for additional security guidance.

When implementing authentication:

- Avoid authentication in the context of the Web View. For example, avoid using a JavaScript OAuth library to perform the authentication flow. In a single-page app, authentication tokens aren't hidden in JavaScript and can be easily discovered by malicious users and used for nefarious purposes. Native apps don't suffer this risk because native apps are only able to obtain tokens outside of the browser context, which means that rogue third-party scripts can't steal the tokens and compromise the app.
- Avoid implementing the authentication workflow yourself. In most cases, platform libraries securely handle the authentication workflow, using the system's browser instead of using a custom Web View that can be hijacked.
- Avoid using the platform's Web View control to perform authentication. Instead, rely on the system's browser when possible.
- Avoid passing the tokens to the document context (JavaScript). In some situations, a JavaScript library within the document is required to perform an authorized call to an external service. Instead of making the token available to JavaScript via JS interop:
  - Provide a generated temporary token to the library and within the Web View.
  - Intercept the outgoing network request in code.

    - Replace the temporary token with the real token and confirm that the destination of the request is valid.

Additional resources

- ASP.NET Core Blazor authentication and authorization
- ASP.NET Core Blazor Hybrid security considerations
- Entra ID documentation for .NET MAUI
    - Tutorial: Register and configure .NET MAUI mobile app in an external tenant
    - Sign in users in a sample .NET MAUI Android application
    - Sign in users in a sample .NET MAUI desktop application
- Azure documentation for .NET MAUI
    - Add authentication to your .NET MAUI app
    - MAUI mobile or desktop application using Microsoft Entra ID for authentication

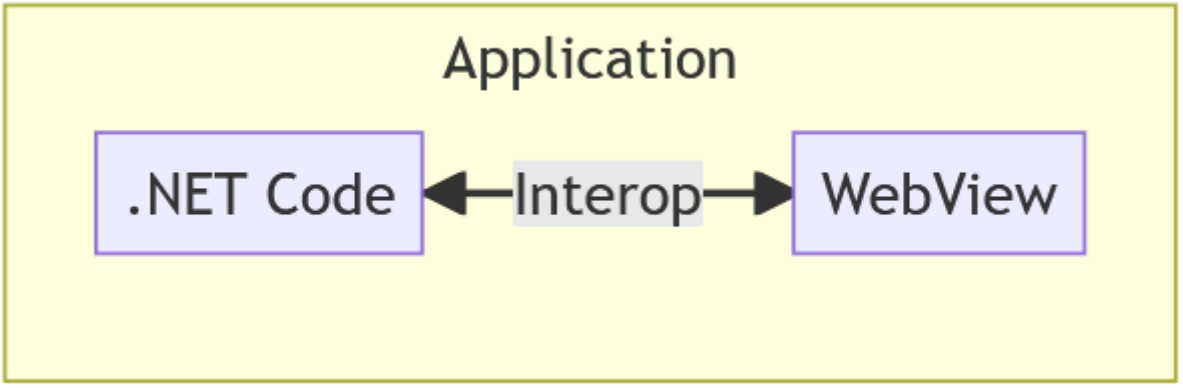
ASP.NET Core Blazor Hybrid security considerations

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes security considerations for Blazor Hybrid apps.

Blazor Hybrid apps that render web content execute .NET code inside a platform Web View. The .NET code interacts with the web content via an interop channel between the .NET code and the Web View.



The web content rendered into the Web View can come from assets provided by the app from either of the following locations:

- The `wwwroot` folder in the app.
- A source external to the app. For example, a network source, such as the Internet.

A trust boundary exists between the .NET code and the code that runs inside the Web View. .NET code is provided by the app and any trusted third-party packages that you've installed. After the app is built, the .NET code Web View content sources can't change.

In contrast to the .NET code sources of content, content sources from the code that runs inside the Web View can come not only from the app but also from external sources. For example, static assets from an external Content Delivery Network (CDN) might be used or rendered by an app's Web View.

Consider the code inside the Web View as **untrusted** in the same way that code running inside the browser for a web app isn't trusted. The same threats and general security

recommendations apply to untrusted resources in Blazor Hybrid apps as for other types of apps.

If possible, avoid loading content from a third-party origin. To mitigate risk, you might be able to serve content directly from the app by downloading the external assets, verifying that they're safe to serve to users, and placing them into the app's `wwwroot` folder for packaging with the rest of the app. When the external content is downloaded for inclusion in the app, we recommend scanning it for viruses and malware before placing it into the `wwwroot` folder of the app.

If your app must reference content from an external origin, we recommend that you use common web security approaches to provide the app with an opportunity to block the content from loading if the content is compromised:

- Serve content securely with TLS/HTTPS.
- Institute a Content Security Policy (CSP).
- Perform subresource integrity checks.

Even if all of the resources are packed into the app and don't load from any external origin, remain cautious about problems in the resources' code that run inside the Web View, as the resources might have vulnerabilities that could allow cross-site scripting (XSS) attacks.

In general, the Blazor framework protects against XSS by dealing with HTML in safe ways. However, some programming patterns allow Razor components to inject raw HTML into rendered output, such as rendering content from an untrusted source. For example, rendering HTML content directly from a database should be avoided. Additionally, JavaScript libraries used by the app might manipulate HTML in unsafe ways to inadvertently or deliberately render unsafe output.

For these reasons, it's best to apply the same protections against XSS that are normally applied to web apps. Prevent loading scripts from unknown sources and don't implement potentially unsafe JavaScript features, such as eval and other unsafe JavaScript primitives. Establishing a CSP is recommended to reduce these security risks.

If the code inside the Web View is compromised, the code gains access to all of the content inside the Web View and might interact with the host via the interop channel. For that reason, any content coming from the Web View (events, JS interop) must be treated as **untrusted** and validated in the same way as for other sensitive contexts, such as in a compromised Blazor Server app that can lead to malicious attacks on the host system.

Don't store sensitive information, such as credentials, security tokens, or sensitive user data, in the context of the Web View, as it makes the information available to a cyberattacker if the Web View is compromised. There are safer alternatives, such as handling the sensitive information directly within the native portion of the app.

External content rendered in an `iframe`

When using an iframe to display external content within a Blazor Hybrid page, we recommend that users leverage sandboxing features to ensure that the content is isolated from the parent page containing the app. In the following Razor component example, the sandbox attribute is present for the `<iframe>` tag to apply sandboxing features to the `admin.html` page:

razor

```razor
<iframe sandbox src="https://contoso.com/admin.html" />
```

> ⚠ **Warning**
>
> The sandbox attribute isn't supported in early browser versions. For more information, see Can I use: sandbox.

Links to external URLs

Links to URLs outside of the app are opened in an appropriate external app, not loaded within the Web View. We don't recommend overriding the default behavior.

Keep the Web View current in deployed apps

The BlazorWebView control uses the currently-installed, platform-specific native Web View. Since the native Web View is periodically updated with support for new APIs and fixes for security issues, it may be necessary to ensure that an app is using a Web View version that meets the app's requirements.

Use one of the following approaches to keep the Web View current in deployed apps:

- **On all platforms**: Check the Web View version and prompt the user to take any necessary steps to update it.
- **Only on Windows**: Package a fixed-version Web View within the app, using it in place of the system's shared Web View.

Android

The Android Web View is distributed and updated via the Google Play Store. Check the Web View version by reading the User-Agent string. Read the Web View's navigator.userAgent property using JavaScript interop and optionally cache the value using a singleton service if the user agent string is required outside of a Razor component context.

When using the Android Emulator:

- Use an emulated device with **Google Play Services** preinstalled. Emulated devices without Google Play Services preinstalled aren't supported.
- Install Google Chrome from the Google Play Store. If Google Chrome is already installed, update Chrome from the Google Play Store. If an emulated device doesn't have the latest version of Chrome installed, it might not have the latest version of the Android Web View installed.

iOS/Mac Catalyst

iOS and Mac Catalyst both use WKWebView, a Safari-based control, which is updated by the operating system. Similar to the Android case, determine the Web View version by reading the Web View's User-Agent string.

Windows (.NET MAUI, WPF, Windows Forms)

On Windows, the Chromium-based Microsoft Edge WebView2 is required to run Blazor web apps.

The newest installed version of `WebView2`, known as the *Evergreen distribution*, is used. If you wish to ship a specific version of `WebView2` with the app, use the *Fixed Version distribution*.

For more information on checking the currently-installed `WebView2` version and the distribution modes, see the WebView2 distribution documentation.

Additional resources

- ASP.NET Core Blazor Hybrid authentication and authorization
- ASP.NET Core Blazor authentication and authorization

Publish ASP.NET Core Blazor Hybrid apps

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to publish Blazor Hybrid apps.

Publish for a specific framework

Blazor Hybrid supports .NET MAUI, WPF, and Windows Forms. The publishing steps for apps using Blazor Hybrid are nearly identical to the publishing steps for the target platform.

- WPF and Windows Forms
  - .NET application publishing overview
- .NET MAUI
  - Windows
  - Android
  - iOS
  - macOS

Blazor-specific considerations

Blazor Hybrid apps require a Web View on the host platform. For more information, see Keep the Web View current in deployed Blazor Hybrid apps.

Troubleshoot ASP.NET Core Blazor Hybrid

Article • 11/14/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

BlazorWebView has built-in logging that can help you diagnose problems in your Blazor Hybrid app.

This article explains the steps to use BlazorWebView logging:

- Enable BlazorWebView and related components to log diagnostic information.
- Configure logging providers.
- View logger output.

Enable `BlazorWebView` logging

Enable logging configuration during service registration. To enable maximum logging for BlazorWebView and related components under the Microsoft.AspNetCore.Components.WebView namespace, add the following code in the `Program` file:

C#

```
services.AddLogging(logging =>
{
    logging.AddFilter("Microsoft.AspNetCore.Components.WebView",
LogLevel.Trace);
});
```

Alternatively, use the following code to enable maximum logging for every component that uses Microsoft.Extensions.Logging:

C#

```
services.AddLogging(logging =>
{
```

```
    logging.SetMinimumLevel(LogLevel.Trace);
});
```

Configure logging providers

After configuring components to write log information, configure where the loggers should write log information.

The **Debug** logging providers write the output using Debug statements.

To configure the **Debug** logging provider, add a reference to the Microsoft.Extensions.Logging.Debug NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Register the provider inside the call to AddLogging added in the previous step by calling the AddDebug extension method:

C#

```
services.AddLogging(logging =>
{
    logging.AddFilter("Microsoft.AspNetCore.Components.WebView",
LogLevel.Trace);
    logging.AddDebug();
});
```

View logger output

When the app is run from Visual Studio with debugging enabled, the debug output appears in Visual Studio's **Output** window.

Additional resources

- Logging in C# and .NET
- Logging in .NET Core and ASP.NET Core

ASP.NET Core Blazor project structure

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes the files and folders that make up a Blazor app generated from a Blazor project template.

Blazor Web App

Blazor Web App project template: `blazor`

The Blazor Web App project template provides a single starting point for using Razor components (`.razor`) to build any style of web UI, both server-side rendered and client-side rendered. It combines the strengths of the existing Blazor Server and Blazor WebAssembly hosting models with server-side rendering, streaming rendering, enhanced navigation and form handling, and the ability to add interactivity using either Blazor Server or Blazor WebAssembly on a per-component basis.

If both client-side rendering (CSR) and interactive server-side rendering (interactive SSR) are selected on app creation, the project template uses the Interactive Auto render mode. The automatic rendering mode initially uses interactive SSR while the .NET app bundle and runtime are downloaded to the browser. After the .NET WebAssembly runtime is activated, rendering switches to CSR.

The Blazor Web App template enables both static and interactive server-side rendering using a single project. If you also enable Interactive WebAssembly rendering, the project includes an additional client project (`.Client`) for your WebAssembly-based components. The built output from the client project is downloaded to the browser and executed on the client. Components using the Interactive WebAssembly or Interactive Auto render modes must be located in the `.Client` project.

The component folder structure of the `.Client` project differs from the Blazor Web App's main project folder structure because the main project is a standard ASP.NET Core project. The main project must take into account other assets for ASP.NET Core projects that are unrelated to Blazor. You're welcome to use whatever component folder structure you wish in the `.Client` project. You're free to mirror the component folder

layout of the main project in the `.Client` project if you wish. Note that namespaces might require adjustments for such assets as layout files if you move components into different folders than the project template uses.

More information on components and render modes is found in the ASP.NET Core Razor components and ASP.NET Core Blazor render modes articles.

Based on the interactive render mode selected at app creation, the `Layout` folder is either in the server project in the `Components` folder or at the root of the `.Client` project. The folder contains the following layout components and stylesheets:

- The `MainLayout` component (`MainLayout.razor`) is the app's layout component.
- The `MainLayout.razor.css` is the stylesheet for the app's main layout.
- The `NavMenu` component (`NavMenu.razor`) implements sidebar navigation. The component includes NavLink components (NavLink), which render navigation links to other Razor components. The NavLink component indicates to the user which component is currently displayed.
- The `NavMenu.razor.css` is the stylesheet for the app's navigation menu.

The `Routes` component (`Routes.razor`) is either in the server project or the `.Client` project and sets up routing using the Router component. For client-side interactive components, the Router component intercepts browser navigation and renders the page that matches the requested address.

The `Components` folder of the server project holds the app's server-side Razor components. Shared components are often placed at the root of the `Components` folder, while layout and page components are usually placed in folders within the `Components` folder.

The `Components/Pages` folder of the server project contains the app's routable server-side Razor components. The route for each page is specified using the @page directive.

The `App` component (`App.razor`) is the root component of the app with HTML `<head>` markup, the `Routes` component, and the Blazor `<script>` tag. The root component is the first component that the app loads.

An `_Imports.razor` file in each of the server and `.Client` projects includes common Razor directives for Razor components of either project, such as @using directives for namespaces.

The `Properties` folder of the server project holds development environment configuration in the `launchSettings.json` file.

> **Note**
>
> The `http` profile precedes the `https` profile in the `launchSettings.json` file. When an app is run with the .NET CLI, the app runs at an HTTP endpoint because the first profile found is `http`. The profile order eases the transition of adopting HTTPS for Linux and macOS users. If you prefer to start the app with the .NET CLI without having to pass the `-lp https` or `--launch-profile https` option to the `dotnet watch` (or `dotnet run`) command, simply place the `https` profile above the `http` profile in the file.

The `wwwroot` folder of the server project is the Web Root folder for the server project that holds the app's public static assets.

The `Program.cs` file of the server project is the project's entry point that sets up the ASP.NET Core web application host and contains the app's startup logic, including service registrations, configuration, logging, and request processing pipeline:

- Services for Razor components are added by calling AddRazorComponents. AddInteractiveServerComponents adds services to support rendering Interactive Server components. AddInteractiveWebAssemblyComponents adds services to support rendering Interactive WebAssembly components.
- MapRazorComponents discovers available components and specifies the root component for the app (the first component loaded), which by default is the `App` component (`App.razor`). AddInteractiveServerRenderMode configures interactive server-side rendering (interactive SSR) for the app. AddInteractiveWebAssemblyRenderMode configures the Interactive WebAssembly render mode for the app.

The app settings files (`appsettings.Development.json`, `appsettings.json`) in either the server or `.Client` project provide configuration settings. In the server project, settings files are at the root of the project. In the `.Client` project, settings files are consumed from the Web Root folder, `wwwroot`.

In the `.Client` project:

- The `Pages` folder contains routable client-side Razor components. The route for each page is specified using the @page directive.
- The `wwwroot` folder is the Web Root folder for the `.Client` project that holds the app's public static assets.

- The `Program.cs` file is the project's entry point that sets up the WebAssembly host and contains the project's startup logic, including service registrations, configuration, logging, and request processing pipeline.

Additional files and folders may appear in an app produced from a Blazor Web App project template when additional options are configured. For example, generating an app with ASP.NET Core Identity includes additional assets for authentication and authorization features.

Standalone Blazor WebAssembly

Standalone Blazor WebAssembly project template: `blazorwasm`

The Blazor WebAssembly template creates the initial files and directory structure for a standalone Blazor WebAssembly app:

- If the `blazorwasm` template is used, the app is populated with the following:
  - Demonstration code for a `Weather` component that loads data from a static asset (`weather.json`) and user interaction with a `Counter` component.
  - Bootstrap frontend toolkit.
- The `blazorwasm` template can also be generated without sample pages and styling.

Project structure:

- `Layout` folder: Contains the following layout components and stylesheets:
  - `MainLayout` component (`MainLayout.razor`): The app's layout component.
  - `MainLayout.razor.css`: Stylesheet for the app's main layout.
  - `NavMenu` component (`NavMenu.razor`): Implements sidebar navigation. Includes the NavLink component (NavLink), which renders navigation links to other Razor components. The NavLink component automatically indicates a selected state when its component is loaded, which helps the user understand which component is currently displayed.
  - `NavMenu.razor.css`: Stylesheet for the app's navigation menu.

- `Pages` folder: Contains the Blazor app's routable Razor components (`.razor`). The route for each page is specified using the @page directive. The template includes the following components:
  - `Counter` component (`Counter.razor`): Implements the Counter page.
  - `Index` component (`Index.razor`): Implements the Home page.
  - `Weather` component (`Weather.razor`): Implements the Weather page.

- `_Imports.razor`: Includes common Razor directives to include in the app's components (`.razor`), such as @using directives for namespaces.
- `App.razor`: The root component of the app that sets up client-side routing using the Router component. The Router component intercepts browser navigation and renders the page that matches the requested address.
- `Properties` folder: Holds development environment configuration in the `launchSettings.json` file.

  > **Note**
  >
  > The `http` profile precedes the `https` profile in the `launchSettings.json` file. When an app is run with the .NET CLI, the app runs at an HTTP endpoint because the first profile found is `http`. The profile order eases the transition of adopting HTTPS for Linux and macOS users. If you prefer to start the app with the .NET CLI without having to pass the `-lp https` or `--launch-profile https` option to the `dotnet watch` (or `dotnet run`) command, simply place the `https` profile above the `http` profile in the file.

- `wwwroot` folder: The Web Root folder for the app containing the app's public static assets, including `appsettings.json` and environmental app settings files for configuration settings and sample weather data (`sample-data/weather.json`). The `index.html` webpage is the root page of the app implemented as an HTML page:
  - When any page of the app is initially requested, this page is rendered and returned in the response.
  - The page specifies where the root `App` component is rendered. The component is rendered at the location of the `div` DOM element with an `id` of `app` (`<div id="app">Loading...</div>`).
- `Program.cs`: The app's entry point that sets up the WebAssembly host:
  - The `App` component is the root component of the app. The `App` component is specified as the `div` DOM element with an `id` of `app` (`<div id="app">Loading...</div>` in `wwwroot/index.html`) to the root component collection (`builder.RootComponents.Add<App>("#app")`).
  - Services are added and configured (for example, `builder.Services.AddSingleton<IMyDependency, MyDependency>()`).

Additional files and folders may appear in an app produced from a Blazor WebAssembly project template when additional options are configured. For example, generating an

app with ASP.NET Core Identity includes additional assets for authentication and authorization features.

Location of the Blazor script

The Blazor script is served from an embedded resource in the ASP.NET Core shared framework.

In a Blazor Web App, the Blazor script is located in the `Components/App.razor` file:

```
HTML
<script src="_framework/blazor.web.js"></script>
```

In a Blazor Server app, the Blazor script is located in the `Pages/_Host.cshtml` file:

```
HTML
<script src="_framework/blazor.server.js"></script>
```

In a Blazor WebAssembly app, the Blazor script content is located in the `wwwroot/index.html` file:

```
HTML
<script src="_framework/blazor.webassembly.js"></script>
```

Location of `<head>` and `<body>` content

In a Blazor Web App, `<head>` and `<body>` content is located in the `Components/App.razor` file.

In a Blazor WebAssembly app, `<head>` and `<body>` content is located in the `wwwroot/index.html` file.

Additional resources

- Tooling for ASP.NET Core Blazor
- ASP.NET Core Blazor hosting models
- Minimal APIs overview
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor fundamentals

Article • 12/20/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Fundamentals articles provide guidance on foundational Blazor concepts. Some of the concepts are connected to a basic understanding of *Razor components*, which are described further in the next section of this article and covered in detail in the *Components* articles.

Static and interactive rendering concepts

Razor components are either *statically* rendered or *interactively* rendered.

Static or *static rendering* is a server-side scenario that means the component is rendered without the capacity for interplay between the user and .NET/C# code. JavaScript and HTML DOM events remain unaffected, but no user events on the client can be processed with .NET running on the server.

Interactive or *interactive rendering* means that the component has the capacity to process .NET events via C# code. The .NET events are either processed on the server by the ASP.NET Core runtime or in the browser on the client by the WebAssembly-based Blazor runtime.

> **Important**
>
> When using a Blazor Web App, most of the Blazor documentation example components ***require*** interactivity to function and demonstrate the concepts covered by the articles. When you test an example component provided by an article, make sure that either the app adopts global interactivity or the component adopts an interactive render mode.

More information on these concepts and how to control static and interactive rendering is found in the ASP.NET Core Blazor render modes article later in the Blazor documentation.

Client and server rendering concepts

Throughout the Blazor documentation, activity that takes place on the user's system is said to occur *on the client* or *client-side*. Activity that takes place on a server is said to occur *on the server* or *server-side*.

The term *rendering* means to produce the HTML markup that browsers display.

- **Client-side rendering (CSR)** means that the final HTML markup is generated by the .NET WebAssembly runtime on the client. No HTML for the app's client-generated UI is sent from a server to the client for this type of rendering. User interactivity with the page is assumed. There's no such concept as *static* client-side rendering. CSR is assumed to be interactive, so "*interactive* client-side rendering" and "*interactive* CSR" aren't used by the industry or in the Blazor documentation.
- **Server-side rendering (SSR)** means that the final HTML markup is generated by the ASP.NET Core runtime on the server. The HTML is sent to the client over a network for display by the client's browser. No HTML for the app's server-generated UI is created by the client for this type of rendering. SSR can be of two varieties:
  - **Static SSR**: The server produces static HTML that doesn't provide for user interactivity or maintaining Razor component state.
  - **Interactive SSR**: Blazor events permit user interactivity and Razor component state is maintained by the Blazor framework.
- **Prerendering** is the process of initially rendering page content on the server without enabling event handlers for rendered controls. The server outputs the HTML UI of the page as soon as possible in response to the initial request, which makes the app feel more responsive to users. Prerendering can also improve Search Engine Optimization (SEO) by rendering content for the initial HTTP response that search engines use to calculate page rank. Prerendering is always followed by final rendering, either on the server or the client.

Razor components

Blazor apps are based on *Razor components*, often referred to as just *components*. A *component* is an element of UI, such as a page, dialog, or data entry form. Components are .NET C# classes built into .NET assemblies.

Razor refers to how components are usually written in the form of a Razor markup page for client-side UI logic and composition. Razor is a syntax for combining HTML markup

with C# code designed for developer productivity. Razor files use the `.razor` file extension.

Although some Blazor developers and online resources use the term "Blazor components," the documentation avoids that term and universally uses "Razor components" or "components."

Blazor documentation adopts several conventions for showing and discussing components:

- Generally, examples adhere to ASP.NET Core/C# coding conventions and engineering guidelines. For more information see the following resources:
  - ASP.NET Core framework engineering guidelines (dotnet/aspnetcore GitHub repository)
  - C# Coding Conventions (C# guide)
- Project code, file paths and names, project template names, and other specialized terms are in United States English and usually code-fenced.
- Components are usually referred to by their C# class name (Pascal case) followed by the word "component." For example, a typical file upload component is referred to as the "`FileUpload` component."
- Usually, a component's C# class name is the same as its file name.
- Routable components usually set their relative URLs to the component's class name in kebab-case. For example, a `FileUpload` component includes routing configuration to reach the rendered component at the relative URL `/file-upload`. Routing and navigation is covered in ASP.NET Core Blazor routing and navigation.
- When multiple versions of a component are used, they're numbered sequentially. For example, the `FileUpload3` component is reached at `/file-upload-3`.
- Razor directives at the top of a component definition (`.razor file`) are placed in the following order: `@page`, `@rendermode` (.NET 8 or later), `@using` statements, other directives in alphabetical order.
- Although not required for `private` members, access modifiers are used in article examples and sample apps. For example, `private` is stated for declaring a field named `maxAllowedFiles` as `private int maxAllowedFiles = 3;`.
- Component parameter values lead with a Razor reserved @ symbol, but it isn't required. Literals (for example, boolean values), keywords (for example, `this`), and `null` as component parameter values aren't prefixed with `@`, but this is also merely a documentation convention. Your own code can prefix literals with `@` if you wish.
- C# classes use the this keyword and avoid prefixing fields with an underscore (`_`) that are assigned to in constructors, which differs from the ASP.NET Core framework engineering guidelines.

- In examples that use primary constructors (C# 12 or later), primary constructor parameters are typically used directly by class members. In article examples, code lines are split to reduce horizontal scrolling. These breaks don't affect execution but can be removed when pasting into your project.

Additional information on Razor component syntax is provided in the *Razor syntax* section of ASP.NET Core Razor components.

The following is an example counter component and part of an app created from a Blazor project template. Detailed components coverage is found in the *Components* articles later in the documentation. The following example demonstrates component concepts seen in the *Fundamentals* articles before reaching the *Components* articles later in the documentation.

`Counter.razor`:

The component assumes that an interactive render mode is inherited from a parent component or applied globally to the app.

razor

```razor
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount() => currentCount++;
}
```

The preceding `Counter` component:

- Sets its route with the `@page` directive in the first line.
- Sets its page title and heading.
- Renders the current count with `@currentCount`. `currentCount` is an integer variable defined in the C# code of the `@code` block.
- Displays a button to trigger the `IncrementCount` method, which is also found in the `@code` block and increases the value of the `currentCount` variable.

Render modes

Articles in the *Fundamentals* node make reference to the concept of *render modes*. This subject is covered in detail in the ASP.NET Core Blazor render modes article in the *Components* node, which appears after the *Fundamentals* node of articles.

For the early references in this node of articles to render mode concepts, merely note the following at this time:

Every component in a Blazor Web App adopts a *render mode* to determine the hosting model that it uses, where it's rendered, and whether or not it's rendered statically on the server, rendered with for user interactivity on the server, or rendered for user interactivity on the client (usually with prerendering on the server).

Blazor Server and Blazor WebAssembly apps for ASP.NET Core releases prior to .NET 8 remain fixated on *hosting model* concepts, not render modes. Render modes are conceptually applied to Blazor Web Apps in .NET 8 or later.

The following table shows the available render modes for rendering Razor components in a Blazor Web App. Render modes are applied to components with the `@rendermode` directive on the component instance or on the component definition. It's also possible to set a render mode for the entire app.

Expand table

Name	Description	Render location	Interactive
Static Server	Static server-side rendering (static SSR)	Server	✗
Interactive Server	Interactive server-side rendering (interactive SSR) using Blazor Server	Server	✓
Interactive WebAssembly	Client-side rendering (CSR) using Blazor WebAssembly†	Client	✓
Interactive Auto	Interactive SSR using Blazor Server initially and then CSR on subsequent visits after the Blazor bundle is downloaded	Server, then client	✓

†Client-side rendering (CSR) is assumed to be interactive. "*Interactive* client-side rendering" and "*interactive* CSR" aren't used by the industry or in the Blazor documentation.

The preceding information on render modes is all that you need to know to understand the *Fundamentals* node articles. If you're new to Blazor and reading Blazor articles in

order down the table of contents, you can delay consuming in-depth information on render modes until you reach the ASP.NET Core Blazor render modes article in the *Components* node.

Document Object Model (DOM)

References to the *Document Object Model* use the abbreviation *DOM*.

For more information, see the following resources:

- Introduction to the DOM (MDN documentation)
- Level 1 Document Object Model Specification (W3C)

Subset of .NET APIs for Blazor WebAssembly apps

A curated list of specific .NET APIs that are supported on the browser for Blazor WebAssembly isn't available. However, you can manually search for a list of .NET APIs annotated with [UnsupportedOSPlatform("browser")] to discover .NET APIs that aren't supported in WebAssembly.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

For more information, see the following resources:

- Class libraries: Client-side browser compatibility analyzer
- Annotating APIs as unsupported on specific platforms (dotnet/designs GitHub repository

Sample apps

Documentation sample apps are available for inspection and download:

Blazor samples GitHub repository (dotnet/blazor-samples)

Locate a sample app by first selecting the version folder that matches the version of .NET that you're working with.

Samples apps in the repository:

- Blazor Web App
- Blazor WebAssembly
- Blazor Web App Movies tutorial sample (Build a Blazor movie database app (Overview))
- Blazor Web App with SignalR (Use ASP.NET Core SignalR with Blazor)
- Two Blazor Web Apps and a Blazor WebAssembly app for calling web (server) APIs (Call a web API from an ASP.NET Core Blazor app)
- Blazor Web App with OIDC (BFF and non-BFF patterns) (Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC))
- Blazor WebAssembly scopes-enabled logging (ASP.NET Core Blazor logging)
- Blazor WebAssembly with ASP.NET Core Identity (Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity)
- .NET MAUI Blazor Hybrid app with a Blazor Web App and a shared UI provided by a Razor class library (RCL) (Build a .NET MAUI Blazor Hybrid app with a Blazor Web App)
- Additional samples (see the Blazor samples GitHub repository README file)

For more information, a list of the samples in the repository, and download instructions, see the Blazor samples GitHub repository README file.

The ASP.NET Core repository's Basic Test App is also a helpful set of samples for various Blazor scenarios:

BasicTestApp in ASP.NET Core reference source (dotnet/aspnetcore)

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Byte multiples

.NET byte sizes use metric prefixes for non-decimal multiples of bytes based on powers of 1024.

Expand table

Name (abbreviation)	Size	Example
Kilobyte (KB)	1,024 bytes	1 KB = 1,024 bytes
Megabyte (MB)	$1{,}024^2$ bytes	1 MB = 1,048,576 bytes
Gigabyte (GB)	$1{,}024^3$ bytes	1 GB = 1,073,741,824 bytes

Support requests

Only documentation-related issues are appropriate for the `dotnet/AspNetCore.Docs` repository. ***For product support, don't open a documentation issue.*** Seek assistance through one or more of the following support channels:

- Stack Overflow (tagged: blazor)
- General ASP.NET Core Slack Team
- Blazor Gitter

For a potential bug in the framework or product feedback, open an issue for the ASP.NET Core product unit at dotnet/aspnetcore issues. Bug reports usually ***require*** the following:

- **Clear explanation of the problem**: Follow the instructions in the GitHub issue template provided by the product unit when opening the issue.
- **Minimal repro project**: Place a project on GitHub for the product unit engineers to download and run. Cross-link the project into the issue's opening comment.

For a potential problem with a Blazor article, open a documentation issue. To open a documentation issue, use the **Open a documentation issue** feedback link at the bottom of the article. Metadata added to your issue provides tracking data and automatically pings the author of the article. If the subject was discussed with the product unit prior to opening the documentation issue, place a cross-link to the engineering issue in the documentation issue's opening comment.

For problems or feedback on Visual Studio, use the **Report a Problem** or **Suggest a Feature** gestures from within Visual Studio, which open internal issues for Visual Studio. For more information, see Visual Studio Feedback.

For problems with Visual Studio Code, ask for support on community support forums. For bug reports and product feedback, open an issue on the microsoft/vscode GitHub repo.

GitHub issues for Blazor documentation are automatically marked for triage on the Blazor.Docs project (dotnet/AspNetCore.Docs GitHub repository). Please wait a short while for a response, especially over weekends and holidays. Usually, documentation authors respond within 24 hours on weekdays.

Community links to Blazor resources

For a collection of links to Blazor resources maintained by the community, visit Awesome Blazor.

> **Note**
>
> Microsoft doesn't own, maintain, or support *Awesome Blazor* and most of the community products and services described and linked there.

ASP.NET Core Blazor routing and navigation

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to manage Blazor app request routing and how to use the NavLink component to create navigation links.

> **Important**
>
> Code examples throughout this article show methods called on `Navigation`, which is an injected NavigationManager in classes and components.

Static versus interactive routing

This section applies to Blazor Web Apps.

If prerendering is enabled, the Blazor router (`Router` component, `<Router>` in `Routes.razor`) performs static routing to components during static server-side rendering (static SSR). This type of routing is called *static routing*.

When an interactive render mode is assigned to the `Routes` component, the Blazor router becomes interactive after static SSR with static routing on the server. This type of routing is called *interactive routing*.

Static routers use endpoint routing and the HTTP request path to determine which component to render. When the router becomes interactive, it uses the document's URL (the URL in the browser's address bar) to determine which component to render. This means that the interactive router can dynamically change which component is rendered if the document's URL dynamically changes to another valid internal URL, and it can do so without performing an HTTP request to fetch new page content.

Interactive routing also prevents prerendering because new page content isn't requested from the server with a normal page request. For more information, see Prerender ASP.NET Core Razor components.

Route templates

The Router component enables routing to Razor components and is located in the app's `Routes` component (`Components/Routes.razor`).

When a Razor component (`.razor`) with an @page directive is compiled, the generated component class is provided a RouteAttribute specifying the component's route template.

When the app starts, the assembly specified as the Router's `AppAssembly` is scanned to gather route information for the app's components that have a RouteAttribute.

At runtime, the RouteView component:

- Receives the RouteData from the Router along with any route parameters.
- Renders the specified component with its layout, including any further nested layouts.

Optionally specify a DefaultLayout parameter with a layout class for components that don't specify a layout with the @layout directive. The framework's Blazor project templates specify the `MainLayout` component (`MainLayout.razor`) as the app's default layout. For more information on layouts, see ASP.NET Core Blazor layouts.

Components support multiple route templates using multiple @page directives. The following example component loads on requests for `/blazor-route` and `/different-blazor-route`.

`BlazorRoute.razor`:

razor

```razor
@page "/blazor-route"
@page "/different-blazor-route"

<PageTitle>Routing</PageTitle>

<h1>Routing Example</h1>

<p>
    This page is reached at either <code>/blazor-route</code> or
    <code>/different-blazor-route</code>.
</p>
```

> **Important**
>
> For URLs to resolve correctly, the app must include a `<base>` tag (location of <head> content) with the app base path specified in the `href` attribute. For more information, see Host and deploy ASP.NET Core Blazor.

As an alternative to specifying the route template as a string literal with the `@page` directive, constant-based route templates can be specified with the @attribute directive.

In the following example, the `@page` directive in a component is replaced with the `@attribute` directive and the constant-based route template in `Constants.CounterRoute`, which is set elsewhere in the app to "`/counter`":

diff

```diff
- @page "/counter"
+ @attribute [Route(Constants.CounterRoute)]
```

Focus an element on navigation

The FocusOnNavigate component sets the UI focus to an element based on a CSS selector after navigating from one page to another.

razor

```razor
<FocusOnNavigate RouteData="routeData" Selector="h1" />
```

When the Router component navigates to a new page, the FocusOnNavigate component sets the focus to the page's top-level header (`<h1>`). This is a common strategy for ensuring that a page navigation is announced when using a screen reader.

Provide custom content when content isn't found

The Router component allows the app to specify custom content if content isn't found for the requested route.

Set custom content for the Router component's NotFound parameter:

```razor
<Router ...>
    ...
    <NotFound>
        ...
    </NotFound>
</Router>
```

Arbitrary items are supported as content of the NotFound parameter, such as other interactive components. To apply a default layout to NotFound content, see ASP.NET Core Blazor layouts.

> **Important**
>
> Blazor Web Apps don't use the **NotFound** parameter (`<NotFound>...</NotFound>` markup), but the parameter is supported† for backward compatibility to avoid a breaking change in the framework. The server-side ASP.NET Core middleware pipeline processes requests on the server. Use server-side techniques to handle bad requests.
>
> †*Supported* in this context means that placing `<NotFound>...</NotFound>` markup doesn't result in an exception, but using the markup isn't effective either.
>
> For more information, including a recommended approach for handling bad requests, see **ASP.NET Core Blazor render modes**.

Route to components from multiple assemblies

This section applies to Blazor Web Apps.

Use the Router component's AdditionalAssemblies parameter and the endpoint convention builder AddAdditionalAssemblies to discover routable components in additional assemblies. The following subsections explain when and how to use each API.

Static routing

To discover routable components from additional assemblies for static server-side rendering (static SSR), even if the router later becomes interactive for interactive rendering, the assemblies must be disclosed to the Blazor framework. Call the AddAdditionalAssemblies method with the additional assemblies chained to MapRazorComponents in the server project's `Program` file.

The following example includes the routable components in the `BlazorSample.Client` project's assembly using the project's `_Imports.razor` file:

```C#
app.MapRazorComponents<App>()
    .AddAdditionalAssemblies(typeof(BlazorSample.Client._Imports).Assembly);
```

> **Note**

The preceding guidance also applies in component class library scenarios. Additional important guidance for class libraries and static SSR is found in ASP.NET Core Razor class libraries (RCLs) with static server-side rendering (static SSR).

Interactive routing

An interactive render mode can be assigned to the `Routes` component (`Routes.razor`) that makes the Blazor router become interactive after static SSR and static routing on the server. For example, `<Routes @rendermode="InteractiveServer" />` assigns interactive server-side rendering (interactive SSR) to the `Routes` component. The `Router` component inherits interactive server-side rendering (interactive SSR) from the `Routes` component. The router becomes interactive after static routing on the server.

Internal navigation for interactive routing doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests. For more information, see Prerender ASP.NET Core Razor components.

If the `Routes` component is defined in the server project, the AdditionalAssemblies parameter of the `Router` component should include the `.Client` project's assembly. This allows the router to work correctly when rendered interactively.

In the following example, the `Routes` component is in the server project, and the `_Imports.razor` file of the `BlazorSample.Client` project indicates the assembly to search for routable components:

razor

```
<Router
    AppAssembly="..."
    AdditionalAssemblies="new[] { typeof(BlazorSample.Client._Imports).Assembly }">
    ...
</Router>
```

Additional assemblies are scanned in addition to the assembly specified to AppAssembly.

> **Note**
>
> The preceding guidance also applies in component class library scenarios.

Alternatively, routable components only exist in the `.Client` project with global Interactive WebAssembly or Auto rendering applied, and the `Routes` component is defined in the `.Client` project, not the server project. In this case, there aren't external assemblies with routable components, so it isn't necessary to specify a value for AdditionalAssemblies.

Route parameters

The router uses route parameters to populate the corresponding component parameters with the same name. Route parameter names are case insensitive. In the following example, the `text` parameter assigns the value of the route segment to the component's `Text` property. When a request is made for `/route-parameter-1/amazing`, the content is rendered as `Blazor is amazing!`.

`RouteParameter1.razor`:

razor

```razor
@page "/route-parameter-1/{text}"

<PageTitle>Route Parameter 1</PageTitle>

<h1>Route Parameter Example 1</h1>

<p>Blazor is @Text!</p>

@code {
    [Parameter]
    public string? Text { get; set; }
}
```

Optional parameters are supported. In the following example, the `text` optional parameter assigns the value of the route segment to the component's `Text` property. If the segment isn't present, the value of `Text` is set to `fantastic`.

`RouteParameter2.razor`:

razor

```razor
@page "/route-parameter-2/{text?}"

<PageTitle>Route Parameter 2</PageTitle>

<h1>Route Parameter Example 2</h1>

<p>Blazor is @Text!</p>

@code {
    [Parameter]
    public string? Text { get; set; }

    protected override void OnParametersSet() => Text = Text ?? "fantastic";
}
```

When the OnInitialized{Async} lifecycle method is used instead of the OnParametersSet{Async} lifecycle method, the default assignment of the `Text` property to `fantastic` doesn't occur if the user navigates within the same component. For example, this situation arises when the user navigates from `/route-parameter-2/amazing` to `/route-parameter-2`. As the component instance persists and accepts new parameters, the `OnInitialized` method isn't invoked again.

Route constraints

A route constraint enforces type matching on a route segment to a component.

In the following example, the route to the `User` component only matches if:

- An `Id` route segment is present in the request URL.
- The `Id` segment is an integer (`int`) type.

`User.razor`:

razor

```razor
@page "/user/{Id:int}"

<PageTitle>User</PageTitle>
```

```razor
<h1>User Example</h1>

<p>User Id: @Id</p>

@code {
    [Parameter]
    public int Id { get; set; }
}
```

The route constraints shown in the following table are available. For the route constraints that match the invariant culture, see the warning below the table for more information.

Expand table

Constraint	Example	Example Matches	Invariant culture matching
`bool`	`{active:bool}`	`true`, `FALSE`	No
`datetime`	`{dob:datetime}`	`2016-12-31`, `2016-12-31 7:32pm`	Yes
`decimal`	`{price:decimal}`	`49.99`, `-1,000.01`	Yes
`double`	`{weight:double}`	`1.234`, `-1,001.01e8`	Yes
`float`	`{weight:float}`	`1.234`, `-1,001.01e8`	Yes
`guid`	`{id:guid}`	`00001111-aaaa-2222-bbbb-3333cccc4444`, `{00001111-aaaa-2222-bbbb-3333cccc4444}`	No
`int`	`{id:int}`	`123456789`, `-123456789`	Yes
`long`	`{ticks:long}`	`123456789`, `-123456789`	Yes
`nonfile`	`{parameter:nonfile}`	Not `BlazorSample.styles.css`, not `favicon.ico`	Yes

> ⚠ **Warning**
>
> Route constraints that verify the URL and are converted to a CLR type (such as `int` or DateTime) always use the invariant culture. These constraints assume that the URL is non-localizable.

Route constraints also work with optional parameters. In the following example, `Id` is required, but `Option` is an optional boolean route parameter.

`User.razor`:

```razor
@page "/user/{id:int}/{option:bool?}"

<p>
    Id: @Id
</p>

<p>
    Option: @Option
</p>

@code {
```

```
    [Parameter]
    public int Id { get; set; }

    [Parameter]
    public bool Option { get; set; }
}
```

Avoid file capture in a route parameter

The following route template inadvertently captures static asset paths in its optional route parameter (`Optional`). For example, the app's stylesheet (`.styles.css`) is captured, which breaks the app's styles:

razor

```
@page "/{optional?}"

...

@code {
    [Parameter]
    public string? Optional { get; set; }
}
```

To restrict a route parameter to capturing non-file paths, use the :nonfile constraint in the route template:

razor

```
@page "/{optional:nonfile?}"
```

Catch-all route parameters

Catch-all route parameters, which capture paths across multiple folder boundaries, are supported in components.

Catch-all route parameters are:

- Named to match the route segment name. Naming isn't case-sensitive.
- A `string` type. The framework doesn't provide automatic casting.
- At the end of the URL.

`CatchAll.razor`:

razor

```
@page "/catch-all/{*pageRoute}"

<PageTitle>Catch All</PageTitle>

<h1>Catch All Parameters Example</h1>

<p>Add some URI segments to the route and request the page again.</p>

<p>
    PageRoute: @PageRoute
</p>

@code {
    [Parameter]
```

```
    public string? PageRoute { get; set; }
}
```

For the URL `/catch-all/this/is/a/test` with a route template of `/catch-all/{*pageRoute}`, the value of `PageRoute` is set to `this/is/a/test`.

Slashes and segments of the captured path are decoded. For a route template of `/catch-all/{*pageRoute}`, the URL `/catch-all/this/is/a%2Ftest%2A` yields `this/is/a/test*`.

URI and navigation state helpers

Use NavigationManager to manage URIs and navigation in C# code. NavigationManager provides the event and methods shown in the following table.

Expand table

Member	Description
Uri	Gets the current absolute URI.
BaseUri	Gets the base URI (with a trailing slash) that can be prepended to relative URI paths to produce an absolute URI. Typically, BaseUri corresponds to the `href` attribute on the document's `<base>` element (location of <head> content).
NavigateTo	Navigates to the specified URI. If `forceLoad` is `false`: • And enhanced navigation is available at the current URL, Blazor's enhanced navigation is activated. • Otherwise, Blazor performs a full-page reload for the requested URL. If `forceLoad` is `true`: • Client-side routing is bypassed. • The browser is forced to load the new page from the server, whether or not the URI is normally handled by the client-side interactive router. For more information, see the Enhanced navigation and form handling section. If `replace` is `true`, the current URI in the browser history is replaced instead of pushing a new URI onto the history stack.
LocationChanged	An event that fires when the navigation location has changed. For more information, see the Location changes section.
ToAbsoluteUri	Converts a relative URI into an absolute URI.
ToBaseRelativePath	Based on the app's base URI, converts an absolute URI into a URI relative to the base URI prefix. For an example, see the Produce a URI relative to the base URI prefix section.
RegisterLocationChangingHandler	Registers a handler to process incoming navigation events. Calling NavigateTo always invokes the handler.
GetUriWithQueryParameter	Returns a URI constructed by updating NavigationManager.Uri with a single parameter added, updated, or removed. For more information, see the Query strings section.

Location changes

For the LocationChanged event, LocationChangedEventArgs provides the following information about navigation events:

- Location: The URL of the new location.
- IsNavigationIntercepted: If `true`, Blazor intercepted the navigation from the browser. If `false`, NavigationManager.NavigateTo caused the navigation to occur.

The following component:

- Navigates to the app's `Counter` component (`Counter.razor`) when the button is selected using NavigateTo.
- Handles the location changed event by subscribing to NavigationManager.LocationChanged.
  - The `HandleLocationChanged` method is unhooked when `Dispose` is called by the framework. Unhooking the method permits garbage collection of the component.
  - The logger implementation logs the following information when the button is selected:

    > BlazorSample.Pages.Navigate: Information: URL of new location: https://localhost:{PORT}/counter

`Navigate.razor`:

razor

```
@page "/navigate"
@implements IDisposable
@inject ILogger<Navigate> Logger
@inject NavigationManager Navigation

<PageTitle>Navigate</PageTitle>

<h1>Navigate Example</h1>

<button class="btn btn-primary" @onclick="NavigateToCounterComponent">
    Navigate to the Counter component
</button>

@code {
    private void NavigateToCounterComponent() => Navigation.NavigateTo("counter");

    protected override void OnInitialized() =>
        Navigation.LocationChanged += HandleLocationChanged;

    private void HandleLocationChanged(object? sender, LocationChangedEventArgs e) =>
        Logger.LogInformation("URL of new location: {Location}", e.Location);

    public void Dispose() => Navigation.LocationChanged -= HandleLocationChanged;
}
```

For more information on component disposal, see ASP.NET Core Razor component lifecycle.

Enhanced navigation and form handling

This section applies to Blazor Web Apps.

Blazor Web Apps are capable of two types of routing for page navigation and form handling requests:

- Normal navigation (cross-document navigation): a full-page reload is triggered for the request URL.
- Enhanced navigation (same-document navigation): Blazor intercepts the request and performs a `fetch` request instead. Blazor then patches the response content into the page's DOM. Blazor's enhanced navigation and form handling avoid the need for a full-page reload and preserves more of the page state, so pages load faster, usually without losing the user's scroll position on the page.

Enhanced navigation is available when:

- The Blazor Web App script (`blazor.web.js`) is used, not the Blazor Server script (`blazor.server.js`) or Blazor WebAssembly script (`blazor.webassembly.js`).
- The feature isn't explicitly disabled.
- The destination URL is within the internal base URI space (the app's base path).

If server-side routing and enhanced navigation are enabled, location changing handlers are only invoked for programmatic navigation initiated from an interactive runtime. In future releases, additional types of navigation, such as following a link, may also invoke location changing handlers.

When an enhanced navigation occurs, LocationChanged event handlers registered with Interactive Server and WebAssembly runtimes are typically invoked. There are cases when location changing handlers might not intercept an enhanced navigation. For example, the user might switch to another page before an interactive runtime becomes available. Therefore, it's important that app logic not rely on invoking a location changing handler, as there's no guarantee of the handler executing.

When calling NavigateTo:

- If `forceLoad` is `false`, which is the default:
  - And enhanced navigation is available at the current URL, Blazor's enhanced navigation is activated.
  - Otherwise, Blazor performs a full-page reload for the requested URL.
- If `forceLoad` is `true`: Blazor performs a full-page reload for the requested URL, whether enhanced navigation is available or not.

You can refresh the current page by calling `NavigationManager.Refresh(bool forceLoad = false)`, which always performs an enhanced navigation, if available. If enhanced navigation isn't available, Blazor performs a full-page reload.

C#

```
Navigation.Refresh();
```

Pass `true` to the `forceLoad` parameter to ensure a full-page reload is always performed, even if enhanced navigation is available:

C#

```
Navigation.Refresh(true);
```

Enhanced navigation is enabled by default, but it can be controlled hierarchically and on a per-link basis using the `data-enhance-nav` HTML attribute.

The following examples disable enhanced navigation:

HTML

```
<a href="redirect" data-enhance-nav="false">
    GET without enhanced navigation
</a>
```

razor

```
<ul data-enhance-nav="false">
    <li>
        <a href="redirect">GET without enhanced navigation</a>
```

```
    </li>
    <li>
        <a href="redirect-2">GET without enhanced navigation</a>
    </li>
</ul>
```

If the destination is a non-Blazor endpoint, enhanced navigation doesn't apply, and the client-side JavaScript retries as a full page load. This ensures no confusion to the framework about external pages that shouldn't be patched into an existing page.

To enable enhanced form handling, add the Enhance parameter to EditForm forms or the `data-enhance` attribute to HTML forms (`<form>`):

razor

```
<EditForm ... Enhance ...>
    ...
</EditForm>
```

HTML

```
<form ... data-enhance ...>
    ...
</form>
```

Enhanced form handling isn't hierarchical and doesn't flow to child forms:

❌ You can't set enhanced navigation on a form's ancestor element to enable enhanced navigation for the form.

HTML

```
<div ... data-enhance ...>
    <form ...>
        <!-- NOT enhanced -->
    </form>
</div>
```

Enhanced form posts only work with Blazor endpoints. Posting an enhanced form to non-Blazor endpoint results in an error.

To disable enhanced navigation:

- For an EditForm, remove the Enhance parameter from the form element (or set it to `false`: `Enhance="false"`).
- For an HTML `<form>`, remove the `data-enhance` attribute from form element (or set it to `false`: `data-enhance="false"`).

Blazor's enhanced navigation and form handing may undo dynamic changes to the DOM if the updated content isn't part of the server rendering. To preserve the content of an element, use the `data-permanent` attribute.

In the following example, the content of the `<div>` element is updated dynamically by a script when the page loads:

HTML

```
<div data-permanent>
    ...
</div>
```

Once Blazor has started on the client, you can use the `enhancedload` event to listen for enhanced page updates. This allows for re-applying changes to the DOM that may have been undone by an enhanced page update.

JavaScript

```
Blazor.addEventListener('enhancedload', () => console.log('Enhanced update!'));
```

To disable enhanced navigation and form handling globally, see ASP.NET Core Blazor startup.

Enhanced navigation with static server-side rendering (static SSR) requires special attention when loading JavaScript. For more information, see ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR).

Produce a URI relative to the base URI prefix

Based on the app's base URI, ToBaseRelativePath converts an absolute URI into a URI relative to the base URI prefix.

Consider the following example:

C#

```
try
{
    baseRelativePath = Navigation.ToBaseRelativePath(inputURI);
}
catch (ArgumentException ex)
{
    ...
}
```

If the base URI of the app is `https://localhost:8000`, the following results are obtained:

- Passing `https://localhost:8000/segment` in `inputURI` results in a `baseRelativePath` of `segment`.
- Passing `https://localhost:8000/segment1/segment2` in `inputURI` results in a `baseRelativePath` of `segment1/segment2`.

If the base URI of the app doesn't match the base URI of `inputURI`, an ArgumentException is thrown.

Passing `https://localhost:8001/segment` in `inputURI` results in the following exception:

> System.ArgumentException: 'The URI 'https://localhost:8001/segment' is not contained by the base URI 'https://localhost:8000/'.'

Navigation history state

The NavigationManager uses the browser's History API to maintain navigation history state associated with each location change made by the app. Maintaining history state is particularly useful in external redirect scenarios, such as when authenticating users with external identity providers. For more information, see the Navigation options section.

Navigation options

Pass NavigationOptions to NavigateTo to control the following behaviors:

- ForceLoad: Bypass client-side routing and force the browser to load the new page from the server, whether or not the URI is handled by the client-side router. The default value is `false`.
- ReplaceHistoryEntry: Replace the current entry in the history stack. If `false`, append the new entry to the history stack. The default value is `false`.
- HistoryEntryState: Gets or sets the state to append to the history entry.

C#

```
Navigation.NavigateTo("/path", new NavigationOptions
{
    HistoryEntryState = "Navigation state"
});
```

For more information on obtaining the state associated with the target history entry while handling location changes, see the Handle/prevent location changes section.

Query strings

Use the [SupplyParameterFromQuery] attribute to specify that a component parameter comes from the query string.

Component parameters supplied from the query string support the following types:

- `bool`, `DateTime`, `decimal`, `double`, `float`, `Guid`, `int`, `long`, `string`.
- Nullable variants of the preceding types.
- Arrays of the preceding types, whether they're nullable or not nullable.

The correct culture-invariant formatting is applied for the given type (CultureInfo.InvariantCulture).

Specify the `[SupplyParameterFromQuery]` attribute's Name property to use a query parameter name different from the component parameter name. In the following example, the C# name of the component parameter is `{COMPONENT PARAMETER NAME}`. A different query parameter name is specified for the `{QUERY PARAMETER NAME}` placeholder:

Unlike component parameter properties (`[Parameter]`), `[SupplyParameterFromQuery]` properties can be marked `private` in addition to `public`.

C#

```
[SupplyParameterFromQuery(Name = "{QUERY PARAMETER NAME}")]
private string? {COMPONENT PARAMETER NAME} { get; set; }
```

In the following example with a URL of `/search?filter=scifi%20stars&page=3&star=LeVar%20Burton&star=Gary%20Oldman`:

- The `Filter` property resolves to `scifi stars`.
- The `Page` property resolves to `3`.
- The `Stars` array is filled from query parameters named `star` (`Name = "star"`) and resolves to `LeVar Burton` and `Gary Oldman`.

> **Note**
>
> The query string parameters in the following routable page component also work in a *non-routable* component without an `@page` directive (for example, `Search.razor` for a shared `Search` component used in

other components).

`Search.razor`:

razor

```razor
@page "/search"

<h1>Search Example</h1>

<p>Filter: @Filter</p>

<p>Page: @Page</p>

@if (Stars is not null)
{
    <p>Stars:</p>

    <ul>
        @foreach (var name in Stars)
        {
            <li>@name</li>
        }
    </ul>
}

@code {
    [SupplyParameterFromQuery]
    private string? Filter { get; set; }

    [SupplyParameterFromQuery]
    private int? Page { get; set; }

    [SupplyParameterFromQuery(Name = "star")]
    private string[]? Stars { get; set; }
}
```

Use GetUriWithQueryParameter to add, change, or remove one or more query parameters on the current URL:

razor

```razor
@inject NavigationManager Navigation

...

Navigation.GetUriWithQueryParameter("{NAME}", {VALUE})
```

For the preceding example:

- The `{NAME}` placeholder specifies the query parameter name. The `{VALUE}` placeholder specifies the value as a supported type. Supported types are listed later in this section.
- A string is returned equal to the current URL with a single parameter:
  - Added if the query parameter name doesn't exist in the current URL.
  - Updated to the value provided if the query parameter exists in the current URL.
  - Removed if the type of the provided value is nullable and the value is `null`.
- The correct culture-invariant formatting is applied for the given type (CultureInfo.InvariantCulture).
- The query parameter name and value are URL-encoded.
- All of the values with the matching query parameter name are replaced if there are multiple instances of the type.

Call GetUriWithQueryParameters to create a URI constructed from Uri with multiple parameters added, updated, or removed. For each value, the framework uses `value?.GetType()` to determine the runtime type for each query parameter and selects the correct culture-invariant formatting. The framework throws an error for unsupported types.

```razor
@inject NavigationManager Navigation

...

Navigation.GetUriWithQueryParameters({PARAMETERS})
```

The `{PARAMETERS}` placeholder is an `IReadOnlyDictionary<string, object>`.

Pass a URI string to GetUriWithQueryParameters to generate a new URI from a provided URI with multiple parameters added, updated, or removed. For each value, the framework uses `value?.GetType()` to determine the runtime type for each query parameter and selects the correct culture-invariant formatting. The framework throws an error for unsupported types. Supported types are listed later in this section.

```razor
@inject NavigationManager Navigation

...

Navigation.GetUriWithQueryParameters("{URI}", {PARAMETERS})
```

- The `{URI}` placeholder is the URI with or without a query string.
- The `{PARAMETERS}` placeholder is an `IReadOnlyDictionary<string, object>`.

Supported types are identical to supported types for route constraints:

- `bool`
- `DateTime`
- `decimal`
- `double`
- `float`
- `Guid`
- `int`
- `long`
- `string`

Supported types include:

- Nullable variants of the preceding types.
- Arrays of the preceding types, whether they're nullable or not nullable.

Replace a query parameter value when the parameter exists

```C#
Navigation.GetUriWithQueryParameter("full name", "Morena Baccarin")
```

Expand table

Current URL	Generated URL
`scheme://host/?full%20name=David%20Krumholtz&age=42`	`scheme://host/?full%20name=Morena%20Baccarin&age=42`
`scheme://host/?fUlL%20nAmE=David%20Krumholtz&AgE=42`	`scheme://host/?full%20name=Morena%20Baccarin&AgE=42`
`scheme://host/?full%20name=Jewel%20Staite&age=42&full%20name=Summer%20Glau`	`scheme://host/?full%20name=Morena%20Baccarin&age=42&full%20name=Morena%20Baccarin`
`scheme://host/?full%20name=&age=42`	`scheme://host/?full%20name=Morena%20Baccarin&age=42`
`scheme://host/?full%20name=`	`scheme://host/?full%20name=Morena%20Baccarin`

Append a query parameter and value when the parameter doesn't exist

C#

```
Navigation.GetUriWithQueryParameter("name", "Morena Baccarin")
```

Expand table

Current URL	Generated URL
`scheme://host/?age=42`	`scheme://host/?age=42&name=Morena%20Baccarin`
`scheme://host/`	`scheme://host/?name=Morena%20Baccarin`
`scheme://host/?`	`scheme://host/?name=Morena%20Baccarin`

Remove a query parameter when the parameter value is `null`

C#

```
Navigation.GetUriWithQueryParameter("full name", (string)null)
```

Expand table

Current URL	Generated URL
`scheme://host/?full%20name=David%20Krumholtz&age=42`	`scheme://host/?age=42`
`scheme://host/?full%20name=Sally%20Smith&age=42&full%20name=Summer%20Glau`	`scheme://host/?age=42`
`scheme://host/?full%20name=Sally%20Smith&age=42&FuLl%20NaMe=Summer%20Glau`	`scheme://host/?age=42`
`scheme://host/?full%20name=&age=42`	`scheme://host/?age=42`
`scheme://host/?full%20name=`	`scheme://host/`

Add, update, and remove query parameters

In the following example:

- `name` is removed, if present.
- `age` is added with a value of `25` (`int`), if not present. If present, `age` is updated to a value of `25`.
- `eye color` is added or updated to a value of `green`.

C#

```
Navigation.GetUriWithQueryParameters(
    new Dictionary<string, object?>
    {
        ["name"] = null,
        ["age"] = (int?)25,
        ["eye color"] = "green"
    })
```

Expand table

Current URL	Generated URL
`scheme://host/?name=David%20Krumholtz&age=42`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/?NaMe=David%20Krumholtz&AgE=42`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/?name=David%20Krumholtz&age=42&keepme=true`	`scheme://host/?age=25&keepme=true&eye%20color=green`
`scheme://host/?age=42&eye%20color=87`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/?`	`scheme://host/?age=25&eye%20color=green`
`scheme://host/`	`scheme://host/?age=25&eye%20color=green`

Support for enumerable values

In the following example:

- `full name` is added or updated to `Morena Baccarin`, a single value.
- `ping` parameters are added or replaced with `35`, `16`, `87` and `240`.

C#

```
Navigation.GetUriWithQueryParameters(
    new Dictionary<string, object?>
    {
        ["full name"] = "Morena Baccarin",
        ["ping"] = new int?[] { 35, 16, null, 87, 240 }
    })
```

Expand table

Current URL	Generated URL
`scheme://host/?full%20name=David%20Krumholtz&ping=8&ping=300`	`scheme://host/?full%20name=Morena%20Baccarin&ping=35&ping=16&ping=87&ping=240`
`scheme://host/?ping=8&full%20name=David%20Krumholtz&ping=300`	`scheme://host/?ping=35&full%20name=Morena%20Baccarin&ping=16&ping=87&ping=240`
`scheme://host/?ping=8&ping=300&ping=50&ping=68&ping=42`	`scheme://host/?ping=35&ping=16&ping=87&ping=240&full%20name=Morena%20Baccarin`

Navigate with an added or modified query string

To navigate with an added or modified query string, pass a generated URL to NavigateTo.

The following example calls:

- GetUriWithQueryParameter to add or replace the `name` query parameter using a value of `Morena Baccarin`.
- Calls NavigateTo to trigger navigation to the new URL.

C#

```
Navigation.NavigateTo(
    Navigation.GetUriWithQueryParameter("name", "Morena Baccarin"));
```

Hashed routing to named elements

Navigate to a named element using the following approaches with a hashed (`#`) reference to the element. Routes to elements within the component and routes to elements in external components use root-relative paths. A leading forward slash (`/`) is optional.

Examples for each of the following approaches demonstrate navigation to an element with an `id` of `targetElement` in the `Counter` component:

- Anchor element (`<a>`) with an `href`:

  razor

  ```
  <a href="/counter#targetElement">
  ```

- NavLink component with an `href`:

  razor

  ```
  <NavLink href="/counter#targetElement">
  ```

- NavigationManager.NavigateTo passing the relative URL:

  C#

  ```
  Navigation.NavigateTo("/counter#targetElement");
  ```

The following example demonstrates hashed routing to named H2 headings within a component and to external components.

In the `Home` (`Home.razor`) and `Counter` (`Counter.razor`) components, place the following markup at the bottoms of the existing component markup to serve as navigation targets. The `<div>` creates artificial vertical space to demonstrate browser scrolling behavior:

razor

```
<div class="border border-info rounded bg-info" style="height:500px"></div>

<h2 id="targetElement">Target H2 heading</h2>
<p>Content!</p>
```

Add the following `HashedRouting` component to the app.

`HashedRouting.razor`:

razor

```razor
@page "/hashed-routing"
@inject NavigationManager Navigation

<PageTitle>Hashed routing</PageTitle>

<h1>Hashed routing to named elements</h1>

<ul>
    <li>
        <a href="/hashed-routing#targetElement">
            Anchor in this component
        </a>
    </li>
    <li>
        <a href="/#targetElement">
            Anchor to the <code>Home</code> component
        </a>
    </li>
    <li>
        <a href="/counter#targetElement">
            Anchor to the <code>Counter</code> component
        </a>
    </li>
    <li>
        <NavLink href="/hashed-routing#targetElement">
            Use a `NavLink` component in this component
        </NavLink>
    </li>
    <li>
        <button @onclick="NavigateToElement">
            Navigate with <code>NavigationManager</code> to the
            <code>Counter</code> component
        </button>
    </li>
</ul>

<div class="border border-info rounded bg-info" style="height:500px"></div>

<h2 id="targetElement">Target H2 heading</h2>
<p>Content!</p>

@code {
    private void NavigateToElement()
    {
        Navigation.NavigateTo("/counter#targetElement");
    }
}
```

User interaction with `<Navigating>` content

If there's a significant delay during navigation, such as while lazy-loading assemblies in a Blazor WebAssembly app or for a slow network connection to a Blazor server-side app, the Router component can indicate to the user that a page transition is occurring.

At the top of the component that specifies the Router component, add an @using directive for the Microsoft.AspNetCore.Components.Routing namespace:

razor

```
@using Microsoft.AspNetCore.Components.Routing
```

Provide content to the Navigating parameter for display during page transition events.

In the router element (`<Router>...</Router>`) content:

razor

```
<Navigating>
    <p>Loading the requested page&hellip;</p>
</Navigating>
```

For an example that uses the Navigating property, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Handle asynchronous navigation events with `OnNavigateAsync`

The Router component supports an OnNavigateAsync feature. The OnNavigateAsync handler is invoked when the user:

- Visits a route for the first time by navigating to it directly in their browser.
- Navigates to a new route using a link or a NavigationManager.NavigateTo invocation.

razor

```
<Router AppAssembly="typeof(App).Assembly"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private async Task OnNavigateAsync(NavigationContext args)
    {
        ...
    }
}
```

For an example that uses OnNavigateAsync, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

When prerendering on the server, OnNavigateAsync is executed *twice*:

- Once when the requested endpoint component is initially rendered statically.
- A second time when the browser renders the endpoint component.

To prevent developer code in OnNavigateAsync from executing twice, the `Routes` component can store the NavigationContext for use in the OnAfterRender{Async} lifecycle method, where `firstRender` can be checked. For more information, see Prerendering with JavaScript interop.

Handle cancellations in `OnNavigateAsync`

The NavigationContext object passed to the OnNavigateAsync callback contains a CancellationToken that's set when a new navigation event occurs. The OnNavigateAsync callback must throw when this cancellation token is set to avoid continuing to run the OnNavigateAsync callback on an outdated navigation.

If a user navigates to an endpoint but then immediately navigates to a new endpoint, the app shouldn't continue running the OnNavigateAsync callback for the first endpoint.

In the following example:

- The cancellation token is passed in the call to `PostAsJsonAsync`, which can cancel the POST if the user navigates away from the `/about` endpoint.
- The cancellation token is set during a product prefetch operation if the user navigates away from the `/store` endpoint.

razor

```
@inject HttpClient Http
@inject ProductCatalog Products

<Router AppAssembly="typeof(App).Assembly"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private async Task OnNavigateAsync(NavigationContext context)
    {
        if (context.Path == "/about")
        {
            var stats = new Stats { Page = "/about" };
            await Http.PostAsJsonAsync("api/visited", stats,
                context.CancellationToken);
        }
        else if (context.Path == "/store")
        {
            var productIds = new[] { 345, 789, 135, 689 };

            foreach (var productId in productIds)
            {
                context.CancellationToken.ThrowIfCancellationRequested();
                Products.Prefetch(productId);
            }
        }
    }
}
```

> **Note**
>
> Not throwing if the cancellation token in NavigationContext is canceled can result in unintended behavior, such as rendering a component from a previous navigation.

Handle/prevent location changes

RegisterLocationChangingHandler registers a handler to process incoming navigation events. The handler's context provided by LocationChangingContext includes the following properties:

- TargetLocation: Gets the target location.
- HistoryEntryState: Gets the state associated with the target history entry.
- IsNavigationIntercepted: Gets whether the navigation was intercepted from a link.
- CancellationToken: Gets a CancellationToken to determine if the navigation was canceled, for example, to determine if the user triggered a different navigation.
- PreventNavigation: Called to prevent the navigation from continuing.

A component can register multiple location changing handlers in the OnAfterRender{Async} lifecycle method. Navigation invokes all of the location changing handlers registered across the entire app (across multiple components), and any internal navigation executes them all in parallel. In addition to NavigateTo handlers are invoked:

- When selecting internal links, which are links that point to URLs under the app's base path.
- When navigating using the forward and back buttons in a browser.

Handlers are only executed for internal navigation within the app. If the user selects a link that navigates to a different site or changes the address bar to a different site manually, location changing handlers aren't executed.

Implement IDisposable and dispose registered handlers to unregister them. For more information, see ASP.NET Core Razor component lifecycle.

> **Important**
>
> Don't attempt to execute DOM cleanup tasks via JavaScript (JS) interop when handling location changes. Use the MutationObserver pattern in JS on the client. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

In the following example, a location changing handler is registered for navigation events.

`NavHandler.razor`:

razor

```razor
@page "/nav-handler"
@implements IDisposable
@inject NavigationManager Navigation

<p>
    <button @onclick="@(() => Navigation.NavigateTo("/"))">
        Home (Allowed)
    </button>
    <button @onclick="@(() => Navigation.NavigateTo("/counter"))">
        Counter (Prevented)
    </button>
</p>

@code {
    private IDisposable? registration;

    protected override void OnAfterRender(bool firstRender)
    {
        if (firstRender)
        {
            registration =
                Navigation.RegisterLocationChangingHandler(OnLocationChanging);
        }
    }

    private ValueTask OnLocationChanging(LocationChangingContext context)
    {
        if (context.TargetLocation == "/counter")
        {
            context.PreventNavigation();
        }

        return ValueTask.CompletedTask;
    }
```

```
    public void Dispose() => registration?.Dispose();
}
```

Since internal navigation can be canceled asynchronously, multiple overlapping calls to registered handlers may occur. For example, multiple handler calls may occur when the user rapidly selects the back button on a page or selects multiple links before a navigation is executed. The following is a summary of the asynchronous navigation logic:

- If any location changing handlers are registered, all navigation is initially reverted, then replayed if the navigation isn't canceled.
- If overlapping navigation requests are made, the latest request always cancels earlier requests, which means the following:
  - The app may treat multiple back and forward button selections as a single selection.
  - If the user selects multiple links before the navigation completes, the last link selected determines the navigation.

For more information on passing NavigationOptions to NavigateTo to control entries and state of the navigation history stack, see the Navigation options section.

For additional example code, see the NavigationManagerComponent in the BasicTestApp (dotnet/aspnetcore reference source).

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

The NavigationLock component intercepts navigation events as long as it is rendered, effectively "locking" any given navigation until a decision is made to either proceed or cancel. Use `NavigationLock` when navigation interception can be scoped to the lifetime of a component.

NavigationLock parameters:

- ConfirmExternalNavigation sets a browser dialog to prompt the user to either confirm or cancel external navigation. The default value is `false`. Displaying the confirmation dialog requires initial user interaction with the page before triggering external navigation with the URL in the browser's address bar. For more information on the interaction requirement, see Window: beforeunload event (MDN documentation).
- OnBeforeInternalNavigation sets a callback for internal navigation events.

In the following `NavLock` component:

- An attempt to follow the link to Microsoft's website must be confirmed by the user before the navigation to `https://www.microsoft.com` succeeds.
- PreventNavigation is called to prevent navigation from occurring if the user declines to confirm the navigation via a JavaScript (JS) interop call that spawns the JS confirm dialog.

`NavLock.razor`:

razor

```
@page "/nav-lock"
@inject IJSRuntime JSRuntime
@inject NavigationManager Navigation
```

```razor
<NavigationLock ConfirmExternalNavigation="true"
    OnBeforeInternalNavigation="OnBeforeInternalNavigation" />

<p>
    <button @onclick="Navigate">Navigate</button>
</p>

<p>
    <a href="https://www.microsoft.com">Microsoft homepage</a>
</p>

@code {
    private void Navigate()
    {
        Navigation.NavigateTo("/");
    }

    private async Task OnBeforeInternalNavigation(LocationChangingContext context)
    {
        var isConfirmed = await JSRuntime.InvokeAsync<bool>("confirm",
            "Are you sure you want to navigate to the root page?");

        if (!isConfirmed)
        {
            context.PreventNavigation();
        }
    }
}
```

For additional example code, see the ConfigurableNavigationLock component in the BasicTestApp (dotnet/aspnetcore reference source).

`NavLink` component

Use a NavLink component in place of HTML hyperlink elements (`<a>`) when creating navigation links. A NavLink component behaves like an `<a>` element, except it toggles an `active` CSS class based on whether its `href` matches the current URL. The `active` class helps a user understand which page is the active page among the navigation links displayed. Optionally, assign a CSS class name to NavLink.ActiveClass to apply a custom CSS class to the rendered link when the current route matches the `href`.

There are two NavLinkMatch options that you can assign to the `Match` attribute of the `<NavLink>` element:

- NavLinkMatch.All: The NavLink is active when it matches the entire current URL.
- NavLinkMatch.Prefix (*default*): The NavLink is active when it matches any prefix of the current URL.

In the preceding example, the Home NavLink `href=""` matches the home URL and only receives the `active` CSS class at the app's default base path (`/`). The second NavLink receives the `active` class when the user visits any URL with a `component` prefix (for example, `/component` and `/component/another-segment`).

Additional NavLink component attributes are passed through to the rendered anchor tag. In the following example, the NavLink component includes the `target` attribute:

razor

```razor
<NavLink href="example-page" target="_blank">Example page</NavLink>
```

The following HTML markup is rendered:

HTML

```
<a href="example-page" target="_blank">Example page</a>
```

> ⚠ **Warning**
>
> Due to the way that Blazor renders child content, rendering `NavLink` components inside a `for` loop requires a local index variable if the incrementing loop variable is used in the `NavLink` (child) component's content:
>
> razor
>
> ```
> @for (int c = 1; c < 4; c++)
> {
>     var ct = c;
>     <li ...>
>         <NavLink ...>
>             <span ...> Product #@ct
>         </NavLink>
>     </li>
> }
> ```
>
> Using an index variable in this scenario is a requirement for **any** child component that uses a loop variable in its child content, not just the `NavLink` component.
>
> Alternatively, use a `foreach` loop with Enumerable.Range:
>
> razor
>
> ```
> @foreach (var c in Enumerable.Range(1, 3))
> {
>     <li ...>
>         <NavLink ...>
>             <span ...> Product #@c
>         </NavLink>
>     </li>
> }
> ```

NavLink component entries can be dynamically created from the app's components via reflection. The following example demonstrates the general approach for further customization.

For the following demonstration, a consistent, standard naming convention is used for the app's components:

- Routable component file names use Pascal case†, for example `Pages/ProductDetail.razor`.
- Routable component file paths match their URLs in kebab case‡ with hyphens appearing between words in a component's route template. For example, a `ProductDetail` component with a route template of `/product-detail` (`@page "/product-detail"`) is requested in a browser at the relative URL `/product-detail`.

†Pascal case (upper camel case) is a naming convention without spaces and punctuation and with the first letter of each word capitalized, including the first word.
‡Kebab case is a naming convention without spaces and punctuation that uses lowercase letters and dashes between words.

In the Razor markup of the `NavMenu` component (`NavMenu.razor`) under the default `Home` page, NavLink components are added from a collection:

diff

```
<div class="nav-scrollable"
    onclick="document.querySelector('.navbar-toggler').click()">
    <nav class="flex-column">
        <div class="nav-item px-3">
            <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
                <span class="bi bi-house-door-fill-nav-menu"
                    aria-hidden="true"></span> Home
            </NavLink>
        </div>

+       @foreach (var name in GetRoutableComponents())
+       {
+           <div class="nav-item px-3">
+               <NavLink class="nav-link"
+                       href="@Regex.Replace(name, @"(\B[A-Z]|\d+)", "-$1").ToLower()">
+                   @Regex.Replace(name, @"(\B[A-Z]|\d+)", " $1")
+               </NavLink>
+           </div>
+       }

    </nav>
</div>
```

The `GetRoutableComponents` method in the `@code` block:

C#

```
public IEnumerable<string> GetRoutableComponents() =>
    Assembly.GetExecutingAssembly()
        .ExportedTypes
        .Where(t => t.IsSubclassOf(typeof(ComponentBase)))
        .Where(c => c.GetCustomAttributes(inherit: true)
                     .OfType<RouteAttribute>()
                     .Any())
        .Where(c => c.Name != "Home" && c.Name != "Error")
        .OrderBy(o => o.Name)
        .Select(c => c.Name);
```

The preceding example doesn't include the following pages in the rendered list of components:

- `Home` page: The page is listed separately from the automatically generated links because it should appear at the top of the list and set the `Match` parameter.
- `Error` page: The error page is only navigated to by the framework and shouldn't be listed.

For an example of the preceding code in a sample app that you can run locally, obtain the **Blazor Web App** or **Blazor WebAssembly** sample app.

ASP.NET Core endpoint routing integration

This section applies to Blazor Web Apps operating over a circuit.

A Blazor Web App is integrated into ASP.NET Core Endpoint Routing. An ASP.NET Core app is configured to accept incoming connections for interactive components with MapRazorComponents in the `Program` file. The default root component (first component loaded) is the `App` component (`App.razor`):

C#

```
app.MapRazorComponents<App>();
```

ASP.NET Core Blazor configuration

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure Blazor apps, including app settings, authentication, and logging configuration.

This guidance applies to client-side project configuration in a Blazor Web App or a standalone Blazor WebAssembly app.

Default behavior in Blazor Web Apps:

- For server-side configuration:
  - See Configuration in ASP.NET Core for guidance.
  - Only configuration in the project's root app settings files is loaded.
  - The remainder of this article only applies to client-side configuration in the `.Client` project.
- For client-side configuration (`.Client` project), configuration is loaded from the following app settings files:
  - `wwwroot/appsettings.json`.
  - `wwwroot/appsettings.{ENVIRONMENT}.json`, where the `{ENVIRONMENT}` placeholder is the app's runtime environment.

In standalone Blazor WebAssembly apps, configuration is loaded from the following app settings files:

- `wwwroot/appsettings.json`.
- `wwwroot/appsettings.{ENVIRONMENT}.json`, where the `{ENVIRONMENT}` placeholder is the app's runtime environment.

> **Note**
>
> Logging configuration placed into an app settings file in `wwwroot` isn't loaded by default. For more information, see the Logging configuration section later in this article.

In some scenarios, such as with Azure services, it's important to use an environment file name segment that exactly matches the environment name. For example, use the file name `appsettings.Staging.json` with a capital "S" for the `Staging` environment. For recommended conventions, see the opening remarks of ASP.NET Core Blazor environments.

Other configuration providers registered by the app can also provide configuration, but not all providers or provider features are appropriate:

- Azure Key Vault configuration provider: The provider isn't supported for managed identity and application ID (client ID) with client secret scenarios. Application ID with a client secret isn't recommended for any ASP.NET Core app, especially client-side apps because the client secret can't be secured client-side to access the Azure Key Vault service.
- Azure App configuration provider: The provider isn't appropriate for a client-side app because the app doesn't run on a server in Azure.

For more information on configuration providers, see Configuration in ASP.NET Core.

> ⚠ **Warning**
>
> Configuration and settings files in the web root (`wwwroot` folder) are visible to users on the client, and users can tamper with the data. **Don't store app secrets, credentials, or any other sensitive data in any web root file.**

App settings configuration

Configuration in app settings files are loaded by default. In the following example, a UI configuration value is stored in an app settings file and loaded by the Blazor framework automatically. The value is read by a component.

`wwwroot/appsettings.json`:

JSON

```
{
    "h1FontSize": "50px"
}
```

Inject an IConfiguration instance into a component to access the configuration data.

`ConfigExample.razor`:

```razor
@page "/config-example"
@inject IConfiguration Configuration

<PageTitle>Configuration</PageTitle>

<h1 style="font-size:@Configuration["h1FontSize"]">
    Configuration example (50px)
</h1>
```

Client security restrictions prevent direct access to files via user code, including settings files for app configuration. To read configuration files in addition to `appsettings.json`/`appsettings.{ENVIRONMENT}.json` from the `wwwroot` folder into configuration, use an HttpClient.

> **⚠ Warning**
>
> Configuration and settings files in the web root (`wwwroot` folder) are visible to users on the client, and users can tamper with the data. **Don't store app secrets, credentials, or any other sensitive data in any web root file.**

The following example reads a configuration file (`cars.json`) into the app's configuration.

`wwwroot/cars.json`:

```JSON
{
    "size": "tiny"
}
```

Add the namespace for Microsoft.Extensions.Configuration to the `Program` file:

```C#
using Microsoft.Extensions.Configuration;
```

Modify the existing HttpClient service registration to use the client to read the file:

```C#
```

```
var http = new HttpClient()
{
    BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
};

builder.Services.AddScoped(sp => http);

using var response = await http.GetAsync("cars.json");
using var stream = await response.Content.ReadAsStreamAsync();

builder.Configuration.AddJsonStream(stream);
```

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the app and is typically derived from the `<base>` tag's `href` value in the host page.

Memory Configuration Source

The following example uses a MemoryConfigurationSource in the `Program` file to supply additional configuration.

Add the namespace for Microsoft.Extensions.Configuration.Memory to the `Program` file:

C#

```
using Microsoft.Extensions.Configuration.Memory;
```

In the `Program` file:

C#

```
var vehicleData = new Dictionary<string, string?>()
{
    { "color", "blue" },
    { "type", "car" },
    { "wheels:count", "3" },
    { "wheels:brand", "Blazin" },
    { "wheels:brand:type", "rally" },
    { "wheels:year", "2008" },
};

var memoryConfig = new MemoryConfigurationSource { InitialData = vehicleData
};

builder.Configuration.Add(memoryConfig);
```

Inject an IConfiguration instance into a component to access the configuration data.

`MemoryConfig.razor`:

razor

```razor
@page "/memory-config"
@inject IConfiguration Configuration

<PageTitle>Memory Configuration</PageTitle>

<h1>Memory Configuration Example</h1>

<h2>General specifications</h2>

<ul>
    <li>Color: @Configuration["color"]</li>
    <li>Type: @Configuration["type"]</li>
</ul>

<h2>Wheels</h2>

<ul>
    <li>Count: @Configuration["wheels:count"]</li>
    <li>Brand: @Configuration["wheels:brand"]</li>
    <li>Type: @Configuration["wheels:brand:type"]</li>
    <li>Year: @Configuration["wheels:year"]</li>
</ul>
```

Obtain a section of the configuration in C# code with IConfiguration.GetSection. The following example obtains the `wheels` section for the configuration in the preceding example:

razor

```razor
@code {
    protected override void OnInitialized()
    {
        var wheelsSection = Configuration.GetSection("wheels");

        ...
    }
}
```

Authentication configuration

Provide ***public*** authentication configuration in an app settings file.

`wwwroot/appsettings.json`:

```
JSON

{
  "Local": {
    "Authority": "{AUTHORITY}",
    "ClientId": "{CLIENT ID}"
  }
}
```

Load the configuration for an Identity provider with ConfigurationBinder.Bind in the `Program` file. The following example loads configuration for an OIDC provider:

```
C#

builder.Services.AddOidcAuthentication(options =>
    builder.Configuration.Bind("Local", options.ProviderOptions));
```

> ⚠ **Warning**
>
> Configuration and settings files in the web root (`wwwroot` folder) are visible to users on the client, and users can tamper with the data. **Don't store app secrets, credentials, or any other sensitive data in any web root file.**

Logging configuration

This section applies to apps that configure logging via an app settings file in the `wwwroot` folder.

Add the Microsoft.Extensions.Logging.Configuration package to the app.

> ⓘ **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the app settings file, provide logging configuration. The logging configuration is loaded in the `Program` file.

`wwwroot/appsettings.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

In the `Program` file:

C#

```
builder.Logging.AddConfiguration(
    builder.Configuration.GetSection("Logging"));
```

Host builder configuration

Read host builder configuration from WebAssemblyHostBuilder.Configuration in the `Program` file:

C#

```
var hostname = builder.Configuration["HostName"];
```

Cached configuration

Configuration files are cached for offline use. With Progressive Web Applications (PWAs), you can only update configuration files when creating a new deployment. Editing configuration files between deployments has no effect because:

- Users have cached versions of the files that they continue to use.
- The PWA's `service-worker.js` and `service-worker-assets.js` files must be rebuilt on compilation, which signal to the app on the user's next online visit that the app has been redeployed.

For more information on how background updates are handled by PWAs, see ASP.NET Core Blazor Progressive Web Application (PWA).

Options configuration

Options configuration requires adding a package reference for the Microsoft.Extensions.Options.ConfigurationExtensions NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Example:

```
C#

builder.Services.Configure<MyOptions>(
    builder.Configuration.GetSection("MyOptions"));
```

Not all of the ASP.NET Core Options features are supported in Razor components. For example, IOptionsSnapshot<TOptions> and IOptionsMonitor<TOptions> configuration is supported, but recomputing option values for these interfaces isn't supported outside of reloading the app by either requesting the app in a new browser tab or selecting the browser's reload button. Merely calling StateHasChanged doesn't update snapshot or monitored option values when the underlying configuration changes.

ASP.NET Core Blazor dependency injection

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Rainer Stropek and Mike Rousos

This article explains how Blazor apps can inject services into components.

Dependency injection (DI) is a technique for accessing services configured in a central location:

- Framework-registered services can be injected directly into Razor components.
- Blazor apps define and register custom services and make them available throughout the app via DI.

> **Note**
>
> We recommend reading Dependency injection in ASP.NET Core before reading this topic.

Default services

The services shown in the following table are commonly used in Blazor apps.

Expand table

Service	Lifetime	Description
HttpClient	Scoped	Provides methods for sending HTTP requests and receiving HTTP responses from a resource identified by a URI. Client-side, an instance of HttpClient is registered by the app in the `Program` file and uses the browser for handling the HTTP traffic in the background.

Service	Lifetime	Description
		Server-side, an HttpClient isn't configured as a service by default. In server-side code, provide an HttpClient. For more information, see Call a web API from an ASP.NET Core Blazor app. An HttpClient is registered as a scoped service, not singleton. For more information, see the Service lifetime section.
IJSRuntime	**Client-side**: Singleton **Server-side**: Scoped The Blazor framework registers IJSRuntime in the app's service container.	Represents an instance of a JavaScript runtime where JavaScript calls are dispatched. For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor. When seeking to inject the service into a singleton service on the server, take either of the following approaches: • Change the service registration to scoped to match IJSRuntime's registration, which is appropriate if the service deals with user-specific state. • Pass the IJSRuntime into the singleton service's implementation as an argument of its method calls instead of injecting it into the singleton.
NavigationManager	**Client-side**: Singleton **Server-side**: Scoped The Blazor framework registers NavigationManager in the app's service container.	Contains helpers for working with URIs and navigation state. For more information, see URI and navigation state helpers.

Additional services registered by the Blazor framework are described in the documentation where they're used to describe Blazor features, such as configuration and logging.

A custom service provider doesn't automatically provide the default services listed in the table. If you use a custom service provider and require any of the services shown in the table, add the required services to the new service provider.

Add client-side services

Configure services for the app's service collection in the `Program` file. In the following example, the `ExampleDependency` implementation is registered for `IExampleDependency`:

C#

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
...
builder.Services.AddSingleton<IExampleDependency, ExampleDependency>();
...

await builder.Build().RunAsync();
```

After the host is built, services are available from the root DI scope before any components are rendered. This can be useful for running initialization logic before rendering content:

C#

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
...
builder.Services.AddSingleton<WeatherService>();
...

var host = builder.Build();

var weatherService = host.Services.GetRequiredService<WeatherService>();
await weatherService.InitializeWeatherAsync();

await host.RunAsync();
```

The host provides a central configuration instance for the app. Building on the preceding example, the weather service's URL is passed from a default configuration source (for example, `appsettings.json`) to `InitializeWeatherAsync`:

C#

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
...
builder.Services.AddSingleton<WeatherService>();
...

var host = builder.Build();

var weatherService = host.Services.GetRequiredService<WeatherService>();
await weatherService.InitializeWeatherAsync(
    host.Configuration["WeatherServiceUrl"]);
```

```
await host.RunAsync();
```

Add server-side services

After creating a new app, examine part of the `Program` file:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

The `builder` variable represents a WebApplicationBuilder with an IServiceCollection, which is a list of service descriptor objects. Services are added by providing service descriptors to the service collection. The following example demonstrates the concept with the `IDataAccess` interface and its concrete implementation `DataAccess`:

C#

```
builder.Services.AddSingleton<IDataAccess, DataAccess>();
```

Register common services

If one or more common services are required client- and server-side, you can place the common service registrations in a method client-side and call the method to register the services in both projects.

First, factor common service registrations into a separate method. For example, create a `ConfigureCommonServices` method client-side:

C#

```
public static void ConfigureCommonServices(IServiceCollection services)
{
    services.Add...;
}
```

For the client-side `Program` file, call `ConfigureCommonServices` to register the common services:

```
```

```
C#

var builder = WebAssemblyHostBuilder.CreateDefault(args);

...

ConfigureCommonServices(builder.Services);
```

In the server-side `Program` file, call `ConfigureCommonServices` to register the common services:

```
C#

var builder = WebApplication.CreateBuilder(args);

...

Client.Program.ConfigureCommonServices(builder.Services);
```

For an example of this approach, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Client-side services that fail during prerendering

This section only applies to WebAssembly components in Blazor Web Apps.

Blazor Web Apps normally prerender client-side WebAssembly components. If an app is run with a required service only registered in the `.Client` project, executing the app results in a runtime error similar to the following when a component attempts to use the required service during prerendering:

> InvalidOperationException: Cannot provide a value for {PROPERTY} on type '{ASSEMBLY}}.Client.Pages.{COMPONENT NAME}'. There is no registered service of type '{SERVICE}'.

Use ***either*** of the following approaches to resolve this problem:

- Register the service in the main project to make it available during component prerendering.
- If prerendering isn't required for the component, disable prerendering by following the guidance in ASP.NET Core Blazor render modes. If you adopt this approach, you don't need to register the service in the main project.

For more information, see Client-side services fail to resolve during prerendering.

Service lifetime

Services can be configured with the lifetimes shown in the following table.

Expand table

Lifetime	Description
Scoped	Client-side doesn't currently have a concept of DI scopes. `Scoped`-registered services behave like `Singleton` services. Server-side development supports the `Scoped` lifetime across HTTP requests but not across SignalR connection/circuit messages among components that are loaded on the client. The Razor Pages or MVC portion of the app treats scoped services normally and recreates the services on *each HTTP request* when navigating among pages or views or from a page or view to a component. Scoped services aren't reconstructed when navigating among components on the client, where the communication to the server takes place over the SignalR connection of the user's circuit, not via HTTP requests. In the following component scenarios on the client, scoped services are reconstructed because a new circuit is created for the user: • The user closes the browser's window. The user opens a new window and navigates back to the app. • The user closes a tab of the app in a browser window. The user opens a new tab and navigates back to the app. • The user selects the browser's reload/refresh button. For more information on preserving user state in server-side apps, see ASP.NET Core Blazor state management.
Singleton	DI creates a *single instance* of the service. All components requiring a `Singleton` service receive the same instance of the service.
Transient	Whenever a component obtains an instance of a `Transient` service from the service container, it receives a *new instance* of the service.

The DI system is based on the DI system in ASP.NET Core. For more information, see Dependency injection in ASP.NET Core.

Request a service in a component

After services are added to the service collection, inject one or more services into components with the @inject Razor directive, which has two parameters:

- Type: The type of the service to inject.

- Property: The name of the property receiving the injected app service. The property doesn't require manual creation. The compiler creates the property.

For more information, see Dependency injection into views in ASP.NET Core.

Use multiple @inject statements to inject different services.

The following example demonstrates shows how to use the @inject directive. The service implementing `Services.NavigationManager` is injected into the component's property `Navigation`. Note how the code is only using the `NavigationManager` abstraction.

`PropertyInjection.razor`:

razor

```razor
@page "/property-injection"
@inject NavigationManager Navigation

<button @onclick="@(() => Navigation.NavigateTo("/counter"))">
    Take me to the Counter component
</button>
```

Internally, the generated property (`Navigation`) uses the [Inject] attribute. Typically, this attribute isn't used directly. If a base class is required for components and injected properties are also required for the base class, manually add the [Inject] attribute:

C#

```
using Microsoft.AspNetCore.Components;

public class ComponentBase : IComponent
{
    [Inject]
    protected NavigationManager Navigation { get; set; } = default!;

    ...
}
```

> **Note**
>
> Since injected services are expected to be available, the default literal with the null-forgiving operator (`default!`) is assigned in .NET 6 or later. For more information, see Nullable reference types (NRTs) and .NET compiler null-state static analysis.

In components derived from a base class, the @inject directive isn't required. The InjectAttribute of the base class is sufficient. The component only requires the @inherits directive. In the following example, any injected services of `CustomComponentBase` are available to the `Demo` component:

razor

```
@page "/demo"
@inherits CustomComponentBase
```

Use DI in services

Complex services might require additional services. In the following example, `DataAccess` requires the HttpClient default service. @inject (or the [Inject] attribute) isn't available for use in services. *Constructor injection* must be used instead. Required services are added by adding parameters to the service's constructor. When DI creates the service, it recognizes the services it requires in the constructor and provides them accordingly. In the following example, the constructor receives an HttpClient via DI. HttpClient is a default service.

C#

```
using System.Net.Http;

public class DataAccess : IDataAccess
{
    public DataAccess(HttpClient http)
    {
        ...
    }

    ...
}
```

Constructor injection is supported with primary constructors in C# 12 (.NET 8) or later:

C#

```
using System.Net.Http;

public class DataAccess(HttpClient http) : IDataAccess
{
    ...
}
```

Prerequisites for constructor injection:

- One constructor must exist whose arguments can all be fulfilled by DI. Additional parameters not covered by DI are allowed if they specify default values.
- The applicable constructor must be `public`.
- One applicable constructor must exist. In case of an ambiguity, DI throws an exception.

Inject keyed services into components

Blazor supports injecting keyed services using the `[Inject]` attribute. Keys allow for scoping of registration and consumption of services when using dependency injection. Use the InjectAttribute.Key property to specify the key for the service to inject:

C#

```
[Inject(Key = "my-service")]
public IMyService MyService { get; set; }
```

Utility base component classes to manage a DI scope

In non-Blazor ASP.NET Core apps, scoped and transient services are typically scoped to the current request. After the request completes, scoped and transient services are disposed by the DI system.

In interactive server-side Blazor apps, the DI scope lasts for the duration of the circuit (the SignalR connection between the client and server), which can result in scoped and disposable transient services living much longer than the lifetime of a single component. Therefore, don't directly inject a scoped service into a component if you intend the service lifetime to match the lifetime of the component. Transient services injected into a component that don't implement IDisposable are garbage collected when the component is disposed. However, injected transient services *that implement IDisposable* are maintained by the DI container for the lifetime of the circuit, which prevents service garbage collection when the component is disposed and results in a memory leak. An alternative approach for scoped services based on the OwningComponentBase type is described later in this section, and disposable transient services shouldn't be used at all. For more information, see Design for solving transient disposables on Blazor Server (dotnet/aspnetcore #26676).

Even in client-side Blazor apps that don't operate over a circuit, services registered with a scoped lifetime are treated as singletons, so they live longer than scoped services in typical ASP.NET Core apps. Client-side disposable transient services also live longer than the components where they're injected because the DI container, which holds references to disposable services, persists for the lifetime of the app, preventing garbage collection on the services. Although long-lived disposable transient services are of greater concern on the server, they should be avoided as client service registrations as well. Use of the OwningComponentBase type is also recommended for client-side scoped services to control service lifetime, and disposable transient services shouldn't be used at all.

An approach that limits a service lifetime is use of the OwningComponentBase type. OwningComponentBase is an abstract type derived from ComponentBase that creates a DI scope corresponding to the *lifetime of the component*. Using this scope, a component can inject services with a scoped lifetime and have them live as long as the component. When the component is destroyed, services from the component's scoped service provider are disposed as well. This can be useful for services reused within a component but not shared across components.

Two versions of OwningComponentBase type are available and described in the next two sections:

- OwningComponentBase
- OwningComponentBase<TService>

`OwningComponentBase`

OwningComponentBase is an abstract, disposable child of the ComponentBase type with a protected ScopedServices property of type IServiceProvider. The provider can be used to resolve services that are scoped to the lifetime of the component.

DI services injected into the component using @inject or the [Inject] attribute aren't created in the component's scope. To use the component's scope, services must be resolved using ScopedServices with either GetRequiredService or GetService. Any services resolved using the ScopedServices provider have their dependencies provided in the component's scope.

The following example demonstrates the difference between injecting a scoped service directly and resolving a service using ScopedServices on the server. The following interface and implementation for a time travel class include a `DT` property to hold a DateTime value. The implementation calls DateTime.Now to set `DT` when the `TimeTravel` class is instantiated.

`ITimeTravel.cs`:

```
C#

public interface ITimeTravel
{
    public DateTime DT { get; set; }
}
```

`TimeTravel.cs`:

```
C#

public class TimeTravel : ITimeTravel
{
    public DateTime DT { get; set; } = DateTime.Now;
}
```

The service is registered as scoped in the server-side `Program` file. Server-side, scoped services have a lifetime equal to the duration of the circuit.

In the `Program` file:

```
C#

builder.Services.AddScoped<ITimeTravel, TimeTravel>();
```

In the following `TimeTravel` component:

- The time travel service is directly injected with `@inject` as `TimeTravel1`.
- The service is also resolved separately with ScopedServices and GetRequiredService as `TimeTravel2`.

`TimeTravel.razor`:

```
razor

@page "/time-travel"
@inject ITimeTravel TimeTravel1
@inherits OwningComponentBase

<h1><code>OwningComponentBase</code> Example</h1>

<ul>
    <li>TimeTravel1.DT: @TimeTravel1?.DT</li>
    <li>TimeTravel2.DT: @TimeTravel2?.DT</li>
</ul>
```

```
@code {
    private ITimeTravel TimeTravel2 { get; set; } = default!;

    protected override void OnInitialized()
    {
        TimeTravel2 = ScopedServices.GetRequiredService<ITimeTravel>();
    }
}
```

Initially navigating to the `TimeTravel` component, the time travel service is instantiated twice when the component loads, and `TimeTravel1` and `TimeTravel2` have the same initial value:

> TimeTravel1.DT: 8/31/2022 2:54:45 PM
> TimeTravel2.DT: 8/31/2022 2:54:45 PM

When navigating away from the `TimeTravel` component to another component and back to the `TimeTravel` component:

- `TimeTravel1` is provided the same service instance that was created when the component first loaded, so the value of `DT` remains the same.
- `TimeTravel2` obtains a new `ITimeTravel` service instance in `TimeTravel2` with a new DT value.

> TimeTravel1.DT: 8/31/2022 2:54:45 PM
> TimeTravel2.DT: 8/31/2022 2:54:48 PM

`TimeTravel1` is tied to the user's circuit, which remains intact and isn't disposed until the underlying circuit is deconstructed. For example, the service is disposed if the circuit is disconnected for the disconnected circuit retention period.

In spite of the scoped service registration in the `Program` file and the longevity of the user's circuit, `TimeTravel2` receives a new `ITimeTravel` service instance each time the component is initialized.

`OwningComponentBase<TService>`

OwningComponentBase<TService> derives from OwningComponentBase and adds a Service property that returns an instance of `T` from the scoped DI provider. This type is a convenient way to access scoped services without using an instance of IServiceProvider when there's one primary service the app requires from the DI

container using the component's scope. The ScopedServices property is available, so the app can get services of other types, if necessary.

razor

```
@page "/users"
@attribute [Authorize]
@inherits OwningComponentBase<AppDbContext>

<h1>Users (@Service.Users.Count())</h1>

<ul>
    @foreach (var user in Service.Users)
    {
        <li>@user.UserName</li>
    }
</ul>
```

Detect client-side transient disposables

Custom code can be added to a client-side Blazor app to detect disposable transient services in an app that should use OwningComponentBase. This approach is useful if you're concerned that code added to the app in the future consumes one or more transient disposable services, including services added by libraries. Demonstration code is available in the Blazor samples GitHub repository (how to download).

Inspect the following in .NET 6 or later versions of the `BlazorSample_WebAssembly` sample:

- `DetectIncorrectUsagesOfTransientDisposables.cs`
- `Services/TransientDisposableService.cs`
- In `Program.cs`:
  - The app's `Services` namespace is provided at the top of the file (`using BlazorSample.Services;`).
  - `DetectIncorrectUsageOfTransients` is called immediately after the `builder` is assigned from WebAssemblyHostBuilder.CreateDefault.
  - The `TransientDisposableService` is registered (`builder.Services.AddTransient<TransientDisposableService>();`).
  - `EnableTransientDisposableDetection` is called on the built host in the processing pipeline of the app (`host.EnableTransientDisposableDetection();`).
- The app registers the `TransientDisposableService` service without throwing an exception. However, attempting to resolve the service in `TransientService.razor`

throws an InvalidOperationException when the framework attempts to construct an instance of `TransientDisposableService`.

Detect server-side transient disposables

Custom code can be added to a server-side Blazor app to detect server-side disposable transient services in an app that should use OwningComponentBase. This approach is useful if you're concerned that code added to the app in the future consumes one or more transient disposable services, including services added by libraries. Demonstration code is available in the Blazor samples GitHub repository (how to download).

Inspect the following in .NET 8 or later versions of the `BlazorSample_BlazorWebApp` sample:

- `DetectIncorrectUsagesOfTransientDisposables.cs`
- `Services/TransitiveTransientDisposableDependency.cs`:
- In `Program.cs`:
  - The app's `Services` namespace is provided at the top of the file (`using BlazorSample.Services;`).
  - `DetectIncorrectUsageOfTransients` is called on the host builder (`builder.DetectIncorrectUsageOfTransients();`).
  - The `TransientDependency` service is registered (`builder.Services.AddTransient<TransientDependency>();`).
  - The `TransitiveTransientDisposableDependency` is registered for `ITransitiveTransientDisposableDependency` (`builder.Services.AddTransient<ITransitiveTransientDisposableDependency, TransitiveTransientDisposableDependency>();`).
- The app registers the `TransientDependency` service without throwing an exception. However, attempting to resolve the service in `TransientService.razor` throws an InvalidOperationException when the framework attempts to construct an instance of `TransientDependency`.

Transient service registrations for `IHttpClientFactory`/`HttpClient` handlers

Transient service registrations for IHttpClientFactory/HttpClient handlers are recommended. If the app contains IHttpClientFactory/HttpClient handlers and uses the IRemoteAuthenticationBuilder<TRemoteAuthenticationState,TAccount> to add support

for authentication, the following transient disposables for client-side authentication are also discovered, which is expected and can be ignored:

- BaseAddressAuthorizationMessageHandler
- AuthorizationMessageHandler

Other instances of IHttpClientFactory/HttpClient are also discovered. These instances can also be ignored.

The Blazor sample apps in the Blazor samples GitHub repository (how to download) demonstrate the code to detect transient disposables. However, the code is deactivated because the sample apps include IHttpClientFactory/HttpClient handlers.

To activate the demonstration code and witness its operation:

- Uncomment the transient disposable lines in `Program.cs`.
- Remove the conditional check in `NavLink.razor` that prevents the `TransientService` component from displaying in the app's navigation sidebar:

  diff

  ```
  - else if (name != "TransientService")
  + else
  ```

- Run the sample app and navigate to the `TransientService` component at `/transient-service`.

Use of an Entity Framework Core (EF Core) DbContext from DI

For more information, see ASP.NET Core Blazor with Entity Framework Core (EF Core).

Access server-side Blazor services from a different DI scope

Circuit activity handlers provide an approach for accessing scoped Blazor services from other non-Blazor dependency injection (DI) scopes, such as scopes created using IHttpClientFactory.

Prior to the release of ASP.NET Core in .NET 8, accessing circuit-scoped services from other dependency injection scopes required using a custom base component type. With

circuit activity handlers, a custom base component type isn't required, as the following example demonstrates:

C#

```
public class CircuitServicesAccessor
{
    static readonly AsyncLocal<IServiceProvider> blazorServices = new();

    public IServiceProvider? Services
    {
        get => blazorServices.Value;
        set => blazorServices.Value = value;
    }
}

public class ServicesAccessorCircuitHandler(
    IServiceProvider services, CircuitServicesAccessor servicesAccessor)
    : CircuitHandler
{
    public override Func<CircuitInboundActivityContext, Task>
CreateInboundActivityHandler(
        Func<CircuitInboundActivityContext, Task> next) =>
            async context =>
            {
                servicesAccessor.Services = services;
                await next(context);
                servicesAccessor.Services = null;
            };
}

public static class CircuitServicesServiceCollectionExtensions
{
    public static IServiceCollection AddCircuitServicesAccessor(
        this IServiceCollection services)
    {
        services.AddScoped<CircuitServicesAccessor>();
        services.AddScoped<CircuitHandler, ServicesAccessorCircuitHandler>
();

        return services;
    }
}
```

Access the circuit-scoped services by injecting the `CircuitServicesAccessor` where it's needed.

For an example that shows how to access the AuthenticationStateProvider from a DelegatingHandler set up using IHttpClientFactory, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Additional resources

- Service injection via a top-level imports file (_Imports.razor) in Blazor Web Apps
- Dependency injection in ASP.NET Core
- IDisposable guidance for Transient and shared instances
- Dependency injection into views in ASP.NET Core
- Primary constructors (C# Guide)

ASP.NET Core Blazor startup

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains Blazor app startup configuration.

For general guidance on ASP.NET Core app configuration for server-side development, see Configuration in ASP.NET Core.

Startup process and configuration

The Blazor startup process is automatic and asynchronous via the Blazor script (`blazor.*.js`), where the `*` placeholder is:

- `web` for a Blazor Web App
- `server` for a Blazor Server app
- `webassembly` for a Blazor WebAssembly app

For the location of the script, see ASP.NET Core Blazor project structure.

To manually start Blazor:

Blazor Web App:

- Add an `autostart="false"` attribute and value to the Blazor `<script>` tag.
- Place a script that calls `Blazor.start()` after the Blazor `<script>` tag and inside the closing `</body>` tag.
- Place static server-side rendering (static SSR) options in the `ssr` property.
- Place server-side Blazor-SignalR circuit options in the `circuit` property.
- Place client-side WebAssembly options in the `webAssembly` property.

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  ...
  Blazor.start({
    ssr: {
```

```
      ...
    },
    circuit: {
      ...
    },
    webAssembly: {
      ...
    }
  });
  ...
</script>
```

Standalone Blazor WebAssembly and Blazor Server:

- Add an `autostart="false"` attribute and value to the Blazor `<script>` tag.
- Place a script that calls `Blazor.start()` after the Blazor `<script>` tag and inside the closing `</body>` tag.
- You can provide additional options in the `Blazor.start()` parameter.

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  ...
  Blazor.start({
    ...
  });
  ...
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

JavaScript initializers

JavaScript (JS) initializers execute logic before and after a Blazor app loads. JS initializers are useful in the following scenarios:

- Customizing how a Blazor app loads.
- Initializing libraries before Blazor starts up.
- Configuring Blazor settings.

JS initializers are detected as part of the build process and imported automatically. Use of JS initializers often removes the need to manually trigger script functions from the app when using Razor class libraries (RCLs).

To define a JS initializer, add a JS module to the project named `{NAME}.lib.module.js`, where the `{NAME}` placeholder is the assembly name, library name, or package identifier. Place the file in the project's web root, which is typically the `wwwroot` folder.

For Blazor Web Apps:

- `beforeWebStart(options)`: Called before the Blazor Web App starts. For example, `beforeWebStart` is used to customize the loading process, logging level, and other options. Receives the Blazor Web options (`options`).
- `afterWebStarted(blazor)`: Called after all `beforeWebStart` promises resolve. For example, `afterWebStarted` can be used to register Blazor event listeners and custom event types. The Blazor instance is passed to `afterWebStarted` as an argument (`blazor`).
- `beforeServerStart(options, extensions)`: Called before the first Server runtime is started. Receives SignalR circuit start options (`options`) and any extensions (`extensions`) added during publishing.
- `afterServerStarted(blazor)`: Called after the first Interactive Server runtime is started.
- `beforeWebAssemblyStart(options, extensions)`: Called before the Interactive WebAssembly runtime is started. Receives the Blazor options (`options`) and any extensions (`extensions`) added during publishing. For example, options can specify the use of a custom boot resource loader.
- `afterWebAssemblyStarted(blazor)`: Called after the Interactive WebAssembly runtime is started.

> **Note**
>
> Legacy JS initializers (`beforeStart`, `afterStarted`) aren't invoked by default in a Blazor Web App. You can enable the legacy initializers to run with the `enableClassicInitializers` option. However, legacy initializer execution is unpredictable.
>
> HTML
>
> ```
> <script>
>   Blazor.start({ enableClassicInitializers: true });
> </script>
> ```

For Blazor Server, Blazor WebAssembly, and Blazor Hybrid apps:

- `beforeStart(options, extensions)`: Called before Blazor starts. For example, `beforeStart` is used to customize the loading process, logging level, and other options specific to the hosting model.
  - Client-side, `beforeStart` receives the Blazor options (`options`) and any extensions (`extensions`) added during publishing. For example, options can specify the use of a custom boot resource loader.
  - Server-side, `beforeStart` receives SignalR circuit start options (`options`).
  - In a BlazorWebView, no options are passed.
- `afterStarted(blazor)`: Called after Blazor is ready to receive calls from JS. For example, `afterStarted` is used to initialize libraries by making JS interop calls and registering custom elements. The Blazor instance is passed to `afterStarted` as an argument (`blazor`).

Additional .NET WebAssembly runtime callbacks:

- `onRuntimeConfigLoaded(config)`: Called when the boot configuration is downloaded. Allows the app to modify parameters (config) before the runtime starts (the parameter is `MonoConfig` from dotnet.d.ts):

  JavaScript

  ```
  export function onRuntimeConfigLoaded(config) {
    // Sample: Enable startup diagnostic logging when the URL contains
    // parameter debug=1
    const params = new URLSearchParams(location.search);
    if (params.get("debug") == "1") {
      config.diagnosticTracing = true;
    }
  }
  ```

- `onRuntimeReady({ getAssemblyExports, getConfig })`: Called after the .NET WebAssembly runtime has started (the parameter is `RuntimeAPI` from dotnet.d.ts):

  JavaScript

  ```
  export function onRuntimeReady({ getAssemblyExports, getConfig }) {
    // Sample: After the runtime starts, but before Main method is
  called,
    // call [JSExport]ed method.
    const config = getConfig();
    const exports = await getAssemblyExports(config.mainAssemblyName);
    exports.Sample.Greet();
  }
  ```

Both callbacks can return a `Promise`, and the promise is awaited before the startup continues.

For the file name:

- If the JS initializers are consumed as a static asset in the project, use the format `{ASSEMBLY NAME}.lib.module.js`, where the `{ASSEMBLY NAME}` placeholder is the app's assembly name. For example, name the file `BlazorSample.lib.module.js` for a project with an assembly name of `BlazorSample`. Place the file in the app's `wwwroot` folder.
- If the JS initializers are consumed from an RCL, use the format `{LIBRARY NAME/PACKAGE ID}.lib.module.js`, where the `{LIBRARY NAME/PACKAGE ID}` placeholder is the project's library name or package identifier. For example, name the file `RazorClassLibrary1.lib.module.js` for an RCL with a package identifier of `RazorClassLibrary1`. Place the file in the library's `wwwroot` folder.

For Blazor Web Apps:

The following example demonstrates JS initializers that load custom scripts before and after the Blazor Web App has started by appending them to the `<head>` in `beforeWebStart` and `afterWebStarted`:

JavaScript

```
export function beforeWebStart() {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'beforeStartScripts.js');
  document.head.appendChild(customScript);
}

export function afterWebStarted() {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'afterStartedScripts.js');
  document.head.appendChild(customScript);
}
```

The preceding `beforeWebStart` example only guarantees that the custom script loads before Blazor starts. It doesn't guarantee that awaited promises in the script complete their execution before Blazor starts.

For Blazor Server, Blazor WebAssembly, and Blazor Hybrid apps:

The following example demonstrates JS initializers that load custom scripts before and after Blazor has started by appending them to the `<head>` in `beforeStart` and `afterStarted`:

JavaScript

```
export function beforeStart(options, extensions) {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'beforeStartScripts.js');
  document.head.appendChild(customScript);
}

export function afterStarted(blazor) {
  var customScript = document.createElement('script');
  customScript.setAttribute('src', 'afterStartedScripts.js');
  document.head.appendChild(customScript);
}
```

The preceding `beforeStart` example only guarantees that the custom script loads before Blazor starts. It doesn't guarantee that awaited promises in the script complete their execution before Blazor starts.

> **Note**
>
> MVC and Razor Pages apps don't automatically load JS initializers. However, developer code can include a script to fetch the app's manifest and trigger the load of the JS initializers.

For examples of JS initializers, see the following resources:

- ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)
- Use Razor components in JavaScript apps and SPA frameworks (`quoteContainer2` *example*)
- ASP.NET Core Blazor event handling (*Custom clipboard paste event example*)
- Enable QR code generation for TOTP authenticator apps in an ASP.NET Core Blazor Web App
- Basic Test App in the ASP.NET Core GitHub repository (BasicTestApp.lib.module.js)

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Ensure libraries are loaded in a specific order

Append custom scripts to the `<head>` in `beforeStart` and `afterStarted` in the order that they should load.

The following example loads `script1.js` before `script2.js` and `script3.js` before `script4.js`:

JavaScript

```
export function beforeStart(options, extensions) {
    var customScript1 = document.createElement('script');
    customScript1.setAttribute('src', 'script1.js');
    document.head.appendChild(customScript1);

    var customScript2 = document.createElement('script');
    customScript2.setAttribute('src', 'script2.js');
    document.head.appendChild(customScript2);
}

export function afterStarted(blazor) {
    var customScript1 = document.createElement('script');
    customScript1.setAttribute('src', 'script3.js');
    document.head.appendChild(customScript1);

    var customScript2 = document.createElement('script');
    customScript2.setAttribute('src', 'script4.js');
    document.head.appendChild(customScript2);
}
```

Import additional modules

Use top-level import statements in the JS initializers file to import additional modules.

`additionalModule.js`:

JavaScript

```
export function logMessage() {
  console.log('logMessage is logging');
}
```

In the JS initializers file (`.lib.module.js`):

JavaScript

```
import { logMessage } from "/additionalModule.js";

export function beforeStart(options, extensions) {
  ...

  logMessage();
}
```

Import map

Import maps are supported by ASP.NET Core and Blazor.

Initialize Blazor when the document is ready

The following example starts Blazor when the document is ready:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  document.addEventListener("DOMContentLoaded", function() {
    Blazor.start();
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Chain to the `Promise` that results from a manual start

To perform additional tasks, such as JS interop initialization, use then to chain to the Promise that results from a manual Blazor app start:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start().then(function () {
    ...
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

> **Note**
>
> For a library to automatically execute additional tasks after Blazor has started, use a JavaScript initializer. Use of a JS initializer doesn't require the consumer of the library to chain JS calls to Blazor's manual start.

Load client-side boot resources

When an app loads in the browser, the app downloads boot resources from the server:

- JavaScript code to bootstrap the app
- .NET runtime and assemblies
- Locale specific data

Customize how these boot resources are loaded using the `loadBootResource` API. The `loadBootResource` function overrides the built-in boot resource loading mechanism. Use `loadBootResource` for the following scenarios:

- Load static resources, such as timezone data or `dotnet.wasm`, from a CDN.
- Load compressed assemblies using an HTTP request and decompress them on the client for hosts that don't support fetching compressed contents from the server.
- Alias resources to a different name by redirecting each `fetch` request to a new name.

> **Note**
>
> External sources must return the required Cross-Origin Resource Sharing (CORS) headers for browsers to allow cross-origin resource loading. CDNs usually provide the required headers.

`loadBootResource` parameters appear in the following table.

Expand table

Parameter	Description
`type`	The type of the resource. Permissible types include: `assembly`, `pdb`, `dotnetjs`, `dotnetwasm`, and `timezonedata`. You only need to specify types for custom behaviors.

Parameter	Description
	Types not specified to `loadBootResource` are loaded by the framework per their default loading behaviors. The `dotnetjs` boot resource (`dotnet.*.js`) must either return `null` for default loading behavior or a URI for the source of the `dotnetjs` boot resource.
`name`	The name of the resource.
`defaultUri`	The relative or absolute URI of the resource.
`integrity`	The integrity string representing the expected content in the response.

The `loadBootResource` function can return a URI string to override the loading process. In the following example, the following files from `bin/Release/{TARGET FRAMEWORK}/wwwroot/_framework` are served from a CDN at `https://cdn.example.com/blazorwebassembly/{VERSION}/`:

- `dotnet.*.js`
- `dotnet.wasm`
- Timezone data

The `{TARGET FRAMEWORK}` placeholder is the target framework moniker (for example, `net7.0`). The `{VERSION}` placeholder is the shared framework version (for example, `7.0.0`).

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    webAssembly: {
      loadBootResource: function (type, name, defaultUri, integrity) {
        console.log(`Loading: '${type}', '${name}', '${defaultUri}',
'${integrity}'`);
        switch (type) {
          case 'dotnetjs':
          case 'dotnetwasm':
          case 'timezonedata':
            return
`https://cdn.example.com/blazorwebassembly/{VERSION}/${name}`;
        }
      }
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    loadBootResource: function (type, name, defaultUri, integrity) {
      console.log(`Loading: '${type}', '${name}', '${defaultUri}',
'${integrity}'`);
      switch (type) {
        case 'dotnetjs':
        case 'dotnetwasm':
        case 'timezonedata':
          return
`https://cdn.example.com/blazorwebassembly/{VERSION}/${name}`;
      }
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

To customize more than just the URLs for boot resources, the `loadBootResource` function can call `fetch` directly and return the result. The following example adds a custom HTTP header to the outbound requests. To retain the default integrity checking behavior, pass through the `integrity` parameter.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    webAssembly: {
      loadBootResource: function (type, name, defaultUri, integrity) {
        if (type == 'dotnetjs') {
          return null;
        } else {
          return fetch(defaultUri, {
            cache: 'no-cache',
            integrity: integrity,
            headers: { 'Custom-Header': 'Custom Value' }
          });
        }
      }
    }
```

```
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    loadBootResource: function (type, name, defaultUri, integrity) {
      if (type == 'dotnetjs') {
        return null;
      } else {
        return fetch(defaultUri, {
          cache: 'no-cache',
          integrity: integrity,
          headers: { 'Custom-Header': 'Custom Value' }
        });
      }
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

When the `loadBootResource` function returns `null`, Blazor uses the default loading behavior for the resource. For example, the preceding code returns `null` for the `dotnetjs` boot resource (`dotnet.*.js`) because the `dotnetjs` boot resource must either return `null` for default loading behavior or a URI for the source of the `dotnetjs` boot resource.

The `loadBootResource` function can also return a Response promise. For an example, see Host and deploy ASP.NET Core Blazor WebAssembly.

For more information, see ASP.NET Core Blazor WebAssembly .NET runtime and app bundle caching.

Control headers in C# code

Control headers at startup in C# code using the following approaches.

In the following examples, a Content Security Policy (CSP) is applied to the app via a CSP header. The `{POLICY STRING}` placeholder is the CSP policy string.

Server-side and prerendered client-side scenarios

Use ASP.NET Core Middleware to control the headers collection.

In the `Program` file:

C#

```
app.Use(async (context, next) =>
{
    context.Response.Headers.Append("Content-Security-Policy", "{POLICY
STRING}");
    await next();
});
```

The preceding example uses inline middleware, but you can also create a custom middleware class and call the middleware with an extension method in the `Program` file. For more information, see Write custom ASP.NET Core middleware.

For more information on CSPs, see Enforce a Content Security Policy for ASP.NET Core Blazor.

Client-side loading progress indicators

A loading progress indicator shows the loading progress of the app to users, indicating that the app is loading normally and that the user should wait until loading is finished.

Blazor Web App loading progress

The loading progress indicator used in Blazor WebAssembly apps isn't present in an app created from the Blazor Web App project template. Usually, a loading progress indicator isn't desirable for interactive WebAssembly components because Blazor Web Apps prerender client-side components on the server for fast initial load times. For mixed-render-mode situations, the framework or developer code must also be careful to avoid the following problems:

- Showing multiple loading indicators on the same rendered page.
- Inadvertently discarding prerendered content while the .NET WebAssembly runtime is loading.

A future release of .NET might provide a framework-based loading progress indicator. In the meantime, you can add a custom loading progress indicator to a Blazor Web App.

Create a `LoadingProgress` component in the `.Client` app that calls OperatingSystem.IsBrowser:

- When `false`, display a loading progress indicator while the Blazor bundle is downloaded and before the Blazor runtime activates on the client.
- When `true`, render the requested component's content.

The following demonstration uses the loading progress indicator found in apps created from the Blazor WebAssembly template, including a modification of the styles that the template provides. The styles are loaded into the app's `<head>` content by the HeadContent component. For more information, see Control head content in ASP.NET Core Blazor apps.

`LoadingProgress.razor`:

razor

```
@if (!OperatingSystem.IsBrowser())
{
    <HeadContent>
        <style>
            .loading-progress {
                position: relative;
                display: block;
                width: 8rem;
                height: 8rem;
                margin: 20vh auto 1rem auto;
            }

                .loading-progress circle {
                    fill: none;
                    stroke: #e0e0e0;
                    stroke-width: 0.6rem;
                    transform-origin: 50% 50%;
                    transform: rotate(-90deg);
                }

                    .loading-progress circle:last-child {
                        stroke: #1b6ec2;
                        stroke-dasharray:
                            calc(3.142 * var(--blazor-load-percentage, 0%) *
0.8),
                            500%;
                        transition: stroke-dasharray 0.05s ease-in-out;
                    }

            .loading-progress-text {
                position: relative;
                text-align: center;
                font-weight: bold;
                top: -90px;
```

```
            }

                .loading-progress-text:after {
                    content: var(--blazor-load-percentage-text, "Loading");
                }

            code {
                color: #c02d76;
            }
        </style>
    </HeadContent>
    <svg class="loading-progress">
        <circle r="40%" cx="50%" cy="50%" />
        <circle r="40%" cx="50%" cy="50%" />
    </svg>
    <div class="loading-progress-text"></div>
}
else
{
    @ChildContent
}

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

In a component that adopts Interactive WebAssembly rendering, wrap the component's Razor markup with the `LoadingProgress` component. The following example demonstrates the approach with the `Counter` component of an app created from the Blazor Web App project template.

`Pages/Counter.razor`:

razor

```
@page "/counter"
@rendermode InteractiveWebAssembly

<PageTitle>Counter</PageTitle>

<LoadingProgress>
    <h1>Counter</h1>

    <p role="status">Current count: @currentCount</p>

    <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
</LoadingProgress>

@code {
    private int currentCount = 0;
```

```
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Blazor WebAssembly app loading progress

The project template contains Scalable Vector Graphics (SVG) and text indicators that show the loading progress of the app.

The progress indicators are implemented with HTML and CSS using two CSS custom properties (variables) provided by Blazor:

- `--blazor-load-percentage`: The percentage of app files loaded.
- `--blazor-load-percentage-text`: The percentage of app files loaded, rounded to the nearest whole number.

Using the preceding CSS variables, you can create custom progress indicators that match the styling of your app.

In the following example:

- `resourcesLoaded` is an instantaneous count of the resources loaded during app startup.
- `totalResources` is the total number of resources to load.

JavaScript

```
const percentage = resourcesLoaded / totalResources * 100;
document.documentElement.style.setProperty(
  '--blazor-load-percentage', `${percentage}%`);
document.documentElement.style.setProperty(
  '--blazor-load-percentage-text', `"${Math.floor(percentage)}%"`);
```

The default round progress indicator is implemented in HTML in the `wwwroot/index.html` file:

HTML

```
<div id="app">
    <svg class="loading-progress">
        <circle r="40%" cx="50%" cy="50%" />
        <circle r="40%" cx="50%" cy="50%" />
    </svg>
```

```
    <div class="loading-progress-text"></div>
</div>
```

To review the project template markup and styling for the default progress indicators, see the ASP.NET Core reference source:

- wwwroot/index.html
- app.css

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Instead of using the default round progress indicator, the following example shows how to implement a linear progress indicator.

Add the following styles to `wwwroot/css/app.css`:

```css
.linear-progress {
    background: silver;
    width: 50vw;
    margin: 20% auto;
    height: 1rem;
    border-radius: 10rem;
    overflow: hidden;
    position: relative;
}

.linear-progress:after {
    content: '';
    position: absolute;
    inset: 0;
    background: blue;
    scale: var(--blazor-load-percentage, 0%) 100%;
    transform-origin: left top;
    transition: scale ease-out 0.5s;
}
```

A CSS variable (`var(...)`) is used to pass the value of `--blazor-load-percentage` to the `scale` property of a blue pseudo-element that indicates the loading progress of the

app's files. As the app loads, `--blazor-load-percentage` is updated automatically, which dynamically changes the progress indicator's visual representation.

In `wwwroot/index.html`, remove the default SVG round indicator in `<div id="app">...</div>` and replace it with the following markup:

HTML

```
<div class="linear-progress"></div>
```

Configure the .NET WebAssembly runtime

In advanced programming scenarios, the `configureRuntime` function with the `dotnet` runtime host builder is used to configure the .NET WebAssembly runtime. For example, `dotnet.withEnvironmentVariable` sets an environment variable that:

- Configures the .NET WebAssembly runtime.
- Changes the behavior of a C library.

> **Note**
>
> A documentation request is pending in the `dotnet/runtime` GitHub repository for more information on environment variables that configure the .NET WebAssembly runtime or affect the behavior of C libraries. Although the documentation request is pending, more information and cross-links to additional resources are available in the request, Question/request for documentation on .NET WASM runtime env vars (dotnet/runtime #98225).

The `configureRuntime` function can also be used to enable integration with a browser profiler.

For the placeholders in the following examples that set an environment variable:

- The `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.
- The `{NAME}` placeholder is the environment variable's name.
- The `{VALUE}` placeholder is the environment variable's value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    webAssembly: {
      configureRuntime: dotnet => {
        dotnet.withEnvironmentVariable("{NAME}", "{VALUE}");
      }
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureRuntime: dotnet => {
      dotnet.withEnvironmentVariable("{NAME}", "{VALUE}");
    }
  });
</script>
```

> **Note**
>
> The .NET runtime instance can be accessed using the .NET WebAssembly Runtime API (`Blazor.runtime`). For example, the app's build configuration can be obtained using `Blazor.runtime.runtimeBuildInfo.buildConfiguration`.
>
> For more information on the .NET WebAssembly runtime configuration, see the runtime's TypeScript definition file (dotnet.d.ts) in the dotnet/runtime GitHub repository.
>
> > **Note**
> >
> > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Disable enhanced navigation and form handling

This section applies to Blazor Web Apps.

To disable enhanced navigation and form handling, set `disableDomPreservation` to `true` for `Blazor.start`:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    ssr: { disableDomPreservation: true }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Additional resources

- Environments: Set the app's environment
- SignalR (includes sections on SignalR startup configuration)
- Globalization and localization: Statically set the culture with Blazor.start() (*client-side only*)
- Host and deploy: Blazor WebAssembly: Compression

ASP.NET Core Blazor environments

Article • 12/20/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure and read the environment in a Blazor app.

When running an app locally, the environment defaults to `Development`. When the app is published, the environment defaults to `Production`.

We recommend the following conventions:

- Always use the "`Development`" environment name for local development. This is because the ASP.NET Core framework expects exactly that name when configuring the app and tooling for local development runs of an app.
- For testing, staging, and production environments, always publish and deploy the app. You can use any environment naming scheme that you wish for published apps, but always use app setting file names with casing of the environment segment that exactly matches the environment name. For staging, use "`Staging`" (capital "S") as the environment name, and name the app settings file to match (`appsettings.Staging.json`). For production, use "`Production`" (capital "P") as the environment name, and name the app settings file to match (`appsettings.Production.json`).

Set the environment

The environment is set using any of the following approaches:

- Blazor Web App: Use any of the approaches described in Use multiple environments in ASP.NET Core for general ASP.NET Core apps.
- Blazor Web App or standalone Blazor WebAssembly: Blazor start configuration
- Standalone Blazor WebAssembly: Blazor-Environment header
- Blazor Web App or standalone Blazor WebAssembly: Azure App Service

On the client for a Blazor Web App, the environment is determined from the server via a middleware that communicates the environment to the browser via a header named

`Blazor-Environment`. The header sets the environment when the WebAssemblyHost is created in the client-side `Program` file (WebAssemblyHostBuilder.CreateDefault).

For a standalone Blazor WebAssembly app running locally, the development server adds the `Blazor-Environment` header with the environment name obtained from the hosting environment. The hosting environment sets the environment from the `ASPNETCORE_ENVIRONMENT` environment variable established by the project's `Properties/launchSettings.json` file. The default value of the environment variable in a project created from the Blazor WebAssembly project template is `Development`. For more information, see the Set the client-side environment via header section.

For app's running locally in development, the app defaults to the `Development` environment. Publishing the app defaults the environment to `Production`.

Set the client-side environment via Blazor startup configuration

The following example starts Blazor in the `Staging` environment if the hostname includes `localhost`. Otherwise, the environment is set to its default value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  if (window.location.hostname.includes("localhost")) {
    Blazor.start({
      webAssembly: {
        environment: "Staging"
      }
    });
  } else {
    Blazor.start();
  }
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

> **Note**

For Blazor Web Apps that set the `webAssembly` > `environment` property in `Blazor.start` configuration, it's wise to match the server-side environment to the environment set on the `environment` property. Otherwise, prerendering on the server will operate under a different environment than rendering on the client, which results in arbitrary effects. For general guidance on setting the environment for a Blazor Web App, see Use multiple environments in ASP.NET Core.

Standalone Blazor WebAssembly:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  if (window.location.hostname.includes("localhost")) {
    Blazor.start({
      environment: "Staging"
    });
  } else {
    Blazor.start();
  }
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Using the `environment` property overrides the environment set by the Blazor-Environment header.

The preceding approach sets the client's environment without changing the `Blazor-Environment` header's value, nor does it change the server project's console logging of the startup environment for a Blazor Web App that has adopted global Interactive WebAssembly rendering.

To log the environment to the console in either a standalone Blazor WebAssembly project or the `.Client` project of a Blazor Web App, place the following C# code in the `Program` file after the WebAssemblyHost is created with WebAssemblyHostBuilder.CreateDefault and before the line that builds and runs the project (`await builder.Build().RunAsync();`):

C#

```
Console.WriteLine(
    $"Client Hosting Environment: {builder.HostEnvironment.Environment}");
```

For more information on Blazor startup, see ASP.NET Core Blazor startup.

Set the client-side environment via header

Blazor WebAssembly apps can set the environment with the `Blazor-Environment` header. Specifically, the response header must be set on the `_framework/blazor.boot.json` file, but there's no harm setting the header on file server responses for other Blazor file requests or the entire Blazor deployment.

Although the Blazor framework issues the header name in kebab case with mixed letter case (`Blazor-Environment`), you're welcome to use all-lower or all-upper kebab case (`blazor-environment`, `BLAZOR-ENVIRONMENT`).

For local development runs with Blazor's built-in development server, you can control the value of the `Blazor-Environment` header by setting the value of the `ASPNETCORE_ENVIRONMENT` environment variable in the project's `Properties/launchSettings.json` file. When running locally with the development server, the order of precedence for determining the app's environment is Blazor.start configuration (environment key) > `Blazor-Environment` response header (`blazor.boot.json` file) > `ASPNETCORE_ENVIRONMENT` environment variable (`launchSettings.json`). You can't use the `ASPNETCORE_ENVIRONMENT` environment variable (`launchSettings.json`) approach for a deployed Blazor WebAssembly app. The technique only works with the development server on local runs of the app.

IIS

In the following example for IIS, the custom header (`Blazor-Environment`) is added to the published `web.config` file. The `web.config` file is located in the `bin/Release/{TARGET FRAMEWORK}/publish` folder, where the `{TARGET FRAMEWORK}` placeholder is the target framework:

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.webServer>
    ...
    <httpProtocol>
      <customHeaders>
        <add name="Blazor-Environment" value="Staging" />
      </customHeaders>
    </httpProtocol>
```

```
  </system.webServer>
</configuration>
```

> **Note**
>
> To use a custom `web.config` file for IIS that isn't overwritten when the app is published to the `publish` folder, see Host and deploy ASP.NET Core Blazor WebAssembly.

Nginx

For Nginx servers, use the `add_header` directive from the `ngx_http_headers_module`:

```
http {
    server {
        ...
        location / {
            ...
            add_header Blazor-Environment "Staging";
        }
    }
}
```

For more information, see the following resources:

- Nginx documentation
- Blazor WebAssembly Nginx hosting guidance

Apache

For Apache servers, use the `Header` directive from the `mod_headers` module:

```
<VirtualHost *:80>
    ...
    Header set Blazor-Environment "Staging"
    ...
</VirtualHost>
```

For more information:

- Apache documentation (search the latest release for "mod_headers")
- Blazor WebAssembly Apache hosting guidance

Set the environment for Azure App Service

For a standalone Blazor WebAssembly app, you can set the environment manually via start configuration or the Blazor-Environment header.

For a server-side app, set the environment via an `ASPNETCORE_ENVIRONMENT` app setting in Azure:

1. ***Confirm that the casing of environment segments in app settings file names match their environment name casing exactly***. For example, the matching app settings file name for the `Staging` environment is `appsettings.Staging.json`. If the file name is `appsettings.staging.json` (lowercase "`s`"), the file isn't located, and the settings in the file aren't used in the `Staging` environment.

2. For Visual Studio deployment, confirm that the app is deployed to the correct deployment slot. For an app named `BlazorAzureAppSample`, the app is deployed to the `Staging` deployment slot.

3. In the Azure portal for the environment's deployment slot, set the environment with the `ASPNETCORE_ENVIRONMENT` app setting. For an app named `BlazorAzureAppSample`, the staging App Service Slot is named `BlazorAzureAppSample/Staging`. For the `Staging` slot's configuration, create an app setting for `ASPNETCORE_ENVIRONMENT` with a value of `Staging`. **Deployment slot setting** is enabled for the setting.

When requested in a browser, the `BlazorAzureAppSample/Staging` app loads in the `Staging` environment at `https://blazorazureappsample-staging.azurewebsites.net`.

When the app is loaded in the browser, the response header collection for `blazor.boot.json` indicates that the `Blazor-Environment` header value is `Staging`.

App settings from the `appsettings.{ENVIRONMENT}.json` file are loaded by the app, where the `{ENVIRONMENT}` placeholder is the app's environment. In the preceding example, settings from the `appsettings.Staging.json` file are loaded.

Read the environment in a Blazor WebAssembly app

Obtain the app's environment in a component by injecting IWebAssemblyHostEnvironment and reading the Environment property.

`ReadEnvironment.razor`:

```razor
@page "/read-environment"
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
@inject IWebAssemblyHostEnvironment Env

<h1>Environment example</h1>

<p>Environment: @Env.Environment</p>
```

Read the environment client-side in a Blazor Web App

Assuming that prerendering isn't disabled for a component or the app, a component in the `.Client` project is prerendered on the server. Because the server doesn't have a registered IWebAssemblyHostEnvironment service, it isn't possible to inject the service and use the service implementation's host environment extension methods and properties during server prerendering. Injecting the service into an Interactive WebAssembly or Interactive Auto component results in the following runtime error:

> There is no registered service of type 'Microsoft.AspNetCore.Components.WebAssembly.Hosting.IWebAssemblyHostEnvironment'.

To address this, create a custom service implementation for IWebAssemblyHostEnvironment on the server. Add the following class to the server project.

`ServerHostEnvironment.cs`:

```C#
using Microsoft.AspNetCore.Components.WebAssembly.Hosting;
using Microsoft.AspNetCore.Components;

public class ServerHostEnvironment(IWebHostEnvironment env,
NavigationManager nav) :
    IWebAssemblyHostEnvironment
{
```

```
    public string Environment => env.EnvironmentName;
    public string BaseAddress => nav.BaseUri;
}
```

In the server project's `Program` file, register the service:

C#

```
builder.Services.TryAddScoped<IWebAssemblyHostEnvironment,
ServerHostEnvironment>();
```

At this point, the IWebAssemblyHostEnvironment service can be injected into an interactive WebAssembly or interactive Auto component and used as shown in the Read the environment in a Blazor WebAssembly app section.

The preceding example can demonstrate that it's possible to have a different server environment than client environment, which isn't recommended and may lead to arbitrary results. When setting the environment in a Blazor Web App, it's best to match server and `.Client` project environments. Consider the following scenario in a test app:

- Implement the client-side (`webassembly`) environment property with the `Staging` environment via `Blazor.start`. See the Set the client-side environment via startup configuration section for an example.
- Don't change the server-side `Properties/launchSettings.json` file. Leave the `environmentVariables` section with the `ASPNETCORE_ENVIRONMENT` environment variable set to `Development`.

You can see the value of the IWebAssemblyHostEnvironment.Environment property change in the UI.

When prerendering occurs on the server, the component is rendered in the `Development` environment:

> **Environment:** Development

When the component is rerendered just a second or two later, after the Blazor bundle is downloaded and the .NET WebAssembly runtime activates, the values change to reflect that the client is operating in the `Staging` environment on the client:

> **Environment:** Staging

The preceding example demonstrates why we recommend setting the server environment to match the client environment for development, testing, and production deployments.

For more information, see the Client-side services fail to resolve during prerendering section of the *Render modes* article, which appears later in the Blazor documentation.

Read the client-side environment during startup

During startup, the WebAssemblyHostBuilder exposes the IWebAssemblyHostEnvironment through the HostEnvironment property, which enables environment-specific logic in host builder code.

In the `Program` file:

```
C#

if (builder.HostEnvironment.Environment == "Custom")
{
    ...
};
```

The following convenience extension methods provided through WebAssemblyHostEnvironmentExtensions permit checking the current environment for `Development`, `Production`, `Staging`, and custom environment names:

- IsDevelopment
- IsProduction
- IsStaging
- IsEnvironment

In the `Program` file:

```
C#

if (builder.HostEnvironment.IsStaging())
{
    ...
};

if (builder.HostEnvironment.IsEnvironment("Custom"))
{
```

```
    ...
};
```

The IWebAssemblyHostEnvironment.BaseAddress property can be used during startup when the NavigationManager service isn't available.

Additional resources

- ASP.NET Core Blazor startup
- Use multiple environments in ASP.NET Core
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor logging

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains Blazor app logging, including configuration and how to write log messages from Razor components.

Configuration

Logging configuration can be loaded from app settings files. For more information, see ASP.NET Core Blazor configuration.

At default log levels and without configuring additional logging providers:

- On the server, logging only occurs to the server-side .NET console in the `Development` environment at the LogLevel.Information level or higher.
- On the client, logging only occurs to the client-side browser developer tools console at the LogLevel.Information level or higher.

When the app is configured in the project file to use implicit namespaces (`<ImplicitUsings>enable</ImplicitUsings>`), a `using` directive for Microsoft.Extensions.Logging or any API in the LoggerExtensions class isn't required to support API Visual Studio IntelliSense completions or building apps. If implicit namespaces aren't enabled, Razor components must explicitly define @using directives for logging namespaces that aren't imported via the `_Imports.razor` file.

Log levels

Log levels conform to ASP.NET Core app log levels, which are listed in the API documentation at LogLevel.

Razor component logging

The following example:

- Injects an ILogger (`ILogger<Counter1>`) object to create a logger. The log's *category* is the fully qualified name of the component's type, `Counter`.
- Calls LogWarning to log at the Warning level.

`Counter1.razor`:

razor

```
@page "/counter-1"
@inject ILogger<Counter1> Logger

<PageTitle>Counter 1</PageTitle>

<h1>Counter 1</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        Logger.LogWarning("Someone has clicked me!");

        currentCount++;
    }
}
```

The following example demonstrates logging with an ILoggerFactory in components.

`Counter2.razor`:

razor

```
@page "/counter-2"
@inject ILoggerFactory LoggerFactory

<PageTitle>Counter 2</PageTitle>

<h1>Counter 2</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
```

```
    {
        var logger = LoggerFactory.CreateLogger<Counter2>();
        logger.LogWarning("Someone has clicked me!");

        currentCount++;
    }
}
```

Server-side logging

For general ASP.NET Core logging guidance, see Logging in .NET Core and ASP.NET Core.

Client-side logging

Not every feature of ASP.NET Core logging is supported client-side. For example, client-side components don't have access to the client's file system or network, so writing logs to the client's physical or network storage isn't possible. When using a third-party logging service designed to work with single-page apps (SPAs), follow the service's security guidance. Keep in mind that every piece of data, including keys or secrets stored client-side are ***insecure*** and can be easily discovered by malicious users.

Configure logging in client-side apps with the WebAssemblyHostBuilder.Logging property. The Logging property is of type ILoggingBuilder, so the extension methods of ILoggingBuilder are supported.

To set the minimum logging level, call LoggingBuilderExtensions.SetMinimumLevel on the host builder in the `Program` file with the LogLevel. The following example sets the minimum log level to Warning:

C#

```
builder.Logging.SetMinimumLevel(LogLevel.Warning);
```

Log in the client-side `Program` file

Logging is supported in client-side apps after the WebAssemblyHostBuilder is built using the framework's internal console logger provider (WebAssemblyConsoleLoggerProvider (reference source)).

In the `Program` file:

C#

```
var host = builder.Build();

var logger = host.Services.GetRequiredService<ILoggerFactory>()
    .CreateLogger<Program>();

logger.LogInformation("Logged after the app is built in the Program file.");

await host.RunAsync();
```

Developer tools console output:

```
info: Program[0]
Logged after the app is built in the Program file.
```

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Client-side log category

Log categories are supported.

The following example shows how to use log categories with the `Counter` component of an app created from a Blazor project template.

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`) that injects an ILoggerFactory as `LoggerFactory`:

C#

```
var logger = LoggerFactory.CreateLogger("CustomCategory");
logger.LogWarning("Someone has clicked me!");
```

Developer tools console output:

```
warn: CustomCategory[0]
Someone has clicked me!
```

Client-side log event ID

Log event ID is supported.

The following example shows how to use log event IDs with the `Counter` component of an app created from a Blazor project template.

`LogEvent.cs`:

C#

```
public class LogEvent
{
    public const int Event1 = 1000;
    public const int Event2 = 1001;
}
```

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`):

C#

```
logger.LogInformation(LogEvent.Event1, "Someone has clicked me!");
logger.LogWarning(LogEvent.Event2, "Someone has clicked me!");
```

Developer tools console output:

> info: BlazorSample.Pages.Counter[1000]
> Someone has clicked me!
> warn: BlazorSample.Pages.Counter[1001]
> Someone has clicked me!

Client-side log message template

Log message templates are supported:

The following example shows how to use log message templates with the `Counter` component of an app created from a Blazor project template.

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`):

C#

```
logger.LogInformation("Someone clicked me at {CurrentDT}!",
DateTime.UtcNow);
```

Developer tools console output:

> info: BlazorSample.Pages.Counter[0]
> Someone clicked me at 04/21/2022 12:15:57!

Client-side log exception parameters

Log exception parameters are supported.

The following example shows how to use log exception parameters with the `Counter` component of an app created from a Blazor project template.

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`):

C#

```
currentCount++;

try
{
    if (currentCount == 3)
    {
        currentCount = 4;
        throw new OperationCanceledException("Skip 3");
    }
}
catch (Exception ex)
{
    logger.LogWarning(ex, "Exception (currentCount: {Count})!",
currentCount);
}
```

Developer tools console output:

> warn: BlazorSample.Pages.Counter[0]
> Exception (currentCount: 4)!
> System.OperationCanceledException: Skip 3
> at BlazorSample.Pages.Counter.IncrementCount() in
> C:UsersAlabaDesktopBlazorSamplePagesCounter.razor:line 28

Client-side filter function

Filter functions are supported.

The following example shows how to use a filter with the `Counter` component of an app created from a Blazor project template.

In the `Program` file:

```
C#

builder.Logging.AddFilter((provider, category, logLevel) =>
    category.Equals("CustomCategory2") && logLevel == LogLevel.Information);
```

In the `IncrementCount` method of the app's `Counter` component (`Counter.razor`) that injects an ILoggerFactory as `LoggerFactory`:

```
C#

var logger1 = LoggerFactory.CreateLogger("CustomCategory1");
logger1.LogInformation("Someone has clicked me!");

var logger2 = LoggerFactory.CreateLogger("CustomCategory1");
logger2.LogWarning("Someone has clicked me!");

var logger3 = LoggerFactory.CreateLogger("CustomCategory2");
logger3.LogInformation("Someone has clicked me!");

var logger4 = LoggerFactory.CreateLogger("CustomCategory2");
logger4.LogWarning("Someone has clicked me!");
```

In the developer tools console output, the filter only permits logging for the `CustomCategory2` category and Information log level message:

> info: CustomCategory2[0]
> Someone has clicked me!

The app can also configure log filtering for specific namespaces. For example, set the log level to Trace in the `Program` file:

```
C#

builder.Logging.SetMinimumLevel(LogLevel.Trace);
```

Normally at the Trace log level, developer tools console output at the **Verbose** level includes Microsoft.AspNetCore.Components.RenderTree logging messages, such as the following:

> dbug: Microsoft.AspNetCore.Components.RenderTree.Renderer[3]
> Rendering component 14 of type
> Microsoft.AspNetCore.Components.Web.HeadOutlet

In the `Program` file, logging messages specific to Microsoft.AspNetCore.Components.RenderTree can be disabled using ***either*** of the following approaches:

- C#

```
builder.Logging.AddFilter("Microsoft.AspNetCore.Components.RenderTree.*
", LogLevel.None);
```

- C#

```
builder.Services.PostConfigure<LoggerFilterOptions>(options =>
    options.Rules.Add(
        new LoggerFilterRule(null,

"Microsoft.AspNetCore.Components.RenderTree.*",
                             LogLevel.None,
                             null)
    ));
```

After ***either*** of the preceding filters is added to the app, the console output at the **Verbose** level doesn't show logging messages from the Microsoft.AspNetCore.Components.RenderTree API.

Client-side custom logger provider

The example in this section demonstrates a custom logger provider for further customization.

Add a package reference to the app for the Microsoft.Extensions.Logging.Configuration package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Add the following custom logger configuration. The configuration establishes a `LogLevels` dictionary that sets a custom log format for three log levels: Information, Warning, and Error. A `LogFormat` enum is used to describe short (`LogFormat.Short`) and long (`LogFormat.Long`) formats.

`CustomLoggerConfiguration.cs`:

C#

```
using Microsoft.Extensions.Logging;

public class CustomLoggerConfiguration
{
    public int EventId { get; set; }

    public Dictionary<LogLevel, LogFormat> LogLevels { get; set; } =
        new()
        {
            [LogLevel.Information] = LogFormat.Short,
            [LogLevel.Warning] = LogFormat.Short,
            [LogLevel.Error] = LogFormat.Long
        };

    public enum LogFormat
    {
        Short,
        Long
    }
}
```

Add the following custom logger to the app. The `CustomLogger` outputs custom log formats based on the `logLevel` values defined in the preceding `CustomLoggerConfiguration` configuration.

C#

```
using Microsoft.Extensions.Logging;
using static CustomLoggerConfiguration;

public sealed class CustomLogger : ILogger
{
    private readonly string name;
    private readonly Func<CustomLoggerConfiguration> getCurrentConfig;

    public CustomLogger(
        string name,
        Func<CustomLoggerConfiguration> getCurrentConfig) =>
        (this.name, this.getCurrentConfig) = (name, getCurrentConfig);

    public IDisposable BeginScope<TState>(TState state) => default!;
```

```
    public bool IsEnabled(LogLevel logLevel) =>
        getCurrentConfig().LogLevels.ContainsKey(logLevel);

    public void Log<TState>(
        LogLevel logLevel,
        EventId eventId,
        TState state,
        Exception? exception,
        Func<TState, Exception?, string> formatter)
    {
        if (!IsEnabled(logLevel))
        {
            return;
        }

        CustomLoggerConfiguration config = getCurrentConfig();

        if (config.EventId == 0 || config.EventId == eventId.Id)
        {
            switch (config.LogLevels[logLevel])
            {
                case LogFormat.Short:
                    Console.WriteLine($"{name}: {formatter(state,
exception)}");
                    break;
                case LogFormat.Long:
                    Console.WriteLine($"[{eventId.Id, 2}: {logLevel, -12}]
{name} - {formatter(state, exception)}");
                    break;
                default:
                    // No-op
                    break;
            }
        }
    }
}
```

Add the following custom logger provider to the app. `CustomLoggerProvider` adopts an Options-based approach to configure the logger via built-in logging configuration features. For example, the app can set or change log formats via an `appsettings.json` file without requiring code changes to the custom logger, which is demonstrated at the end of this section.

`CustomLoggerProvider.cs`:

C#

```
using System.Collections.Concurrent;
using Microsoft.Extensions.Options;
```

```
[ProviderAlias("CustomLog")]
public sealed class CustomLoggerProvider : ILoggerProvider
{
    private readonly IDisposable onChangeToken;
    private CustomLoggerConfiguration config;
    private readonly ConcurrentDictionary<string, CustomLogger> loggers =
        new(StringComparer.OrdinalIgnoreCase);

    public CustomLoggerProvider(
        IOptionsMonitor<CustomLoggerConfiguration> config)
    {
        this.config = config.CurrentValue;
        onChangeToken = config.OnChange(updatedConfig => this.config =
updatedConfig);
    }

    public ILogger CreateLogger(string categoryName) =>
        loggers.GetOrAdd(categoryName, name => new CustomLogger(name,
GetCurrentConfig));

    private CustomLoggerConfiguration GetCurrentConfig() => config;

    public void Dispose()
    {
        loggers.Clear();
        onChangeToken.Dispose();
    }
}
```

Add the following custom logger extensions.

`CustomLoggerExtensions.cs`:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Configuration;

public static class CustomLoggerExtensions
{
    public static ILoggingBuilder AddCustomLogger(
        this ILoggingBuilder builder)
    {
        builder.AddConfiguration();

        builder.Services.TryAddEnumerable(
            ServiceDescriptor.Singleton<ILoggerProvider,
CustomLoggerProvider>());

        LoggerProviderOptions.RegisterProviderOptions
            <CustomLoggerConfiguration, CustomLoggerProvider>
```

```
(builder.Services);

        return builder;
    }
}
```

In the `Program` file on the host builder, clear the existing provider by calling ClearProviders and add the custom logging provider:

C#

```
builder.Logging.ClearProviders().AddCustomLogger();
```

In the following `CustomLoggerExample` component:

- The debug message isn't logged.
- The information message is logged in the short format (`LogFormat.Short`).
- The warning message is logged in the short format (`LogFormat.Short`).
- The error message is logged in the long format (`LogFormat.Long`).
- The trace message isn't logged.

`CustomLoggerExample.razor`:

razor

```
@page "/custom-logger-example"
@inject ILogger<CustomLoggerExample> Logger

<p>
    <button @onclick="LogMessages">Log Messages</button>
</p>

@code{
    private void LogMessages()
    {
        Logger.LogDebug(1, "This is a debug message.");
        Logger.LogInformation(3, "This is an information message.");
        Logger.LogWarning(5, "This is a warning message.");
        Logger.LogError(7, "This is an error message.");
        Logger.LogTrace(5!, "This is a trace message.");
    }
}
```

The following output is seen in the browser's developer tools console when the **`Log Messages`** button is selected. The log entries reflect the appropriate formats applied by the custom logger (the client app is named `LoggingTest`):

> LoggingTest.Pages.CustomLoggerExample: This is an information message.
> LoggingTest.Pages.CustomLoggerExample: This is a warning message.
> [7: Error] LoggingTest.Pages.CustomLoggerExample - This is an error message.

From a casual inspection of the preceding example, it's apparent that setting the log line formats via the dictionary in `CustomLoggerConfiguration` isn't strictly necessary. The line formats applied by the custom logger (`CustomLogger`) could have been applied by merely checking the `logLevel` in the `Log` method. The purpose of assigning the log format via configuration is that the developer can change the log format easily via app configuration, as the following example demonstrates.

In the client-side app, add or update the `appsettings.json` file to include logging configuration. Set the log format to `Long` for all three log levels:

JSON

```
{
  "Logging": {
    "CustomLog": {
      "LogLevels": {
        "Information": "Long",
        "Warning": "Long",
        "Error": "Long"
      }
    }
  }
}
```

In the preceding example, notice that the entry for the custom logger configuration is `CustomLog`, which was applied to the custom logger provider (`CustomLoggerProvider`) as an alias with `[ProviderAlias("CustomLog")]`. The logging configuration could have been applied with the name `CustomLoggerProvider` instead of `CustomLog`, but use of the alias `CustomLog` is more user friendly.

In the `Program` file, consume the logging configuration. Add the following code:

C#

```
builder.Logging.AddConfiguration(
    builder.Configuration.GetSection("Logging"));
```

The call to LoggingBuilderConfigurationExtensions.AddConfiguration can be placed either before or after adding the custom logger provider.

Run the app again. Select the **`Log Messages`** button. Notice that the logging configuration is applied from the `appsettings.json` file. All three log entries are in the long (`LogFormat.Long`) format (the client app is named `LoggingTest`):

> [3: Information] LoggingTest.Pages.CustomLoggerExample - This is an information message.
> [5: Warning] LoggingTest.Pages.CustomLoggerExample - This is a warning message.
> [7: Error] LoggingTest.Pages.CustomLoggerExample - This is an error message.

Client-side log scopes

The developer tools console logger doesn't support log scopes. However, a custom logger can support log scopes. For an unsupported example that you can further develop to suit your needs, see the `BlazorWebAssemblyScopesLogger` sample app in the Blazor samples GitHub repository (how to download).

The sample app uses standard ASP.NET Core BeginScope logging syntax to indicate scopes for logged messages. The `Logger` service in the following example is an `ILogger<CustomLoggerExample>`, which is injected into the app's `CustomLoggerExample` component (`CustomLoggerExample.razor`).

C#

```
using (Logger.BeginScope("L1"))
{
    Logger.LogInformation(3, "INFO: ONE scope.");
}

using (Logger.BeginScope("L1"))
{
    using (Logger.BeginScope("L2"))
    {
        Logger.LogInformation(3, "INFO: TWO scopes.");
    }
}

using (Logger.BeginScope("L1"))
{
    using (Logger.BeginScope("L2"))
    {
        using (Logger.BeginScope("L3"))
        {
            Logger.LogInformation(3, "INFO: THREE scopes.");
        }
```

```
    }
}
```

Output:

```
[ 3: Information ] {CLASS} - INFO: ONE scope. => L1 blazor.webassembly.js:1:35542
[ 3: Information ] {CLASS} - INFO: TWO scopes. => L1 => L2
blazor.webassembly.js:1:35542
[ 3: Information ] {CLASS} - INFO: THREE scopes. => L1 => L2 => L3
```

The `{CLASS}` placeholder in the preceding example is `BlazorWebAssemblyScopesLogger.Pages.CustomLoggerExample`.

Prerendered component logging

Prerendered components execute component initialization code twice. Logging takes place server-side on the first execution of initialization code and client-side on the second execution of initialization code. Depending on the goal of logging during initialization, check logs server-side, client-side, or both.

SignalR client logging with the SignalR client builder

This section applies to server-side apps.

In Blazor script start configuration, pass in the `configureSignalR` configuration object that calls `configureLogging` with the log level.

For the `configureLogging` log level value, pass the argument as either the string or integer log level shown in the following table.

Expand table

LogLevel	String setting	Integer setting
Trace	`trace`	0
Debug	`debug`	1
Information	`information`	2
Warning	`warning`	3

LogLevel	String setting	Integer setting
Error	`error`	4
Critical	`critical`	5
None	`none`	6

Example 1: Set the Information log level with a string value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.configureLogging("information");
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.configureLogging("information");
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Example 2: Set the Information log level with an integer value.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
```

```
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.configureLogging(2); // LogLevel.Information
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.configureLogging(2); // LogLevel.Information
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

> ⓘ Note
>
> Using an integer to specify the logging level in Example 2, often referred to as a *magic number* or *magic constant*, is considered a poor coding practice because the integer doesn't clearly identify the logging level when viewing the source code. If minimizing the bytes transferred to the browser is a priority, using an integer might be justified (consider removing the comment in such cases).

For more information on Blazor startup (`Blazor.start()`), see ASP.NET Core Blazor startup.

SignalR client logging with app configuration

Set up app settings configuration as described in ASP.NET Core Blazor configuration. Place app settings files in `wwwroot` that contain a `Logging:LogLevel:HubConnection` app setting.

> ⓘ Note

As an alternative to using app settings, you can pass the LogLevel as the argument to LoggingBuilderExtensions.SetMinimumLevel when the hub connection is created in a Razor component. However, accidentally deploying the app to a production hosting environment with verbose logging may result in a performance penalty. We recommend using app settings to set the log level.

Provide a `Logging:LogLevel:HubConnection` app setting in the default `appsettings.json` file and in the `Development` environment app settings file. Use a typical less-verbose log level for the default, such as LogLevel.Warning. The default app settings value is what is used in `Staging` and `Production` environments if no app settings files for those environments are present. Use a verbose log level in the `Development` environment app settings file, such as LogLevel.Trace.

`wwwroot/appsettings.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "HubConnection": "Warning"
    }
  }
}
```

`wwwroot/appsettings.Development.json`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "HubConnection": "Trace"
    }
  }
}
```

ⓘ Important

Configuration in the preceding app settings files is only used by the app if the guidance in ASP.NET Core Blazor configuration is followed.

At the top of the Razor component file (`.razor`):

- Inject an ILoggerProvider to add a `WebAssemblyConsoleLogger` to the logging providers passed to HubConnectionBuilder. Unlike a ConsoleLoggerProvider, `WebAssemblyConsoleLogger` is a wrapper around browser-specific logging APIs (for example, `console.log`). Use of `WebAssemblyConsoleLogger` makes logging possible within Mono inside a browser context.
- Inject an `IConfiguration` to read the `Logging:LogLevel:HubConnection` app setting.

> ⓘ Note
>
> `WebAssemblyConsoleLogger` is internal and not supported for direct use in developer code.

C#

```
@inject ILoggerProvider LoggerProvider
@inject IConfiguration Config
```

> ⓘ Note
>
> The following example is based on the demonstration in the SignalR with Blazor tutorial. Consult the tutorial for further details.

In the component's OnInitializedAsync method, use HubConnectionBuilderExtensions.ConfigureLogging to add the logging provider and set the minimum log level from configuration:

C#

```
protected override async Task OnInitializedAsync()
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
        .ConfigureLogging(builder =>
        {
            builder.AddProvider(LoggerProvider);
            builder.SetMinimumLevel(
                Config.GetValue<LogLevel>
("Logging:LogLevel:HubConnection"));
```

```
        })
        .Build();

    hubConnection.On<string, string>("ReceiveMessage", (user, message) =>
...

    await hubConnection.StartAsync();
}
```

> **Note**
>
> In the preceding example, `Navigation` is an injected NavigationManager.

For more information on setting the app's environment, see ASP.NET Core Blazor environments.

Client-side authentication logging

Log Blazor authentication messages at the LogLevel.Debug or LogLevel.Trace logging levels with a logging configuration in app settings or by using a log filter for Microsoft.AspNetCore.Components.WebAssembly.Authentication in the `Program` file.

Use ***either*** of the following approaches:

- In an app settings file (for example, `wwwroot/appsettings.Development.json`):

  JSON

  ```
  "Logging": {
    "LogLevel": {
      "Microsoft.AspNetCore.Components.WebAssembly.Authentication":
  "Debug"
    }
  }
  ```

  For more information on how to configure a client-side app to read app settings files, see ASP.NET Core Blazor configuration.

- Using a log filter, the following example:
  - Activates logging for the `Debug` build configuration using a C# preprocessor directive.
  - Logs Blazor authentication messages at the Debug log level.

  C#

```
#if DEBUG
    builder.Logging.AddFilter(
        "Microsoft.AspNetCore.Components.WebAssembly.Authentication",
        LogLevel.Debug);
#endif
```

> **Note**
>
> Razor components rendered on the client only log to the client-side browser developer tools console.

Additional resources

- Logging in .NET Core and ASP.NET Core
- Loglevel Enum (API documentation)
- Implement a custom logging provider in .NET
- Browser developer tools documentation:
  - Chrome DevTools
  - Microsoft Edge Developer Tools overview
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Handle errors in ASP.NET Core Blazor apps

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes how Blazor manages unhandled exceptions and how to develop apps that detect and handle errors.

Detailed errors during development

When a Blazor app isn't functioning properly during development, receiving detailed error information from the app assists in troubleshooting and fixing the issue. When an error occurs, Blazor apps display a light yellow bar at the bottom of the screen:

- During development, the bar directs you to the browser console, where you can see the exception.
- In production, the bar notifies the user that an error has occurred and recommends refreshing the browser.

The UI for this error handling experience is part of the Blazor project templates. Not all versions of the Blazor project templates use the data-nosnippet attribute to signal to browsers not to cache the contents of the error UI, but all versions of the Blazor documentation apply the attribute.

In a Blazor Web App, customize the experience in the `MainLayout` component. Because the Environment Tag Helper (for example, `<environment include="Production">...</environment>`) isn't supported in Razor components, the following example injects IHostEnvironment to configure error messages for different environments.

At the top of `MainLayout.razor`:

```razor
@inject IHostEnvironment HostEnvironment
```

Create or modify the Blazor error UI markup:

razor

```razor
<div id="blazor-error-ui" data-nosnippet>
    @if (HostEnvironment.IsProduction())
    {
        <span>An error has occurred.</span>
    }
    else
    {
        <span>An unhandled exception occurred.</span>
    }
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
</div>
```

In a Blazor WebAssembly app, customize the experience in the `wwwroot/index.html` file:

HTML

```html
<div id="blazor-error-ui" data-nosnippet>
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
</div>
```

The `blazor-error-ui` element is normally hidden due to the presence of the `display: none` style of the `blazor-error-ui` CSS class in the app's auto-generated stylesheet. When an error occurs, the framework applies `display: block` to the element.

Detailed circuit errors

This section applies to Blazor Web Apps operating over a circuit.

Client-side errors don't include the call stack and don't provide detail on the cause of the error, but server logs do contain such information. For development purposes, sensitive circuit error information can be made available to the client by enabling detailed errors.

Set CircuitOptions.DetailedErrors to `true`. For more information and an example, see ASP.NET Core Blazor SignalR guidance.

An alternative to setting CircuitOptions.DetailedErrors is to set the `DetailedErrors` configuration key to `true` in the app's `Development` environment settings file

(`appsettings.Development.json`). Additionally, set SignalR server-side logging (`Microsoft.AspNetCore.SignalR`) to Debug or Trace for detailed SignalR logging.

`appsettings.Development.json`:

JSON

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.AspNetCore.SignalR": "Debug"
    }
  }
}
```

The DetailedErrors configuration key can also be set to `true` using the `ASPNETCORE_DETAILEDERRORS` environment variable with a value of `true` on `Development`/`Staging` environment servers or on your local system.

> ⚠ **Warning**
>
> Always avoid exposing error information to clients on the Internet, which is a security risk.

Detailed errors for Razor component server-side rendering

This section applies to Blazor Web Apps.

Use the RazorComponentsServiceOptions.DetailedErrors option to control producing detailed information on errors for Razor component server-side rendering. The default value is `false`.

The following example enables detailed errors:

C#

```
builder.Services.AddRazorComponents(options =>
    options.DetailedErrors = builder.Environment.IsDevelopment());
```

> ⚠ **Warning**
>
> **Only enable detailed errors in the `Development` environment.** Detailed errors may contain sensitive information about the app that malicious users can use in an attack.
>
> The preceding example provides a degree of safety by setting the value of DetailedErrors based on the value returned by IsDevelopment. When the app is in the `Development` environment, DetailedErrors is set to `true`. This approach isn't foolproof because it's possible to host a production app on a public server in the `Development` environment.

Manage unhandled exceptions in developer code

For an app to continue after an error, the app must have error handling logic. Later sections of this article describe potential sources of unhandled exceptions.

In production, don't render framework exception messages or stack traces in the UI. Rendering exception messages or stack traces could:

- Disclose sensitive information to end users.
- Help a malicious user discover weaknesses in an app that can compromise the security of the app, server, or network.

Unhandled exceptions for circuits

This section applies to server-side apps operating over a circuit.

Razor components with server interactivity enabled are stateful on the server. While users interact with the component on the server, they maintain a connection to the server known as a *circuit*. The circuit holds active component instances, plus many other aspects of state, such as:

- The most recent rendered output of components.
- The current set of event-handling delegates that could be triggered by client-side events.

If a user opens the app in multiple browser tabs, the user creates multiple independent circuits.

Blazor treats most unhandled exceptions as fatal to the circuit where they occur. If a circuit is terminated due to an unhandled exception, the user can only continue to interact with the app by reloading the page to create a new circuit. Circuits outside of the one that's terminated, which are circuits for other users or other browser tabs, aren't affected. This scenario is similar to a desktop app that crashes. The crashed app must be restarted, but other apps aren't affected.

The framework terminates a circuit when an unhandled exception occurs for the following reasons:

- An unhandled exception often leaves the circuit in an undefined state.
- The app's normal operation can't be guaranteed after an unhandled exception.
- Security vulnerabilities may appear in the app if the circuit continues in an undefined state.

Global exception handling

For approaches to handling exceptions globally, see the following sections:

- Error boundaries: Applies to all Blazor apps.
- Alternative global exception handling: Applies to Blazor Server, Blazor WebAssembly, and Blazor Web Apps (8.0 or later) that adopt a global interactive render mode.

Error boundaries

Error boundaries provide a convenient approach for handling exceptions. The ErrorBoundary component:

- Renders its child content when an error hasn't occurred.
- Renders error UI when an unhandled exception is thrown by any component within the error boundary.

To define an error boundary, use the ErrorBoundary component to wrap one or more other components. The error boundary manages unhandled exceptions thrown by the components that it wraps.

razor

```
<ErrorBoundary>
    ...
</ErrorBoundary>
```

To implement an error boundary in a global fashion, add the boundary around the body content of the app's main layout.

In `MainLayout.razor`:

razor

```razor
<article class="content px-4">
    <ErrorBoundary>
        @Body
    </ErrorBoundary>
</article>
```

In Blazor Web Apps with the error boundary only applied to a static `MainLayout` component, the boundary is only active during static server-side rendering (static SSR). The boundary doesn't activate just because a component further down the component hierarchy is interactive.

An interactive render mode can't be applied to the `MainLayout` component because the component's `Body` parameter is a RenderFragment delegate, which is arbitrary code and can't be serialized. To enable interactivity broadly for the `MainLayout` component and the rest of the components further down the component hierarchy, the app must adopt a global interactive render mode by applying the interactive render mode to the `HeadOutlet` and `Routes` component instances in the app's root component, which is typically the `App` component. The following example adopts the Interactive Server (`InteractiveServer`) render mode globally.

In `Components/App.razor`:

razor

```razor
<HeadOutlet @rendermode="InteractiveServer" />

...

<Routes @rendermode="InteractiveServer" />
```

If you prefer not to enable global interactivity, place the error boundary farther down the component hierarchy. The important concepts to keep in mind are that wherever the error boundary is placed:

- If the component where the error boundary is placed isn't interactive, the error boundary is only capable of activating on the server during static SSR. For example, the boundary can activate when an error is thrown in a component lifecycle

method but not for an event triggered by user interactivity within the component, such as an error thrown by a button click handler.
- If the component where the error boundary is placed is interactive, the error boundary is capable of activating for interactive components that it wraps.

> **Note**
>
> The preceding considerations aren't relevant for standalone Blazor WebAssembly apps because the client-side rendering (CSR) of a Blazor WebAssembly app is completely interactive.

Consider the following example, where an exception thrown by an embedded counter component is caught by an error boundary in the `Home` component, which adopts an interactive render mode.

`EmbeddedCounter.razor`:

razor

```razor
<h1>Embedded Counter</h1>

<p role="status">Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;

        if (currentCount > 5)
        {
            throw new InvalidOperationException("Current count is too
big!");
        }
    }
}
```

`Home.razor`:

razor

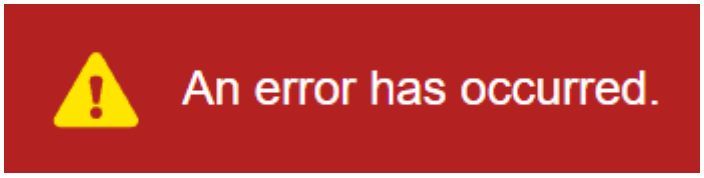
```razor
@page "/"
@rendermode InteractiveServer

<PageTitle>Home</PageTitle>
```

```
<h1>Home</h1>

<ErrorBoundary>
    <EmbeddedCounter />
</ErrorBoundary>
```

If the unhandled exception is thrown for a `currentCount` over five:

- The error is logged normally (`System.InvalidOperationException: Current count is too big!`).
- The exception is handled by the error boundary.
- The default error UI is rendered by the error boundary.

The ErrorBoundary component renders an empty `<div>` element using the `blazor-error-boundary` CSS class for its error content. The colors, text, and icon for the default UI are defined in the app's stylesheet in the `wwwroot` folder, so you're free to customize the error UI.



To change the default error content:

- Wrap the components of the error boundary in the ChildContent property.
- Set the ErrorContent property to the error content.

The following example wraps the `EmbeddedCounter` component and supplies custom error content:

razor

```
<ErrorBoundary>
    <ChildContent>
        <EmbeddedCounter />
    </ChildContent>
    <ErrorContent>
        <p class="errorUI">😈 A rotten gremlin got us. Sorry!</p>
    </ErrorContent>
</ErrorBoundary>
```

For the preceding example, the app's stylesheet presumably includes an `errorUI` CSS class to style the content. The error content is rendered from the ErrorContent property

without a block-level element. A block-level element, such as a division (`<div>`) or a paragraph (`<p>`) element, can wrap the error content markup, but it isn't required.

Optionally, use the context (`@context`) of the ErrorContent to obtain error data:

razor

```
<ErrorContent>
    @context.HelpLink
</ErrorContent>
```

The ErrorContent can also name the context. In the following example, the context is named `exception`:

razor

```
<ErrorContent Context="exception">
    @exception.HelpLink
</ErrorContent>
```

> ⚠ **Warning**
>
> Always avoid exposing error information to clients on the Internet, which is a security risk.

If the error boundary is defined in the app's layout, the error UI is seen regardless of which page the user navigates to after the error occurs. We recommend narrowly scoping error boundaries in most scenarios. If you broadly scope an error boundary, you can reset it to a non-error state on subsequent page navigation events by calling the error boundary's Recover method.

In `MainLayout.razor`:

- Add a field for the ErrorBoundary to capture a reference to it with the @ref attribute directive.
- In the OnParameterSet lifecycle method, you can trigger a recovery on the error boundary with Recover to clear the error when the user navigates to a different component.

razor

```
...

<ErrorBoundary @ref="errorBoundary">
```

```
    @Body
</ErrorBoundary>

...

@code {
    private ErrorBoundary? errorBoundary;

    protected override void OnParametersSet()
    {
        errorBoundary?.Recover();
    }
}
```

To avoid the infinite loop where recovering merely rerenders a component that throws the error again, don't call Recover from rendering logic. Only call Recover when:

- The user performs a UI gesture, such as selecting a button to indicate that they want to retry a procedure or when the user navigates to a new component.
- Additional logic that executes also clears the exception. When the component is rerendered, the error doesn't reoccur.

The following example permits the user to recover from the exception with a button:

razor

```
<ErrorBoundary @ref="errorBoundary">
    <ChildContent>
        <EmbeddedCounter />
    </ChildContent>
    <ErrorContent>
        <div class="alert alert-danger" role="alert">
            <p class="fs-3 fw-bold">😈 A rotten gremlin got us. Sorry!</p>
            <p>@context.HelpLink</p>
            <button class="btn btn-info" @onclick="_ =>
errorBoundary?.Recover()">
                Clear
            </button>
        </div>
    </ErrorContent>
</ErrorBoundary>

@code {
    private ErrorBoundary? errorBoundary;
}
```

You can also subclass ErrorBoundary for custom processing by overriding OnErrorAsync. The following example merely logs the error, but you can implement any error handling

code you wish. You can remove the line that returns a CompletedTask if your code awaits an asynchronous task.

`CustomErrorBoundary.razor`:

razor

```razor
@inherits ErrorBoundary
@inject ILogger<CustomErrorBoundary> Logger

@if (CurrentException is null)
{
    @ChildContent
}
else if (ErrorContent is not null)
{
    @ErrorContent(CurrentException)
}

@code {
    protected override Task OnErrorAsync(Exception ex)
    {
        Logger.LogError(ex, "😈 A rotten gremlin got us. Sorry!");
        return Task.CompletedTask;
    }
}
```

The preceding example can also be implemented as a class.

`CustomErrorBoundary.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;

namespace BlazorSample;

public class CustomErrorBoundary : ErrorBoundary
{
    [Inject]
    ILogger<CustomErrorBoundary> Logger {  get; set; } = default!;

    protected override Task OnErrorAsync(Exception ex)
    {
        Logger.LogError(ex, "😈 A rotten gremlin got us. Sorry!");
        return Task.CompletedTask;
    }
}
```

Either of the preceding implementations used in a component:

razor

```
<CustomErrorBoundary>
    ...
</CustomErrorBoundary>
```

Alternative global exception handling

The approach described in this section applies to Blazor Server, Blazor WebAssembly, and Blazor Web Apps that adopt a global interactive render mode (`InteractiveServer`, `InteractiveWebAssembly`, or `InteractiveAuto`). The approach doesn't work with Blazor Web Apps that adopt per-page/component render modes or static server-side rendering (static SSR) because the approach relies on a CascadingValue/CascadingParameter, which don't work across render mode boundaries or with components that adopt static SSR.

An alternative to using Error boundaries (ErrorBoundary) is to pass a custom error component as a CascadingValue to child components. An advantage of using a component over using an injected service or a custom logger implementation is that a cascaded component can render content and apply CSS styles when an error occurs.

The following `ProcessError` component example merely logs errors, but methods of the component can process errors in any way required by the app, including through the use of multiple error processing methods.

`ProcessError.razor`:

razor

```
@inject ILogger<ProcessError> Logger

<CascadingValue Value="this">
    @ChildContent
</CascadingValue>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    public void LogError(Exception ex)
    {
        Logger.LogError("ProcessError.LogError: {Type} Message: {Message}",
            ex.GetType(), ex.Message);
```

```
        // Call StateHasChanged if LogError directly participates in
        // rendering. If LogError only logs or records the error,
        // there's no need to call StateHasChanged.
        //StateHasChanged();
    }
}
```

> **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

When using this approach in a Blazor Web App, open the `Routes` component and wrap the Router component (`<Router>...</Router>`) with the `ProcessError` component. This permits the `ProcessError` component to cascade down to any component of the app where the `ProcessError` component is received as a CascadingParameter.

In `Routes.razor`:

razor

```
<ProcessError>
    <Router ...>
        ...
    </Router>
</ProcessError>
```

When using this approach in a Blazor Server or Blazor WebAssembly app, open the `App` component, wrap the Router component (`<Router>...</Router>`) with the `ProcessError` component. This permits the `ProcessError` component to cascade down to any component of the app where the `ProcessError` component is received as a CascadingParameter.

In `App.razor`:

razor

```
<ProcessError>
    <Router ...>
        ...
    </Router>
</ProcessError>
```

To process errors in a component:

- Designate the `ProcessError` component as a CascadingParameter in the `@code` block. In an example `Counter` component in an app based on a Blazor project template, add the following `ProcessError` property:

  C#

  ```
  [CascadingParameter]
  public ProcessError? ProcessError { get; set; }
  ```

- Call an error processing method in any `catch` block with an appropriate exception type. The example `ProcessError` component only offers a single `LogError` method, but the error processing component can provide any number of error processing methods to address alternative error processing requirements throughout the app. The following `Counter` component `@code` block example includes the `ProcessError` cascading parameter and traps an exception for logging when the count is greater than five:

  razor

  ```
  @code {
      private int currentCount = 0;

      [CascadingParameter]
      public ProcessError? ProcessError { get; set; }

      private void IncrementCount()
      {
          try
          {
              currentCount++;

              if (currentCount > 5)
              {
                  throw new InvalidOperationException("Current count is
  over five!");
              }
          }
          catch (Exception ex)
          {
              ProcessError?.LogError(ex);
          }
      }
  }
  ```

The logged error:

> fail: {COMPONENT NAMESPACE}.ProcessError[0]
> ProcessError.LogError: System.InvalidOperationException Message: Current count is over five!

If the `LogError` method directly participates in rendering, such as showing a custom error message bar or changing the CSS styles of the rendered elements, call StateHasChanged at the end of the `LogError` method to rerender the UI.

Because the approaches in this section handle errors with a try-catch statement, an app's SignalR connection between the client and server isn't broken when an error occurs and the circuit remains alive. Other unhandled exceptions remain fatal to a circuit. For more information, see the section on how a circuit reacts to unhandled exceptions.

Log errors with a persistent provider

If an unhandled exception occurs, the exception is logged to ILogger instances configured in the service container. Blazor apps log console output with the Console Logging Provider. Consider logging to a location on the server (or backend web API for client-side apps) with a provider that manages log size and log rotation. Alternatively, the app can use an Application Performance Management (APM) service, such as Azure Application Insights (Azure Monitor).

> **Note**
>
> Native Application Insights features to support client-side apps and native Blazor framework support for Google Analytics might become available in future releases of these technologies. For more information, see Support App Insights in Blazor WASM Client Side (microsoft/ApplicationInsights-dotnet #2143) and Web analytics and diagnostics (includes links to community implementations) (dotnet/aspnetcore #5461). In the meantime, a client-side app can use the Application Insights JavaScript SDK with JS interop to log errors directly to Application Insights from a client-side app.

During development in a Blazor app operating over a circuit, the app usually sends the full details of exceptions to the browser's console to aid in debugging. In production, detailed errors aren't sent to clients, but an exception's full details are logged on the server.

You must decide which incidents to log and the level of severity of logged incidents. Hostile users might be able to trigger errors deliberately. For example, don't log an incident from an error where an unknown `ProductId` is supplied in the URL of a component that displays product details. Not all errors should be treated as incidents for logging.

For more information, see the following articles:

- ASP.NET Core Blazor logging
- Handle errors in ASP.NET Core‡
- Create web APIs with ASP.NET Core

‡Applies to server-side Blazor apps and other server-side ASP.NET Core apps that are web API backend apps for Blazor. Client-side apps can trap and send error information on the client to a web API, which logs the error information to a persistent logging provider.

Places where errors may occur

Framework and app code may trigger unhandled exceptions in any of the following locations, which are described further in the following sections of this article:

- Component instantiation
- Lifecycle methods
- Rendering logic
- Event handlers
- Component disposal
- JavaScript interop
- Prerendering

Component instantiation

When Blazor creates an instance of a component:

- The component's constructor is invoked.
- The constructors of DI services supplied to the component's constructor via the @inject directive or the [Inject] attribute are invoked.

An error in an executed constructor or a setter for any `[Inject]` property results in an unhandled exception and stops the framework from instantiating the component. If the app is operating over a circuit, the circuit fails. If constructor logic may throw exceptions,

the app should trap the exceptions using a try-catch statement with error handling and logging.

Lifecycle methods

During the lifetime of a component, Blazor invokes lifecycle methods. If any lifecycle method throws an exception, synchronously or asynchronously, the exception is fatal to a circuit. For components to deal with errors in lifecycle methods, add error handling logic.

In the following example where OnParametersSetAsync calls a method to obtain a product:

- An exception thrown in the `ProductRepository.GetProductByIdAsync` method is handled by a try-catch statement.
- When the `catch` block is executed:
  - `loadFailed` is set to `true`, which is used to display an error message to the user.
  - The error is logged.

razor

```
@page "/product-details/{ProductId:int?}"
@inject ILogger<ProductDetails> Logger
@inject IProductRepository Product

<PageTitle>Product Details</PageTitle>

<h1>Product Details Example</h1>

@if (details != null)
{
    <h2>@details.ProductName</h2>
    <p>
        @details.Description
        <a href="@details.Url">Company Link</a>
    </p>

}
else if (loadFailed)
{
    <h1>Sorry, we could not load this product due to an error.</h1>
}
else
{
    <h1>Loading...</h1>
}

@code {
    private ProductDetail? details;
```

```
    private bool loadFailed;

    [Parameter]
    public int ProductId { get; set; }

    protected override async Task OnParametersSetAsync()
    {
        try
        {
            loadFailed = false;

            // Reset details to null to display the loading indicator
            details = null;

            details = await Product.GetProductByIdAsync(ProductId);
        }
        catch (Exception ex)
        {
            loadFailed = true;
            Logger.LogWarning(ex, "Failed to load product {ProductId}",
ProductId);
        }
    }

    public class ProductDetail
    {
        public string? ProductName { get; set; }
        public string? Description { get; set; }
        public string? Url { get; set; }
    }

    /*
    * Register the service in Program.cs:
    * using static BlazorSample.Components.Pages.ProductDetails;
    * builder.Services.AddScoped<IProductRepository, ProductRepository>();
    */

    public interface IProductRepository
    {
        public Task<ProductDetail> GetProductByIdAsync(int id);
    }

    public class ProductRepository : IProductRepository
    {
        public Task<ProductDetail> GetProductByIdAsync(int id)
        {
            return Task.FromResult(
                new ProductDetail()
                {
                    ProductName = "Flowbee ",
                    Description = "The Revolutionary Haircutting System
You've Come to Love!",
                    Url = "https://flowbee.com/"
                });
        }
```

```
    }
}
```

Rendering logic

The declarative markup in a Razor component file (`.razor`) is compiled into a C# method called BuildRenderTree. When a component renders, BuildRenderTree executes and builds up a data structure describing the elements, text, and child components of the rendered component.

Rendering logic can throw an exception. An example of this scenario occurs when `@someObject.PropertyName` is evaluated but `@someObject` is `null`. For Blazor apps operating over a circuit, an unhandled exception thrown by rendering logic is fatal to the app's circuit.

To prevent a NullReferenceException in rendering logic, check for a `null` object before accessing its members. In the following example, `person.Address` properties aren't accessed if `person.Address` is `null`:

razor

```
@if (person.Address != null)
{
    <div>@person.Address.Line1</div>
    <div>@person.Address.Line2</div>
    <div>@person.Address.City</div>
    <div>@person.Address.Country</div>
}
```

The preceding code assumes that `person` isn't `null`. Often, the structure of the code guarantees that an object exists at the time the component is rendered. In those cases, it isn't necessary to check for `null` in rendering logic. In the prior example, `person` might be guaranteed to exist because `person` is created when the component is instantiated, as the following example shows:

razor

```
@code {
    private Person person = new();

    ...
}
```

Event handlers

Client-side code triggers invocations of C# code when event handlers are created using:

- `@onclick`
- `@onchange`
- Other `@on...` attributes
- `@bind`

Event handler code might throw an unhandled exception in these scenarios.

If the app calls code that could fail for external reasons, trap exceptions using a try-catch statement with error handling and logging.

If an event handler throws an unhandled exception (for example, a database query fails) that isn't trapped and handled by developer code:

- The framework logs the exception.
- In a Blazor app operating over a circuit, the exception is fatal to the app's circuit.

Component disposal

A component may be removed from the UI, for example, because the user has navigated to another page. When a component that implements System.IDisposable is removed from the UI, the framework calls the component's Dispose method.

If the component's `Dispose` method throws an unhandled exception in a Blazor app operating over a circuit, the exception is fatal to the app's circuit.

If disposal logic may throw exceptions, the app should trap the exceptions using a try-catch statement with error handling and logging.

For more information on component disposal, see ASP.NET Core Razor component lifecycle.

JavaScript interop

IJSRuntime is registered by the Blazor framework. IJSRuntime.InvokeAsync allows .NET code to make asynchronous calls to the JavaScript (JS) runtime in the user's browser.

The following conditions apply to error handling with InvokeAsync:

- If a call to InvokeAsync fails synchronously, a .NET exception occurs. A call to InvokeAsync may fail, for example, because the supplied arguments can't be

serialized. Developer code must catch the exception. If app code in an event handler or component lifecycle method doesn't handle an exception in a Blazor app operating over a circuit, the resulting exception is fatal to the app's circuit.

- If a call to InvokeAsync fails asynchronously, the .NET Task fails. A call to InvokeAsync may fail, for example, because the JS-side code throws an exception or returns a `Promise` that completed as `rejected`. Developer code must catch the exception. If using the await operator, consider wrapping the method call in a try-catch statement with error handling and logging. Otherwise in a Blazor app operating over a circuit, the failing code results in an unhandled exception that's fatal to the app's circuit.
- Calls to InvokeAsync must complete within a certain period or else the call times out. The default timeout period is one minute. The timeout protects the code against a loss in network connectivity or JS code that never sends back a completion message. If the call times out, the resulting System.Threading.Tasks fails with an OperationCanceledException. Trap and process the exception with logging.

Similarly, JS code may initiate calls to .NET methods indicated by the [JSInvokable] attribute. If these .NET methods throw an unhandled exception:

- In a Blazor app operating over a circuit, the exception isn't treated as fatal to the app's circuit.
- The JS-side `Promise` is rejected.

You have the option of using error handling code on either the .NET side or the JS side of the method call.

For more information, see the following articles:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Prerendering

Razor components are prerendered by default so that their rendered HTML markup is returned as part of the user's initial HTTP request.

In a Blazor app operating over a circuit, prerendering works by:

- Creating a new circuit for all of the prerendered components that are part of the same page.
- Generating the initial HTML.

- Treating the circuit as `disconnected` until the user's browser establishes a SignalR connection back to the same server. When the connection is established, interactivity on the circuit is resumed and the components' HTML markup is updated.

For prerendered client-side components, prerendering works by:

- Generating initial HTML on the server for all of the prerendered components that are part of the same page.
- Making the component interactive on the client after the browser has loaded the app's compiled code and the .NET runtime (if not already loaded) in the background.

If a component throws an unhandled exception during prerendering, for example, during a lifecycle method or in rendering logic:

- In a Blazor app operating over a circuit, the exception is fatal to the circuit. For prerendered client-side components, the exception prevents rendering the component.
- The exception is thrown up the call stack from the ComponentTagHelper.

Under normal circumstances when prerendering fails, continuing to build and render the component doesn't make sense because a working component can't be rendered.

To tolerate errors that may occur during prerendering, error handling logic must be placed inside a component that may throw exceptions. Use try-catch statements with error handling and logging. Instead of wrapping the ComponentTagHelper in a try-catch statement, place error handling logic in the component rendered by the ComponentTagHelper.

Advanced scenarios

Recursive rendering

Components can be nested recursively. This is useful for representing recursive data structures. For example, a `TreeNode` component can render more `TreeNode` components for each of the node's children.

When rendering recursively, avoid coding patterns that result in infinite recursion:

- Don't recursively render a data structure that contains a cycle. For example, don't render a tree node whose children includes itself.

- Don't create a chain of layouts that contain a cycle. For example, don't create a layout whose layout is itself.
- Don't allow an end user to violate recursion invariants (rules) through malicious data entry or JavaScript interop calls.

Infinite loops during rendering:

- Causes the rendering process to continue forever.
- Is equivalent to creating an unterminated loop.

In these scenarios, the Blazor fails and usually attempts to:

- Consume as much CPU time as permitted by the operating system, indefinitely.
- Consume an unlimited amount of memory. Consuming unlimited memory is equivalent to the scenario where an unterminated loop adds entries to a collection on every iteration.

To avoid infinite recursion patterns, ensure that recursive rendering code contains suitable stopping conditions.

Custom render tree logic

Most Razor components are implemented as Razor component files (`.razor`) and are compiled by the framework to produce logic that operates on a RenderTreeBuilder to render their output. However, a developer may manually implement RenderTreeBuilder logic using procedural C# code. For more information, see ASP.NET Core Blazor advanced scenarios (render tree construction).

> ⚠ **Warning**
>
> Use of manual render tree builder logic is considered an advanced and unsafe scenario, not recommended for general component development.

If RenderTreeBuilder code is written, the developer must guarantee the correctness of the code. For example, the developer must ensure that:

- Calls to OpenElement and CloseElement are correctly balanced.
- Attributes are only added in the correct places.

Incorrect manual render tree builder logic can cause arbitrary undefined behavior, including crashes, the app or server to stop responding, and security vulnerabilities.

Consider manual render tree builder logic on the same level of complexity and with the same level of *danger* as writing assembly code or Microsoft Intermediate Language (MSIL) instructions by hand.

Additional resources

- Handle caught exceptions outside of a Razor component's lifecycle
- ASP.NET Core Blazor logging
- Handle errors in ASP.NET Core†
- Create web APIs with ASP.NET Core
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

†Applies to backend ASP.NET Core web API apps that client-side Blazor apps use for logging.

ASP.NET Core Blazor SignalR guidance

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure and manage SignalR connections in Blazor apps.

For general guidance on ASP.NET Core SignalR configuration, see the topics in the Overview of ASP.NET Core SignalR area of the documentation, especially ASP.NET Core SignalR configuration.

Server-side apps use ASP.NET Core SignalR to communicate with the browser. SignalR's hosting and scaling conditions apply to server-side apps.

Blazor works best when using WebSockets as the SignalR transport due to lower latency, reliability, and security. Long Polling is used by SignalR when WebSockets isn't available or when the app is explicitly configured to use Long Polling.

Azure SignalR Service with stateful reconnect

The Azure SignalR Service with SDK v1.26.1 or later supports SignalR stateful reconnect (WithStatefulReconnect).

Disable response compression for Hot Reload

When using Hot Reload, disable Response Compression Middleware in the `Development` environment. Whether or not the default code from a project template is used, always call UseResponseCompression first in the request processing pipeline.

In the `Program` file:

C#

```
if (!app.Environment.IsDevelopment())
{
    app.UseResponseCompression();
}
```

Client-side SignalR cross-origin negotiation for authentication

This section explains how to configure SignalR's underlying client to send credentials, such as cookies or HTTP authentication headers.

Use SetBrowserRequestCredentials to set Include on cross-origin fetch requests.

`IncludeRequestCredentialsMessageHandler.cs`:

C#

```
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components.WebAssembly.Http;

public class IncludeRequestCredentialsMessageHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {

request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        return base.SendAsync(request, cancellationToken);
    }
}
```

Where a hub connection is built, assign the HttpMessageHandler to the HttpMessageHandlerFactory option:

C#

```
private HubConnectionBuilder? hubConnection;

...

hubConnection = new HubConnectionBuilder()
    .WithUrl(new Uri(Navigation.ToAbsoluteUri("/chathub")), options =>
    {
        options.HttpMessageHandlerFactory = innerHandler =>
            new IncludeRequestCredentialsMessageHandler { InnerHandler =
innerHandler };
    }).Build();
```

The preceding example configures the hub connection URL to the absolute URI address at `/chathub`. The URI can also be set via a string, for example

`https://signalr.example.com`, or via configuration. `Navigation` is an injected NavigationManager.

For more information, see ASP.NET Core SignalR configuration.

Client-side rendering

If prerendering is configured, prerendering occurs before the client connection to the server is established. For more information, see Prerender ASP.NET Core Razor components.

Prerendered state size and SignalR message size limit

A large prerendered state size may exceed Blazor's SignalR circuit message size limit, which results in the following:

- The SignalR circuit fails to initialize with an error on the client: Circuit host not initialized.
- The reconnection UI on the client appears when the circuit fails. Recovery isn't possible.

To resolve the problem, use ***either*** of the following approaches:

- Reduce the amount of data that you are putting into the prerendered state.
- Increase the SignalR message size limit. ***WARNING***: Increasing the limit may increase the risk of Denial of Service (DoS) attacks.

Additional client-side resources

- Secure a SignalR hub
- Overview of ASP.NET Core SignalR
- ASP.NET Core SignalR configuration
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Use session affinity (sticky sessions) for server-side webfarm hosting

When more than one backend server is in use, the app must implement session affinity, also called *sticky sessions*. Session affinity ensures that a client's circuit reconnects to the

same server if the connection is dropped, which is important because client state is only held in the memory of the server that first established the client's circuit.

The following error is thrown by an app that hasn't enabled session affinity in a webfarm:

> Uncaught (in promise) Error: Invocation canceled due to the underlying connection being closed.

For more information on session affinity with Azure App Service hosting, see Host and deploy ASP.NET Core server-side Blazor apps.

Azure SignalR Service

The optional Azure SignalR Service works in conjunction with the app's SignalR hub for scaling up a server-side app to a large number of concurrent connections. In addition, the service's global reach and high-performance data centers significantly aid in reducing latency due to geography.

The service isn't required for Blazor apps hosted in Azure App Service or Azure Container Apps but can be helpful in other hosting environments:

- To facilitate connection scale out.
- Handle global distribution.

For more information, see Host and deploy ASP.NET Core server-side Blazor apps.

Server-side circuit handler options

Configure the circuit with CircuitOptions. View default values in the reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Read or set the options in the `Program` file with an options delegate to AddInteractiveServerComponents. The `{OPTION}` placeholder represents the option, and

the `{VALUE}` placeholder is the value.

In the `Program` file:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents(options
=>
{
    options.{OPTION} = {VALUE};
});
```

To configure the HubConnectionContext, use HubConnectionContextOptions with AddHubOptions. View the defaults for the hub connection context options in reference source. For option descriptions in the SignalR documentation, see ASP.NET Core SignalR configuration. The `{OPTION}` placeholder represents the option, and the `{VALUE}` placeholder is the value.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

In the `Program` file:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents().AddHu
bOptions(options =>
{
    options.{OPTION} = {VALUE};
});
```

> **Warning**
>
> The default value of MaximumReceiveMessageSize is 32 KB. Increasing the value may increase the risk of Denial of Service (DoS) attacks.
>
> Blazor relies on MaximumParallelInvocationsPerClient set to 1, which is the default value. For more information, see MaximumParallelInvocationsPerClient > 1 breaks

file upload in Blazor Server mode (dotnet/aspnetcore #53951).

For information on memory management, see Host and deploy ASP.NET Core server-side Blazor apps.

Blazor hub options

Configure MapBlazorHub options to control HttpConnectionDispatcherOptions of the Blazor hub. View the defaults for the hub connection dispatcher options in reference source. The `{OPTION}` placeholder represents the option, and the `{VALUE}` placeholder is the value.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Place the call to `app.MapBlazorHub` after the call to `app.MapRazorComponents` in the app's `Program` file:

C#

```
app.MapBlazorHub(options =>
{
    options.{OPTION} = {VALUE};
});
```

Configuring the hub used by AddInteractiveServerRenderMode with MapBlazorHub fails with an `AmbiguousMatchException`:

> Microsoft.AspNetCore.Routing.Matching.AmbiguousMatchException: The request matched multiple endpoints.

To workaround the problem for apps targeting .NET 8, give the custom-configured Blazor hub higher precedence using the WithOrder method:

C#

```
app.MapBlazorHub(options =>
{
    options.CloseOnAuthenticationExpiration = true;
}).WithOrder(-1);
```

For more information, see the following resources:

- MapBlazorHub configuration in NET8 throws a The request matched multiple endpoints exception (dotnet/aspnetcore #51698)
- Attempts to map multiple blazor entry points with MapBlazorHub causes Ambiguous Route Error. This worked with net7 (dotnet/aspnetcore #52156)

Maximum receive message size

This section only applies to projects that implement SignalR.

The maximum incoming SignalR message size permitted for hub methods is limited by the HubOptions.MaximumReceiveMessageSize (default: 32 KB). SignalR messages larger than MaximumReceiveMessageSize throw an error. The framework doesn't impose a limit on the size of a SignalR message from the hub to a client.

When SignalR logging isn't set to Debug or Trace, a message size error only appears in the browser's developer tools console:

> Error: Connection disconnected with error 'Error: Server returned an error on close: Connection closed with an error.'.

When SignalR server-side logging is set to Debug or Trace, server-side logging surfaces an InvalidDataException for a message size error.

`appsettings.Development.json`:

JSON

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      ...
      "Microsoft.AspNetCore.SignalR": "Debug"
    }
  }
}
```

Error:

> System.IO.InvalidDataException: The maximum message size of 32768B was exceeded. The message size can be configured in AddHubOptions.

One approach involves increasing the limit by setting MaximumReceiveMessageSize in the `Program` file. The following example sets the maximum receive message size to 64 KB:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents()
    .AddHubOptions(options => options.MaximumReceiveMessageSize = 64 *
1024);
```

Increasing the SignalR incoming message size limit comes at the cost of requiring more server resources, and it increases the risk of Denial of Service (DoS) attacks. Additionally, reading a large amount of content in to memory as strings or byte arrays can also result in allocations that work poorly with the garbage collector, resulting in additional performance penalties.

A better option for reading large payloads is to send the content in smaller chunks and process the payload as a Stream. This can be used when reading large JavaScript (JS) interop JSON payloads or if JS interop data is available as raw bytes. For an example that demonstrates sending large binary payloads in server-side apps that uses techniques similar to the InputFile component, see the Binary Submit sample app and the Blazor InputLargeTextArea Component Sample.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Forms that process large payloads over SignalR can also use streaming JS interop directly. For more information, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor. For a forms example that streams `<textarea>` data to the server, see Troubleshoot ASP.NET Core Blazor forms.

Consider the following guidance when developing code that transfers a large amount of data:

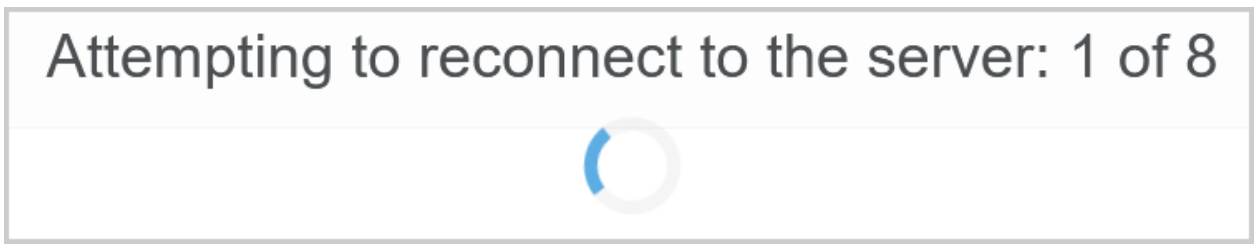
- Leverage the native streaming JS interop support to transfer data larger than the SignalR incoming message size limit:
  - Call JavaScript functions from .NET methods in ASP.NET Core Blazor
  - Call .NET methods from JavaScript functions in ASP.NET Core Blazor
  - Form payload example: Troubleshoot ASP.NET Core Blazor forms
- General tips:
  - Don't allocate large objects in JS and C# code.
  - Free consumed memory when the process is completed or cancelled.
  - Enforce the following additional requirements for security purposes:
    - Declare the maximum file or data size that can be passed.
    - Declare the minimum upload rate from the client to the server.
  - After the data is received by the server, the data can be:
    - Temporarily stored in a memory buffer until all of the segments are collected.
    - Consumed immediately. For example, the data can be stored immediately in a database or written to disk as each segment is received.

Blazor server-side Hub endpoint route configuration

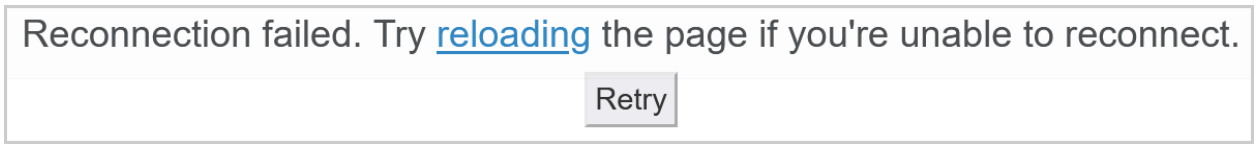
In the `Program` file, call MapBlazorHub to map the Blazor Hub to the app's default path. The Blazor script (`blazor.*.js`) automatically points to the endpoint created by MapBlazorHub.

Reflect the server-side connection state in the UI

If the client detects a lost connection to the server, a default UI is displayed to the user while the client attempts to reconnect:



If reconnection fails, the user is instructed to retry or reload the page:



If reconnection succeeds, user state is often lost. Custom code can be added to any component to save and reload user state across connection failures. For more information, see ASP.NET Core Blazor state management.

To customize the UI, define a single element with an `id` of `components-reconnect-modal` in the `<body>` element content. The following example places the element in the `App` component.

`App.razor`:

CSHTML

```
<div id="components-reconnect-modal">
    Connection lost.<br>Attempting to reconnect...
</div>
```

> **Note**
>
> If more than one element with an `id` of `components-reconnect-modal` are rendered by the app, only the first rendered element receives CSS class changes to display or hide the element.

Add the following CSS styles to the site's stylesheet.

`wwwroot/app.css`:

css

```
#components-reconnect-modal {
    display: none;
}

#components-reconnect-modal.components-reconnect-show,
#components-reconnect-modal.components-reconnect-failed,
#components-reconnect-modal.components-reconnect-rejected {
    display: block;
    background-color: white;
    padding: 2rem;
    border-radius: 0.5rem;
    text-align: center;
    box-shadow: 0 3px 6px 2px rgba(0, 0, 0, 0.3);
    margin: 50px 50px;
```

```
    position: fixed;
    top: 0;
    z-index: 10001;
}
```

The following table describes the CSS classes applied to the `components-reconnect-modal` element by the Blazor framework.

Expand table

CSS class	Indicates...
`components-reconnect-show`	A lost connection. The client is attempting to reconnect. Show the modal.
`components-reconnect-hide`	An active connection is re-established to the server. Hide the modal.
`components-reconnect-failed`	Reconnection failed, probably due to a network failure. To attempt reconnection, call `window.Blazor.reconnect()` in JavaScript.
`components-reconnect-rejected`	Reconnection rejected. The server was reached but refused the connection, and the user's state on the server is lost. To reload the app, call `location.reload()` in JavaScript. This connection state may result when: • A crash in the server-side circuit occurs. • The client is disconnected long enough for the server to drop the user's state. Instances of the user's components are disposed. • The server is restarted, or the app's worker process is recycled.

Customize the delay before the reconnection UI appears by setting the `transition-delay` property in the site's CSS for the modal element. The following example sets the transition delay from 500 ms (default) to 1,000 ms (1 second).

`wwwroot/app.css`:

```
css

#components-reconnect-modal {
    transition: visibility 0s linear 1000ms;
}
```

To display the current reconnect attempt, define an element with an `id` of `components-reconnect-current-attempt`. To display the maximum number of reconnect retries, define an element with an `id` of `components-reconnect-max-retries`. The following example

places these elements inside a reconnect attempt modal element following the previous example.

CSHTML

```
<div id="components-reconnect-modal">
    There was a problem with the connection!
    (Current reconnect attempt:
    <span id="components-reconnect-current-attempt"></span> /
    <span id="components-reconnect-max-retries"></span>)
</div>
```

When the custom reconnect modal appears, it renders the following content with a reconnection attempt counter:

> There was a problem with the connection! (Current reconnect attempt: 1 / 8)

Server-side rendering

By default, components are prerendered on the server before the client connection to the server is established. For more information, see Prerender ASP.NET Core Razor components.

Monitor server-side circuit activity

Monitor inbound circuit activity using the CreateInboundActivityHandler method on CircuitHandler. Inbound circuit activity is any activity sent from the browser to the server, such as UI events or JavaScript-to-.NET interop calls.

For example, you can use a circuit activity handler to detect if the client is idle and log its circuit ID (Circuit.Id):

C#

```
using Microsoft.AspNetCore.Components.Server.Circuits;
using Microsoft.Extensions.Options;
using Timer = System.Timers.Timer;

public sealed class IdleCircuitHandler : CircuitHandler, IDisposable
{
    private Circuit? currentCircuit;
    private readonly ILogger logger;
    private readonly Timer timer;

    public IdleCircuitHandler(ILogger<IdleCircuitHandler> logger,
        IOptions<IdleCircuitOptions> options)
```

```
    {
        timer = new Timer
        {
            Interval = options.Value.IdleTimeout.TotalMilliseconds,
            AutoReset = false
        };

        timer.Elapsed += CircuitIdle;
        this.logger = logger;
    }

    private void CircuitIdle(object? sender, System.Timers.ElapsedEventArgs
e)
    {
        logger.LogInformation("{CircuitId} is idle", currentCircuit?.Id);
    }

    public override Task OnCircuitOpenedAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        currentCircuit = circuit;

        return Task.CompletedTask;
    }

    public override Func<CircuitInboundActivityContext, Task>
CreateInboundActivityHandler(
        Func<CircuitInboundActivityContext, Task> next)
    {
        return context =>
        {
            timer.Stop();
            timer.Start();

            return next(context);
        };
    }

    public void Dispose() => timer.Dispose();
}

public class IdleCircuitOptions
{
    public TimeSpan IdleTimeout { get; set; } = TimeSpan.FromMinutes(5);
}

public static class IdleCircuitHandlerServiceCollectionExtensions
{
    public static IServiceCollection AddIdleCircuitHandler(
        this IServiceCollection services,
        Action<IdleCircuitOptions> configureOptions)
    {
        services.Configure(configureOptions);
        services.AddIdleCircuitHandler();
```

```
        return services;
    }

    public static IServiceCollection AddIdleCircuitHandler(
        this IServiceCollection services)
    {
        services.AddScoped<CircuitHandler, IdleCircuitHandler>();

        return services;
    }
}
```

Register the service in the `Program` file. The following example configures the default idle timeout of five minutes to five seconds in order to test the preceding `IdleCircuitHandler` implementation:

C#

```
builder.Services.AddIdleCircuitHandler(options =>
    options.IdleTimeout = TimeSpan.FromSeconds(5));
```

Circuit activity handlers also provide an approach for accessing scoped Blazor services from other non-Blazor dependency injection (DI) scopes. For more information and examples, see:

- ASP.NET Core Blazor dependency injection
- ASP.NET Core server-side and Blazor Web App additional security scenarios

Blazor startup

Configure the manual start of Blazor's SignalR circuit in the `App.razor` file of a Blazor Web App:

- Add an `autostart="false"` attribute to the `<script>` tag for the `blazor.*.js` script.
- Place a script that calls `Blazor.start()` after the Blazor script is loaded and inside the closing `</body>` tag.

When `autostart` is disabled, any aspect of the app that doesn't depend on the circuit works normally. For example, client-side routing is operational. However, any aspect that depends on the circuit isn't operational until `Blazor.start()` is called. App behavior is unpredictable without an established circuit. For example, component methods fail to execute while the circuit is disconnected.

For more information, including how to initialize Blazor when the document is ready and how to chain to a JS Promise, see ASP.NET Core Blazor startup.

Configure SignalR timeouts and Keep-Alive on the client

Configure the following values for the client:

- `withServerTimeout`: Configures the server timeout in milliseconds. If this timeout elapses without receiving any messages from the server, the connection is terminated with an error. The default timeout value is 30 seconds. The server timeout should be at least double the value assigned to the Keep-Alive interval (`withKeepAliveInterval`).
- `withKeepAliveInterval`: Configures the Keep-Alive interval in milliseconds (default interval at which to ping the server). This setting allows the server to detect hard disconnects, such as when a client unplugs their computer from the network. The ping occurs at most as often as the server pings. If the server pings every five seconds, assigning a value lower than `5000` (5 seconds) pings every five seconds. The default value is 15 seconds. The Keep-Alive interval should be less than or equal to half the value assigned to the server timeout (`withServerTimeout`).

The following example for the `App.razor` file (Blazor Web App) shows the assignment of default values.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.withServerTimeout(30000).withKeepAliveInterval(15000);
      }
    }
  });
</script>
```

The following example for the `Pages/_Host.cshtml` file (Blazor Server, all versions except ASP.NET Core in .NET 6) or `Pages/_Layout.cshtml` file (Blazor Server, ASP.NET Core in .NET 6).

Blazor Server:

```HTML
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.withServerTimeout(30000).withKeepAliveInterval(15000);
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

When creating a hub connection in a component, set the ServerTimeout (default: 30 seconds) and KeepAliveInterval (default: 15 seconds) on the HubConnectionBuilder. Set the HandshakeTimeout (default: 15 seconds) on the built HubConnection. The following example shows the assignment of default values:

```C#
protected override async Task OnInitializedAsync()
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
        .WithServerTimeout(TimeSpan.FromSeconds(30))
        .WithKeepAliveInterval(TimeSpan.FromSeconds(15))
        .Build();

    hubConnection.HandshakeTimeout = TimeSpan.FromSeconds(15);

    hubConnection.On<string, string>("ReceiveMessage", (user, message) =>
...

    await hubConnection.StartAsync();
}
```

When changing the values of the server timeout (ServerTimeout) or the Keep-Alive interval (KeepAliveInterval):

- The server timeout should be at least double the value assigned to the Keep-Alive interval.
- The Keep-Alive interval should be less than or equal to half the value assigned to the server timeout.

For more information, see the *Global deployment and connection failures* sections of the following articles:

- Host and deploy ASP.NET Core server-side Blazor apps
- Host and deploy ASP.NET Core Blazor WebAssembly

Modify the server-side reconnection handler

The reconnection handler's circuit connection events can be modified for custom behaviors, such as:

- To notify the user if the connection is dropped.
- To perform logging (from the client) when a circuit is connected.

To modify the connection events, register callbacks for the following connection changes:

- Dropped connections use `onConnectionDown`.
- Established/re-established connections use `onConnectionUp`.

Both `onConnectionDown` and `onConnectionUp` must be specified.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      reconnectionHandler: {
        onConnectionDown: (options, error) => console.error(error),
        onConnectionUp: () => console.log("Up, up, and away!")
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    reconnectionHandler: {
      onConnectionDown: (options, error) => console.error(error),
```

```
      onConnectionUp: () => console.log("Up, up, and away!")
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

Automatically refresh the page when server-side reconnection fails

The default reconnection behavior requires the user to take manual action to refresh the page after reconnection fails. However, a custom reconnection handler can be used to automatically refresh the page:

`App.razor`:

HTML

```
<div id="reconnect-modal" style="display: none;"></div>
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script src="boot.js"></script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

Create the following `wwwroot/boot.js` file.

Blazor Web App:

JavaScript

```
(() => {
  const maximumRetryCount = 3;
  const retryIntervalMilliseconds = 5000;
  const reconnectModal = document.getElementById('reconnect-modal');

  const startReconnectionProcess = () => {
    reconnectModal.style.display = 'block';

    let isCanceled = false;

    (async () => {
      for (let i = 0; i < maximumRetryCount; i++) {
```

```
        reconnectModal.innerText = `Attempting to reconnect: ${i + 1} of
${maximumRetryCount}`;

        await new Promise(resolve => setTimeout(resolve,
retryIntervalMilliseconds));

        if (isCanceled) {
          return;
        }

        try {
          const result = await Blazor.reconnect();
          if (!result) {
            // The server was reached, but the connection was rejected;
reload the page.
            location.reload();
            return;
          }

          // Successfully reconnected to the server.
          return;
        } catch {
          // Didn't reach the server; try again.
        }
      }

      // Retried too many times; reload the page.
      location.reload();
    })();

    return {
      cancel: () => {
        isCanceled = true;
        reconnectModal.style.display = 'none';
      },
    };
  };

  let currentReconnectionProcess = null;

  Blazor.start({
    circuit: {
      reconnectionHandler: {
        onConnectionDown: () => currentReconnectionProcess ??=
startReconnectionProcess(),
        onConnectionUp: () => {
          currentReconnectionProcess?.cancel();
          currentReconnectionProcess = null;
        }
      }
    }
  });
})();
```

Blazor Server:

JavaScript

```
(() => {
  const maximumRetryCount = 3;
  const retryIntervalMilliseconds = 5000;
  const reconnectModal = document.getElementById('reconnect-modal');

  const startReconnectionProcess = () => {
    reconnectModal.style.display = 'block';

    let isCanceled = false;

    (async () => {
      for (let i = 0; i < maximumRetryCount; i++) {
        reconnectModal.innerText = `Attempting to reconnect: ${i + 1} of
${maximumRetryCount}`;

        await new Promise(resolve => setTimeout(resolve,
retryIntervalMilliseconds));

        if (isCanceled) {
          return;
        }

        try {
          const result = await Blazor.reconnect();
          if (!result) {
            // The server was reached, but the connection was rejected;
reload the page.
            location.reload();
            return;
          }

          // Successfully reconnected to the server.
          return;
        } catch {
          // Didn't reach the server; try again.
        }
      }

      // Retried too many times; reload the page.
      location.reload();
    })();

    return {
      cancel: () => {
        isCanceled = true;
        reconnectModal.style.display = 'none';
      },
    };
  };
```

```
  let currentReconnectionProcess = null;

  Blazor.start({
    reconnectionHandler: {
      onConnectionDown: () => currentReconnectionProcess ??=
startReconnectionProcess(),
      onConnectionUp: () => {
        currentReconnectionProcess?.cancel();
        currentReconnectionProcess = null;
      }
    }
  });
})();
```

For more information on Blazor startup, see ASP.NET Core Blazor startup.

Adjust the server-side reconnection retry count and interval

To adjust the reconnection retry count and interval, set the number of retries (`maxRetries`) and period in milliseconds permitted for each retry attempt (`retryIntervalMilliseconds`).

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      reconnectionOptions: {
        maxRetries: 3,
        retryIntervalMilliseconds: 2000
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    reconnectionOptions: {
      maxRetries: 3,
```

```
      retryIntervalMilliseconds: 2000
    }
  });
</script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script and the path to use, see ASP.NET Core Blazor project structure.

For more information on Blazor startup, see ASP.NET Core Blazor startup.

Control when the reconnection UI appears

Controlling when the reconnection UI appears can be useful in the following situations:

- A deployed app frequently displays the reconnection UI due to ping timeouts caused by internal network or Internet latency, and you would like to increase the delay.
- An app should report to users that the connection has dropped sooner, and you would like to shorten the delay.

The timing of the appearance of the reconnection UI is influenced by adjusting the Keep-Alive interval and timeouts on the client. The reconnection UI appears when the server timeout is reached on the client (`withServerTimeout`, Client configuration section). However, changing the value of `withServerTimeout` requires changes to other Keep-Alive, timeout, and handshake settings described in the following guidance.

As general recommendations for the guidance that follows:

- The Keep-Alive interval should match between client and server configurations.
- Timeouts should be at least double the value assigned to the Keep-Alive interval.

Server configuration

Set the following:

- ClientTimeoutInterval (default: 30 seconds): The time window clients have to send a message before the server closes the connection.
- HandshakeTimeout (default: 15 seconds): The interval used by the server to timeout incoming handshake requests by clients.
- KeepAliveInterval (default: 15 seconds): The interval used by the server to send keep alive pings to connected clients. Note that there is also a Keep-Alive interval setting on the client, which should match the server's value.

The ClientTimeoutInterval and HandshakeTimeout can be increased, and the KeepAliveInterval can remain the same. The important consideration is that if you change the values, make sure that the timeouts are at least double the value of the Keep-Alive interval and that the Keep-Alive interval matches between server and client. For more information, see the Configure SignalR timeouts and Keep-Alive on the client section.

In the following example:

- The ClientTimeoutInterval is increased to 60 seconds (default value: 30 seconds).
- The HandshakeTimeout is increased to 30 seconds (default value: 15 seconds).
- The KeepAliveInterval isn't set in developer code and uses its default value of 15 seconds. Decreasing the value of the Keep-Alive interval increases the frequency of communication pings, which increases the load on the app, server, and network. Care must be taken to avoid introducing poor performance when lowering the Keep-Alive interval.

Blazor Web App (.NET 8 or later) in the server project's `Program` file:

```
C#

builder.Services.AddRazorComponents().AddInteractiveServerComponents()
    .AddHubOptions(options =>
{
    options.ClientTimeoutInterval = TimeSpan.FromSeconds(60);
    options.HandshakeTimeout = TimeSpan.FromSeconds(30);
});
```

Blazor Server in the `Program` file:

```
C#

builder.Services.AddServerSideBlazor()
    .AddHubOptions(options =>
    {
        options.ClientTimeoutInterval = TimeSpan.FromSeconds(60);
        options.HandshakeTimeout = TimeSpan.FromSeconds(30);
    });
```

For more information, see the Server-side circuit handler options section.

Client configuration

Set the following:

- `withServerTimeout` (default: 30 seconds): Configures the server timeout, specified in milliseconds, for the circuit's hub connection.
- `withKeepAliveInterval` (default: 15 seconds): The interval, specified in milliseconds, at which the connection sends Keep-Alive messages.

The server timeout can be increased, and the Keep-Alive interval can remain the same. The important consideration is that if you change the values, make sure that the server timeout is at least double the value of the Keep-Alive interval and that the Keep-Alive interval values match between server and client. For more information, see the Configure SignalR timeouts and Keep-Alive on the client section.

In the following startup configuration example (location of the Blazor script), a custom value of 60 seconds is used for the server timeout. The Keep-Alive interval (`withKeepAliveInterval`) isn't set and uses its default value of 15 seconds.

Blazor Web App:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    circuit: {
      configureSignalR: function (builder) {
        builder.withServerTimeout(60000);
      }
    }
  });
</script>
```

Blazor Server:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
<script>
  Blazor.start({
    configureSignalR: function (builder) {
      builder.withServerTimeout(60000);
    }
  });
</script>
```

When creating a hub connection in a component, set the server timeout (WithServerTimeout, default: 30 seconds) on the HubConnectionBuilder. Set the HandshakeTimeout (default: 15 seconds) on the built HubConnection. Confirm that the

timeouts are at least double the Keep-Alive interval (WithKeepAliveInterval/KeepAliveInterval) and that the Keep-Alive value matches between server and client.

The following example is based on the `Index` component in the SignalR with Blazor tutorial. The server timeout is increased to 60 seconds, and the handshake timeout is increased to 30 seconds. The Keep-Alive interval isn't set and uses its default value of 15 seconds.

C#

```
protected override async Task OnInitializedAsync()
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
        .WithServerTimeout(TimeSpan.FromSeconds(60))
        .Build();

    hubConnection.HandshakeTimeout = TimeSpan.FromSeconds(30);

    hubConnection.On<string, string>("ReceiveMessage", (user, message) =>
...

    await hubConnection.StartAsync();
}
```

Disconnect Blazor's SignalR circuit from the client

Blazor's SignalR circuit is disconnected when the unload page event is triggered. To disconnect the circuit for other scenarios on the client, invoke `Blazor.disconnect` in the appropriate event handler. In the following example, the circuit is disconnected when the page is hidden (pagehide event):

JavaScript

```
window.addEventListener('pagehide', () => {
  Blazor.disconnect();
});
```

For more information on Blazor startup, see ASP.NET Core Blazor startup.

Server-side circuit handler

You can define a *circuit handler*, which allows running code on changes to the state of a user's circuit. A circuit handler is implemented by deriving from CircuitHandler and registering the class in the app's service container. The following example of a circuit handler tracks open SignalR connections.

`TrackingCircuitHandler.cs`:

C#

```
using Microsoft.AspNetCore.Components.Server.Circuits;

public class TrackingCircuitHandler : CircuitHandler
{
    private HashSet<Circuit> circuits = new();

    public override Task OnConnectionUpAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        circuits.Add(circuit);

        return Task.CompletedTask;
    }

    public override Task OnConnectionDownAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        circuits.Remove(circuit);

        return Task.CompletedTask;
    }

    public int ConnectedCircuits => circuits.Count;
}
```

Circuit handlers are registered using DI. Scoped instances are created per instance of a circuit. Using the `TrackingCircuitHandler` in the preceding example, a singleton service is created because the state of all circuits must be tracked.

In the `Program` file:

C#

```
builder.Services.AddSingleton<CircuitHandler, TrackingCircuitHandler>();
```

If a custom circuit handler's methods throw an unhandled exception, the exception is fatal to the circuit. To tolerate exceptions in a handler's code or called methods, wrap the code in one or more try-catch statements with error handling and logging.

When a circuit ends because a user has disconnected and the framework is cleaning up the circuit state, the framework disposes of the circuit's DI scope. Disposing the scope disposes any circuit-scoped DI services that implement System.IDisposable. If any DI service throws an unhandled exception during disposal, the framework logs the exception. For more information, see ASP.NET Core Blazor dependency injection.

Server-side circuit handler to capture users for custom services

Use a CircuitHandler to capture a user from the AuthenticationStateProvider and set that user in a service. For more information and example code, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Closure of circuits when there are no remaining Interactive Server components

Interactive Server components handle web UI events using a real-time connection with the browser called a circuit. A circuit and its associated state are created when a root Interactive Server component is rendered. The circuit is closed when there are no remaining Interactive Server components on the page, which frees up server resources.

Start the SignalR circuit at a different URL

Prevent automatically starting the app by adding `autostart="false"` to the Blazor `<script>` tag (location of the Blazor start script). Manually establish the circuit URL using `Blazor.start`. The following examples use the path `/signalr`.

Blazor Web Apps:

diff

```
- <script src="_framework/blazor.web.js"></script>
+ <script src="_framework/blazor.web.js" autostart="false"></script>
+ <script>
+   Blazor.start({
+     circuit: {
+       configureSignalR: builder => builder.withUrl("/signalr")
+     },
+   });
+ </script>
```

Blazor Server:

```diff
- <script src="_framework/blazor.server.js"></script>
+ <script src="_framework/blazor.server.js" autostart="false"></script>
+ <script>
+   Blazor.start({
+     configureSignalR: builder => builder.withUrl("/signalr")
+   });
+ </script>
```

Add the following MapBlazorHub call with the hub path to the middleware processing pipeline in the server app's `Program` file.

Blazor Web Apps:

```C#
app.MapBlazorHub("/signalr");
```

Blazor Server:

Leave the existing call to MapBlazorHub in the file and add a new call to MapBlazorHub with the path:

```diff
app.MapBlazorHub();
+ app.MapBlazorHub("/signalr");
```

`IHttpContextAccessor` / `HttpContext`

IHttpContextAccessor generally should be avoided with interactive rendering because a valid HttpContext isn't always available.

IHttpContextAccessor can be used for components that are statically rendered on the server. **However, we recommend avoiding it if possible.**

HttpContext can be used as a cascading parameter only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

C#

```
[CascadingParameter]
public HttpContext? HttpContext { get; set; }
```

For scenarios where the HttpContext is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Additional server-side resources

- Server-side host and deployment guidance: SignalR configuration
- Overview of ASP.NET Core SignalR
- ASP.NET Core SignalR configuration
- Server-side security documentation
  - ASP.NET Core Blazor authentication and authorization
  - ASP.NET Core Blazor authentication and authorization
  - Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering
  - ASP.NET Core server-side and Blazor Web App additional security scenarios
- Server-side reconnection events and component lifecycle events
- What is Azure SignalR Service?
- Performance guide for Azure SignalR Service
- Publish an ASP.NET Core SignalR app to Azure App Service
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor static files

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes Blazor app configuration for serving static files.

Static asset delivery in server-side Blazor apps

Configure Static File Middleware to serve static assets to clients by calling UseStaticFiles in the app's request processing pipeline. For more information, see Static files in ASP.NET Core.

In releases prior to .NET 8, Blazor framework static files, such as the Blazor script, are served via Static File Middleware. In .NET 8 or later, Blazor framework static files are mapped using endpoint routing, and Static File Middleware is no longer used.

Summary of static file `<link>` `href` formats

This section applies to all .NET releases and Blazor apps.

The following tables summarize static file `<link>` `href` formats by .NET release.

For the location of `<head>` content where static file links are placed, see ASP.NET Core Blazor project structure. Static asset links can also be supplied using <HeadContent> components in individual Razor components.

.NET 9 or later

Expand table

App type	`href` value	Examples
Blazor Web App	`@Assets["{PATH}"]`	`<link rel="stylesheet" href="@Assets["app.css"]" />` `<link href="@Assets["_content/ComponentLib/styles.css"]" rel="stylesheet" />`

App type	`href` value	Examples
Blazor Server†	`@Assets["{PATH}"]`	`<link href="@Assets["css/site.css"]" rel="stylesheet" />` `<link href="@Assets["_content/ComponentLib/styles.css"]" rel="stylesheet" />`
Standalone Blazor WebAssembly	`{PATH}`	`<link rel="stylesheet" href="css/app.css" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`

.NET 8.x

Expand table

App type	`href` value	Examples
Blazor Web App	`{PATH}`	`<link rel="stylesheet" href="app.css" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Blazor Server†	`{PATH}`	`<link href="css/site.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Standalone Blazor WebAssembly	`{PATH}`	`<link rel="stylesheet" href="css/app.css" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`

.NET 7.x or earlier

Expand table

App type	`href` value	Examples
Blazor Server†	`{PATH}`	`<link href="css/site.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Hosted Blazor WebAssembly‡	`{PATH}`	`<link href="css/app.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css" rel="stylesheet" />`
Blazor WebAssembly	`{PATH}`	`<link href="css/app.css" rel="stylesheet" />` `<link href="_content/ComponentLib/styles.css"`

App type	`href` value	Examples
		`rel="stylesheet" />`

†Blazor Server is supported in .NET 8 or later but is no longer a project template after .NET 7.
‡We recommend updating Hosted Blazor WebAssembly apps to Blazor Web Apps when adopting .NET 8 or later.

Static Web Asset Project Mode

This section applies to the `.Client` project of a Blazor Web App.

The required `<StaticWebAssetProjectMode>Default</StaticWebAssetProjectMode>` setting in the `.Client` project of a Blazor Web App reverts Blazor WebAssembly static asset behaviors back to the defaults, so that the project behaves as part of the hosted project. The Blazor WebAssembly SDK (`Microsoft.NET.Sdk.BlazorWebAssembly`) configures static web assets in a specific way to work in "standalone" mode with a server simply consuming the outputs from the library. This isn't appropriate for a Blazor Web App, where the WebAssembly portion of the app is a logical part of the host and must behave more like a library. For example, the project doesn't expose the styles bundle (for example, `BlazorSample.Client.styles.css`) and instead only provides the host with the project bundle, so that the host can include it in its own styles bundle.

Changing the value (`Default`) of `<StaticWebAssetProjectMode>` or removing the property from the `.Client` project isn't supported.

Static files in non-`Development` environments

This section applies to server-side static files.

When running an app locally, static web assets are only enabled in the Development environment. To enable static files for environments other than Development during local development and testing (for example, Staging), call UseStaticWebAssets on the WebApplicationBuilder in the `Program` file.

> ⚠ **Warning**
>
> Call UseStaticWebAssets for the ***exact environment*** to prevent activating the feature in production, as it serves files from separate locations on disk *other than*

from the project if called in a production environment. The example in this section checks for the Staging environment by calling IsStaging.

```
C#

if (builder.Environment.IsStaging())
{
    builder.WebHost.UseStaticWebAssets();
}
```

Prefix for Blazor WebAssembly assets

This section applies to Blazor Web Apps.

Use the WebAssemblyComponentsEndpointOptions.PathPrefix endpoint option to set the path string that indicates the prefix for Blazor WebAssembly assets. The path must correspond to a referenced Blazor WebAssembly application project.

```
C#

endpoints.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode(options =>
        options.PathPrefix = "{PATH PREFIX}");
```

In the preceding example, the `{PATH PREFIX}` placeholder is the path prefix and must start with a forward slash (`/`).

In the following example, the path prefix is set to `/path-prefix`:

```
C#

endpoints.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode(options =>
        options.PathPrefix = "/path-prefix");
```

Static web asset base path

This section applies to standalone Blazor WebAssembly apps.

Publishing the app places the app's static assets, including Blazor framework files (`_framework` folder assets), at the root path (`/`) in published output. The

`<StaticWebAssetBasePath>` property specified in the project file (`.csproj`) sets the base path to a non-root path:

```
XML

<PropertyGroup>
  <StaticWebAssetBasePath>{PATH}</StaticWebAssetBasePath>
</PropertyGroup>
```

In the preceding example, the `{PATH}` placeholder is the path.

Without setting the `<StaticWebAssetBasePath>` property, a standalone app is published at `/BlazorStandaloneSample/bin/Release/{TFM}/publish/wwwroot/`.

In the preceding example, the `{TFM}` placeholder is the Target Framework Moniker (TFM) (for example, `net6.0`).

If the `<StaticWebAssetBasePath>` property in a standalone Blazor WebAssembly app sets the published static asset path to `app1`, the root path to the app in published output is `/app1`.

In the standalone Blazor WebAssembly app's project file (`.csproj`):

```
XML

<PropertyGroup>
  <StaticWebAssetBasePath>app1</StaticWebAssetBasePath>
</PropertyGroup>
```

In published output, the path to the standalone Blazor WebAssembly app is `/BlazorStandaloneSample/bin/Release/{TFM}/publish/wwwroot/app1/`.

In the preceding example, the `{TFM}` placeholder is the Target Framework Moniker (TFM) (for example, `net6.0`).

File mappings and static file options

This section applies to server-side static files.

To create additional file mappings with a FileExtensionContentTypeProvider or configure other StaticFileOptions, use **one** of the following approaches. In the following examples, the `{EXTENSION}` placeholder is the file extension, and the `{CONTENT TYPE}` placeholder is

the content type. The namespace for the following API is Microsoft.AspNetCore.StaticFiles.

- Configure options through dependency injection (DI) in the `Program` file using StaticFileOptions:

  C#

  ```
  var provider = new FileExtensionContentTypeProvider();
  provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

  builder.Services.Configure<StaticFileOptions>(options =>
  {
      options.ContentTypeProvider = provider;
  });

  app.UseStaticFiles();
  ```

- Pass the StaticFileOptions directly to UseStaticFiles in the `Program` file:

  C#

  ```
  var provider = new FileExtensionContentTypeProvider();
  provider.Mappings["{EXTENSION}"] = "{CONTENT TYPE}";

  app.UseStaticFiles(new StaticFileOptions { ContentTypeProvider =
  provider });
  ```

Serve files from multiple locations

The guidance in this section only applies to Blazor Web Apps.

To serve files from multiple locations with a CompositeFileProvider:

- Add the namespace for Microsoft.Extensions.FileProviders to the top of the `Program` file of the server project.
- In the server project's `Program` file ***before*** the call to UseStaticFiles:
  - Create a PhysicalFileProvider with the path to the static assets.
  - Create a CompositeFileProvider from the WebRootFileProvider and the PhysicalFileProvider. Assign the composite file provider back to the app's WebRootFileProvider.

Example:

Create a new folder in the server project named `AdditionalStaticAssets`. Place an image into the folder.

Add the following `using` statement to the top of the server project's `Program` file:

C#

```
using Microsoft.Extensions.FileProviders;
```

In the server project's `Program` file ***before*** the call to UseStaticFiles, add the following code:

C#

```
var secondaryProvider = new PhysicalFileProvider(
    Path.Combine(builder.Environment.ContentRootPath,
"AdditionalStaticAssets"));
app.Environment.WebRootFileProvider = new CompositeFileProvider(
    app.Environment.WebRootFileProvider, secondaryProvider);
```

In the app's `Home` component (`Home.razor`) markup, reference the image with an `<img>` tag:

razor

```
<img src="{IMAGE FILE NAME}" alt="{ALT TEXT}" />
```

In the preceding example:

- The `{IMAGE FILE NAME}` placeholder is the image file name. There's no need to provide a path segment if the image file is at the root of the `AdditionalStaticAssets` folder.
- The `{ALT TEXT}` placeholder is the image alternate text.

Run the app.

Additional resources

- App base path
- Avoid file capture in a route parameter

ASP.NET Core Razor components

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to create and use Razor components in Blazor apps, including guidance on Razor syntax, component naming, namespaces, and component parameters.

Razor components

Blazor apps are built using *Razor components*, informally known as *Blazor components* or only *components*. A component is a self-contained portion of user interface (UI) with processing logic to enable dynamic behavior. Components can be nested, reused, shared among projects, and used in MVC and Razor Pages apps.

Components render into an in-memory representation of the browser's Document Object Model (DOM) called a *render tree*, which is used to update the UI in a flexible and efficient way.

Although "Razor components" shares some naming with other ASP.NET Core content-rendering technologies, Razor components must be distinguished from the following different features in ASP.NET Core:

- Razor views, which are Razor-based markup pages for MVC apps.
- View components, which are for rendering chunks of content rather than whole responses in Razor Pages and MVC apps.

> **Important**
>
> When using a Blazor Web App, most of the Blazor documentation example components ***require*** interactivity to function and demonstrate the concepts covered by the articles. When you test an example component provided by an article, make sure that either the app adopts global interactivity or the component adopts an interactive render mode. More information on this subject is provided by

ASP.NET Core Blazor render modes, which is the next article in the table of contents after this article.

Component classes

Components are implemented using a combination of C# and HTML markup in Razor component files with the `.razor` file extension.

ComponentBase is the base class for components described by Razor component files. ComponentBase implements the lowest abstraction of components, the IComponent interface. ComponentBase defines component properties and methods for basic functionality, for example, to process a set of built-in component lifecycle events.

ComponentBase in dotnet/aspnetcore reference source : The reference source contains additional remarks on the built-in lifecycle events. However, keep in mind that the internal implementations of component features are subject to change at any time without notice.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205) .

Developers typically create Razor components from Razor component files (`.razor`) or base their components on ComponentBase, but components can also be built by implementing IComponent. Developer-built components that implement IComponent can take low-level control over rendering at the cost of having to manually trigger rendering with events and lifecycle methods that the developer must create and maintain.

Additional conventions adopted by Blazor documentation example code and sample apps is found in ASP.NET Core Blazor fundamentals.

Razor syntax

Components use Razor syntax. Two Razor features are extensively used by components, *directives* and *directive attributes*. These are reserved keywords prefixed with `@` that

appear in Razor markup:

- Directives: Change the way component markup is compiled or functions. For example, the @page directive specifies a routable component with a route template that can be reached directly by a user's request in the browser at a specific URL.

  By convention, a component's directives at the top of a component definition (`.razor` file) are placed in a consistent order. For repeated directives, directives are placed alphabetically by namespace or type, except `@using` directives, which have special second-level ordering.

  The following order is adopted by Blazor sample apps and documentation. Components provided by a Blazor project template may differ from the following order and use a different format. For example, Blazor framework Identity components include blank lines between blocks of `@using` directives and blocks of `@inject` directives. You're free to use a custom ordering scheme and format in your own apps.

  Documentation and sample app Razor directive order:
  - `@page`
  - `@rendermode` (.NET 8 or later)
  - `@using`
    - `System` namespaces (alphabetical order)
    - `Microsoft` namespaces (alphabetical order)
    - Third-party API namespaces (alphabetical order)
    - App namespaces (alphabetical order)
  - Other directives (alphabetical order)

  No blank lines appear among the directives. One blank line appears between the directives and the first line of Razor markup.

  Example:

  razor

  ```
  @page "/doctor-who-episodes/{season:int}"
  @rendermode InteractiveWebAssembly
  @using System.Globalization
  @using System.Text.Json
  @using Microsoft.AspNetCore.Localization
  @using Mandrill
  @using BlazorSample.Components.Layout
  @attribute [Authorize]
  @implements IAsyncDisposable
  ```

```
@inject IJSRuntime JS
@inject ILogger<DoctorWhoEpisodes> Logger

<PageTitle>Doctor Who Episode List</PageTitle>

...
```

- Directive attributes: Change the way a component element is compiled or functions.

  Example:

  ```razor
  <input @bind="episodeId" />
  ```

  You can prefix directive attribute values with the at symbol (`@`) for non-explicit Razor expressions (`@bind="@episodeId"`), but we don't recommend it, and the docs don't adopt the approach in examples.

Directives and directive attributes used in components are explained further in this article and other articles of the Blazor documentation set. For general information on Razor syntax, see Razor syntax reference for ASP.NET Core.

Component name, class name, and namespace

A component's name must start with an uppercase character:

✔ `ProductDetail.razor`

✘ `productDetail.razor`

Common Blazor naming conventions used throughout the Blazor documentation include:

- File paths and file names use Pascal case† and appear before showing code examples. If a path is present, it indicates the typical folder location. For example, `Components/Pages/ProductDetail.razor` indicates that the `ProductDetail` component has a file name of `ProductDetail.razor` and resides in the `Pages` folder of the `Components` folder of the app.
- Component file paths for routable components match their URLs in kebab case‡ with hyphens appearing between words in a component's route template. For example, a `ProductDetail` component with a route template of `/product-detail`

(`@page "/product-detail"`) is requested in a browser at the relative URL `/product-detail`.

†Pascal case (upper camel case) is a naming convention without spaces and punctuation and with the first letter of each word capitalized, including the first word.
‡Kebab case is a naming convention without spaces and punctuation that uses lowercase letters and dashes between words.

Components are ordinary C# classes and can be placed anywhere within a project. Components that produce webpages usually reside in the `Components/Pages` folder. Non-page components are frequently placed in the `Components` folder or a custom folder added to the project.

Typically, a component's namespace is derived from the app's root namespace and the component's location (folder) within the app. If the app's root namespace is `BlazorSample` and the `Counter` component resides in the `Components/Pages` folder:

- The `Counter` component's namespace is `BlazorSample.Components.Pages`.
- The fully qualified type name of the component is `BlazorSample.Components.Pages.Counter`.

For custom folders that hold components, add an @using directive to the parent component or to the app's `_Imports.razor` file. The following example makes components in the `AdminComponents` folder available:

```razor
@using BlazorSample.AdminComponents
```

> **Note**
>
> @using directives in the `_Imports.razor` file are only applied to Razor files (`.razor`), not C# files (`.cs`).

Aliased using statements are supported. In the following example, the public `WeatherForecast` class of the `GridRendering` component is made available as `WeatherForecast` in a component elsewhere in the app:

```razor
@using WeatherForecast = Components.Pages.GridRendering.WeatherForecast
```

Components can also be referenced using their fully qualified names, which doesn't require an @using directive. The following example directly references the `ProductDetail` component in the `AdminComponents/Pages` folder of the app:

```razor
<BlazorSample.AdminComponents.Pages.ProductDetail />
```

The namespace of a component authored with Razor is based on the following (in priority order):

- The @namespace directive in the Razor file's markup (for example, `@namespace BlazorSample.CustomNamespace`).
- The project's `RootNamespace` in the project file (for example, `<RootNamespace>BlazorSample</RootNamespace>`).
- The project namespace and the path from the project root to the component. For example, the framework resolves `{PROJECT NAMESPACE}/Components/Pages/Home.razor` with a project namespace of `BlazorSample` to the namespace `BlazorSample.Components.Pages` for the `Home` component. `{PROJECT NAMESPACE}` is the project namespace. Components follow C# name binding rules. For the `Home` component in this example, the components in scope are all of the components:
  - In the same folder, `Components/Pages`.
  - The components in the project's root that don't explicitly specify a different namespace.

The following are **not** supported:

- The global:: qualification.
- Partially-qualified names. For example, you can't add `@using BlazorSample.Components` to a component and then reference the `NavMenu` component in the app's `Components/Layout` folder (`Components/Layout/NavMenu.razor`) with `<Layout.NavMenu></Layout.NavMenu>`.

Partial class support

Components are generated as C# partial classes and are authored using either of the following approaches:

- A single file contains C# code defined in one or more @code blocks, HTML markup, and Razor markup. Blazor project templates define their components

using this single-file approach.

- HTML and Razor markup are placed in a Razor file (`.razor`). C# code is placed in a code-behind file defined as a partial class (`.cs`).

> **Note**
>
> A component stylesheet that defines component-specific styles is a separate file (`.css`). Blazor CSS isolation is described later in ASP.NET Core Blazor CSS isolation.

The following example shows the default `Counter` component with an `@code` block in an app generated from a Blazor project template. Markup and C# code are in the same file. This is the most common approach taken in component authoring.

`Counter.razor`:

razor

```razor
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount() => currentCount++;
}
```

The following `Counter` component splits presentation HTML and Razor markup from the C# code using a code-behind file with a partial class. Splitting the markup from the C# code is favored by some organizations and developers to organize their component code to suit how they prefer to work. For example, the organization's UI expert can work on the presentation layer independently of another developer working on the component's C# logic. The approach is also useful when working with automatically-generated code or source generators. For more information, see Partial Classes and Methods (C# Programming Guide).

`CounterPartialClass.razor`:

razor

```
@page "/counter-partial-class"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
```

`CounterPartialClass.razor.cs`:

C#

```
namespace BlazorSample.Components.Pages;

public partial class CounterPartialClass
{
    private int currentCount = 0;

    private void IncrementCount() => currentCount++;
}
```

@using directives in the `_Imports.razor` file are only applied to Razor files (`.razor`), not C# files (`.cs`). Add namespaces to a partial class file as needed.

Typical namespaces used by components:

C#

```
using System.Net.Http;
using System.Net.Http.Json;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Components.Forms;
using Microsoft.AspNetCore.Components.Routing;
using Microsoft.AspNetCore.Components.Sections
using Microsoft.AspNetCore.Components.Web;
using static Microsoft.AspNetCore.Components.Web.RenderMode;
using Microsoft.AspNetCore.Components.Web.Virtualization;
using Microsoft.JSInterop;
```

Typical namespaces also include the namespace of the app and the namespace corresponding to the app's `Components` folder:

C#

```
using BlazorSample;
using BlazorSample.Components;
```

Additional folders can also be included, such as the `Layout` folder:

razor

```
using BlazorSample.Components.Layout;
```

Specify a base class

The @inherits directive is used to specify a base class for a component. Unlike using partial classes, which only split markup from C# logic, using a base class allows you to inherit C# code for use across a group of components that share the base class's properties and methods. Using base classes reduce code redundancy in apps and are useful when supplying base code from class libraries to multiple apps. For more information, see Inheritance in C# and .NET.

In the following example, the `BlazorRocksBase1` base class derives from ComponentBase.

`BlazorRocks1.razor`:

razor

```
@page "/blazor-rocks-1"
@inherits BlazorRocksBase1

<PageTitle>Blazor Rocks!</PageTitle>

<h1>Blazor Rocks! Example 1</h1>

<p>
    @BlazorRocksText
</p>
```

`BlazorRocksBase1.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample;

public class BlazorRocksBase1 : ComponentBase
```

```
{
    public string BlazorRocksText { get; set; } = "Blazor rocks the
browser!";
}
```

Routing

Routing in Blazor is achieved by providing a route template to each accessible component in the app with an @page directive. When a Razor file with an @page directive is compiled, the generated class is given a RouteAttribute specifying the route template. At runtime, the router searches for component classes with a RouteAttribute and renders whichever component has a route template that matches the requested URL.

The following `HelloWorld` component uses a route template of `/hello-world`, and the rendered webpage for the component is reached at the relative URL `/hello-world`.

`HelloWorld.razor`:

razor

```
@page "/hello-world"

<PageTitle>Hello World!</PageTitle>

<h1>Hello World!</h1>
```

The preceding component loads in the browser at `/hello-world` regardless of whether or not you add the component to the app's UI navigation. Optionally, components can be added to the `NavMenu` component so that a link to the component appears in the app's UI-based navigation.

For the preceding `HelloWorld` component, you can add a `NavLink` component to the `NavMenu` component. For more information, including descriptions of the `NavLink` and `NavMenu` components, see ASP.NET Core Blazor routing and navigation.

Markup

A component's UI is defined using Razor syntax, which consists of Razor markup, C#, and HTML. When an app is compiled, the HTML markup and C# rendering logic are converted into a component class. The name of the generated class matches the name of the file.

Members of the component class are defined in one or more @code blocks. In @code blocks, component state is specified and processed with C#:

- Property and field initializers.
- Parameter values from arguments passed by parent components and route parameters.
- Methods for user event handling, lifecycle events, and custom component logic.

Component members are used in rendering logic using C# expressions that start with the `@` symbol. For example, a C# field is rendered by prefixing `@` to the field name. The following `Markup` component evaluates and renders:

- `headingFontStyle` for the CSS property value `font-style` of the heading element.
- `headingText` for the content of the heading element.

`Markup.razor`:

razor

```
@page "/markup"

<PageTitle>Markup</PageTitle>

<h1>Markup Example</h1>

<h2 style="font-style:@headingFontStyle">@headingText</h2>

@code {
    private string headingFontStyle = "italic";
    private string headingText = "Put on your new Blazor!";
}
```

> **Note**
>
> Examples throughout the Blazor documentation specify the private access modifier for private members. Private members are scoped to a component's class. However, C# assumes the `private` access modifier when no access modifier is present, so explicitly marking members "`private`" in your own code is optional. For more information on access modifiers, see Access Modifiers (C# Programming Guide).

The Blazor framework processes a component internally as a *render tree*, which is the combination of a component's DOM and Cascading Style Sheet Object Model (CSSOM). After the component is initially rendered, the component's render tree is regenerated in response to events. Blazor compares the new render tree against the

previous render tree and applies any modifications to the browser's DOM for display. For more information, see ASP.NET Core Razor component rendering.

Razor syntax for C# control structures, directives, and directive attributes are lowercase (examples: @if, @code, @bind). Property names are uppercase (example: `@Body` for LayoutComponentBase.Body).

Asynchronous methods (`async`) don't support returning `void`

The Blazor framework doesn't track `void`-returning asynchronous methods (`async`). As a result, exceptions aren't caught if `void` is returned. Always return a Task from asynchronous methods.

Nested components

Components can include other components by declaring them using HTML syntax. The markup for using a component looks like an HTML tag where the name of the tag is the component type.

Consider the following `Heading` component, which can be used by other components to display a heading.

`Heading.razor`:

razor

```razor
<h1 style="font-style:@headingFontStyle">Heading Example</h1>

@code {
    private string headingFontStyle = "italic";
}
```

The following markup in the `HeadingExample` component renders the preceding `Heading` component at the location where the `<Heading />` tag appears.

`HeadingExample.razor`:

razor

```razor
@page "/heading-example"

<PageTitle>Heading</PageTitle>
```

```
<h1>Heading Example</h1>

<Heading />
```

If a component contains an HTML element with an uppercase first letter that doesn't match a component name within the same namespace, a warning is emitted indicating that the element has an unexpected name. Adding an @using directive for the component's namespace makes the component available, which resolves the warning. For more information, see the Component name, class name, and namespace section.

The `Heading` component example shown in this section doesn't have an @page directive, so the `Heading` component isn't directly accessible to a user via a direct request in the browser. However, any component with an @page directive can be nested in another component. If the `Heading` component was directly accessible by including `@page "/heading"` at the top of its Razor file, then the component would be rendered for browser requests at both `/heading` and `/heading-example`.

Component parameters

Component parameters pass data to components and are defined using public C# properties on the component class with the [Parameter] attribute. In the following example, a built-in reference type (System.String) and a user-defined reference type (`PanelBody`) are passed as component parameters.

`PanelBody.cs`:

C#

```
namespace BlazorSample;

public class PanelBody
{
    public string? Text { get; set; }
    public string? Style { get; set; }
}
```

`ParameterChild.razor`:

razor

```
<div class="card w-25" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">@Title</div>
    <div class="card-body" style="font-style:@Body.Style">
        @Body.Text
```

```
    </div>
</div>

@code {
    [Parameter]
    public string Title { get; set; } = "Set By Child";

    [Parameter]
    public PanelBody Body { get; set; } =
        new()
        {
            Text = "Card content set by child.",
            Style = "normal"
        };
}
```

> ⚠ **Warning**
>
> Providing initial values for component parameters is supported, but don't create a component that writes to its own parameters after the component is rendered for the first time. For more information, see Avoid overwriting parameters in ASP.NET Core Blazor.

The `Title` and `Body` component parameters of the `ParameterChild` component are set by arguments in the HTML tag that renders the instance of the component. The following `ParameterParent` component renders two `ParameterChild` components:

- The first `ParameterChild` component is rendered without supplying parameter arguments.
- The second `ParameterChild` component receives values for `Title` and `Body` from the `ParameterParent` component, which uses an explicit C# expression to set the values of the `PanelBody`'s properties.

`Parameter1.razor`:

razor

```
@page "/parameter-1"

<PageTitle>Parameter 1</PageTitle>

<h1>Parameter Example 1</h1>

<h1>Child component (without attribute values)</h1>

<ParameterChild />
```

```
<h1>Child component (with attribute values)</h1>

<ParameterChild Title="Set by Parent"
    Body="@(new PanelBody() { Text = "Set by parent.", Style = "italic" })"
/>
```

The following rendered HTML markup from the `ParameterParent` component shows `ParameterChild` component default values when the `ParameterParent` component doesn't supply component parameter values. When the `ParameterParent` component provides component parameter values, they replace the `ParameterChild` component's default values.

> **Note**
>
> For clarity, rendered CSS style classes aren't shown in the following rendered HTML markup.

HTML

```
<h1>Child component (without attribute values)</h1>

<div>
    <div>Set By Child</div>
    <div>Set by child.</div>
</div>

<h1>Child component (with attribute values)</h1>

<div>
    <div>Set by Parent</div>
    <div>Set by parent.</div>
</div>
```

Assign a C# field, property, or result of a method to a component parameter as an HTML attribute value. The value of the attribute can typically be any C# expression that matches the type of the parameter. The value of the attribute can optionally lead with a Razor reserved @ symbol, but it isn't required.

If the component parameter is of type string, then the attribute value is instead treated as a C# string literal. If you want to specify a C# expression instead, then use the `@` prefix.

The following `ParameterParent2` component displays four instances of the preceding `ParameterChild` component and sets their `Title` parameter values to:

- The value of the `title` field.
- The result of the `GetTitle` C# method.
- The current local date in long format with ToLongDateString, which uses an implicit C# expression.
- The `panelData` object's `Title` property.

Quotes around parameter attribute values are optional in most cases per the HTML5 specification. For example, `Value=this` is supported, instead of `Value="this"`. However, we recommend using quotes because it's easier to remember and widely adopted across web-based technologies.

Throughout the documentation, code examples:

- Always use quotes. Example: `Value="this"`.
- Don't use the `@` prefix with nonliterals unless required. Example: `Count="ct"`, where `ct` is a number-typed variable. `Count="@ct"` is a valid stylistic approach, but the documentation and examples don't adopt the convention.
- Always avoid `@` for literals, outside of Razor expressions. Example: `IsFixed="true"`. This includes keywords (for example, `this`) and `null`, but you can choose to use them if you wish. For example, `IsFixed="@true"` is uncommon but supported.

`Parameter2.razor`:

razor

```
@page "/parameter-2"

<PageTitle>Parameter 2</PageTitle>

<h1>Parameter Example 2</h1>

<ParameterChild Title="@title" />

<ParameterChild Title="@GetTitle()" />

<ParameterChild Title="@DateTime.Now.ToLongDateString()" />

<ParameterChild Title="@panelData.Title" />

@code {
    private string title = "From Parent field";
    private PanelData panelData = new();

    private string GetTitle() => "From Parent method";

    private class PanelData
    {
        public string Title { get; set; } = "From Parent object";
```

```
    }
}
```

> **Note**
>
> When assigning a C# member to a component parameter, don't prefix the parameter's HTML attribute with `@`.
>
> Correct (`Title` is a string parameter, `Count` is a number-typed parameter):
>
> ```razor
> <ParameterChild Title="@title" Count="ct" />
> ```
>
> ```razor
> <ParameterChild Title="@title" Count="@ct" />
> ```
>
> Incorrect:
>
> ```razor
> <ParameterChild @Title="@title" @Count="ct" />
> ```
>
> ```razor
> <ParameterChild @Title="@title" @Count="@ct" />
> ```

Unlike in Razor pages (`.cshtml`), Blazor can't perform asynchronous work in a Razor expression while rendering a component. This is because Blazor is designed for rendering interactive UIs. In an interactive UI, the screen must always display something, so it doesn't make sense to block the rendering flow. Instead, asynchronous work is performed during one of the asynchronous lifecycle events. After each asynchronous lifecycle event, the component may render again. The following Razor syntax is **not** supported:

```razor
<ParameterChild Title="await ..." />
<ParameterChild Title="@await ..." />
```

The code in the preceding example generates a *compiler error* when the app is built:

> The 'await' operator can only be used within an async method. Consider marking this method with the 'async' modifier and changing its return type to 'Task'.

To obtain a value for the `Title` parameter in the preceding example asynchronously, the component can use the OnInitializedAsync lifecycle event, as the following example demonstrates:

razor

```
<ParameterChild Title="@title" />

@code {
    private string? title;

    protected override async Task OnInitializedAsync()
    {
        title = await ...;
    }
}
```

For more information, see ASP.NET Core Razor component lifecycle.

Use of an explicit Razor expression to concatenate text with an expression result for assignment to a parameter is **not** supported. The following example seeks to concatenate the text "`Set by `" with an object's property value. Although this syntax is supported in a Razor page (`.cshtml`), it isn't valid for assignment to the child's `Title` parameter in a component. The following Razor syntax is **not** supported:

razor

```
<ParameterChild Title="Set by @(panelData.Title)" />
```

The code in the preceding example generates a *compiler error* when the app is built:

> Component attributes do not support complex content (mixed C# and markup).

To support the assignment of a composed value, use a method, field, or property. The following example performs the concatenation of "`Set by `" and an object's property value in the C# method `GetTitle`:

`Parameter3.razor`:

razor

```razor
@page "/parameter-3"

<PageTitle>Parameter 3</PageTitle>

<h1>Parameter Example 3</h1>

<ParameterChild Title="@GetTitle()" />

@code {
    private PanelData panelData = new();

    private string GetTitle() => $"Set by {panelData.Title}";

    private class PanelData
    {
        public string Title { get; set; } = "Parent";
    }
}
```

For more information, see Razor syntax reference for ASP.NET Core.

> ⚠ **Warning**
>
> Providing initial values for component parameters is supported, but don't create a component that writes to its own parameters after the component is rendered for the first time. For more information, see **Avoid overwriting parameters in ASP.NET Core Blazor**.

Component parameters should be declared as *auto-properties*, meaning that they shouldn't contain custom logic in their `get` or `set` accessors. For example, the following `StartData` property is an auto-property:

C#

```
[Parameter]
public DateTime StartData { get; set; }
```

Don't place custom logic in the `get` or `set` accessor because component parameters are purely intended for use as a channel for a parent component to flow information to a child component. If a `set` accessor of a child component property contains logic that causes rerendering of the parent component, an infinite rendering loop results.

To transform a received parameter value:

- Leave the parameter property as an auto-property to represent the supplied raw data.
- Create a different property or method to supply the transformed data based on the parameter property.

Override OnParametersSetAsync to transform a received parameter each time new data is received.

Writing an initial value to a component parameter is supported because initial value assignments don't interfere with the Blazor's automatic component rendering. The following assignment of the current local DateTime with DateTime.Now to `StartData` is valid syntax in a component:

C#

```
[Parameter]
public DateTime StartData { get; set; } = DateTime.Now;
```

After the initial assignment of DateTime.Now, do **not** assign a value to `StartData` in developer code. For more information, see Avoid overwriting parameters in ASP.NET Core Blazor.

Apply the [EditorRequired] attribute to specify a required component parameter. If a parameter value isn't provided, editors or build tools may display warnings to the user. This attribute is only valid on properties also marked with the [Parameter] attribute. The EditorRequiredAttribute is enforced at design-time and when the app is built. The attribute isn't enforced at runtime, and it doesn't guarantee a non-`null` parameter value.

C#

```
[Parameter]
[EditorRequired]
public string? Title { get; set; }
```

Single-line attribute lists are also supported:

C#

```
[Parameter, EditorRequired]
public string? Title { get; set; }
```

Don't use the required modifier or init accessor on component parameter properties. Components are usually instantiated and assigned parameter values using reflection,

which bypasses the guarantees that `init` and `required` are designed to make. Instead, use the [EditorRequired] attribute to specify a required component parameter.

Tuples (API documentation) are supported for component parameters and RenderFragment types. The following component parameter example passes three values in a `Tuple`:

`RenderTupleChild.razor`:

razor

```razor
<div class="card w-50" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Tuple Card</div>
    <div class="card-body">
        <ul>
            <li>Integer: @Data?.Item1</li>
            <li>String: @Data?.Item2</li>
            <li>Boolean: @Data?.Item3</li>
        </ul>
    </div>
</div>

@code {
    [Parameter]
    public (int, string, bool)? Data { get; set; }
}
```

`RenderTupleParent.razor`:

razor

```razor
@page "/render-tuple-parent"

<PageTitle>Render Tuple Parent</PageTitle>

<h1>Render Tuple Parent Example</h1>

<RenderTupleChild Data="data" />

@code {
    private (int, string, bool) data = new(999, "I aim to misbehave.",
true);
}
```

Named tuples are supported, as seen in the following example:

`NamedTupleChild.razor`:

razor

```
<div class="card w-50" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Tuple Card</div>
    <div class="card-body">
        <ul>
            <li>Integer: @Data?.TheInteger</li>
            <li>String: @Data?.TheString</li>
            <li>Boolean: @Data?.TheBoolean</li>
        </ul>
    </div>
</div>

@code {
    [Parameter]
    public (int TheInteger, string TheString, bool TheBoolean)? Data { get;
set; }
}
```

`NamedTuples.razor`:

razor

```
@page "/named-tuples"

<PageTitle>Named Tuples</PageTitle>

<h1>Named Tuples Example</h1>

<NamedTupleChild Data="data" />

@code {
    private (int TheInteger, string TheString, bool TheBoolean) data =
        new(999, "I aim to misbehave.", true);
}
```

Quote ©2005 Universal Pictures: Serenity (Nathan Fillion)

Route parameters

Components can specify route parameters in the route template of the @page directive. The Blazor router uses route parameters to populate corresponding component parameters.

`RouteParameter1.razor`:

razor

```
@page "/route-parameter-1/{text}"
```

```razor
<PageTitle>Route Parameter 1</PageTitle>

<h1>Route Parameter Example 1</h1>

<p>Blazor is @Text!</p>

@code {
    [Parameter]
    public string? Text { get; set; }
}
```

For more information, see the *Route parameters* section of ASP.NET Core Blazor routing and navigation. Optional route parameters are also supported and covered in the same section. For information on catch-all route parameters (`{*pageRoute}`), which capture paths across multiple folder boundaries, see the *Catch-all route parameters* section of ASP.NET Core Blazor routing and navigation.

Child content render fragments

Components can set the content of another component. The assigning component provides the content between the child component's opening and closing tags.

In the following example, the `RenderFragmentChild` component has a `ChildContent` component parameter that represents a segment of the UI to render as a RenderFragment. The position of `ChildContent` in the component's Razor markup is where the content is rendered in the final HTML output.

`RenderFragmentChild.razor`:

razor

```razor
<div class="card w-25" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Child content</div>
    <div class="card-body">@ChildContent</div>
</div>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

> **Important**
>
> The property receiving the **RenderFragment** content must be named `ChildContent` by convention.

Event callbacks aren't supported for RenderFragment.

The following component provides content for rendering the `RenderFragmentChild` by placing the content inside the child component's opening and closing tags.

`RenderFragments.razor`:

```razor
@page "/render-fragments"

<PageTitle>Render Fragments</PageTitle>

<h1>Render Fragments Example</h1>

<RenderFragmentChild>
    Content of the child component is supplied
    by the parent component.
</RenderFragmentChild>
```

Render fragments are used to render child content throughout Blazor apps and are described with examples in the following articles and article sections:

- Blazor layouts
- Pass data across a component hierarchy
- Templated components
- Global exception handling

> ⓘ Note
>
> Blazor framework's built-in Razor components use the same `ChildContent` component parameter convention to set their content. You can see the components that set child content by searching for the component parameter property name `ChildContent` in the API documentation (filters API with the search term "ChildContent").

Render fragments for reusable rendering logic

You can factor out child components purely as a way of reusing rendering logic. In any component's `@code` block, define a RenderFragment and render the fragment from any location as many times as needed:

```razor
```

```
@RenderWelcomeInfo

<p>Render the welcome info a second time:</p>

@RenderWelcomeInfo

@code {
    private RenderFragment RenderWelcomeInfo =  @<p>Welcome to your new app!
</p>;
}
```

For more information, see Reuse rendering logic.

Loop variables with component parameters and child content

Rendering components inside a for loop requires a local index variable if the incrementing loop variable is used by the component's parameters or RenderFragment child content.

Consider the following `RenderFragmentChild2` component that has both a component parameter (`Id`) and a render fragment to display child content (`ChildContent`).

`RenderFragmentChild2.razor`:

razor

```
<div class="card w-25" style="margin-bottom:15px">
    <div class="card-header font-weight-bold">Child content (@Id)</div>
    <div class="card-body">@ChildContent</div>
</div>

@code {
    [Parameter]
    public string? Id { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

When rendering the `RenderFragmentChild2` component in a parent component, use a local index variable (`ct` in the following example) instead of the loop variable (`c`) when assigning the component parameter value and providing the child component's content:

razor

```
@for (int c = 1; c < 4; c++)
{
    var ct = c;

    <RenderFragmentChild2 Id="@($"Child{ct}")">
        Count: @ct
    </RenderFragmentChild2>
}
```

Alternatively, use a foreach loop with Enumerable.Range instead of a for loop:

razor

```
@foreach (var c in Enumerable.Range(1, 3))
{
    <RenderFragmentChild2 Id="@($"Child{c}")">
        Count: @c
    </RenderFragmentChild2>
}
```

Capture references to components

Component references provide a way to reference a component instance for issuing commands. To capture a component reference:

- Add an @ref attribute to the child component.
- Define a field with the same type as the child component.

When the component is rendered, the field is populated with the component instance. You can then invoke .NET methods on the instance.

Consider the following `ReferenceChild` component that logs a message when its `ChildMethod` is called.

`ReferenceChild.razor`:

razor

```
@inject ILogger<ReferenceChild> Logger

@if (value > 0)
{
    <p>
        <code>value</code>: @value
    </p>
```

```
}

@code {
    private int value;

    public void ChildMethod(int value)
    {
        Logger.LogInformation("Received {Value} in ChildMethod", value);

        this.value = value;
        StateHasChanged();
    }
}
```

A component reference is only populated after the component is rendered and its output includes `ReferenceChild`'s element. Until the component is rendered, there's nothing to reference. Don't attempt to call a referenced component method to an event handler directly (for example, `@onclick="childComponent!.ChildMethod(5)"`) because the reference variable may not be assigned at the time the click event is assigned.

To manipulate component references after the component has finished rendering, use the OnAfterRender or OnAfterRenderAsync methods.

The following example uses the preceding `ReferenceChild` component.

`ReferenceParent.razor`:

razor

```
@page "/reference-parent"

<div>
    <button @onclick="@(() => childComponent1!.ChildMethod(5))">
        Call <code>ReferenceChild.ChildMethod</code> (first instance)
        with an argument of 5
    </button>

    <ReferenceChild @ref="childComponent1" />
</div>

<div>
    <button @onclick="CallChildMethod">
        Call <code>ReferenceChild.ChildMethod</code> (second instance)
        with an argument of 5
    </button>

    <ReferenceChild @ref="childComponent2" />
</div>

@code {
    private ReferenceChild? childComponent1;
```

```
    private ReferenceChild? childComponent2;

    private void CallChildMethod() => childComponent2!.ChildMethod(5);
}
```

While capturing component references use a similar syntax to capturing element references, capturing component references isn't a JavaScript interop feature. Component references aren't passed to JavaScript code. Component references are only used in .NET code.

> ⓘ **Important**
>
> Do **not** use component references to mutate the state of child components. Instead, use normal declarative component parameters to pass data to child components. Use of component parameters result in child components that rerender at the correct times automatically. For more information, see the component parameters section and the ASP.NET Core Blazor data binding article.

Apply an attribute

Attributes can be applied to components with the @attribute directive. The following example applies the [Authorize] attribute to the component's class:

razor

```
@page "/"
@attribute [Authorize]
```

Conditional HTML element attributes and DOM properties

Blazor adopts the following general behaviors:

- For HTML attributes, Blazor sets or removes the attribute conditionally based on the .NET value. If the .NET value is `false` or `null`, the attribute isn't set or is removed if it was previously set.
- For DOM properties, such as `checked` or `value`, Blazor sets the DOM property based on the .NET value. If the .NET value is `false` or `null`, the DOM property is reset to a default value.

Which Razor syntax attributes correspond to HTML attributes and which ones correspond to DOM properties remains undocumented because this is a framework implementation detail that might change without notice.

> ⚠ **Warning**
>
> Some HTML attributes, such as aria-pressed, must have a string value of either "true" or "false". Since they require a string value and not a boolean, you must use a .NET `string` and not a `bool` for their value. This is a requirement set by browser DOM APIs.

Raw HTML

Strings are normally rendered using DOM text nodes, which means that any markup they may contain is ignored and treated as literal text. To render raw HTML, wrap the HTML content in a MarkupString value. The value is parsed as HTML or SVG and inserted into the DOM.

> ⚠ **Warning**
>
> Rendering raw HTML constructed from any untrusted source is a **security risk** and should **always** be avoided.

The following example shows using the MarkupString type to add a block of static HTML content to the rendered output of a component.

`MarkupStrings.razor`:

razor

```
@page "/markup-strings"

<PageTitle>Markup Strings</PageTitle>

<h1>Markup Strings Example</h1>

@((MarkupString)myMarkup)

@code {
    private string myMarkup =
        "<p class=\"text-danger\">This is a dangerous <em>markup
string</em>.</p>";
}
```

Razor templates

Render fragments can be defined using Razor template syntax to define a UI snippet. Razor templates use the following format:

```razor
@<{HTML tag}>...</{HTML tag}>
```

The following example illustrates how to specify RenderFragment and RenderFragment<TValue> values and render templates directly in a component. Render fragments can also be passed as arguments to templated components.

`RazorTemplate.razor`:

```razor
@page "/razor-template"

<PageTitle>Razor Template</PageTitle>

<h1>Razor Template Example</h1>

@timeTemplate

@petTemplate(new Pet { Name = "Nutty Rex" })

@code {
    private RenderFragment timeTemplate = @<p>The time is @DateTime.Now.
</p>;
    private RenderFragment<Pet> petTemplate = (pet) => @<p>Pet:
@pet.Name</p>;

    private class Pet
    {
        public string? Name { get; set; }
    }
}
```

Rendered output of the preceding code:

```HTML
<p>The time is 4/19/2021 8:54:46 AM.</p>
<p>Pet: Nutty Rex</p>
```

Static assets

Blazor follows the convention of ASP.NET Core apps for static assets. Static assets are located in the project's web root (wwwroot) folder or folders under the `wwwroot` folder.

Use a base-relative path (`/`) to refer to the web root for a static asset. In the following example, `logo.png` is physically located in the `{PROJECT ROOT}/wwwroot/images` folder. `{PROJECT ROOT}` is the app's project root.

razor

```
<img alt="Company logo" src="/images/logo.png" />
```

Components do **not** support tilde-slash notation (`~/`).

For information on setting an app's base path, see Host and deploy ASP.NET Core Blazor.

Tag Helpers aren't supported in components

Tag Helpers aren't supported in components. To provide Tag Helper-like functionality in Blazor, create a component with the same functionality as the Tag Helper and use the component instead.

Scalable Vector Graphics (SVG) images

Since Blazor renders HTML, browser-supported images, including Scalable Vector Graphics (SVG) images (.svg), are supported via the `<img>` tag:

HTML

```
<img alt="Example image" src="image.svg" />
```

Similarly, SVG images are supported in the CSS rules of a stylesheet file (`.css`):

css

```
.element-class {
    background-image: url("image.svg");
}
```

Blazor supports the <foreignObject> element to display arbitrary HTML within an SVG. The markup can represent arbitrary HTML, a RenderFragment, or a Razor component.

The following example demonstrates:

- Display of a `string` (`@message`).
- Two-way binding with an `<input>` element and a `value` field.
- A `Robot` component.

razor

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
    <rect x="0" y="0" rx="10" ry="10" width="200" height="200"
stroke="black"
        fill="none" />
    <foreignObject x="20" y="20" width="160" height="160">
        <p>@message</p>
    </foreignObject>
</svg>

<svg xmlns="http://www.w3.org/2000/svg">
    <foreignObject width="200" height="200">
        <label>
            Two-way binding:
            <input @bind="value" @bind:event="oninput" />
        </label>
    </foreignObject>
</svg>

<svg xmlns="http://www.w3.org/2000/svg">
    <foreignObject>
        <Robot />
    </foreignObject>
</svg>

@code {
    private string message = "Lorem ipsum dolor sit amet, consectetur
adipiscing " +
        "elit, sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua.";

    private string? value;
}
```

Whitespace rendering behavior

Unless the @preservewhitespace directive is used with a value of `true`, extra whitespace is removed if:

- Leading or trailing within an element.
- Leading or trailing within a RenderFragment/RenderFragment<TValue> parameter (for example, child content passed to another component).
- It precedes or follows a C# code block, such as `@if` or `@foreach`.

Whitespace removal might affect the rendered output when using a CSS rule, such as `white-space: pre`. To disable this performance optimization and preserve the whitespace, take one of the following actions:

- Add the `@preservewhitespace true` directive at the top of the Razor file (`.razor`) to apply the preference to a specific component.
- Add the `@preservewhitespace true` directive inside an `_Imports.razor` file to apply the preference to a subdirectory or to the entire project.

In most cases, no action is required, as apps typically continue to behave normally (but faster). If stripping whitespace causes a rendering problem for a particular component, use `@preservewhitespace true` in that component to disable this optimization.

Root component

A *root Razor component* (*root component*) is the first component loaded of any component hierarchy created by the app.

In an app created from the Blazor Web App project template, the `App` component (`App.razor`) is specified as the default root component by the type parameter declared for the call to MapRazorComponents<TRootComponent> in the server-side `Program` file. The following example shows the use of the `App` component as the root component, which is the default for an app created from the Blazor project template:

C#

```
app.MapRazorComponents<App>();
```

> **Note**
>
> Making a root component interactive, such as the `App` component, isn't supported.

In an app created from the Blazor WebAssembly project template, the `App` component (`App.razor`) is specified as the default root component in the `Program` file:

C#

```
builder.RootComponents.Add<App>("#app");
```

In the preceding code, the CSS selector, `#app`, indicates that the `App` component is specified for the `<div>` in `wwwroot/index.html` with an `id` of `app`:

HTML

```
<div id="app">...</app>
```

MVC and Razor Pages apps can also use the Component Tag Helper to register statically-rendered Blazor WebAssembly root components:

CSHTML

```
<component type="typeof(App)" render-mode="WebAssemblyPrerendered" />
```

Statically-rendered components can only be added to the app. They can't be removed or updated afterwards.

For more information, see the following resources:

- Component Tag Helper in ASP.NET Core
- Integrate ASP.NET Core Razor components with MVC or Razor Pages

`IHttpContextAccessor` / `HttpContext`

IHttpContextAccessor generally should be avoided with interactive rendering because a valid HttpContext isn't always available.

IHttpContextAccessor can be used for components that are statically rendered on the server. **However, we recommend avoiding it if possible.**

HttpContext can be used as a cascading parameter only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

C#

```
[CascadingParameter]
public HttpContext? HttpContext { get; set; }
```

For scenarios where the HttpContext is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

ASP.NET Core Blazor render modes

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains control of Razor component rendering in Blazor Web Apps, either at compile time or runtime.

This guidance doesn't apply to standalone Blazor WebAssembly apps. Blazor WebAssembly apps only render on the client via a client-side WebAssembly-based runtime and have no concept of a *render mode*. If a render mode is applied to a component in a Blazor WebAssembly app, the render mode designation has no influence on rendering the component.

Render modes

Every component in a Blazor Web App adopts a *render mode* to determine the hosting model that it uses, where it's rendered, and whether or not it's interactive.

The following table shows the available render modes for rendering Razor components in a Blazor Web App. To apply a render mode to a component use the `@rendermode` directive on the component instance or on the component definition. Later in this article, examples are shown for each render mode scenario.

Expand table

Name	Description	Render location	Interactive
Static Server	Static server-side rendering (static SSR)	Server	✗
Interactive Server	Interactive server-side rendering (interactive SSR) using Blazor Server.	Server	✓
Interactive WebAssembly	Client-side rendering (CSR) using Blazor WebAssembly†.	Client	✓
Interactive Auto	Interactive SSR using Blazor Server initially and then CSR on subsequent visits after the Blazor	Server, then client	✓

Name	Description	Render location	Interactive
	bundle is downloaded.		

†Client-side rendering (CSR) is assumed to be interactive. "*Interactive* client-side rendering" and "*interactive* CSR" aren't used by the industry or in the Blazor documentation.

Prerendering is enabled by default for interactive components. Guidance on controlling prerendering is provided later in this article. For general industry terminology on client and server rendering concepts, see ASP.NET Core Blazor fundamentals.

The following examples demonstrate setting the component's render mode with a few basic Razor component features.

To test the render mode behaviors locally, you can place the following components in an app created from the *Blazor Web App* project template. When you create the app, select options from dropdown menus (Visual Studio) or apply the CLI options (.NET CLI) to enable both server-side and client-side interactivity. For guidance on how to create a Blazor Web App, see Tooling for ASP.NET Core Blazor.

Enable support for interactive render modes

A Blazor Web App must be configured to support interactive render modes. The following extensions are automatically applied to apps created from the *Blazor Web App* project template during app creation. Individual components are still required to declare their render mode per the *Render modes* section after the component services and endpoints are configured in the app's `Program` file.

Services for Razor components are added by calling AddRazorComponents.

Component builder extensions:

- AddInteractiveServerComponents adds services to support rendering Interactive Server components.
- AddInteractiveWebAssemblyComponents adds services to support rendering Interactive WebAssembly components.

MapRazorComponents discovers available components and specifies the root component for the app (the first component loaded), which by default is the `App` component (`App.razor`).

Endpoint convention builder extensions:

- AddInteractiveServerRenderMode configures interactive server-side rendering (interactive SSR) for the app.
- AddInteractiveWebAssemblyRenderMode configures the Interactive WebAssembly render mode for the app.

> **Note**
>
> For orientation on the placement of the API in the following examples, inspect the `Program` file of an app generated from the Blazor Web App project template. For guidance on how to create a Blazor Web App, see Tooling for ASP.NET Core Blazor.

Example 1: The following `Program` file API adds services and configuration for enabling interactive SSR:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

Example 2: The following `Program` file API adds services and configuration for enabling the Interactive WebAssembly render mode:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveWebAssemblyComponents();
```

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode();
```

Example 3: The following `Program` file API adds services and configuration for enabling the Interactive Server, Interactive WebAssembly, and Interactive Auto render modes:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents()
    .AddInteractiveWebAssemblyComponents();
```

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode()
    .AddInteractiveWebAssemblyRenderMode();
```

Blazor uses the Blazor WebAssembly hosting model to download and execute components that use the Interactive WebAssembly render mode. A separate client project is required to set up Blazor WebAssembly hosting for these components. The client project contains the startup code for the Blazor WebAssembly host and sets up the .NET runtime for running in a browser. The Blazor Web App template adds this client project for you when you select the option to enable WebAssembly interactivity. Any components using the Interactive WebAssembly render mode should be built from the client project, so they get included in the downloaded app bundle.

Apply a render mode to a component instance

To apply a render mode to a component instance use the @rendermode Razor directive attribute where the component is used.

In the following example, interactive server-side rendering (interactive SSR) is applied to the `Dialog` component instance:

razor

```
<Dialog @rendermode="InteractiveServer" />
```

> **Note**
>
> Blazor templates include a static `using` directive for RenderMode in the app's `_Imports` file (`Components/_Imports.razor`) for shorter `@rendermode` syntax:
>
> razor
>
> ```
> @using static Microsoft.AspNetCore.Components.Web.RenderMode
> ```

Without the preceding directive, components must specify the static RenderMode class in `@rendermode` syntax:

```razor
<Dialog @rendermode="RenderMode.InteractiveServer" />
```

You can also reference custom render mode instances instantiated directly with custom configuration. For more information, see the Custom shorthand render modes section later in this article.

Apply a render mode to a component definition

To specify the render mode for a component as part of its definition, use the @rendermode Razor directive and the corresponding render mode attribute.

```razor
@page "..."
@rendermode InteractiveServer
```

Applying a render mode to a component definition is commonly used when applying a render mode to a specific page. Routable pages use the same render mode as the Router component that rendered the page.

Technically, `@rendermode` is both a Razor *directive* and a Razor *directive attribute*. The semantics are similar, but there are differences. The `@rendermode` directive is on the component definition, so the referenced render mode instance must be static. The `@rendermode` directive attribute can take any render mode instance.

> **Note**
>
> Component authors should avoid coupling a component's implementation to a specific render mode. Instead, component authors should typically design components to support any render mode or hosting model. A component's implementation should avoid assumptions on where it's running (server or client) and should degrade gracefully when rendered statically. Specifying the render mode in the component definition may be needed if the component isn't

instantiated directly (such as with a routable page component) or to specify a render mode for all component instances.

Apply a render mode to the entire app

To set the render mode for the entire app, indicate the render mode at the highest-level interactive component in the app's component hierarchy that isn't a root component.

> **Note**
>
> Making a root component interactive, such as the `App` component, isn't supported. Therefore, the render mode for the entire app can't be set directly by the `App` component.

For apps based on the Blazor Web App project template, a render mode assigned to the entire app is typically specified where the `Routes` component is used in the `App` component (`Components/App.razor`):

razor

```
<Routes @rendermode="InteractiveServer" />
```

The Router component propagates its render mode to the pages it routes.

You also typically must set the same interactive render mode on the HeadOutlet component, which is also found in the `App` component of a Blazor Web App generated from the project template:

```
<HeadOutlet @rendermode="InteractiveServer" />
```

For apps that adopt an interactive client-side (WebAssembly or Auto) rendering mode and enable the render mode for the entire app via the `Routes` component:

- Place or move the layout and navigation files of the server app's `Components/Layout` folder into the `.Client` project's `Layout` folder. Create a `Layout` folder in the `.Client` project if it doesn't exist.
- Place or move the components of the server app's `Components/Pages` folder into the `.Client` project's `Pages` folder. Create a `Pages` folder in the `.Client` project if

it doesn't exist.
- Place or move the `Routes` component of the server app's `Components` folder into the `.Client` project's root folder.

To enable global interactivity when creating a Blazor Web App:

- Visual Studio: Set the **Interactivity location** dropdown list to **Global**.
- .NET CLI: Use the `-ai|--all-interactive` option.

For more information, see Tooling for ASP.NET Core Blazor.

Apply a render mode programatically

Properties and fields can assign a render mode.

The second approach described in this section, setting the render mode by component instance, is especially useful when your app specification calls for one or more components to adopt static SSR in a globally-interactive app. This scenario is covered in the Static SSR pages in a globally-interactive app section later in this article.

Set the render mode by component definition

A component definition can define a render mode via a private field:

razor

```
@rendermode pageRenderMode

...

@code {
    private static IComponentRenderMode pageRenderMode = InteractiveServer;
}
```

Set the render mode by component instance

The following example applies interactive server-side rendering (interactive SSR) to any request.

razor

```
<Routes @rendermode="PageRenderMode" />

...
```

```
@code {
    private IComponentRenderMode? PageRenderMode => InteractiveServer;
}
```

Additional information on render mode propagation is provided in the Render mode propagation section later in this article. The Static SSR pages in a globally-interactive app section shows how to use the preceding approach to adopt static SSR in a globally-interactive app.

Blazor documentation examples for Blazor Web Apps

When using a Blazor Web App, most of the Blazor documentation example components ***require*** interactivity to function and demonstrate the concepts covered by the articles. When you test an example component provided by an article, make sure that either the app adopts global interactivity or the component adopts an interactive render mode.

Prerendering

Prerendering is the process of initially rendering page content on the server without enabling event handlers for rendered controls. The server outputs the HTML UI of the page as soon as possible in response to the initial request, which makes the app feel more responsive to users. Prerendering can also improve Search Engine Optimization (SEO) by rendering content for the initial HTTP response that search engines use to calculate page rank.

Prerendering is enabled by default for interactive components.

Internal navigation for interactive routing doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests, including for enhanced navigation. For more information, see Static versus interactive routing, Interactive routing and prerendering, and Enhanced navigation and form handling.

Disabling prerendering using the following techniques only takes effect for top-level render modes. If a parent component specifies a render mode, the prerendering settings of its children are ignored. This behavior is under investigation for possible changes with the release of .NET 10 in November, 2025.

To disable prerendering for a *component instance*, pass the `prerender` flag with a value of `false` to the render mode:

- `<... @rendermode="new InteractiveServerRenderMode(prerender: false)" />`
- `<... @rendermode="new InteractiveWebAssemblyRenderMode(prerender: false)" />`
- `<... @rendermode="new InteractiveAutoRenderMode(prerender: false)" />`

To disable prerendering in a *component definition*:

- `@rendermode @(new InteractiveServerRenderMode(prerender: false))`
- `@rendermode @(new InteractiveWebAssemblyRenderMode(prerender: false))`
- `@rendermode @(new InteractiveAutoRenderMode(prerender: false))`

To disable prerendering for the entire app, indicate the render mode at the highest-level interactive component in the app's component hierarchy that isn't a root component.

For apps based on the Blazor Web App project template, a render mode assigned to the entire app is specified where the `Routes` component is used in the `App` component (`Components/App.razor`). The following example sets the app's render mode to Interactive Server with prerendering disabled:

razor

```razor
<Routes @rendermode="new InteractiveServerRenderMode(prerender: false)" />
```

Also, disable prerendering for the HeadOutlet component in the `App` component:

razor

```razor
<HeadOutlet @rendermode="new InteractiveServerRenderMode(prerender: false)"
/>
```

Making a root component, such as the `App` component, interactive with the `@rendermode` directive at the top of the root component's definition file (`.razor`) isn't supported. Therefore, prerendering can't be disabled directly by the `App` component.

Static server-side rendering (static SSR)

Components use static server-side rendering (static SSR). The component renders to the response stream and interactivity isn't enabled.

In the following example, there's no designation for the component's render mode, so the component inherits its render mode from its parent. Since no ancestor component specifies a render mode, the following component is *statically rendered* on the server. The button isn't interactive and doesn't call the `UpdateMessage` method when selected. The value of `message` doesn't change, and the component isn't rerendered in response to UI events.

`RenderMode1.razor`:

razor

```razor
@page "/render-mode-1"

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component locally in a Blazor Web App, place the component in the server project's `Components/Pages` folder. The server project is the solution's project with a name that doesn't end in `.Client`. When the app is running, navigate to `/render-mode-1` in the browser's address bar.

During static SSR, Razor component page requests are processed by server-side ASP.NET Core middleware pipeline request processing for routing and authorization. Dedicated Blazor features for routing and authorization aren't operational because Razor components aren't rendered during server-side request processing. Blazor router features in the `Routes` component that aren't available during static SSR include displaying:

- Not authorized content (<NotAuthorized>...</NotAuthorized>) (NotAuthorized): Blazor Web Apps typically process unauthorized requests on the server by customizing the behavior of Authorization Middleware.
- Not found content (<NotFound>...</NotFound>) (NotFound): Blazor Web Apps typically process bad URL requests on the server by either displaying the browser's built-in 404 UI or returning a custom 404 page (or other response) via ASP.NET Core middleware (for example, UseStatusCodePagesWithRedirects / API

documentation). For more information, see Create simpler way to show Not Found 404 UI with Blazor (dotnet/aspnetcore #45654).

If the app exhibits root-level interactivity, server-side ASP.NET Core request processing isn't involved after the initial static SSR, which means that the preceding Blazor features work as expected.

Enhanced navigation with static SSR requires special attention when loading JavaScript. For more information, see ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR).

Interactive server-side rendering (interactive SSR)

Interactive server-side rendering (interactive SSR) renders the component interactively from the server using Blazor Server. User interactions are handled over a real-time connection with the browser. The circuit connection is established when the Server component is rendered.

In the following example, the render mode is set interactive SSR by adding `@rendermode InteractiveServer` to the component definition. The button calls the `UpdateMessage` method when selected. The value of `message` changes, and the component is rerendered to update the message in the UI.

`RenderMode2.razor`:

razor

```razor
@page "/render-mode-2"
@rendermode InteractiveServer

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component in a Blazor Web App, place the component in the server project's `Components/Pages` folder. The server project is the solution's project with

a name that doesn't end in `.Client`. When the app is running, navigate to `/render-mode-2` in the browser's address bar.

Client-side rendering (CSR)

Client-side rendering (CSR) renders the component interactively on the client using Blazor WebAssembly. The .NET runtime and app bundle are downloaded and cached when the WebAssembly component is initially rendered. Components using CSR must be built from a separate client project that sets up the Blazor WebAssembly host.

In the following example, the render mode is set to CSR with `@rendermode InteractiveWebAssembly`. The button calls the `UpdateMessage` method when selected. The value of `message` changes, and the component is rerendered to update the message in the UI.

`RenderMode3.razor`:

razor

```razor
@page "/render-mode-3"
@rendermode InteractiveWebAssembly

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component locally in a Blazor Web App, place the component in the client project's `Pages` folder. The client project is the solution's project with a name that ends in `.Client`. When the app is running, navigate to `/render-mode-3` in the browser's address bar.

Automatic (Auto) rendering

Automatic (Auto) rendering determines how to render the component at runtime. The component is initially rendered with interactive server-side rendering (interactive SSR) using the Blazor Server hosting model. The .NET runtime and app bundle are

downloaded to the client in the background and cached so that they can be used on future visits.

The Auto render mode never dynamically changes the render mode of a component already on the page. The Auto render mode makes an initial decision about which type of interactivity to use for a component, then the component keeps that type of interactivity for as long as it's on the page. One factor in this initial decision is considering whether components already exist on the page with WebAssembly/Server interactivity. Auto mode prefers to select a render mode that matches the render mode of existing interactive components. The reason that the Auto mode prefers to use an existing interactivity mode is to avoid introducing a new interactive runtime that doesn't share state with the existing runtime.

Components using the Auto render mode must be built from a separate client project that sets up the Blazor WebAssembly host.

In the following example, the component is interactive throughout the process. The button calls the `UpdateMessage` method when selected. The value of `message` changes, and the component is rerendered to update the message in the UI. Initially, the component is rendered interactively from the server, but on subsequent visits it's rendered from the client after the .NET runtime and app bundle are downloaded and cached.

`RenderMode4.razor`:

razor

```razor
@page "/render-mode-4"
@rendermode InteractiveAuto

<button @onclick="UpdateMessage">Click me</button> @message

@code {
    private string message = "Not updated yet.";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

If using the preceding component locally in a Blazor Web App, place the component in the client project's `Pages` folder. The client project is the solution's project with a name that ends in `.Client`. When the app is running, navigate to `/render-mode-4` in the browser's address bar.

Render mode propagation

Render modes propagate down the component hierarchy.

Rules for applying render modes:

- The default render mode is Static.
- The Interactive Server (InteractiveServer), Interactive WebAssembly (InteractiveWebAssembly), and Interactive Auto (InteractiveAuto) render modes can be used from a component, including using different render modes for sibling components.
- You can't switch to a different interactive render mode in a child component. For example, a Server component can't be a child of a WebAssembly component.
- Parameters passed to an interactive child component from a Static parent must be JSON serializable. This means that you can't pass render fragments or child content from a Static parent component to an interactive child component.

The following examples use a non-routable, non-page `SharedMessage` component. The render mode agnostic `SharedMessage` component doesn't apply a render mode with an @attribute directive. If you're testing these scenarios with a Blazor Web App, place the following component in the app's `Components` folder.

`SharedMessage.razor`:

razor

```razor
<p>@Greeting</p>

<button @onclick="UpdateMessage">Click me</button> @message

<p>@ChildContent</p>

@code {
    private string message = "Not updated yet.";

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    [Parameter]
    public string Greeting { get; set; } = "Hello!";

    private void UpdateMessage()
    {
        message = "Somebody updated me!";
    }
}
```

Render mode inheritance

If the `SharedMessage` component is placed in a statically-rendered parent component, the `SharedMessage` component is also rendered statically and isn't interactive. The button doesn't call `UpdateMessage`, and the message isn't updated.

`RenderMode5.razor`:

```razor
@page "/render-mode-5"

<SharedMessage />
```

If the `SharedMessage` component is placed in a component that defines the render mode, it inherits the applied render mode.

In the following example, the `SharedMessage` component is interactive over a SignalR connection to the client. The button calls `UpdateMessage`, and the message is updated.

`RenderMode6.razor`:

```razor
@page "/render-mode-6"
@rendermode InteractiveServer

<SharedMessage />
```

Child components with different render modes

In the following example, both `SharedMessage` components are prerendered and appear when the page is displayed in the browser.

- The first `SharedMessage` component with interactive server-side rendering (interactive SSR) is interactive after Blazor's SignalR circuit is established.
- The second `SharedMessage` component with client-side rendering (CSR) is interactive *after* the Blazor app bundle is downloaded and the .NET runtime is active on the client.

`RenderMode7.razor`:

```razor
```

```
@page "/render-mode-7"

<SharedMessage @rendermode="InteractiveServer" />
<SharedMessage @rendermode="InteractiveWebAssembly" />
```

Child component with a serializable parameter

The following example demonstrates an interactive child component that takes a parameter. Parameters must be serializable.

`RenderMode8.razor`:

razor

```
@page "/render-mode-8"

<SharedMessage @rendermode="InteractiveServer" Greeting="Welcome!" />
```

Non-serializable component parameters, such as child content or a render fragment, aren't supported. In the following example, passing child content to the `SharedMessage` component results in a runtime error.

`RenderMode9.razor`:

razor

```
@page "/render-mode-9"

<SharedMessage @rendermode="InteractiveServer">
    Child content
</SharedMessage>
```

❌ **Error**:

> System.InvalidOperationException: Cannot pass the parameter 'ChildContent' to component 'SharedMessage' with rendermode 'InteractiveServerRenderMode'. This is because the parameter is of the delegate type 'Microsoft.AspNetCore.Components.RenderFragment', which is arbitrary code and cannot be serialized.

To circumvent the preceding limitation, wrap the child component in another component that doesn't have the parameter. This is the approach taken in the Blazor

Web App project template with the `Routes` component (`Components/Routes.razor`) to wrap the Router component.

`WrapperComponent.razor`:

```razor
<SharedMessage>
    Child content
</SharedMessage>
```

`RenderMode10.razor`:

```razor
@page "/render-mode-10"

<WrapperComponent @rendermode="InteractiveServer" />
```

In the preceding example:

- The child content is passed to the `SharedMessage` component without generating a runtime error.
- The `SharedMessage` component renders interactively on the server.

Child component with a different render mode than its parent

Don't try to apply a different interactive render mode to a child component than its parent's render mode.

The following component results in a runtime error when the component is rendered:

`RenderMode11.razor`:

```razor
@page "/render-mode-11"
@rendermode InteractiveServer

<SharedMessage @rendermode="InteractiveWebAssembly" />
```

❌ **Error**:

> Cannot create a component of type 'BlazorSample.Components.SharedMessage' because its render mode 'Microsoft.AspNetCore.Components.Web.InteractiveWebAssemblyRenderMode' is not supported by Interactive Server rendering.

Static SSR pages in a globally-interactive app

There are cases where the app's specification calls for components to adopt static server-side rendering (static SSR) and only run on the server, while the rest of the app uses an interactive render mode.

This approach is only useful when the app has specific pages that can't work with interactive Server or WebAssembly rendering. For example, adopt this approach for pages that depend on reading/writing HTTP cookies and can only work in a request/response cycle instead of interactive rendering. For pages that work with interactive rendering, you shouldn't force them to use static SSR rendering, as it's less efficient and less responsive for the end user.

There are two approaches that can be taken for fine control of render modes, each of which is described in the following subsections:

- Area (folder) of static SSR components: You have an area (folder) of the app with components that must adopt static SSR and share the same route path prefix. The app controls the render mode globally by setting the render mode on the `Routes` component in the `App` component based on the path to the folder.
- Static SSR components spread out across the app: You have components spread around the app in various locations that must adopt static SSR and only run on the server. The static SSR-only components aren't in a single folder and don't share a common route path prefix. The app controls the render mode on a per-component basis by setting the render mode with the `@rendermode` directive in component instances. Reflection is used in the `App` component to set the render mode on the `Routes` component.

In both cases, the component that must adopt static SSR must also force a full-page reload.

The following examples use the HttpContext cascading parameter to determine if the page is statically-rendered. A `null` HttpContext indicates that the component is rendering interactively, which is useful as a signal in app code to trigger a full-page reload.

Area (folder) of static SSR components

The approach described in this subsection is used by the Blazor Web App project template with individual authentication and global interactivity.

An area (folder) of the app contains the components that must adopt static SSR and only run on the server. The components in the folder share the same route path prefix. For example, the Identity Razor components of the Blazor Web App project template are in the `Components/Account/Pages` folder and share the root path prefix `/Account`.

The folder also contains an `_Imports.razor` file, which applies a custom account layout to the components in the folder:

razor

```
@using BlazorSample.Components.Account.Shared
@layout AccountLayout
```

The `Shared` folder maintains the `AccountLayout` layout component. The component makes use of HttpContext to determine if the component has adopted static SSR. Identity components must render on the server with static SSR because they set Identity cookies. If the value of HttpContext is `null`, the component is rendering interactively, and a full-page reload is performed by calling NavigationManager.Refresh with `forceLoad` set to `true`. This forces a full rerender of the page using static SSR.

`Components/Account/Shared/AccountLayout.razor`:

razor

```
@inherits LayoutComponentBase
@layout BlazorSample.Components.Layout.MainLayout
@inject NavigationManager Navigation

@if (HttpContext is null)
{
    <p>Loading...</p>
}
else
{
    @Body
}

@code {
    [CascadingParameter]
    private HttpContext? HttpContext { get; set; }

    protected override void OnParametersSet()
```

```
    {
        if (HttpContext is null)
        {
            Navigation.Refresh(forceReload: true);
        }
    }
}
```

> **Note**
>
> In the Blazor Web App project template, there's a second layout file (`ManageLayout.razor` in the `Components/Account/Shared` folder) for Identity components in the `Components/Account/Pages/Manage` folder. The `Manage` folder has its own `_Imports.razor` file to apply to the `ManageLayout` to components in the folder. In your own apps, using nested `_Imports.razor` files is a useful approach for applying custom layouts to groups of pages.

In the `App` component, any request for a component in the `Account` folder applies a `null` render mode, which enforces static SSR. Other component requests receive a global application of the interactive SSR render mode (`InteractiveServer`).

> **Important**
>
> Applying a `null` render mode doesn't always enforce static SSR. It just happens to behave that way using the approach shown in this section.
>
> A `null` render mode is effectively the same as not specifying a render mode, which results in the component inheriting its parent's render mode. In this case, the `App` component is rendered using static SSR, so a `null` render mode results in the `Routes` component inheriting static SSR from the `App` component. If a null render mode is specified for a child component whose parent uses an interactive render mode, the child inherits the same interactive render mode.

`Components/App.razor`:

razor

```razor
<Routes @rendermode="RenderModeForPage" />

...

@code {
    [CascadingParameter]
```

```
    private HttpContext HttpContext { get; set; } = default!;

    private IComponentRenderMode? RenderModeForPage =>
        HttpContext.Request.Path.StartsWithSegments("/Account")
            ? null
            : {INTERACTIVE RENDER MODE};
}
```

In the preceding code, change the `{INTERACTIVE RENDER MODE}` placeholder to the appropriate value, depending on if the rest of the application should adopt global InteractiveServer, InteractiveWebAssembly, or InteractiveAuto rendering.

The components that must adopt static SSR in the `Account` folder aren't required to set the layout, which is applied via the `_Imports.razor` file. The components don't set a render mode because they should render with static SSR. Nothing further must be done for the components in the `Account` folder to enforce static SSR.

Static SSR components spread out across the app

In the preceding subsection, the app controls the render mode of the components by setting the render mode globally in the `App` component. Alternatively, the `App` component can also adopt ***per-component*** render modes for setting the render mode, which permits components spread around the app to enforce adoption of static SSR. This subsection describes the approach.

The app has a custom layout that can be applied to components around the app. Usually, a shared component for the app is placed in the `Components/Layout` folder. The component makes use of HttpContext to determine if the component has adopted static SSR. If the value of HttpContext is `null`, the component is rendering interactively, and a full-page reload is performed by calling NavigationManager.Refresh with `forceLoad` set to `true`. This triggers a request to the server for the component.

`Components/Layout/StaticSsrLayout.razor`:

razor

```
@inherits LayoutComponentBase
@layout MainLayout
@inject NavigationManager Navigation

@if (HttpContext is null)
{
    <p>Loading...</p>
}
else
```

```
{
    @Body
}

@code {
    [CascadingParameter]
    private HttpContext? HttpContext { get; set; }

    protected override void OnParametersSet()
    {
        if (HttpContext is null)
        {
            Navigation.Refresh(forceReload: true);
        }
    }
}
```

In the `App` component, reflection is used to set the render mode. Whatever render mode is assigned to the individual component definition file is applied to the `Routes` component.

`Components/App.razor`:

razor

```
<Routes @rendermode="RenderModeForPage" />

...

@code {
    [CascadingParameter]
    private HttpContext HttpContext { get; set; } = default!;

    private IComponentRenderMode? RenderModeForPage =>
        HttpContext.GetEndpoint()?.Metadata.GetMetadata<RenderModeAttribute>
()?
            .Mode;
}
```

Each component that must adopt static SSR sets the custom layout and doesn't specify a render mode. Not specifying a render mode results in a `null` value of RenderModeAttribute.Mode in the `App` component, which results in no render mode assigned to the `Routes` component instance and enforcement of static SSR.

> **ⓘ Important**
>
> Applying a `null` render mode doesn't always enforce static SSR. It just happens to behave that way using the approach shown in this section.

A `null` render mode is effectively the same as not specifying a render mode, which results in the component inheriting its parent's render mode. In this case, the `App` component is rendered using static SSR, so a `null` render mode results in the `Routes` component inheriting static SSR from the `App` component. If a null render mode is specified for a child component whose parent uses an interactive render mode, the child inherits the same interactive render mode.

Nothing further must be done for the components to enforce static SSR than applying the custom layout ***without setting an interactive render mode***:

razor

```
@layout BlazorSample.Components.Layout.StaticSsrLayout
```

Interactive components around the app ***avoid*** applying the custom static SSR layout and ***only set an appropriate interactive render mode***, which upon reflection in the `App` component is applied to the `Routes` component:

razor

```
@rendermode {INTERACTIVE RENDER MODE}
```

In the preceding code, change the `{INTERACTIVE RENDER MODE}` placeholder to the appropriate value, depending on if the component should adopt InteractiveServer, InteractiveWebAssembly, or InteractiveAuto rendering.

Client-side services fail to resolve during prerendering

Assuming that prerendering isn't disabled for a component or for the app, a component in the `.Client` project is prerendered on the server. Because the server doesn't have access to registered client-side Blazor services, it isn't possible to inject these services into a component without receiving an error that the service can't be found during prerendering.

For example, consider the following `Home` component in the `.Client` project in a Blazor Web App with global Interactive WebAssembly or Interactive Auto rendering. The component attempts to inject IWebAssemblyHostEnvironment to obtain the environment's name.

```
```

razor

```
@page "/"
@inject IWebAssemblyHostEnvironment Environment

<PageTitle>Home</PageTitle>

<h1>Home</h1>

<p>
    Environment: @Environment.Environment
</p>
```

No compile time error occurs, but a runtime error occurs during prerendering:

> Cannot provide a value for property 'Environment' on type 'BlazorSample.Client.Pages.Home'. There is no registered service of type 'Microsoft.AspNetCore.Components.WebAssembly.Hosting.IWebAssemblyHostEnvironment'.

This error occurs because the component must compile and execute on the server during prerendering, but IWebAssemblyHostEnvironment isn't a registered service on the server.

If the app doesn't require the value during prerendering, this problem can be solved by injecting IServiceProvider to obtain the service instead of the service type itself:

razor

```
@page "/"
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
@inject IServiceProvider Services

<PageTitle>Home</PageTitle>

<h1>Home</h1>

<p>
    <b>Environment:</b> @environmentName
</p>

@code {
    private string? environmentName;

    protected override void OnInitialized()
    {
        if (Services.GetService<IWebAssemblyHostEnvironment>() is { } env)
        {
            environmentName = env.Environment;
```

```
        }
    }
}
```

However, the preceding approach isn't useful if your logic requires a value during prerendering.

You can also avoid the problem if you disable prerendering for the component, but that's an extreme measure to take in many cases that may not meet your component's specifications.

There are a three approaches that you can take to address this scenario. The following are listed from most recommended to least recommended:

- *Recommended* for shared framework services: For shared framework services that merely aren't registered server-side in the main project, register the services in the main project, which makes them available during prerendering. For an example of this scenario, see the guidance for HttpClient services in Call a web API from an ASP.NET Core Blazor app.
- *Recommended* for services outside of the shared framework: Create a custom service implementation for the service on the server. Use the service normally in interactive components of the `.Client` project. For a demonstration of this approach, see ASP.NET Core Blazor environments.
- Create a service abstraction and create implementations for the service in the `.Client` and server projects. Register the services in each project. Inject the custom service in the component.
- You might be able to add a `.Client` project package reference to a server-side package and fall back to using the server-side API when prerendering on the server.

Discover components from additional assemblies

Additional assemblies must be disclosed to the Blazor framework to discover routable Razor components in referenced projects. For more information, see ASP.NET Core Blazor routing and navigation.

Closure of circuits when there are no remaining Interactive Server components

Interactive Server components handle web UI events using a real-time connection with the browser called a circuit. A circuit and its associated state are created when a root Interactive Server component is rendered. The circuit is closed when there are no remaining Interactive Server components on the page, which frees up server resources.

Custom shorthand render modes

The `@rendermode` directive takes a single parameter that's a static instance of type IComponentRenderMode. The `@rendermode` directive attribute can take any render mode instance, static or not. The Blazor framework provides the RenderMode static class with some predefined render modes for convenience, but you can create your own.

Normally, a component uses the following `@rendermode` directive to disable prerendering:

razor

```
@rendermode @(new InteractiveServerRenderMode(prerender: false))
```

However, consider the following example that creates a shorthand interactive server-side render mode without prerendering via the app's `_Imports` file (`Components/_Imports.razor`):

C#

```
public static IComponentRenderMode InteractiveServerWithoutPrerendering {
get; } =
    new InteractiveServerRenderMode(prerender: false);
```

Use the shorthand render mode in components throughout the `Components` folder:

razor

```
@rendermode InteractiveServerWithoutPrerendering
```

Alternatively, a single component instance can define a custom render mode via a private field:

razor

```razor
@rendermode interactiveServerWithoutPrerendering

...

@code {
    private static IComponentRenderMode interactiveServerWithoutPrerendering
=
        new InteractiveServerRenderMode(prerender: false);
}
```

At the moment, the shorthand render mode approach is probably only useful for reducing the verbosity of specifying the `prerender` flag. The shorthand approach might be more useful in the future if additional flags become available for interactive rendering and you would like to create shorthand render modes with different combinations of flags.

Service injection via a top-level imports file (`_Imports.razor`)

This section only applies to Blazor Web Apps.

A top-level imports file in the `Components` folder (`Components/_Imports.razor`) injects its references into all of the components in the folder hierarchy, which includes the `App` component (`App.razor`). The `App` component is always rendered statically even if prerendering of a page component is disabled. Therefore, injecting services via the top-level imports file results in resolving *two instances* of the service in page components.

To address this scenario, inject the service in a new imports file placed in the `Pages` folder (`Components/Pages/_Imports.razor`). From that location, the service is only resolved once in page components.

Additional resources

- ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)
- Cascading values/parameters and render mode boundaries: Also see the Root-level cascading parameters section earlier in the article.
- ASP.NET Core Razor class libraries (RCLs) with static server-side rendering (static SSR)

Prerender ASP.NET Core Razor components

Article • 01/10/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains Razor component prerendering scenarios for server-rendered components in Blazor Web Apps.

Prerendering is the process of initially rendering page content on the server without enabling event handlers for rendered controls. The server outputs the HTML UI of the page as soon as possible in response to the initial request, which makes the app feel more responsive to users. Prerendering can also improve Search Engine Optimization (SEO) by rendering content for the initial HTTP response that search engines use to calculate page rank.

Persist prerendered state

Without persisting prerendered state, state used during prerendering is lost and must be recreated when the app is fully loaded. If any state is created asynchronously, the UI may flicker as the prerendered UI is replaced when the component is rerendered.

Consider the following `PrerenderedCounter1` counter component. The component sets an initial random counter value during prerendering in OnInitialized lifecycle method. After the SignalR connection to the client is established, the component rerenders, and the initial count value is replaced when `OnInitialized` executes a second time.

`PrerenderedCounter1.razor`:

razor

```razor
@page "/prerendered-counter-1"
@rendermode @(new InteractiveServerRenderMode(prerender: true))
@inject ILogger<PrerenderedCounter1> Logger

<PageTitle>Prerendered Counter 1</PageTitle>

<h1>Prerendered Counter 1</h1>
```

```razor
<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount;

    protected override void OnInitialized()
    {
        currentCount = Random.Shared.Next(100);
        Logger.LogInformation("currentCount set to {Count}", currentCount);
    }

    private void IncrementCount() => currentCount++;
}
```

Run the app and inspect logging from the component. The following is example output.

> **Note**
>
> If the app adopts interactive routing and the page is reached via an internal enhanced navigation, prerendering doesn't occur. Therefore, you must perform a full page reload for the `PrerenderedCounter1` component to see the following output. For more information, see the Interactive routing and prerendering section.

```
info: BlazorSample.Components.Pages.PrerenderedCounter1[0]
currentCount set to 41
info: BlazorSample.Components.Pages.PrerenderedCounter1[0]
currentCount set to 92
```

The first logged count occurs during prerendering. The count is set again after prerendering when the component is rerendered. There's also a flicker in the UI when the count updates from 41 to 92.

To retain the initial value of the counter during prerendering, Blazor supports persisting state in a prerendered page using the PersistentComponentState service (and for components embedded into pages or views of Razor Pages or MVC apps, the Persist Component State Tag Helper).

To preserve prerendered state, decide what state to persist using the PersistentComponentState service. PersistentComponentState.RegisterOnPersisting

registers a callback to persist the component state before the app is paused. The state is retrieved when the app resumes.

The following example demonstrates the general pattern:

- The `{TYPE}` placeholder represents the type of data to persist.
- The `{TOKEN}` placeholder is a state identifier string. Consider using `nameof({VARIABLE})`, where the `{VARIABLE}` placeholder is the name of the variable that holds the state. Using nameof() for the state identifier avoids the use of a quoted string.

razor

```
@implements IDisposable
@inject PersistentComponentState ApplicationState

...

@code {
    private {TYPE} data;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistData);

        if (!ApplicationState.TryTakeFromJson<{TYPE}>(
            "{TOKEN}", out var restored))
        {
            data = await ...;
        }
        else
        {
            data = restored!;
        }
    }

    private Task PersistData()
    {
        ApplicationState.PersistAsJson("{TOKEN}", data);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

The following counter component example persists counter state during prerendering and retrieves the state to initialize the component.

`PrerenderedCounter2.razor`:

razor

```razor
@page "/prerendered-counter-2"
@implements IDisposable
@inject ILogger<PrerenderedCounter2> Logger
@inject PersistentComponentState ApplicationState

<PageTitle>Prerendered Counter 2</PageTitle>

<h1>Prerendered Counter 2</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override void OnInitialized()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistCount);

        if (!ApplicationState.TryTakeFromJson<int>(
            nameof(currentCount), out var restoredCount))
        {
            currentCount = Random.Shared.Next(100);
            Logger.LogInformation("currentCount set to {Count}",
currentCount);
        }
        else
        {
            currentCount = restoredCount!;
            Logger.LogInformation("currentCount restored to {Count}",
currentCount);
        }
    }

    private Task PersistCount()
    {
        ApplicationState.PersistAsJson(nameof(currentCount), currentCount);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose() => persistingSubscription.Dispose();
```

```
    private void IncrementCount() => currentCount++;
}
```

When the component executes, `currentCount` is only set once during prerendering. The value is restored when the component is rerendered. The following is example output.

> **Note**
>
> If the app adopts interactive routing and the page is reached via an internal enhanced navigation, prerendering doesn't occur. Therefore, you must perform a full page reload for the `PrerenderedCounter2` component to see the following output. For more information, see the Interactive routing and prerendering section.

```
info: BlazorSample.Components.Pages.PrerenderedCounter2[0]
currentCount set to 96
info: BlazorSample.Components.Pages.PrerenderedCounter2[0]
currentCount restored to 96
```

By initializing components with the same state used during prerendering, any expensive initialization steps are only executed once. The rendered UI also matches the prerendered UI, so no flicker occurs in the browser.

The persisted prerendered state is transferred to the client, where it's used to restore the component state. During client-side rendering (CSR, `InteractiveWebAssembly`), the data is exposed to the browser and must not contain sensitive, private information. During interactive server-side rendering (interactive SSR, `InteractiveServer`), ASP.NET Core Data Protection ensures that the data is transferred securely. The `InteractiveAuto` render mode combines WebAssembly and Server interactivity, so it's necessary to consider data exposure to the browser, as in the CSR case.

Components embedded into pages and views (Razor Pages/MVC)

For components embedded into a page or view of a Razor Pages or MVC app, you must add the Persist Component State Tag Helper with the `<persist-component-state />` HTML tag inside the closing `</body>` tag of the app's layout. **This is only required for Razor Pages and MVC apps.** For more information, see Persist Component State Tag Helper in ASP.NET Core.

`Pages/Shared/_Layout.cshtml`:

CSHTML

```
<body>
    ...

    <persist-component-state />
</body>
```

Interactive routing and prerendering

When the `Routes` component doesn't define a render mode, the app is using per-page/component interactivity and navigation. Using per-page/component navigation, internal† navigation is handled by enhanced routing after the app becomes interactive. †*Internal* in this context means that the URL destination of the navigation event is a Blazor endpoint inside the app.

The PersistentComponentState service only works on the initial page load and not across internal enhanced page navigation events.

If the app performs a full (non-enhanced) navigation to a page utilizing persistent component state, the persisted state is made available for the app to use when it becomes interactive.

If an interactive circuit has already been established and an enhanced navigation is performed to a page utilizing persistent component state, the state *isn't made available in the existing circuit for the component to use*. There's no prerendering for the internal page request, and the PersistentComponentState service isn't aware that an enhanced navigation has occurred. There's no mechanism to deliver state updates to components that are already running on an existing circuit. The reason for this is that Blazor only supports passing state from the server to the client at the time the runtime initializes, not after the runtime has started.

Additional work on the Blazor framework to address this scenario is under consideration for .NET 10 (November, 2025). For more information and community discussion of *unsupported workarounds*‡, see Support persistent component state across enhanced page navigations (dotnet/aspnetcore #51584). ‡Unsupported workarounds aren't sanctioned by Microsoft for use in Blazor apps. *Use third-party packages, approaches, and code at your own risk*.

Disabling enhanced navigation, which reduces performance but also avoids the problem of loading state with PersistentComponentState for internal page requests, is covered in

ASP.NET Core Blazor routing and navigation.

Prerendering guidance

Prerendering guidance is organized in the Blazor documentation by subject matter. The following links cover all of the prerendering guidance throughout the documentation set by subject:

- Fundamentals
  - OnNavigateAsync is executed *twice* when prerendering: Handle asynchronous navigation events with OnNavigateAsync
  - Startup: Control headers in C# code
  - Handle Errors: Prerendering
  - SignalR: Prerendered state size and SignalR message size limit
- Render modes: Prerendering
- Components
  - Control <head> content during prerendering
  - Razor component lifecycle subjects that pertain to prerendering
    - Component initialization (OnInitialized{Async})
    - After component render (OnAfterRender{Async})
    - Stateful reconnection after prerendering
    - Prerendering with JavaScript interop: This section also appears in the two JS interop articles on calling JavaScript from .NET and calling .NET from JavaScript.
    - Handle incomplete asynchronous actions at render: Guidance for delayed rendering due to long-running lifecycle tasks during prerendering on the server.
  - QuickGrid component sample app: The **QuickGrid for Blazor** sample app is hosted on GitHub Pages. The site loads fast thanks to static prerendering using the community-maintained BlazorWasmPrerendering.Build GitHub project.
  - Prerendering when integrating components into Razor Pages and MVC apps
- Authentication and authorization
  - Server-side threat mitigation: Cross-site scripting (XSS)
  - Server-side unauthorized content display while prerendering with a custom AuthenticationStateProvider
  - Blazor WebAssembly rendered component authentication with prerendering
- State management: Handle prerendering: Besides the *Handle prerendering* section, several of the article's other sections include remarks on prerendering.

ASP.NET Core Razor component generic type support

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes generic type support in Razor components.

If you're new to generic types, see Generic classes and methods (C# Guide) for general guidance on the use of generics before reading this article.

The example code in this article is only available for the latest .NET release in the Blazor sample apps.

Generic type parameter support

The @typeparam directive declares a generic type parameter for the generated component class:

```razor
@typeparam TItem
```

C# syntax with where type constraints is supported:

```razor
@typeparam TEntity where TEntity : IEntity
```

A RenderFragment generic type is supported, but components aren't supported as the generic type. To render components by type, consider using a DynamicComponent. For more information, see Dynamically-rendered ASP.NET Core Razor components.

In the following example, the `ListItems1` component is generically typed as `TExample`, which represents the type of the `ExampleList` collection.

`ListItems1.razor`:

razor

```razor
@typeparam TExample

<h2>List Items 1</h2>

@if (ExampleList is not null)
{
    <ul style="color:@Color">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>

    <p>
        Type of <code>TExample</code>: @typeof(TExample)
    </p>
}

@code {
    [Parameter]
    public string? Color { get; set; }

    [Parameter]
    public IEnumerable<TExample>? ExampleList { get; set; }
}
```

The following component renders two `ListItems1` components:

- String or integer data is assigned to the `ExampleList` parameter of each component.
- Type `string` or `int` that matches the type of the assigned data is set for the type parameter (`TExample`) of each component.

`Generics1.razor`:

razor

```razor
@page "/generics-1"

<PageTitle>Generics 1</PageTitle>

<h1>Generic Type Example 1</h1>

<ListItems1 Color="blue"
            ExampleList="@(new List<string> { "Item 1", "Item 2" })"
            TExample="string" />

<ListItems1 Color="red"
```

```
            ExampleList="@(new List<int> { 1, 2 })"
            TExample="int" />
```

For more information, see Razor syntax reference for ASP.NET Core. For an example of generic typing with templated components, see ASP.NET Core Blazor templated components.

Cascaded generic type support

An ancestor component can cascade a type parameter by name to descendants using the [CascadingTypeParameter] attribute. This attribute allows a generic type inference to use the specified type parameter automatically with descendants that have a type parameter with the same name.

By adding `@attribute [CascadingTypeParameter(...)]` to a component, the specified generic type argument is automatically used by descendants that:

- Are nested as child content for the component in the same `.razor` document.
- Also declare a @typeparam with the exact same name.
- Don't have another value explicitly supplied or implicitly inferred for the type parameter. If another value is supplied or inferred, it takes precedence over the cascaded generic type.

When receiving a cascaded type parameter, components obtain the parameter value from the closest ancestor that has a [CascadingTypeParameter] attribute with a matching name. Cascaded generic type parameters are overridden within a particular subtree.

Matching is only performed by name. Therefore, we recommend avoiding a cascaded generic type parameter with a generic name, for example `T` or `TItem`. If a developer opts into cascading a type parameter, they're implicitly promising that its name is unique enough not to clash with other cascaded type parameters from unrelated components.

Generic types can be cascaded to child components with either of the following approaches for ancestor (parent) components, which are demonstrated in the following two sub-sections:

- Explicitly set the cascaded generic type.
- Infer the cascaded generic type.

The following subsections provide examples of the preceding approaches using the following `ListDisplay1` component. The component receives and renders list data

generically typed as `TExample`. To make each instance of `ListDisplay1` stand out, an additional component parameter controls the color of the list.

`ListDisplay1.razor`:

razor

```razor
@typeparam TExample

@if (ExampleList is not null)
{
    <ul style="color:@Color">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>
}

@code {
    [Parameter]
    public string? Color { get; set; }

    [Parameter]
    public IEnumerable<TExample>? ExampleList { get; set; }
}
```

Explicit generic types based on ancestor components

The demonstration in this section cascades a type explicitly for `TExample`.

> ⓘ **Note**
>
> This section uses the preceding `ListDisplay1` component in the Cascaded generic type support section.

The following `ListItems2` component receives data and cascades a generic type parameter named `TExample` to its descendent components. In the upcoming parent component, the `ListItems2` component is used to display list data with the preceding `ListDisplay1` component.

`ListItems2.razor`:

razor

```
@attribute [CascadingTypeParameter(nameof(TExample))]
@typeparam TExample

<h2>List Items 2</h2>

@ChildContent

<p>
    Type of <code>TExample</code>: @typeof(TExample)
</p>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

The following parent component sets the child content (RenderFragment) of two `ListItems2` components specifying the `ListItems2` types (`TExample`), which are cascaded to child components. `ListDisplay1` components are rendered with the list item data shown in the example. String data is used with the first `ListItems2` component, and integer data is used with the second `ListItems2` component.

`Generics2.razor`:

razor

```
@page "/generics-2"

<PageTitle>Generics 2</PageTitle>

<h1>Generic Type Example 2</h1>

<ListItems2 TExample="string">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<string> { "Item 1", "Item 2" })"
/>
    <ListDisplay1 Color="red"
                  ExampleList="@(new List<string> { "Item 3", "Item 4" })"
/>
</ListItems2>

<ListItems2 TExample="int">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<int> { 1, 2 })" />
    <ListDisplay1 Color="red"
                  ExampleList="@(new List<int> { 3, 4 })" />
</ListItems2>
```

Specifying the type explicitly also allows the use of cascading values and parameters to provide data to child components, as the following demonstration shows.

`ListDisplay2.razor`:

razor

```razor
@typeparam TExample

@if (ExampleList is not null)
{
    <ul style="color:@Color">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>
}

@code {
    [Parameter]
    public string? Color { get; set; }

    [CascadingParameter]
    protected IEnumerable<TExample>? ExampleList { get; set; }
}
```

`ListItems3.razor`:

razor

```razor
@attribute [CascadingTypeParameter(nameof(TExample))]
@typeparam TExample

<h2>List Items 3</h2>

@ChildContent

@if (ExampleList is not null)
{
    <ul style="color:green">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>

    <p>
        Type of <code>TExample</code>: @typeof(TExample)
    </p>
}
```

```
@code {
    [CascadingParameter]
    protected IEnumerable<TExample>? ExampleList { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

When cascading the data in the following example, the type must be provided to the component.

`Generics3.razor`:

razor

```
@page "/generics-3"

<PageTitle>Generics 3</PageTitle>

<h1>Generic Type Example 3</h1>

<CascadingValue Value="stringData">
    <ListItems3 TExample="string">
        <ListDisplay2 Color="blue" />
        <ListDisplay2 Color="red" />
    </ListItems3>
</CascadingValue>

<CascadingValue Value="integerData">
    <ListItems3 TExample="int">
        <ListDisplay2 Color="blue" />
        <ListDisplay2 Color="red" />
    </ListItems3>
</CascadingValue>

@code {
    private List<string> stringData = new() { "Item 1", "Item 2" };
    private List<int> integerData = new() { 1, 2 };
}
```

When multiple generic types are cascaded, values for all generic types in the set must be passed. In the following example, `TItem`, `TValue`, and `TEdit` are `GridColumn` generic types, but the parent component that places `GridColumn` doesn't specify the `TItem` type:

razor

```
<GridColumn TValue="string" TEdit="TextEdit" />
```

The preceding example generates a compile-time error that the `GridColumn` component is missing the `TItem` type parameter. Valid code specifies all of the types:

razor

```
<GridColumn TValue="string" TEdit="TextEdit" TItem="User" />
```

Infer generic types based on ancestor components

The demonstration in this section cascades a type inferred for `TExample`.

> **Note**
>
> This section uses the `ListDisplay` component in the Cascaded generic type support section.

`ListItems4.razor`:

razor

```
@attribute [CascadingTypeParameter(nameof(TExample))]
@typeparam TExample

<h2>List Items 4</h2>

@ChildContent

@if (ExampleList is not null)
{
    <ul style="color:green">
        @foreach (var item in ExampleList)
        {
            <li>@item</li>
        }
    </ul>

    <p>
        Type of <code>TExample</code>: @typeof(TExample)
    </p>
}

@code {
    [Parameter]
    public IEnumerable<TExample>? ExampleList { get; set; }

    [Parameter]
```

```
    public RenderFragment? ChildContent { get; set; }
}
```

The following component with inferred cascaded types provides different data for display.

`Generics4.razor`:

razor

```
@page "/generics-4"

<PageTitle>Generics 4</PageTitle>

<h1>Generic Type Example 4</h1>

<ListItems4 ExampleList="@(new List<string> { "Item 5", "Item 6" })">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<string> { "Item 1", "Item 2" })"
/>
    <ListDisplay1 Color="red"
                  ExampleList="@(new List<string> { "Item 3", "Item 4" })"
/>
</ListItems4>

<ListItems4 ExampleList="@(new List<int> { 5, 6 })">
    <ListDisplay1 Color="blue"
                  ExampleList="@(new List<int> { 1, 2 })" />
    <ListDisplay1 Color="red"
                  ExampleList="@(new List<int> { 3, 4 })" />
</ListItems4>
```

The following component with inferred cascaded types provides the same data for display. The following example directly assigns the data to the components.

`Generics5.razor`:

razor

```
@page "/generics-5"

<PageTitle>Generics 5</PageTitle>

<h1>Generic Type Example 5</h1>

<ListItems4 ExampleList="stringData">
    <ListDisplay1 Color="blue" ExampleList="stringData" />
    <ListDisplay1 Color="red" ExampleList="stringData" />
</ListItems4>
```

```
<ListItems4 ExampleList="integerData">
    <ListDisplay1 Color="blue" ExampleList="integerData" />
    <ListDisplay1 Color="red" ExampleList="integerData" />
</ListItems4>

@code {
    private List<string> stringData = new() { "Item 1", "Item 2" };
    private List<int> integerData = new() { 1, 2 };
}
```

ASP.NET Core Blazor synchronization context

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Blazor uses a synchronization context (SynchronizationContext) to enforce a single logical thread of execution. A component's lifecycle methods and event callbacks raised by Blazor are executed on the synchronization context.

Blazor's server-side synchronization context attempts to emulate a single-threaded environment so that it closely matches the WebAssembly model in the browser, which is single threaded. This emulation is scoped only to an individual circuit, meaning two different circuits can run in parallel. At any given point in time within a circuit, work is performed on exactly one thread, which yields the impression of a single logical thread. No two operations execute concurrently within the same circuit.

Avoid thread-blocking calls

Generally, don't call the following methods in components. The following methods block the execution thread and thus block the app from resuming work until the underlying Task is complete:

- Result
- Wait
- WaitAny
- WaitAll
- Sleep
- GetResult

> **Note**
>
> Blazor documentation examples that use the thread-blocking methods mentioned in this section are only using the methods for demonstration purposes, not as

recommended coding guidance. For example, a few component code demonstrations simulate a long-running process by calling Thread.Sleep.

Invoke component methods externally to update state

In the event a component must be updated based on an external event, such as a timer or other notification, use the `InvokeAsync` method, which dispatches code execution to Blazor's synchronization context. For example, consider the following *notifier service* that can notify any listening component about updated state. The `Update` method can be called from anywhere in the app.

`TimerService.cs`:

C#

```
namespace BlazorSample;

public class TimerService(NotifierService notifier,
    ILogger<TimerService> logger) : IDisposable
{
    private int elapsedCount;
    private readonly static TimeSpan heartbeatTickRate =
TimeSpan.FromSeconds(5);
    private readonly ILogger<TimerService> logger = logger;
    private readonly NotifierService notifier = notifier;
    private PeriodicTimer? timer;

    public async Task Start()
    {
        if (timer is null)
        {
            timer = new(heartbeatTickRate);
            logger.LogInformation("Started");

            using (timer)
            {
                while (await timer.WaitForNextTickAsync())
                {
                    elapsedCount += 1;
                    await notifier.Update("elapsedCount", elapsedCount);
                    logger.LogInformation("ElapsedCount {Count}",
elapsedCount);
                }
            }
        }
    }
```

```
    public void Dispose()
    {
        timer?.Dispose();

        // The following prevents derived types that introduce a
        // finalizer from needing to re-implement IDisposable.
        GC.SuppressFinalize(this);
    }
}
```

`NotifierService.cs`:

C#

```
namespace BlazorSample;

public class NotifierService
{
    public async Task Update(string key, int value)
    {
        if (Notify != null)
        {
            await Notify.Invoke(key, value);
        }
    }

    public event Func<string, int, Task>? Notify;
}
```

Register the services:

- For client-side development, register the services as singletons in the client-side `Program` file:

  C#

  ```
  builder.Services.AddSingleton<NotifierService>();
  builder.Services.AddSingleton<TimerService>();
  ```

- For server-side development, register the services as scoped in the server `Program` file:

  C#

  ```
  builder.Services.AddScoped<NotifierService>();
  builder.Services.AddScoped<TimerService>();
  ```

Use the `NotifierService` to update a component.

`Notifications.razor`:

razor

```razor
@page "/notifications"
@implements IDisposable
@inject NotifierService Notifier
@inject TimerService Timer

<PageTitle>Notifications</PageTitle>

<h1>Notifications Example</h1>

<h2>Timer Service</h2>

<button @onclick="StartTimer">Start Timer</button>

<h2>Notifications</h2>

<p>
    Status:
    @if (lastNotification.key is not null)
    {
        <span>@lastNotification.key = @lastNotification.value</span>
    }
    else
    {
        <span>Awaiting notification</span>
    }
</p>

@code {
    private (string key, int value) lastNotification;

    protected override void OnInitialized() => Notifier.Notify += OnNotify;

    public async Task OnNotify(string key, int value)
    {
        await InvokeAsync(() =>
        {
            lastNotification = (key, value);
            StateHasChanged();
        });
    }

    private void StartTimer() => _ = Task.Run(Timer.Start);

    public void Dispose() => Notifier.Notify -= OnNotify;
}
```

In the preceding example:

- The timer is initiated outside of Blazor's synchronization context with `_ = Task.Run(Timer.Start)`.
- `NotifierService` invokes the component's `OnNotify` method. `InvokeAsync` is used to switch to the correct context and enqueue a rerender. For more information, see ASP.NET Core Razor component rendering.
- The component implements IDisposable. The `OnNotify` delegate is unsubscribed in the `Dispose` method, which is called by the framework when the component is disposed. For more information, see ASP.NET Core Razor component lifecycle.

> **Important**
>
> If a Razor component defines an event that's triggered from a background thread, the component might be required to capture and restore the execution context (ExecutionContext) at the time the handler is registered. For more information, see Calling InvokeAsync(StateHasChanged) causes page to fallback to default culture (dotnet/aspnetcore #28521).

To dispatch caught exceptions from the background `TimerService` to the component to treat the exceptions like normal lifecycle event exceptions, see the Handle caught exceptions outside of a Razor component's lifecycle section.

Handle caught exceptions outside of a Razor component's lifecycle

Use ComponentBase.DispatchExceptionAsync in a Razor component to process exceptions thrown outside of the component's lifecycle call stack. This permits the component's code to treat exceptions as though they're lifecycle method exceptions. Thereafter, Blazor's error handling mechanisms, such as error boundaries, can process the exceptions.

> **Note**
>
> ComponentBase.DispatchExceptionAsync is used in Razor component files (`.razor`) that inherit from ComponentBase. When creating components that implement IComponent directly, use RenderHandle.DispatchExceptionAsync.

To handle caught exceptions outside of a Razor component's lifecycle, pass the exception to DispatchExceptionAsync and await the result:

C#

```
try
{
    ...
}
catch (Exception ex)
{
    await DispatchExceptionAsync(ex);
}
```

A common scenario for the preceding approach is when a component starts an asynchronous operation but doesn't await a Task, often called the *fire and forget* pattern because the method is *fired* (started) and the result of the method is *forgotten* (thrown away). If the operation fails, you may want the component to treat the failure as a component lifecycle exception for any of the following goals:

- Put the component into a faulted state, for example, to trigger an error boundary.
- Terminate the circuit if there's no error boundary.
- Trigger the same logging that occurs for lifecycle exceptions.

In the following example, the user selects the **Send report** button to trigger a background method, `ReportSender.SendAsync`, that sends a report. In most cases, a component awaits the Task of an asynchronous call and updates the UI to indicate the operation completed. In the following example, the `SendReport` method doesn't await a Task and doesn't report the result to the user. Because the component intentionally discards the Task in `SendReport`, any asynchronous failures occur off of the normal lifecycle call stack, hence aren't seen by Blazor:

razor

```
<button @onclick="SendReport">Send report</button>

@code {
    private void SendReport()
    {
        _ = ReportSender.SendAsync();
    }
}
```

To treat failures like lifecycle method exceptions, explicitly dispatch exceptions back to the component with DispatchExceptionAsync, as the following example demonstrates:

razor

```
<button @onclick="SendReport">Send report</button>

@code {
    private void SendReport()
    {
        _ = SendReportAsync();
    }

    private async Task SendReportAsync()
    {
        try
        {
            await ReportSender.SendAsync();
        }
        catch (Exception ex)
        {
            await DispatchExceptionAsync(ex);
        }
    }
}
```

An alternative approach leverages Task.Run:

C#

```
private void SendReport()
{
    _ = Task.Run(async () =>
    {
        try
        {
            await ReportSender.SendAsync();
        }
        catch (Exception ex)
        {
            await DispatchExceptionAsync(ex);
        }
    });
}
```

For a working demonstration, implement the timer notification example in Invoke component methods externally to update state. In a Blazor app, add the following files from the timer notification example and register the services in the `Program` file as the section explains:

- `TimerService.cs`
- `NotifierService.cs`
- `Notifications.razor`

The example uses a timer outside of a Razor component's lifecycle, where an unhandled exception normally isn't processed by Blazor's error handling mechanisms, such as an error boundary.

First, change the code in `TimerService.cs` to create an artificial exception outside of the component's lifecycle. In the `while` loop of `TimerService.cs`, throw an exception when the `elapsedCount` reaches a value of two:

C#

```
if (elapsedCount == 2)
{
    throw new Exception("I threw an exception! Somebody help me!");
}
```

Place an error boundary in the app's main layout. Replace the `<article>...</article>` markup with the following markup.

In `MainLayout.razor`:

razor

```
<article class="content px-4">
    <ErrorBoundary>
        <ChildContent>
            @Body
        </ChildContent>
        <ErrorContent>
            <p class="alert alert-danger" role="alert">
                Oh, dear! Oh, my! - George Takei
            </p>
        </ErrorContent>
    </ErrorBoundary>
</article>
```

In Blazor Web Apps with the error boundary only applied to a static `MainLayout` component, the boundary is only active during the static server-side rendering (static SSR) phase. The boundary doesn't activate just because a component further down the component hierarchy is interactive. To enable interactivity broadly for the `MainLayout` component and the rest of the components further down the component hierarchy, enable interactive rendering for the `HeadOutlet` and `Routes` component instances in the `App` component (`Components/App.razor`). The following example adopts the Interactive Server (`InteractiveServer`) render mode:

razor

```
<HeadOutlet @rendermode="InteractiveServer" />

...

<Routes @rendermode="InteractiveServer" />
```

If you run the app at this point, the exception is thrown when the elapsed count reaches a value of two. However, the UI doesn't change. The error boundary doesn't show the error content.

To dispatch exceptions from the timer service back to the `Notifications` component, the following changes are made to the component:

- Start the timer in a try-catch statement. In the `catch` clause of the `try-catch` block, exceptions are dispatched back to the component by passing the Exception to DispatchExceptionAsync and awaiting the result.
- In the `StartTimer` method, start the asynchronous timer service in the Action delegate of Task.Run and intentionally discard the returned Task.

The `StartTimer` method of the `Notifications` component (`Notifications.razor`):

C#

```
private void StartTimer()
{
    _ = Task.Run(async () =>
    {
        try
        {
            await Timer.Start();
        }
        catch (Exception ex)
        {
            await DispatchExceptionAsync(ex);
        }
    });
}
```

When the timer service executes and reaches a count of two, the exception is dispatched to the Razor component, which in turn triggers the error boundary to display the error content of the `<ErrorBoundary>` in the `MainLayout` component:

> Oh, dear! Oh, my! - George Takei

Retain element, component, and model relationships in ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use the `@key` directive attribute to retain element, component, and model relationships when rendering and the elements or components subsequently change.

Use of the `@key` directive attribute

When rendering a list of elements or components and the elements or components subsequently change, Blazor must decide which of the previous elements or components are retained and how model objects should map to them. Normally, this process is automatic and sufficient for general rendering, but there are often cases where controlling the process using the @key directive attribute is required.

Consider the following example that demonstrates a collection mapping problem that's solved by using @key.

For the following components:

- The `Details` component receives data (`Data`) from the parent component, which is displayed in an `<input>` element. Any given displayed `<input>` element can receive the focus of the page from the user when they select one of the `<input>` elements.
- The parent component creates a list of person objects for display using the `Details` component. Every three seconds, a new person is added to the collection.

This demonstration allows you to:

- Select an `<input>` from among several rendered `Details` components.
- Study the behavior of the page's focus as the people collection automatically grows.

`Details.razor`:

razor

```razor
<input value="@Data" />

@code {
    [Parameter]
    public string? Data { get; set; }
}
```

In the following parent component, each iteration of adding a person in `OnTimerCallback` results in Blazor rebuilding the entire collection. The page's focus remains on the *same index* position of `<input>` elements, so the focus shifts each time a person is added. *Shifting the focus away from what the user selected isn't desirable behavior.* After demonstrating the poor behavior with the following component, the @key directive attribute is used to improve the user's experience.

`People.razor`:

razor

```razor
@page "/people"
@using System.Timers
@implements IDisposable

<PageTitle>People</PageTitle>

<h1>People Example</h1>

@foreach (var person in people)
{
    <Details Data="@person.Data" />
}

@code {
    private Timer timer = new Timer(3000);

    public List<Person> people =
        new()
        {
            { new Person { Data = "Person 1" } },
            { new Person { Data = "Person 2" } },
            { new Person { Data = "Person 3" } }
        };

    protected override void OnInitialized()
    {
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
    }

    private void OnTimerCallback()
```

```
    {
        _ = InvokeAsync(() =>
        {
            people.Insert(0,
                new Person
                {
                    Data = $"INSERTED {DateTime.Now.ToString("hh:mm:ss
tt")}"
                });
            StateHasChanged();
        });
    }

    public void Dispose() => timer.Dispose();

    public class Person
    {
        public string? Data { get; set; }
    }
}
```

The contents of the `people` collection changes with inserted, deleted, or re-ordered entries. Rerendering can lead to visible behavior differences. For example, each time a person is inserted into the `people` collection, the user's focus is lost.

The mapping process of elements or components to a collection can be controlled with the @key directive attribute. Use of @key guarantees the preservation of elements or components based on the key's value. If the `Details` component in the preceding example is keyed on the `person` item, Blazor ignores rerendering `Details` components that haven't changed.

To modify the parent component to use the @key directive attribute with the `people` collection, update the `<Details>` element to the following:

razor

```
<Details @key="person" Data="@person.Data" />
```

When the `people` collection changes, the association between `Details` instances and `person` instances is retained. When a `Person` is inserted at the beginning of the collection, one new `Details` instance is inserted at that corresponding position. Other instances are left unchanged. Therefore, the user's focus isn't lost as people are added to the collection.

Other collection updates exhibit the same behavior when the @key directive attribute is used:

- If an instance is deleted from the collection, only the corresponding component instance is removed from the UI. Other instances are left unchanged.
- If collection entries are re-ordered, the corresponding component instances are preserved and re-ordered in the UI.

> **ⓘ Important**
>
> Keys are local to each container element or component. Keys aren't compared globally across the document.

When to use `@key`

Typically, it makes sense to use @key whenever a list is rendered (for example, in a foreach block) and a suitable value exists to define the @key.

You can also use @key to preserve an element or component subtree when an object doesn't change, as the following examples show.

Example 1:

razor

```razor
<li @key="person">
    <input value="@person.Data" />
</li>
```

Example 2:

razor

```razor
<div @key="person">
    @* other HTML elements *@
</div>
```

If an `person` instance changes, the @key attribute directive forces Blazor to:

- Discard the entire `<li>` or `<div>` and their descendants.
- Rebuild the subtree within the UI with new elements and components.

This is useful to guarantee that no UI state is preserved when the collection changes within a subtree.

Scope of `@key`

The @key attribute directive is scoped to its own siblings within its parent.

Consider the following example. The `first` and `second` keys are compared against each other within the same scope of the outer `<div>` element:

razor

```
<div>
    <div @key="first">...</div>
    <div @key="second">...</div>
</div>
```

The following example demonstrates `first` and `second` keys in their own scopes, unrelated to each other and without influence on each other. Each @key scope only applies to its parent `<div>` element, not across the parent `<div>` elements:

razor

```
<div>
    <div @key="first">...</div>
</div>
<div>
    <div @key="second">...</div>
</div>
```

For the `Details` component shown earlier, the following examples render `person` data within the same @key scope and demonstrate typical use cases for @key:

razor

```
<div>
    @foreach (var person in people)
    {
        <Details @key="person" Data="@person.Data" />
    }
</div>
```

razor

```
@foreach (var person in people)
{
    <div @key="person">
        <Details Data="@person.Data" />
```

```
    </div>
}
```

razor

```
<ol>
    @foreach (var person in people)
    {
        <li @key="person">
            <Details Data="@person.Data" />
        </li>
    }
</ol>
```

The following examples only scope @key to the `<div>` or `<li>` element that surrounds each `Details` component instance. Therefore, `person` data for each member of the `people` collection is **not** keyed on each `person` instance across the rendered `Details` components. Avoid the following patterns when using @key:

razor

```
@foreach (var person in people)
{
    <div>
        <Details @key="person" Data="@person.Data" />
    </div>
}
```

razor

```
<ol>
    @foreach (var person in people)
    {
        <li>
            <Details @key="person" Data="@person.Data" />
        </li>
    }
</ol>
```

When not to use `@key`

There's a performance cost when rendering with @key. The performance cost isn't large, but only specify @key if preserving the element or component benefits the app.

Even if @key isn't used, Blazor preserves child element and component instances as much as possible. The only advantage to using @key is control over *how* model instances are mapped to the preserved component instances, instead of Blazor selecting the mapping.

Values to use for `@key`

Generally, it makes sense to supply one of the following values for @key:

- Model object instances. For example, the `Person` instance (`person`) was used in the earlier example. This ensures preservation based on object reference equality.
- Unique identifiers. For example, unique identifiers can be based on primary key values of type `int`, `string`, or `Guid`.

Ensure that values used for @key don't clash. If clashing values are detected within the same parent element, Blazor throws an exception because it can't deterministically map old elements or components to new elements or components. Only use distinct values, such as object instances or primary key values.

Avoid overwriting parameters in ASP.NET Core Blazor

Article • 12/13/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Robert Haken

This article explains how to avoid overwriting parameters in Blazor apps during rerendering.

Overwritten parameters

The Blazor framework generally imposes safe parent-to-child parameter assignment:

- Parameters aren't overwritten unexpectedly.
- Side effects are minimized. For example, additional renders are avoided because they may create infinite rendering loops.

A child component receives new parameter values that possibly overwrite existing values when the parent component rerenders. Accidentally overwriting parameter values in a child component often occurs when developing the component with one or more data-bound parameters and the developer writes directly to a parameter in the child:

- The child component is rendered with one or more parameter values from the parent component.
- The child writes directly to the value of a parameter.
- The parent component rerenders and overwrites the value of the child's parameter.

The potential for overwriting parameter values extends into the child component's property `set` accessors, too.

> **Important**
>
> Our general guidance is not to create components that directly write to their own parameters after the component is rendered for the first time.

Consider the following `ShowMoreExpander` component that:

- Renders the title.
- Shows the child content when selected.
- Allows you to set initial state with a component parameter (`InitiallyExpanded`).

After the following `ShowMoreExpander` component demonstrates an overwritten parameter, a modified `ShowMoreExpander` component is shown to demonstrate the correct approach for this scenario. The following examples can be placed in a local sample app to experience the behaviors described.

`ShowMoreExpander.razor`:

```
<div @onclick="ShowMore" class="card bg-light mb-3" style="width:30rem">
    <div class="card-header">
        <h2 class="card-title">Show more (<code>Expanded</code> =
@InitiallyExpanded)</h2>
    </div>
    @if (InitiallyExpanded)
    {
        <div class="card-body">
            <p class="card-text">@ChildContent</p>
        </div>
    }
</div>

@code {
    [Parameter]
    public bool InitiallyExpanded { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    private void ShowMore() => InitiallyExpanded = true;
}
```

The `ShowMoreExpander` component is added to the following `Expanders` parent component that may call StateHasChanged:

- Calling StateHasChanged in developer code notifies a component that its state has changed and typically enqueues component rerendering to update the UI. StateHasChanged is covered in more detail later in ASP.NET Core Razor component lifecycle and ASP.NET Core Razor component rendering.
- The button's `@onclick` directive attribute attaches an event handler to the button's `onclick` event. Event handling is covered in more detail later in ASP.NET Core Blazor event handling.

`Expanders.razor`:

```
@page "/expanders"

<PageTitle>Expanders</PageTitle>

<h1>Expanders Example</h1>

<ShowMoreExpander InitiallyExpanded="false">
    Expander 1 content
</ShowMoreExpander>

<ShowMoreExpander InitiallyExpanded="false" />

<button @onclick="StateHasChanged">Call StateHasChanged</button>
```

Initially, the `ShowMoreExpander` components behave independently when their `InitiallyExpanded` properties are set. The child components maintain their states as expected.

If StateHasChanged is called in a parent component, the Blazor framework rerenders child components if their parameters might have changed:

- For a group of parameter types that Blazor explicitly checks, Blazor rerenders a child component if it detects that any of the parameters have changed.
- For unchecked parameter types, Blazor rerenders the child component *regardless of whether or not the parameters have changed*. Child content falls into this category of parameter types because child content is of type RenderFragment, which is a delegate that refers to other mutable objects.

For the `Expanders` component:

- The first `ShowMoreExpander` component sets child content in a potentially mutable RenderFragment, so a call to StateHasChanged in the parent component automatically rerenders the component and potentially overwrites the value of `InitiallyExpanded` to its initial value of `false`.
- The second `ShowMoreExpander` component doesn't set child content. Therefore, a potentially mutable RenderFragment doesn't exist. A call to StateHasChanged in the parent component doesn't automatically rerender the child component, so the component's `InitiallyExpanded` value isn't overwritten.

To maintain state in the preceding scenario, use a *private field* in the `ShowMoreExpander` component to maintain its state.

The following revised `ShowMoreExpander` component:

- Accepts the `InitiallyExpanded` component parameter value from the parent.
- Assigns the component parameter value to a *private field* (`expanded`) in the OnInitialized event.
- Uses the private field to maintain its internal toggle state, which demonstrates how to avoid writing directly to a parameter.

> **Note**
>
> The advice in this section extends to similar logic in component parameter `set` accessors, which can result in similar undesirable side effects.

`ShowMoreExpander.razor`:

```
<div @onclick="Expand" class="card bg-light mb-3" style="width:30rem">
    <div class="card-header">
        <h2 class="card-title">Show more (<code>Expanded</code> = @expanded)
</h2>
    </div>
    @if (expanded)
    {
        <div class="card-body">
            <p class="card-text">@ChildContent</p>
        </div>
    }
</div>

@code {
    private bool expanded;

    [Parameter]
    public bool InitiallyExpanded { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    protected override void OnInitialized() => expanded = InitiallyExpanded;

    private void Expand() => expanded = true;
}
```

> **Note**
>
> The revised `ShowMoreExpander` doesn't reflect changes to the `InitiallyExpanded` parameter after initialization (`OnInitialized`). In certain scenarios, an already initialized component might receive new parameter values. This can happen, for example, in a primary-subordinate view where the same component is used to

render different detail views or when the `/item/{id}` route parameter changes to display a different item.

Consider following `ToggleExpander` component that:

- Allows you to change the state both from inside and outside.
- Handles new parameter values even if the same component instance is reused.

`ToggleExpander.razor`:

```
<div class="card bg-light mb-3" style="width:30rem">
    <div @onclick="Toggle" class="card-header">
        <h2 class="card-title">Toggle (<code>Expanded</code> = @expanded)
</h2>
    </div>
    @if (expanded)
    {
        <div class="card-body">
            <p class="card-text">@ChildContent</p>
        </div>
    }
</div>

@code {
    [Parameter]
    public bool Expanded { get; set; }

    [Parameter]
    public EventCallback<bool> ExpandedChanged { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    private bool expanded;

    protected override void OnParametersSet() => expanded = Expanded;

    private async Task Toggle()
    {
        expanded = !expanded;
        await ExpandedChanged.InvokeAsync(expanded);
    }
}
```

The `ToggleExpander` component should be used with the `@bind-Expanded="{field}"` binding syntax, allowing two-way synchronization of the parameter.

`ExpandersToggle.razor`:

```
@page "/expanders-toggle"

<PageTitle>Expanders Toggle</PageTitle>

<h1>Expanders Toggle</h1>

<ToggleExpander @bind-Expanded="expanded">
    Expander content
</ToggleExpander>

<button @onclick="Toggle">Toggle</button>

<button @onclick="StateHasChanged">Call StateHasChanged</button>

@code {
    private bool expanded;

    private void Toggle() => expanded = !expanded;
}
```

For more information on parent-child binding, see the following resources:

- Binding with component parameters
- Bind across more than two components
- Blazor Two Way Binding Error (dotnet/aspnetcore #24599)

For more information on change detection, including information on the exact types that Blazor checks, see ASP.NET Core Razor component rendering.

ASP.NET Core Blazor attribute splatting and arbitrary parameters

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Components can capture and render additional attributes in addition to the component's declared parameters. Additional attributes can be captured in a dictionary and then applied to an element, called *splatting*, when the component is rendered using the @attributes Razor directive attribute. This scenario is useful for defining a component that produces a markup element that supports a variety of customizations. For example, it can be tedious to define attributes separately for an `<input>` that supports many parameters.

Attribute splatting

In the following `Splat` component:

- The first `<input>` element (`id="useIndividualParams"`) uses individual component parameters.
- The second `<input>` element (`id="useAttributesDict"`) uses attribute splatting.

`Splat.razor`:

razor

```
@page "/splat"

<PageTitle>SPLAT!</PageTitle>

<h1>Splat Parameters Example</h1>

<input id="useIndividualParams"
       maxlength="@maxlength"
       placeholder="@placeholder"
       required="@required"
       size="@size" />

<input id="useAttributesDict"
```

```
        @attributes="InputAttributes" />

@code {
    private string maxlength = "10";
    private string placeholder = "Input placeholder text";
    private string required = "required";
    private string size = "50";

    private Dictionary<string, object> InputAttributes { get; set; } =
        new()
        {
            { "maxlength", "10" },
            { "placeholder", "Input placeholder text" },
            { "required", "required" },
            { "size", "50" }
        };
}
```

The rendered `<input>` elements in the webpage are identical:

HTML

```
<input id="useIndividualParams"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">

<input id="useAttributesDict"
       maxlength="10"
       placeholder="Input placeholder text"
       required="required"
       size="50">
```

Arbitrary attributes

To accept arbitrary attributes, define a component parameter with the CaptureUnmatchedValues property set to `true`:

razor

```
@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object>? InputAttributes { get; set; }
}
```

The CaptureUnmatchedValues property on [Parameter] allows the parameter to match all attributes that don't match any other parameter. A component can only define a

single parameter with CaptureUnmatchedValues. The property type used with CaptureUnmatchedValues must be assignable from Dictionary<string, object> with string keys. Use of IEnumerable<KeyValuePair<string, object>> or IReadOnlyDictionary<string, object> are also options in this scenario.

The position of @attributes relative to the position of element attributes is important. When @attributes are splatted on the element, the attributes are processed from right to left (last to first). Consider the following example of a parent component that consumes a child component:

`AttributeOrderChild1.razor`:

razor

```razor
<div @attributes="AdditionalAttributes" extra="5" />

@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public IDictionary<string, object>? AdditionalAttributes { get; set; }
}
```

`AttributeOrder1.razor`:

razor

```razor
@page "/attribute-order-1"

<PageTitle>Attribute Order 1</PageTitle>

<h1>Attribute Order Example 1</h1>

<AttributeOrderChild1 extra="10" />

<p>
    View the HTML markup in your browser to inspect the attributes on
    the AttributeOrderChild1 component.
</p>
```

The `AttributeOrderChild1` component's `extra` attribute is set to the right of @attributes. The `AttributeOrderParent1` component's rendered `<div>` contains `extra="5"` when passed through the additional attribute because the attributes are processed right to left (last to first):

HTML

```html
<div extra="5" />
```

In the following example, the order of `extra` and @attributes is reversed in the child component's `<div>`:

`AttributeOrderChild2.razor`:

razor

```razor
<div extra="5" @attributes="AdditionalAttributes" />

@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public IDictionary<string, object>? AdditionalAttributes { get; set; }
}
```

`AttributeOrder2.razor`:

razor

```razor
@page "/attribute-order-2"

<PageTitle>Attribute Order 2</PageTitle>

<h1>Attribute Order Example 2</h1>

<AttributeOrderChild2 extra="10" />

<p>
    View the HTML markup in your browser to inspect the attributes on
    the AttributeOrderChild2 component.
</p>
```

The `<div>` in the parent component's rendered webpage contains `extra="10"` when passed through the additional attribute:

HTML

```html
<div extra="10" />
```

ASP.NET Core Blazor layouts

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to create reusable layout components for Blazor apps.

Usefulness of Blazor layouts

Some app elements, such as menus, copyright messages, and company logos, are usually part of app's overall presentation. Placing a copy of the markup for these elements into all of the components of an app isn't efficient. Every time that one of these elements is updated, every component that uses the element must be updated. This approach is costly to maintain and can lead to inconsistent content if an update is missed. *Layouts* solve these problems.

A Blazor layout is a Razor component that shares markup with components that reference it. Layouts can use data binding, dependency injection, and other features of components.

Layout components

Create a layout component

To create a layout component:

- Create a Razor component defined by a Razor template or C# code. Layout components based on a Razor template use the `.razor` file extension just like ordinary Razor components. Because layout components are shared across an app's components, they're usually placed in the app's `Shared` or `Layout` folder. However, layouts can be placed in any location accessible to the components that use it. For example, a layout can be placed in the same folder as the components that use it.
- Inherit the component from LayoutComponentBase. The LayoutComponentBase defines a Body property (RenderFragment type) for the rendered content inside

the layout.

- Use the Razor syntax `@Body` to specify the location in the layout markup where the content is rendered.

> ⓘ **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

The following `DoctorWhoLayout` component shows the Razor template of a layout component. The layout inherits LayoutComponentBase and sets the `@Body` between the navigation bar (`<nav>...</nav>`) and the footer (`<footer>...</footer>`).

`DoctorWhoLayout.razor`:

razor

```razor
@inherits LayoutComponentBase

<PageTitle>Doctor Who® Database</PageTitle>

<header>
    <h1>Doctor Who® Database</h1>
</header>

<nav>
    <a href="main-list">Main Episode List</a>
    <a href="search">Search</a>
    <a href="new">Add Episode</a>
</nav>

@Body

<footer>
    @TrademarkMessage
</footer>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/ https://www.bbc.com";
}
```

`MainLayout` component

In an app created from a Blazor project template, the `MainLayout` component is the app's default layout. Blazor's layout adopts the Flexbox layout model (MDN

documentation) (W3C specification).

Blazor's CSS isolation feature applies isolated CSS styles to the `MainLayout` component. By convention, the styles are provided by the accompanying stylesheet of the same name, `MainLayout.razor.css`. The ASP.NET Core framework implementation of the stylesheet is available for inspection in the ASP.NET Core reference source (`dotnet/aspnetcore` GitHub repository):

- Blazor Web App MainLayout.razor.css
- Blazor WebAssembly MainLayout.razor.css

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Apply a layout

Make the layout namespace available

Layout file locations and namespaces changed over time for the Blazor framework. Depending on the version of Blazor and type of Blazor app that you're building, you may need to indicate the layout's namespace when using it. When referencing a layout implementation and the layout isn't found without indicating the layout's namespace, take any of the following approaches:

- Add an `@using` directive to the `_Imports.razor` file for the location of the layouts. In the following example, a folder of layouts with the name `Layout` is inside a `Components` folder, and the app's namespace is `BlazorSample`:

  ```razor
  @using BlazorSample.Components.Layout
  ```

- Add an `@using` directive at the top of the component definition where the layout is used:

  ```
  ```

razor

```razor
@using BlazorSample.Components.Layout
@layout DoctorWhoLayout
```

- Fully qualify the namespace of the layout where it's used:

razor

```razor
@layout BlazorSample.Components.Layout.DoctorWhoLayout
```

Apply a layout to a component

Use the @layout Razor directive to apply a layout to a routable Razor component that has an @page directive. The compiler converts `@layout` into a LayoutAttribute and applies the attribute to the component class.

The content of the following `Episodes` component is inserted into the `DoctorWhoLayout` at the position of `@Body`.

`Episodes.razor`:

razor

```razor
@page "/episodes"
@layout DoctorWhoLayout

<h2>Doctor Who® Episodes</h2>

<ul>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vfknq">
            <em>The Ribos Operation</em>
        </a>
    </li>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vfdsb">
            <em>The Sunmakers</em>
        </a>
    </li>
    <li>
        <a href="https://www.bbc.co.uk/programmes/p00vhc26">
            <em>Nightmare of Eden</em>
        </a>
    </li>
</ul>
```

The following rendered HTML markup is produced by the preceding `DoctorWhoLayout` and `Episodes` component. Extraneous markup doesn't appear in order to focus on the content provided by the two components involved:

- The H1 "database" heading (`<h1>...</h1>`) in the header (`<header>...</header>`), navigation bar (`<nav>...</nav>`), and trademark information in the footer (`<footer>...</footer>`) come from the `DoctorWhoLayout` component.
- The H2 "episodes" heading (`<h2>...</h2>`) and episode list (`<ul>...</ul>`) come from the `Episodes` component.

HTML

```
<header>
    <h1 ...>...</h1>
</header>

<nav>
    ...
</nav>

<h2>...</h2>

<ul>
    <li>...</li>
    <li>...</li>
    <li>...</li>
</ul>

<footer>
    ...
</footer>
```

Specifying the layout directly in a component overrides a *default layout*:

- Set by an `@layout` directive imported from an `_Imports.razor` file, as described in the following Apply a layout to a folder of components section.
- Set as the app's default layout, as described in the Apply a default layout to an app section later in this article.

Apply a layout to a folder of components

Every folder of an app can optionally contain a template file named `_Imports.razor`. The compiler includes the directives specified in the imports file in all of the Razor templates in the same folder and recursively in all of its subfolders. Therefore, an `_Imports.razor` file containing `@layout DoctorWhoLayout` ensures that all of the components in a folder

use the `DoctorWhoLayout` component. There's no need to repeatedly add `@layout DoctorWhoLayout` to all of the Razor components (`.razor`) within the folder and subfolders.

`_Imports.razor`:

```razor
@layout DoctorWhoLayout
...
```

The `_Imports.razor` file is similar to the _ViewImports.cshtml file for Razor views and pages but applied specifically to Razor component files.

Specifying a layout in `_Imports.razor` overrides a layout specified as the router's default app layout, which is described in the following section.

> **⚠ Warning**
>
> Do **not** add a Razor `@layout` directive to the root `_Imports.razor` file, which results in an infinite loop of layouts. To control the default app layout, specify the layout in the Router component. For more information, see the following Apply a default layout to an app section.

> **ⓘ Note**
>
> The @layout Razor directive only applies a layout to routable Razor components with an @page directive.

Apply a default layout to an app

Specify the default app layout in the Router component's RouteView component. Use the DefaultLayout parameter to set the layout type:

```razor
<RouteView RouteData="routeData" DefaultLayout="typeof({LAYOUT})" />
```

In the preceding example, the `{LAYOUT}` placeholder is the layout (for example, `DoctorWhoLayout` if the layout file name is `DoctorWhoLayout.razor`). You may need to

idenfity the layout's namespace depending on the .NET version and type of Blazor app. For more information, see the Make the layout namespace available section.

Specifying the layout as a default layout in the Router component's RouteView is a useful practice because you can override the layout on a per-component or per-folder basis, as described in the preceding sections of this article. We recommend using the Router component to set the app's default layout because it's the most general and flexible approach for using layouts.

Apply a layout to arbitrary content (`LayoutView` component)

To set a layout for arbitrary Razor template content, specify the layout with a LayoutView component. You can use a LayoutView in any Razor component. The following example sets a layout component named `ErrorLayout` for the `MainLayout` component's NotFound template (`<NotFound>...</NotFound>`).

razor

```
<Router ...>
    <Found ...>
        ...
    </Found>
    <NotFound>
        <LayoutView Layout="typeof(ErrorLayout)">
            <h1>Page not found</h1>
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

You may need to identity the layout's namespace depending on the .NET version and type of Blazor app. For more information, see the Make the layout namespace available section.

> **Important**
>
> Blazor Web Apps don't use the **NotFound** parameter (`<NotFound>...</NotFound>` markup), but the parameter is supported for backward compatibility to avoid a breaking change in the framework. The server-side ASP.NET Core middleware pipeline processes requests on the server. Use server-side techniques to handle bad requests. For more information, see **ASP.NET Core Blazor render modes**.

Nested layouts

A component can reference a layout that in turn references another layout. For example, nested layouts are used to create a multi-level menu structures.

The following example shows how to use nested layouts. The `Episodes` component shown in the Apply a layout to a component section is the component to display. The component references the `DoctorWhoLayout` component.

The following `DoctorWhoLayout` component is a modified version of the example shown earlier in this article. The header and footer elements are removed, and the layout references another layout, `ProductionsLayout`. The `Episodes` component is rendered where `@Body` appears in the `DoctorWhoLayout`.

`DoctorWhoLayout.razor`:

razor

```
@inherits LayoutComponentBase
@layout ProductionsLayout

<PageTitle>Doctor Who® Database</PageTitle>

<h1>Doctor Who® Database</h1>

<nav>
    <a href="main-episode-list">Main Episode List</a>
    <a href="episode-search">Search</a>
    <a href="new-episode">Add Episode</a>
</nav>

@Body

<div>
    @TrademarkMessage
</div>

@code {
    public string TrademarkMessage { get; set; } =
        "Doctor Who is a registered trademark of the BBC. " +
        "https://www.doctorwho.tv/ https://www.bbc.com";
}
```

The `ProductionsLayout` component contains the top-level layout elements, where the header (`<header>...</header>`) and footer (`<footer>...</footer>`) elements now reside. The `DoctorWhoLayout` with the `Episodes` component is rendered where `@Body` appears.

`ProductionsLayout.razor`:

razor

```razor
@inherits LayoutComponentBase

<header>
    <h1>Productions</h1>
</header>

<nav>
    <a href="main-production-list">Main Production List</a>
    <a href="production-search">Search</a>
    <a href="new-production">Add Production</a>
</nav>

@Body

<footer>
    Footer of Productions Layout
</footer>
```

The following rendered HTML markup is produced by the preceding nested layout. Extraneous markup doesn't appear in order to focus on the nested content provided by the three components involved:

- The header (`<header>...</header>`), production navigation bar (`<nav>...</nav>`), and footer (`<footer>...</footer>`) elements and their content come from the `ProductionsLayout` component.
- The H1 "database" heading (`<h1>...</h1>`), episode navigation bar (`<nav>...</nav>`), and trademark information (`<div>...</div>`) come from the `DoctorWhoLayout` component.
- The H2 "episodes" heading (`<h2>...</h2>`) and episode list (`<ul>...</ul>`) come from the `Episodes` component.

HTML

```html
<header>
    ...
</header>

<nav>
    <a href="main-production-list">Main Production List</a>
    <a href="production-search">Search</a>
    <a href="new-production">Add Production</a>
</nav>

<h1>...</h1>
```

```
<nav>
    <a href="main-episode-list">Main Episode List</a>
    <a href="episode-search">Search</a>
    <a href="new-episode">Add Episode</a>
</nav>

<h2>...</h2>

<ul>
    <li>...</li>
    <li>...</li>
    <li>...</li>
</ul>

<div>
    ...
</div>

<footer>
    ...
</footer>
```

Share a Razor Pages layout with integrated components

When routable components are integrated into a Razor Pages app, the app's shared layout can be used with the components. For more information, see Integrate ASP.NET Core Razor components with MVC or Razor Pages.

Sections

To control the content in a layout from a child Razor component, see ASP.NET Core Blazor sections.

Additional resources

- Layout in ASP.NET Core
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor sections

Article • 11/14/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to control the content in a Razor component from a child Razor component.

Blazor sections

To control the content in a Razor component from a child Razor component, Blazor supports *sections* using the following built-in components:

- SectionOutlet: Renders content provided by SectionContent components with matching SectionName or SectionId arguments. Two or more SectionOutlet components can't have the same SectionName or SectionId.
- SectionContent: Provides content as a RenderFragment to SectionOutlet components with a matching SectionName or SectionId. If several SectionContent components have the same SectionName or SectionId, the matching SectionOutlet component renders the content of the last rendered SectionContent.

Sections can be used in both layouts and across nested parent-child components.

Although the argument passed to SectionName can use any type of casing, the documentation adopts kebab casing (for example, `top-bar`), which is a common casing choice for HTML element IDs. SectionId receives a static `object` field, and we always recommend Pascal casing for C# field names (for example, `TopbarSection`).

In the following example, the app's main layout component implements an increment counter button for the app's `Counter` component.

If the namespace for sections isn't in the `_Imports.razor` file, add it:

razor

```
@using Microsoft.AspNetCore.Components.Sections
```

In the `MainLayout` component (`MainLayout.razor`), place a SectionOutlet component and pass a string to the SectionName parameter to indicate the section's name. The following example uses the section name `top-bar`:

razor

```
<SectionOutlet SectionName="top-bar" />
```

In the `Counter` component (`Counter.razor`), create a SectionContent component and pass the matching string (`top-bar`) to its SectionName parameter:

razor

```
<SectionContent SectionName="top-bar">
    <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
</SectionContent>
```

When the `Counter` component is accessed at `/counter`, the `MainLayout` component renders the increment count button from the `Counter` component where the SectionOutlet component is placed. When any other component is accessed, the increment count button isn't rendered.

Instead of using a named section, you can pass a static `object` with the SectionId parameter to identify the section. The following example also implements an increment counter button for the app's `Counter` component in the app's main layout.

If you don't want other SectionContent components to accidentally match the name of a SectionOutlet, pass an object SectionId parameter to identify the section. This can be useful when designing a Razor class library (RCL). When a SectionOutlet in the RCL uses an object reference with SectionId and the consumer places a SectionContent component with a matching SectionId object, an accidental match by name isn't possible when consumers of the RCL implement other SectionContent components.

The following example also implements an increment counter button for the app's `Counter` component in the app's main layout, using an object reference instead of a section name.

Add a `TopbarSection` static `object` to the `MainLayout` component in an `@code` block:

razor

```
@code {
    internal static object TopbarSection = new();
```

```
}
```

In the `MainLayout` component's Razor markup, place a SectionOutlet component and pass `TopbarSection` to the SectionId parameter to indicate the section:

```razor
<SectionOutlet SectionId="TopbarSection" />
```

Add a SectionContent component to the app's `Counter` component that renders an increment count button. Use the `MainLayout` component's `TopbarSection` section static `object` as the SectionId (`MainLayout.TopbarSection`).

In `Counter.razor`:

```razor
<SectionContent SectionId="MainLayout.TopbarSection">
    <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
</SectionContent>
```

When the `Counter` component is accessed, the `MainLayout` component renders the increment count button where the SectionOutlet component is placed.

> ⓘ Note
>
> SectionOutlet and SectionContent components can only set either SectionId or SectionName, not both.

Section interaction with other Blazor features

A section interacts with other Blazor features in the following ways:

- Cascading values flow into section content from where the content is defined by the SectionContent component.
- Unhandled exceptions are handled by error boundaries defined around a SectionContent component.
- A Razor component configured for streaming rendering also configures section content provided by a SectionContent component to use streaming rendering.

Control `<head>` content in ASP.NET Core Blazor apps

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Razor components can modify the HTML `<head>` element content of a page, including setting the page's title (`<title>` element) and modifying metadata (`<meta>` elements).

Control `<head>` content in a Razor component

Specify the page's title with the PageTitle component, which enables rendering an HTML `<title>` element to a HeadOutlet component.

Specify `<head>` element content with the HeadContent component, which provides content to a HeadOutlet component.

The following example sets the page's title and description using Razor.

`ControlHeadContent.razor`:

razor

```razor
@page "/control-head-content"

<PageTitle>@title</PageTitle>

<h1>Control <head> Content Example</h1>

<p>
    Title: @title
</p>

<p>
    Description: @description
</p>

<HeadContent>
    <meta name="description" content="@description">
</HeadContent>
```

```
@code {
    private string description = "This description is set by the
component.";
    private string title = "Control <head> Content";
}
```

Set a page title for components via a layout

Set the page title in a layout component:

razor

```
@inherits LayoutComponentBase

<PageTitle>Page Title</PageTitle>

<div class="page">
    ...
</div>
```

`HeadOutlet` component

The HeadOutlet component renders content provided by PageTitle and HeadContent components.

In a Blazor Web App created from the project template, the HeadOutlet component in `App.razor` renders `<head>` content:

razor

```
<head>
    ...
    <HeadOutlet />
</head>
```

In an app created from the Blazor WebAssembly project template, the HeadOutlet component is added to the RootComponents collection of the WebAssemblyHostBuilder in the client-side `Program` file:

C#

```
builder.RootComponents.Add<HeadOutlet>("head::after");
```

When the ::after pseudo-selector is specified, the contents of the root component are appended to the existing head contents instead of replacing the content. This allows the app to retain static head content in `wwwroot/index.html` without having to repeat the content in the app's Razor components.

Set a default page title in a Blazor Web App

Set the page title in the `App` component (`App.razor`):

razor

```razor
<head>
    ...
    <HeadOutlet />
    <PageTitle>Page Title</PageTitle>
</head>
```

Not found page title in a Blazor WebAssembly app

In Blazor apps created from the Blazor WebAssembly Standalone App project template, the `NotFound` component template in the `App` component (`App.razor`) sets the page title to `Not found`.

`App.razor`:

razor

```razor
<NotFound>
    <PageTitle>Not found</PageTitle>
    ...
</NotFound>
```

Additional resources

- Control headers in C# code at startup
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Mozilla MDN Web Docs documentation:

- What's in the head? Metadata in HTML
- <head>: The Document Metadata (Header) element

- <title>: The Document Title element
- <meta>: The metadata element

ASP.NET Core Blazor cascading values and parameters

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to flow data from an ancestor Razor component to descendent components.

Cascading values and parameters provide a convenient way to flow data down a component hierarchy from an ancestor component to any number of descendent components. Unlike Component parameters, cascading values and parameters don't require an attribute assignment for each descendent component where the data is consumed. Cascading values and parameters also allow components to coordinate with each other across a component hierarchy.

> **Note**
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from the `CascadingType?`, `@ActiveTab?`, `RenderFragment?`, `ITab?`, `TabSet?`, and `string?` types in the article's examples.

Root-level cascading values

Root-level cascading values can be registered for the entire component hierarchy. Named cascading values and subscriptions for update notifications are supported.

The following class is used in this section's examples.

`Dalek.cs`:

```
C#
```

```
// "Dalek" ©Terry Nation https://www.imdb.com/name/nm0622334/
// "Doctor Who" ©BBC https://www.bbc.co.uk/programmes/b006q2x0

namespace BlazorSample;

public class Dalek
{
    public int Units { get; set; }
}
```

The following registrations are made in the app's `Program` file with AddCascadingValue:

- `Dalek` with a property value for `Units` is registered as a fixed cascading value.
- A second `Dalek` registration with a different property value for `Units` is named "`AlphaGroup`".

C#

```
builder.Services.AddCascadingValue(sp => new Dalek { Units = 123 });
builder.Services.AddCascadingValue("AlphaGroup", sp => new Dalek { Units =
456 });
```

The following `Daleks` component displays the cascaded values.

`Daleks.razor`:

razor

```
@page "/daleks"

<PageTitle>Daleks</PageTitle>

<h1>Root-level Cascading Value Example</h1>

<ul>
    <li>Dalek Units: @Dalek?.Units</li>
    <li>Alpha Group Dalek Units: @AlphaGroupDalek?.Units</li>
</ul>

<p>
    Dalek© <a href="https://www.imdb.com/name/nm0622334/">Terry Nation</a>
<br>
    Doctor Who© <a href="https://www.bbc.co.uk/programmes/b006q2x0">BBC</a>
</p>

@code {
    [CascadingParameter]
    public Dalek? Dalek { get; set; }
```

```
    [CascadingParameter(Name = "AlphaGroup")]
    public Dalek? AlphaGroupDalek { get; set; }
}
```

In the following example, `Dalek` is registered as a cascading value using CascadingValueSource<T>, where `<T>` is the type. The `isFixed` flag indicates whether the value is fixed. If false, all recipients are subscribed for update notifications, which are issued by calling NotifyChangedAsync. Subscriptions create overhead and reduce performance, so set `isFixed` to `true` if the value doesn't change.

C#

```
builder.Services.AddCascadingValue(sp =>
{
    var dalek = new Dalek { Units = 789 };
    var source = new CascadingValueSource<Dalek>(dalek, isFixed: false);

    return source;
});
```

> ⚠ **Warning**
>
> Registering a component type as a root-level cascading value doesn't register additional services for the type or permit service activation in the component.
>
> Treat required services separately from cascading values, registering them separately from the cascaded type.
>
> Avoid using AddCascadingValue to register a component type as a cascading value. Instead, wrap the `<Router>...</Router>` in the `Routes` component (`Components/Routes.razor`) with the component and adopt global interactive server-side rendering (interactive SSR). For an example, see the CascadingValue component section.

`CascadingValue` component

An ancestor component provides a cascading value using the Blazor framework's CascadingValue component, which wraps a subtree of a component hierarchy and supplies a single value to all of the components within its subtree.

The following example demonstrates the flow of theme information down the component hierarchy to provide a CSS style class to buttons in child components.

The following `ThemeInfo` C# class specifies the theme information.

> **Note**
>
> For the examples in this section, the app's namespace is `BlazorSample`. When experimenting with the code in your own sample app, change the app's namespace to your sample app's namespace.

`ThemeInfo.cs`:

```
C#

namespace BlazorSample;

public class ThemeInfo
{
    public string? ButtonClass { get; set; }
}
```

The following layout component specifies theme information (`ThemeInfo`) as a cascading value for all components that make up the layout body of the Body property. `ButtonClass` is assigned a value of btn-success, which is a Bootstrap button style. Any descendent component in the component hierarchy can use the `ButtonClass` property through the `ThemeInfo` cascading value.

`MainLayout.razor`:

```
razor

@inherits LayoutComponentBase

<div class="page">
    <div class="sidebar">
        <NavMenu />
    </div>

    <main>
        <div class="top-row px-4">
            <a href="https://learn.microsoft.com/aspnet/core/"
target="_blank">About</a>
        </div>

        <CascadingValue Value="theme">
            <article class="content px-4">
                @Body
            </article>
        </CascadingValue>
```

```
    </main>
</div>

<div id="blazor-error-ui" data-nosnippet>
    An unhandled error has occurred.
    <a href="" class="reload">Reload</a>
    <a class="dismiss">✕</a>
</div>

@code {
    private ThemeInfo theme = new() { ButtonClass = "btn-success" };
}
```

Blazor Web Apps provide alternative approaches for cascading values that apply more broadly to the app than furnishing them via a single layout file:

- Wrap the markup of the `Routes` component in a CascadingValue component to specify the data as a cascading value for all of the app's components.

  The following example cascades `ThemeInfo` data from the `Routes` component.

  `Routes.razor`:

  ```razor
  <CascadingValue Value="theme">
      <Router ...>
          ...
      </Router>
  </CascadingValue>

  @code {
      private ThemeInfo theme = new() { ButtonClass = "btn-success" };
  }
  ```

  > **Note**
  >
  > Wrapping the `Routes` component instance in the `App` component (`Components/App.razor`) with a CascadingValue component isn't supported.

- Specify a *root-level cascading value* as a service by calling the AddCascadingValue extension method on the service collection builder.

  The following example cascades `ThemeInfo` data from the `Program` file.

  `Program.cs`

C#

```
builder.Services.AddCascadingValue(sp =>
    new ThemeInfo() { ButtonClass = "btn-primary" });
```

For more information, see the following sections of this article:

- Root-level cascading values
- Cascading values/parameters and render mode boundaries

`[CascadingParameter]` attribute

To make use of cascading values, descendent components declare cascading parameters using the [CascadingParameter] attribute. Cascading values are bound to cascading parameters **by type**. Cascading multiple values of the same type is covered in the Cascade multiple values section later in this article.

The following component binds the `ThemeInfo` cascading value to a cascading parameter, optionally using the same name of `ThemeInfo`. The parameter is used to set the CSS class for the **`Increment Counter (Themed)`** button.

`ThemedCounter.razor`:

razor

```
@page "/themed-counter"

<PageTitle>Themed Counter</PageTitle>

<h1>Themed Counter Example</h1>

<p>Current count: @currentCount</p>

<p>
    <button @onclick="IncrementCount">
        Increment Counter (Unthemed)
    </button>
</p>

<p>
    <button
        class="btn @(ThemeInfo is not null ? ThemeInfo.ButtonClass :
string.Empty)"
        @onclick="IncrementCount">
        Increment Counter (Themed)
    </button>
</p>
```

```
@code {
    private int currentCount = 0;

    [CascadingParameter]
    protected ThemeInfo? ThemeInfo { get; set; }

    private void IncrementCount() => currentCount++;
}
```

Similar to a regular component parameter, components accepting a cascading parameter are rerendered when the cascading value is changed. For instance, configuring a different theme instance causes the `ThemedCounter` component from the CascadingValue component section to rerender.

`MainLayout.razor`:

razor

```
<main>
    <div class="top-row px-4">
        <a href="https://docs.microsoft.com/aspnet/"
target="_blank">About</a>
    </div>

    <CascadingValue Value="theme">
        <article class="content px-4">
            @Body
        </article>
    </CascadingValue>
    <button @onclick="ChangeToDarkTheme">Dark mode</button>
</main>

@code {
    private ThemeInfo theme = new() { ButtonClass = "btn-success" };

    private void ChangeToDarkTheme()
    {
        theme = new() { ButtonClass = "btn-secondary" };
    }
}
```

CascadingValue<TValue>.IsFixed can be used to indicate that a cascading parameter doesn't change after initialization.

Cascading values/parameters and render mode boundaries

Cascading parameters don't pass data across render mode boundaries:

- Interactive sessions run in a different context than the pages that use static server-side rendering (static SSR). There's no requirement that the server producing the page is even the same machine that hosts some later Interactive Server session, including for WebAssembly components where the server is a different machine to the client. The benefit of static server-side rendering (static SSR) is to gain the full performance of pure stateless HTML rendering.
- State crossing the boundary between static and interactive rendering must be serializable. Components are arbitrary objects that reference a vast chain of other objects, including the renderer, the DI container, and every DI service instance. You must explicitly cause state to be serialized from static SSR to make it available in subsequent interactively-rendered components. Two approaches are adopted:
  - Via the Blazor framework, parameters passed across a static SSR to interactive rendering boundary are serialized automatically if they're JSON-serializable, or an error is thrown.
  - State stored in PersistentComponentState is serialized and recovered automatically if it's JSON-serializable, or an error is thrown.

Cascading parameters aren't JSON-serializable because the typical usage patterns for cascading parameters are somewhat like DI services. There are often platform-specific variants of cascading parameters, so it would be unhelpful to developers if the framework stopped developers from having server-interactive-specific versions or WebAssembly-specific versions. Also, many cascading parameter values in general aren't serializable, so it would be impractical to update existing apps if you had to stop using all nonserializable cascading parameter values.

Recommendations:

- If you need to make state available to all interactive components as a cascading parameter, we recommend using root-level cascading values. A factory pattern is available, and the app can emit updated values after app startup. Root-level cascading values are available to all components, including interactive components, since they're processed as DI services.
- For component library authors, you can create an extension method for library consumers similar to the following:

```
C#

builder.Services.AddLibraryCascadingParameters();
```

Instruct developers to call your extension method. This is a sound alternative to instructing them to add a `<RootComponent>` component in their `MainLayout` component.

Cascade multiple values

To cascade multiple values of the same type within the same subtree, provide a unique Name string to each CascadingValue component and their corresponding [CascadingParameter] attributes.

In the following example, two CascadingValue components cascade different instances of `CascadingType`:

razor

```
<CascadingValue Value="parentCascadeParameter1" Name="CascadeParam1">
    <CascadingValue Value="ParentCascadeParameter2" Name="CascadeParam2">
        ...
    </CascadingValue>
</CascadingValue>

@code {
    private CascadingType? parentCascadeParameter1;

    [Parameter]
    public CascadingType? ParentCascadeParameter2 { get; set; }
}
```

In a descendant component, the cascaded parameters receive their cascaded values from the ancestor component by Name:

razor

```
@code {
    [CascadingParameter(Name = "CascadeParam1")]
    protected CascadingType? ChildCascadeParameter1 { get; set; }

    [CascadingParameter(Name = "CascadeParam2")]
    protected CascadingType? ChildCascadeParameter2 { get; set; }
}
```

Pass data across a component hierarchy

Cascading parameters also enable components to pass data across a component hierarchy. Consider the following UI tab set example, where a tab set component

maintains a series of individual tabs.

> **Note**
>
> For the examples in this section, the app's namespace is `BlazorSample`. When experimenting with the code in your own sample app, change the namespace to your sample app's namespace.

Create an `ITab` interface that tabs implement in a folder named `UIInterfaces`.

`UIInterfaces/ITab.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.UIInterfaces;

public interface ITab
{
    RenderFragment ChildContent { get; }
}
```

> **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

The following `TabSet` component maintains a set of tabs. The tab set's `Tab` components, which are created later in this section, supply the list items (`<li>...</li>`) for the list (`<ul>...</ul>`).

Child `Tab` components aren't explicitly passed as parameters to the `TabSet`. Instead, the child `Tab` components are part of the child content of the `TabSet`. However, the `TabSet` still needs a reference each `Tab` component so that it can render the headers and the active tab. To enable this coordination without requiring additional code, the `TabSet` component *can provide itself as a cascading value* that is then picked up by the descendent `Tab` components.

`TabSet.razor`:

razor

```razor
@using BlazorSample.UIInterfaces

<!-- Display the tab headers -->

<CascadingValue Value="this">
    <ul class="nav nav-tabs">
        @ChildContent
    </ul>
</CascadingValue>

<!-- Display body for only the active tab -->

<div class="nav-tabs-body p-4">
    @ActiveTab?.ChildContent
</div>

@code {
    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    public ITab? ActiveTab { get; private set; }

    public void AddTab(ITab tab)
    {
        if (ActiveTab is null)
        {
            SetActiveTab(tab);
        }
    }

    public void SetActiveTab(ITab tab)
    {
        if (ActiveTab != tab)
        {
            ActiveTab = tab;
            StateHasChanged();
        }
    }
}
```

Descendent `Tab` components capture the containing `TabSet` as a cascading parameter. The `Tab` components add themselves to the `TabSet` and coordinate to set the active tab.

`Tab.razor`:

razor

```razor
@using BlazorSample.UIInterfaces
@implements ITab
```

```
<li>
    <a @onclick="ActivateTab" class="nav-link @TitleCssClass" role="button">
        @Title
    </a>
</li>

@code {
    [CascadingParameter]
    public TabSet? ContainerTabSet { get; set; }

    [Parameter]
    public string? Title { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    private string? TitleCssClass =>
        ContainerTabSet?.ActiveTab == this ? "active" : null;

    protected override void OnInitialized()
    {
        ContainerTabSet?.AddTab(this);
    }

    private void ActivateTab()
    {
        ContainerTabSet?.SetActiveTab(this);
    }
}
```

The following `ExampleTabSet` component uses the `TabSet` component, which contains three `Tab` components.

`ExampleTabSet.razor`:

razor

```
@page "/example-tab-set"

<TabSet>
    <Tab Title="First tab">
        <h4>Greetings from the first tab!</h4>

        <label>
            <input type="checkbox" @bind="showThirdTab" />
            Toggle third tab
        </label>
    </Tab>

    <Tab Title="Second tab">
        <h4>Hello from the second tab!</h4>
    </Tab>
```

```
    @if (showThirdTab)
    {
        <Tab Title="Third tab">
            <h4>Welcome to the disappearing third tab!</h4>
            <p>Toggle this tab from the first tab.</p>
        </Tab>
    }
</TabSet>

@code {
    private bool showThirdTab;
}
```

Additional resources

- Generic type support: Explicit generic types based on ancestor components
- State management: Factor out the state preservation to a common location

ASP.NET Core Blazor event handling

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains Blazor's event handling features, including event argument types, event callbacks, and managing default browser events.

Delegate event handlers

Specify delegate event handlers in Razor component markup with `@on{DOM EVENT}="{DELEGATE}"` Razor syntax:

- The `{DOM EVENT}` placeholder is a DOM event (for example, `click`).
- The `{DELEGATE}` placeholder is the C# delegate event handler.

For event handling:

- Delegate event handlers in Blazor Web Apps are only called in components that adopt an interactive render mode. The examples throughout this article assume that the app adopts an interactive render mode globally in the app's root component, typically the `App` component. For more information, see ASP.NET Core Blazor render modes.
- Asynchronous delegate event handlers that return a Task are supported.
- Delegate event handlers automatically trigger a UI render, so there's no need to manually call StateHasChanged.
- Exceptions are logged.

The following code:

- Calls the `UpdateHeading` method when the button is selected in the UI.
- Calls the `CheckChanged` method when the checkbox is changed in the UI.

`EventHandler1.razor`:

```
razor
```

```razor
@page "/event-handler-1"

<PageTitle>Event Handler 1</PageTitle>

<h1>Event Handler Example 1</h1>

<h2>@headingValue</h2>

<p>
    <button @onclick="UpdateHeading">
        Update heading
    </button>
</p>

<p>
    <label>
        <input type="checkbox" @onchange="CheckChanged" />
        @checkedMessage
    </label>
</p>

@code {
    private string headingValue = "Initial heading";
    private string checkedMessage = "Not changed yet";

    private void UpdateHeading() => headingValue = $"New heading
({DateTime.Now})";

    private void CheckChanged() => checkedMessage = $"Last change
{DateTime.Now}";
}
```

In the following example, `UpdateHeading`:

- Is called asynchronously when the button is selected.
- Waits two seconds before updating the heading.

`EventHandler2.razor`:

razor

```razor
@page "/event-handler-2"

<PageTitle>Event Handler 2</PageTitle>

<h1>Event Handler Example 2</h1>

<h2>@headingValue</h2>

<p>
    <button @onclick="UpdateHeading">
```

```
        Update heading
    </button>
</p>

@code {
    private string headingValue = "Initial heading";

    private async Task UpdateHeading()
    {
        await Task.Delay(2000);

        headingValue = $"New heading ({DateTime.Now})";
    }
}
```

Built-in event arguments

For events that support an event argument type, specifying an event parameter in the event method definition is only necessary if the event type is used in the method. In the following example, MouseEventArgs is used in the `ReportPointerLocation` method to set message text that reports the mouse coordinates when the user selects a button in the UI.

`EventHandler3.razor`:

razor

```
@page "/event-handler-3"

<PageTitle>Event Handler 3</PageTitle>

<h1>Event Handler Example 3</h1>

@for (var i = 0; i < 4; i++)
{
    <p>
        <button @onclick="ReportPointerLocation">
            Where's my mouse pointer for this button?
        </button>
    </p>
}

<p>@mousePointerMessage</p>

@code {
    private string? mousePointerMessage;

    private void ReportPointerLocation(MouseEventArgs e) =>
```

```
        mousePointerMessage = $"Mouse coordinates: {e.ScreenX}:{e.ScreenY}";
}
```

Supported EventArgs are shown in the following table.

Expand table

Event	Class	DOM notes
Clipboard	ClipboardEventArgs	
Drag	DragEventArgs	DataTransfer and DataTransferItem hold dragged item data. Implement drag and drop in Blazor apps using JS interop with HTML Drag and Drop API.
Error	ErrorEventArgs	
Event	EventArgs	EventHandlers holds attributes to configure the mappings between event names and event argument types.
Focus	FocusEventArgs	Doesn't include support for `relatedTarget`.
Input	ChangeEventArgs	
Keyboard	KeyboardEventArgs	
Mouse	MouseEventArgs	
Mouse pointer	PointerEventArgs	
Mouse wheel	WheelEventArgs	
Progress	ProgressEventArgs	
Touch	TouchEventArgs	TouchPoint represents a single contact point on a touch-sensitive device.

For more information, see the following resources:

- EventArgs classes in the ASP.NET Core reference source (dotnet/aspnetcore main branch)

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release

of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

- EventHandlers holds attributes to configure the mappings between event names and event argument types.

Custom event arguments

Blazor supports custom event arguments, which enable you to pass arbitrary data to .NET event handlers with custom events.

General configuration

Custom events with custom event arguments are generally enabled with the following steps.

In JavaScript, define a function for building the custom event argument object from the source event:

JavaScript

```
function eventArgsCreator(event) {
  return {
    customProperty1: 'any value for property 1',
    customProperty2: event.srcElement.id
  };
}
```

The `event` parameter is a DOM Event (MDN documentation).

Register the custom event with the preceding handler in a JavaScript initializer. Provide the appropriate browser event name to `browserEventName`, which for the example shown in this section is `click` for a button selection in the UI.

`wwwroot/{PACKAGE ID/ASSEMBLY NAME}.lib.module.js` (the `{PACKAGE ID/ASSEMBLY NAME}` placeholder is the package ID or assembly name of the app):

For a Blazor Web App:

JavaScript

```
export function afterWebStarted(blazor) {
  blazor.registerCustomEventType('customevent', {
```

```
    browserEventName: 'click',
    createEventArgs: eventArgsCreator
  });
}
```

For a Blazor Server or Blazor WebAssembly app:

JavaScript

```
export function afterStarted(blazor) {
  blazor.registerCustomEventType('customevent', {
    browserEventName: 'click',
    createEventArgs: eventArgsCreator
  });
}
```

The call to `registerCustomEventType` is performed in a script only once per event.

For the call to `registerCustomEventType`, use the `blazor` parameter (lowercase `b`) provided by the Blazor start event. Although the registration is valid when using the `Blazor` object (uppercase `B`), the preferred approach is to use the parameter.

The custom event name, `customevent` in the preceding example, must not match a reserved Blazor event name. The reserved names can be found in the Blazor framework reference source (see the calls to the registerBuiltInEventType function).

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Define a class for the event arguments:

C#

```
namespace BlazorSample.CustomEvents;

public class CustomEventArgs : EventArgs
{
    public string? CustomProperty1 {get; set;}
    public string? CustomProperty2 {get; set;}
}
```

Wire up the custom event with the event arguments by adding an [EventHandler] attribute annotation for the custom event:

- In order for the compiler to find the `[EventHandler]` class, it must be placed into a C# class file (`.cs`), making it a normal top-level class.
- Mark the class `public`.
- The class doesn't require members.
- The class *must* be called "`EventHandlers`" in order to be found by the Razor compiler.
- Place the class under a namespace specific to your app.
- Import the namespace into the Razor component (`.razor`) where the event is used.

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.CustomEvents;

[EventHandler("oncustomevent", typeof(CustomEventArgs),
    enableStopPropagation: true, enablePreventDefault: true)]
public static class EventHandlers
{
}
```

Register the event handler on one or more HTML elements. Access the data that was passed in from JavaScript in the delegate handler method:

razor

```
@using BlazorSample.CustomEvents

<button id="buttonId" @oncustomevent="HandleCustomEvent">Handle</button>

@if (!string.IsNullOrEmpty(propVal1) && !string.IsNullOrEmpty(propVal2))
{
    <ul>
        <li>propVal1: @propVal1</li>
        <li>propVal2: @propVal2</li>
    </ul>
}

@code
{
    private string? propVal1;
    private string? propVal2;

    private void HandleCustomEvent(CustomEventArgs eventArgs)
```

```
    {
        propVal1 = eventArgs.CustomProperty1;
        propVal2 = eventArgs.CustomProperty2;
    }
}
```

If the `@oncustomevent` attribute isn't recognized by IntelliSense, make sure that the component or the `_Imports.razor` file contains an `@using` statement for the namespace containing the `EventHandler` class.

Whenever the custom event is fired on the DOM, the event handler is called with the data passed from the JavaScript.

If you're attempting to fire a custom event, bubbles must be enabled by setting its value to `true`. Otherwise, the event doesn't reach the Blazor handler for processing into the C# custom [EventHandler] attribute class. For more information, see MDN Web Docs: Event bubbling.

Custom clipboard paste event example

The following example receives a custom clipboard paste event that includes the time of the paste and the user's pasted text.

Declare a custom name (`oncustompaste`) for the event and a .NET class (`CustomPasteEventArgs`) to hold the event arguments for this event:

`CustomEvents.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.CustomEvents;

[EventHandler("oncustompaste", typeof(CustomPasteEventArgs),
    enableStopPropagation: true, enablePreventDefault: true)]
public static class EventHandlers
{
}

public class CustomPasteEventArgs : EventArgs
{
    public DateTime EventTimestamp { get; set; }
    public string? PastedData { get; set; }
}
```

Add JavaScript code to supply data for the EventArgs subclass with the preceding handler in a JavaScript initializer. The following example only handles pasting text, but you could use arbitrary JavaScript APIs to deal with users pasting other types of data, such as images.

`wwwroot/{PACKAGE ID/ASSEMBLY NAME}.lib.module.js`:

For a Blazor Web App:

JavaScript

```
export function afterWebStarted(blazor) {
  blazor.registerCustomEventType('custompaste', {
    browserEventName: 'paste',
    createEventArgs: event => {
      return {
        eventTimestamp: new Date(),
        pastedData: event.clipboardData.getData('text')
      };
    }
  });
}
```

For a Blazor Server or Blazor WebAssembly app:

JavaScript

```
export function afterStarted(blazor) {
  blazor.registerCustomEventType('custompaste', {
    browserEventName: 'paste',
    createEventArgs: event => {
      return {
        eventTimestamp: new Date(),
        pastedData: event.clipboardData.getData('text')
      };
    }
  });
}
```

In the preceding example, the `{PACKAGE ID/ASSEMBLY NAME}` placeholder of the file name represents the package ID or assembly name of the app.

> **Note**

For the call to `registerCustomEventType`, use the `blazor` parameter (lowercase `b`) provided by the Blazor start event. Although the registration is valid when using the `Blazor` object (uppercase `B`), the preferred approach is to use the parameter.

The preceding code tells the browser that when a native paste event occurs:

- Raise a `custompaste` event.
- Supply the event arguments data using the custom logic stated:
  - For the `eventTimestamp`, create a new date.
  - For the `pastedData`, get the clipboard data as text. For more information, see MDN Web Docs: ClipboardEvent.clipboardData.

Event name conventions differ between .NET and JavaScript:

- In .NET, event names are prefixed with "`on`".
- In JavaScript, event names don't have a prefix.

In a Razor component, attach the custom handler to an element.

`CustomPasteArguments.razor`:

razor

```
@page "/custom-paste-arguments"
@using BlazorSample.CustomEvents

<label>
    Try pasting into the following text box:
    <input @oncustompaste="HandleCustomPaste" />
</label>

<p>
    @message
</p>

@code {
    private string? message;

    private void HandleCustomPaste(CustomPasteEventArgs eventArgs)
    {
        message = $"At {eventArgs.EventTimestamp.ToShortTimeString()}, " +
            $"you pasted: {eventArgs.PastedData}";
    }
}
```

Lambda expressions

Lambda expressions are supported as the delegate event handler.

`EventHandler4.razor`:

razor

```
@page "/event-handler-4"

<PageTitle>Event Handler 4</PageTitle>

<h1>Event Handler Example 4</h1>

<h2>@heading</h2>

<p>
    <button @onclick="@(e => heading = "New heading!!!")">
        Update heading
    </button>
</p>

@code {
    private string heading = "Initial heading";
}
```

It's often convenient to close over additional values using C# method parameters, such as when iterating over a set of elements. The following example creates three buttons, each of which calls `UpdateHeading` and passes the following data:

- An event argument (MouseEventArgs) in `e`.
- The button number in `buttonNumber`.

`EventHandler5.razor`:

razor

```
@page "/event-handler-5"

<PageTitle>Event Handler 5</PageTitle>

<h1>Event Handler Example 5</h1>

<h2>@heading</h2>

@for (var i = 1; i < 4; i++)
{
    var buttonNumber = i;

    <p>
        <button @onclick="@(e => UpdateHeading(e, buttonNumber))">
            Button #@i
```

```
        </button>
    </p>
}

@code {
    private string heading = "Select a button to learn its position";

    private void UpdateHeading(MouseEventArgs e, int buttonNumber) =>
        heading = $"Selected #{buttonNumber} at {e.ClientX}:{e.ClientY}";
}
```

Creating a large number of event delegates in a loop may cause poor rendering performance. For more information, see ASP.NET Core Blazor performance best practices.

Avoid using a loop variable directly in a lambda expression, such as `i` in the preceding `for` loop example. Otherwise, the same variable is used by all lambda expressions, which results in use of the same value in all lambdas. Capture the variable's value in a local variable. In the preceding example:

- The loop variable `i` is assigned to `buttonNumber`.
- `buttonNumber` is used in the lambda expression.

Alternatively, use a `foreach` loop with Enumerable.Range, which doesn't suffer from the preceding problem:

razor

```
@foreach (var buttonNumber in Enumerable.Range(1, 3))
{
    <p>
        <button @onclick="@(e => UpdateHeading(e, buttonNumber))">
            Button #@buttonNumber
        </button>
    </p>
}
```

EventCallback

A common scenario with nested components is executing a method in a parent component when a child component event occurs. An `onclick` event occurring in the child component is a common use case. To expose events across components, use an EventCallback. A parent component can assign a callback method to a child component's EventCallback.

The following `Child` component demonstrates how a button's `onclick` handler is set up to receive an EventCallback delegate from the sample's `ParentComponent`. The EventCallback is typed with MouseEventArgs, which is appropriate for an `onclick` event from a peripheral device.

`Child.razor`:

razor

```razor
<p>
    <button @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</p>

@code {
    [Parameter]
    public string? Title { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

The parent component sets the child's EventCallback<TValue> (`OnClickCallback`) to its `ShowMessage` method.

`ParentChild.razor`:

razor

```razor
@page "/parent-child"

<PageTitle>Parent Child</PageTitle>

<h1>Parent Child Example</h1>

<Child Title="Panel Title from Parent" OnClickCallback="ShowMessage">
    Content of the child component is supplied by the parent component.
</Child>

<p>@message</p>

@code {
    private string? message;

    private void ShowMessage(MouseEventArgs e) =>
        message = $"Blaze a new trail with Blazor! ({e.ScreenX}:
```

```
{e.ScreenY})";
}
```

When the button is selected in the `ChildComponent`:

- The `Parent` component's `ShowMessage` method is called. `message` is updated and displayed in the `Parent` component.
- A call to StateHasChanged isn't required in the callback's method (`ShowMessage`). StateHasChanged is called automatically to rerender the `Parent` component, just as child events trigger component rerendering in event handlers that execute within the child. For more information, see ASP.NET Core Razor component rendering.

Use EventCallback and EventCallback<TValue> for event handling and binding component parameters.

Prefer the strongly typed EventCallback<TValue> over EventCallback. EventCallback<TValue> provides enhanced error feedback when an inappropriate type is used, guiding users of the component towards correct implementation. Similar to other UI event handlers, specifying the event parameter is optional. Use EventCallback when there's no value passed to the callback.

EventCallback and EventCallback<TValue> permit asynchronous delegates. EventCallback is weakly typed and allows passing any type argument in `InvokeAsync(Object)`. EventCallback<TValue> is strongly typed and requires passing a `T` argument in `InvokeAsync(T)` that's assignable to `TValue`.

Invoke an EventCallback or EventCallback<TValue> with InvokeAsync and await the Task:

C#

```
await OnClickCallback.InvokeAsync({ARGUMENT});
```

In the preceding example, the `{ARGUMENT}` placeholder is an optional argument.

The following parent-child example demonstrates the technique.

`Child2.razor`:

razor

```
<h3>Child2 Component</h3>

<button @onclick="TriggerEvent">Click Me</button>
```

```
@code {
    [Parameter]
    public EventCallback<string> OnClickCallback { get; set; }

    private async Task TriggerEvent()
    {
        await OnClickCallback.InvokeAsync("Blaze It!");
    }
}
```

`ParentChild2.razor`:

razor

```
@page "/parent-child-2"

<PageTitle>Parent Child 2</PageTitle>

<h1>Parent Child 2 Example</h1>

<div>
    <Child2 OnClickCallback="(value) => { message1 = value; }" />
    @message1
</div>

<div>
    <Child2 OnClickCallback=
        "async (value) => { await Task.Delay(2000); message2 = value; }" />
    @message2
</div>

@code {
    private string message1 = string.Empty;
    private string message2 = string.Empty;
}
```

The second occurrence of the `Child2` component demonstrates an asynchronous callback, and the new `message2` value is assigned and rendered with a delay of two seconds.

Prevent default actions

Use the @on{DOM EVENT}:preventDefault directive attribute to prevent the default action for an event, where the `{DOM EVENT}` placeholder is a DOM event.

When a key is selected on an input device and the element focus is on a text box, a browser normally displays the key's character in the text box. In the following example, the default behavior is prevented by specifying the `@onkeydown:preventDefault` directive

attribute. When the focus is on the `<input>` element, the counter increments with the key sequence `Shift`+`+`. The `+` character isn't assigned to the `<input>` element's value. For more information on `keydown`, see MDN Web Docs: Document: keydown event.

`EventHandler6.razor`:

razor

```razor
@page "/event-handler-6"

<PageTitle>Event Handler 6</PageTitle>

<h1>Event Handler Example 6</h1>

<p>For this example, give the <code><input></code> focus.</p>

<p>
    <label>
        Count of '+' key presses:
        <input value="@count" @onkeydown="KeyHandler"
@onkeydown:preventDefault />
    </label>
</p>

@code {
    private int count = 0;

    private void KeyHandler(KeyboardEventArgs e)
    {
        if (e.Key == "+")
        {
            count++;
        }
    }
}
```

Specifying the `@on{DOM EVENT}:preventDefault` attribute without a value is equivalent to `@on{DOM EVENT}:preventDefault="true"`.

An expression is also a permitted value of the attribute. In the following example, `shouldPreventDefault` is a `bool` field set to either `true` or `false`:

razor

```razor
<input @onkeydown:preventDefault="shouldPreventDefault" />

...

@code {
```

```
    private bool shouldPreventDefault = true;
}
```

Stop event propagation

Use the @on{DOM EVENT}:stopPropagation directive attribute to stop event propagation within the Blazor scope. `{DOM EVENT}` is a placeholder for a DOM event.

The `stopPropagation` directive attribute's effect is limited to the Blazor scope and doesn't extend to the HTML DOM. Events must propagate to the HTML DOM root before Blazor can act upon them. For a mechanism to prevent HTML DOM event propagation, consider the following approach:

- Obtain the event's path by calling Event.composedPath().
- Filter events based on the composed event targets (EventTarget).

In the following example, selecting the checkbox prevents click events from the second child `<div>` from propagating to the parent `<div>`. Since propagated click events normally fire the `OnSelectParentDiv` method, selecting the second child `<div>` results in the parent `<div>` message appearing unless the checkbox is selected.

`EventHandler7.razor`:

razor

```
@page "/event-handler-7"

<PageTitle>Event Handler 7</PageTitle>

<h1>Event Handler Example 7</h1>

<div>
    <b>stopPropagation</b>: @stopPropagation
</div>

<div>
    <button @onclick="StopPropagation">
        Stop Propagation (stopPropagation = true)
    </button>
    <button @onclick="EnablePropagation">
        Enable Propagation (stopPropagation = false)
    </button>
</div>

<div class="m-1 p-1 border border-primary" @onclick="OnSelectParentDiv">
    <h3>Parent div</h3>
```

```
    <div class="m-1 p-1 border" @onclick="OnSelectChildDiv">
        Child div that never stops propagation to the parent div when
        selected.
    </div>

    <div class="m-1 p-1 border" @onclick="OnSelectChildDiv"
            @onclick:stopPropagation="stopPropagation">
        Child div that stops propagation when selected if
        <b>stopPropagation</b> is <b>true</b>.
    </div>
</div>

<p>
    @message
</p>

@code {
    private bool stopPropagation = false;
    private string? message;

    private void StopPropagation() => stopPropagation = true;

    private void EnablePropagation() => stopPropagation = false;

    private void OnSelectParentDiv() =>
        message = $"The parent div was selected. {DateTime.Now}";

    private void OnSelectChildDiv() =>
        message = $"The child div was selected. {DateTime.Now}";
}
```

Focus an element

Call FocusAsync on an element reference to focus an element in code. In the following example, select the button to focus the `<input>` element.

`EventHandler8.razor`:

razor

```
@page "/event-handler-8"

<PageTitle>Event Handler 8</PageTitle>

<h1>Event Handler Example 8</h1>

<p>Select the button to give the <code><input></code> focus.</p>

<p>
    <label>
        Input:
```

```
        <input @ref="exampleInput" />
    </label>

</p>

<button @onclick="ChangeFocus">
    Focus the Input Element
</button>

@code {
    private ElementReference exampleInput;

    private async Task ChangeFocus()
    {
        await exampleInput.FocusAsync();
    }
}
```

ASP.NET Core Blazor data binding

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains data binding features for Razor components and DOM elements in Blazor apps.

Binding features

Razor components provide data binding features with the @bind Razor directive attribute with a field, property, or Razor expression value.

The following example binds:

- An `<input>` element value to the C# `inputValue` field.
- A second `<input>` element value to the C# `InputValue` property.

When an `<input>` element loses focus, its bound field or property is updated.

`Bind.razor`:

razor

```
@page "/bind"

<PageTitle>Bind</PageTitle>

<h1>Bind Example</h1>

<p>
    <label>
        inputValue:
        <input @bind="inputValue" />
    </label>
</p>

<p>
    <label>
        InputValue:
        <input @bind="InputValue" />
    </label>
```

```
</p>

<ul>
    <li><code>inputValue</code>: @inputValue</li>
    <li><code>InputValue</code>: @InputValue</li>
</ul>

@code {
    private string? inputValue;

    private string? InputValue { get; set; }
}
```

The text box is updated in the UI only when the component is rendered, not in response to changing the field's or property's value. Since components render themselves after event handler code executes, field and property updates are usually reflected in the UI immediately after an event handler is triggered.

As a demonstration of how data binding composes in HTML, the following example binds the `InputValue` property to the second `<input>` element's `value` and `onchange` attributes (change). *The second `<input>` element in the following example is a concept demonstration and isn't meant to suggest how you should bind data in Razor components.*

`BindTheory.razor`:

razor

```
@page "/bind-theory"

<PageTitle>Bind Theory</PageTitle>

<h1>Bind Theory Example</h1>

<p>
    <label>
        Normal Blazor binding:
        <input @bind="InputValue" />
    </label>
</p>

<p>
    <label>
        Demonstration of equivalent HTML binding:
        <input value="@InputValue" @onchange="@((ChangeEventArgs __e) =>
            InputValue = __e?.Value?.ToString())" />
    </label>
</p>

<p>
    <code>InputValue</code>: @InputValue
```

```
</p>

@code {
    private string? InputValue { get; set; }
}
```

When the `BindTheory` component is rendered, the `value` of the HTML demonstration `<input>` element comes from the `InputValue` property. When the user enters a value in the text box and changes element focus, the `onchange` event is fired and the `InputValue` property is set to the changed value. In reality, code execution is more complex because @bind handles cases where type conversions are performed. In general, @bind associates the current value of an expression with the `value` attribute of the `<input>` and handles changes using the registered handler.

Bind a property or field on other DOM events by including an `@bind:event="{EVENT}"` attribute with a DOM event for the `{EVENT}` placeholder. The following example binds the `InputValue` property to the `<input>` element's value when the element's `oninput` event (input) is triggered. Unlike the `onchange` event (change), which fires when the element loses focus, `oninput` (input) fires when the value of the text box changes.

`Page/BindEvent.razor`:

razor

```
@page "/bind-event"

<PageTitle>Bind Event</PageTitle>

<h1>Bind Event Example</h1>

<p>
    <label>
        InputValue:
        <input @bind="InputValue" @bind:event="oninput" />
    </label>

</p>

<p>
    <code>InputValue</code>: @InputValue
</p>

@code {
    private string? InputValue { get; set; }
}
```

To execute asynchronous logic after binding, use `@bind:after="{DELEGATE}"`, where the `{DELEGATE}` placeholder is a C# delegate (method). An assigned C# delegate isn't executed until the bound value is assigned synchronously.

Using an event callback parameter (EventCallback/EventCallback<T>) with `@bind:after` isn't supported. Instead, pass a method that returns an Action or Task to `@bind:after`.

In the following example:

- Each `<input>` element's `value` is bound to the `searchText` field synchronously.
- The `PerformSearch` method executes asynchronously:
  - When the first box loses focus (`onchange` event) after the value is changed.
  - After each keystroke (`oninput` event) in the second box.
- `PerformSearch` calls a service with an asynchronous method (`FetchAsync`) to return search results.

razor

```
@inject ISearchService SearchService

<input @bind="searchText" @bind:after="PerformSearch" />
<input @bind="searchText" @bind:event="oninput" @bind:after="PerformSearch"
/>

@code {
    private string? searchText;
    private string[]? searchResult;

    private async Task PerformSearch() =>
        searchResult = await SearchService.FetchAsync(searchText);
}
```

Additional examples

`BindAfter.razor`:

razor

```
@page "/bind-after"
@using Microsoft.AspNetCore.Components.Forms

<h1>Bind After Examples</h1>

<h2>Elements</h2>

<input type="text" @bind="text" @bind:after="() => { }" />

<input type="text" @bind="text" @bind:after="After" />
```

```
<input type="text" @bind="text" @bind:after="AfterAsync" />

<h2>Components</h2>

<InputText @bind-Value="text" @bind-Value:after="() => { }" />

<InputText @bind-Value="text" @bind-Value:after="After" />

<InputText @bind-Value="text" @bind-Value:after="AfterAsync" />

@code {
    private string text = "";

    private void After() {}
    private Task AfterAsync() { return Task.CompletedTask; }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

Components support two-way data binding by defining a pair of parameters:

- `@bind:get`: Specifies the value to bind.
- `@bind:set`: Specifies a callback for when the value changes.

The `@bind:get` and `@bind:set` modifiers are always used together.

Examples

`BindGetSet.razor`:

razor

```
@page "/bind-get-set"
@using Microsoft.AspNetCore.Components.Forms

<h1>Bind Get Set Examples</h1>

<h2>Elements</h2>

<input type="text" @bind:get="text" @bind:set="(value) => { text = value; }"
/>
<input type="text" @bind:get="text" @bind:set="Set" />
<input type="text" @bind:get="text" @bind:set="SetAsync" />

<h2>Components</h2>

<InputText @bind-Value:get="text" @bind-Value:set="(value) => { text =
value; }" />
<InputText @bind-Value:get="text" @bind-Value:set="Set" />
```

```razor
<InputText @bind-Value:get="text" @bind-Value:set="SetAsync" />

@code {
    private string text = "";

    private void Set(string value)
    {
        text = value;
    }

    private Task SetAsync(string value)
    {
        text = value;
        return Task.CompletedTask;
    }
}
```

For more information on the `InputText` component, see ASP.NET Core Blazor input components.

For another example use of `@bind:get` and `@bind:set`, see the Bind across more than two components section later in this article.

Razor attribute binding is case-sensitive:

- `@bind`, `@bind:event`, and `@bind:after` are valid.
- `@Bind`/`@bind:Event`/`@bind:aftEr` (capital letters) or `@BIND`/`@BIND:EVENT`/`@BIND:AFTER` (all capital letters) **are invalid**.

Use `@bind:get`/`@bind:set` modifiers and avoid event handlers for two-way data binding

Two-way data binding isn't possible to implement with an event handler. Use `@bind:get`/`@bind:set` modifiers for two-way data binding.

❌ Consider the following ***dysfunctional approach*** for two-way data binding using an event handler:

razor

```razor
<p>
    <input value="@inputValue" @oninput="OnInput" />
</p>

<p>
    <code>inputValue</code>: @inputValue
</p>
```

```
@code {
    private string? inputValue;

    private void OnInput(ChangeEventArgs args)
    {
        var newValue = args.Value?.ToString() ?? string.Empty;

        inputValue = newValue.Length > 4 ? "Long!" : newValue;
    }
}
```

The `OnInput` event handler updates the value of `inputValue` to `Long!` after a fourth character is provided. However, the user can continue adding characters to the element value in the UI. The value of `inputValue` isn't bound back to the element's value with each keystroke. The preceding example is only capable of one-way data binding.

The reason for this behavior is that Blazor isn't aware that your code intends to modify the value of `inputValue` in the event handler. Blazor doesn't try to force DOM element values and .NET variable values to match unless they're bound with `@bind` syntax. In earlier versions of Blazor, two-way data binding is implemented by binding the element to a property and controlling the property's value with its setter. In ASP.NET Core in .NET 7 or later, `@bind:get`/`@bind:set` modifier syntax is used to implement two-way data binding, as the next example demonstrates.

✔ Consider the following ***correct approach*** using `@bind:get`/`@bind:set` for two-way data binding:

razor

```
<p>
    <input @bind:event="oninput" @bind:get="inputValue" @bind:set="OnInput"
/>
</p>

<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private string? inputValue;

    private void OnInput(string value)
    {
        var newValue = value ?? string.Empty;

        inputValue = newValue.Length > 4 ? "Long!" : newValue;
```

```
    }
}
```

Using `@bind:get`/`@bind:set` modifiers both controls the underlying value of `inputValue` via `@bind:set` and binds the value of `inputValue` to the element's value via `@bind:get`. The preceding example demonstrates the correct approach for implementing two-way data binding.

Binding to a property with C# `get` and `set` accessors

C# get and set accessors can be used to create custom binding format behavior, as the following `DecimalBinding` component demonstrates. The component binds a positive or negative decimal with up to three decimal places to an `<input>` element by way of a `string` property (`DecimalValue`).

`DecimalBinding.razor`:

razor

```
@page "/decimal-binding"
@using System.Globalization

<PageTitle>Decimal Binding</PageTitle>

<h1>Decimal Binding Example</h1>

<p>
    <label>
        Decimal value (±0.000 format):
        <input @bind="DecimalValue" />
    </label>
</p>

<p>
    <code>decimalValue</code>: @decimalValue
</p>

@code {
    private decimal decimalValue = 1.1M;
    private NumberStyles style =
        NumberStyles.AllowDecimalPoint | NumberStyles.AllowLeadingSign;
    private CultureInfo culture = CultureInfo.CreateSpecificCulture("en-
US");

    private string DecimalValue
    {
```

```
        get => decimalValue.ToString("0.000", culture);
        set
        {
            if (Decimal.TryParse(value, style, culture, out var number))
            {
                decimalValue = Math.Round(number, 3);
            }
        }
    }
}
```

> **Note**
>
> Two-way binding to a property with `get`/`set` accessors requires discarding the Task returned by EventCallback.InvokeAsync. For two-way data binding, we recommend using `@bind:get`/`@bind:set` modifiers. For more information, see the `@bind:get`/`@bind:set` guidance in the earlier in this article.

Multiple option selection with `<select>` elements

Binding supports multiple option selection with `<select>` elements. The @onchange event provides an array of the selected elements via event arguments (ChangeEventArgs). The value must be bound to an array type.

`BindMultipleInput.razor`:

razor

```
@page "/bind-multiple-input"

<h1>Bind Multiple <code>input</code>Example</h1>

<p>
    <label>
        Select one or more cars:
        <select @onchange="SelectedCarsChanged" multiple>
            <option value="audi">Audi</option>
            <option value="jeep">Jeep</option>
            <option value="opel">Opel</option>
            <option value="saab">Saab</option>
            <option value="volvo">Volvo</option>
        </select>
    </label>
</p>
```

```
<p>
    Selected Cars: @string.Join(", ", SelectedCars)
</p>

<p>
    <label>
        Select one or more cities:
        <select @bind="SelectedCities" multiple>
            <option value="bal">Baltimore</option>
            <option value="la">Los Angeles</option>
            <option value="pdx">Portland</option>
            <option value="sf">San Francisco</option>
            <option value="sea">Seattle</option>
        </select>
    </label>
</p>

<span>
    Selected Cities: @string.Join(", ", SelectedCities)
</span>

@code {
    public string[] SelectedCars { get; set; } = new string[] { };
    public string[] SelectedCities { get; set; } = new[] { "bal", "sea" };

    private void SelectedCarsChanged(ChangeEventArgs e)
    {
        if (e.Value is not null)
        {
            SelectedCars = (string[])e.Value;
        }
    }
}
```

For information on how empty strings and `null` values are handled in data binding, see the Binding <select> element options to C# object null values section.

Binding `<select>` element options to C# object `null` values

There's no sensible way to represent a `<select>` element option value as a C# object `null` value, because:

- HTML attributes can't have `null` values. The closest equivalent to `null` in HTML is absence of the HTML `value` attribute from the `<option>` element.
- When selecting an `<option>` with no `value` attribute, the browser treats the value as the *text content* of that `<option>`'s element.

The Blazor framework doesn't attempt to suppress the default behavior because it would involve:

- Creating a chain of special-case workarounds in the framework.
- Breaking changes to current framework behavior.

The most plausible `null` equivalent in HTML is an *empty string* `value`. The Blazor framework handles `null` to empty string conversions for two-way binding to a `<select>`'s value.

Unparsable values

When a user provides an unparsable value to a data-bound element, the unparsable value is automatically reverted to its previous value when the bind event is triggered.

Consider the following component, where an `<input>` element is bound to an `int` type with an initial value of `123`.

`UnparsableValues.razor`:

razor

```
@page "/unparsable-values"

<PageTitle>Unparsable Values</PageTitle>

<h1>Unparsable Values Example</h1>

<p>
    <label>
        inputValue:
        <input @bind="inputValue" />
    </label>

</p>

<p>
    <code>inputValue</code>: @inputValue
</p>

@code {
    private int inputValue = 123;
}
```

Binding applies to the element's `onchange` event. If the user updates the value of the text box's entry to `123.45` and changes the focus, the element's value is reverted to `123`

when `onchange` fires. When the value `123.45` is rejected in favor of the original value of `123`, the user understands that their value wasn't accepted.

For the `oninput` event (`@bind:event="oninput"`), a value reversion occurs after any keystroke that introduces an unparsable value. When targeting the `oninput` event with an `int`-bound type, a user is prevented from typing a dot (`.`) character. A dot (`.`) character is immediately removed, so the user receives immediate feedback that only whole numbers are permitted. There are scenarios where reverting the value on the `oninput` event isn't ideal, such as when the user should be allowed to clear an unparsable `<input>` value. Alternatives include:

- Don't use the `oninput` event. Use the default `onchange` event, where an invalid value isn't reverted until the element loses focus.
- Bind to a nullable type, such as `int?` or `string` and either use `@bind:get`/`@bind:set` modifiers (described earlier in this article) or bind to a property with custom get and set accessor logic to handle invalid entries.
- Use an input component, such as InputNumber<TValue> or InputDate<TValue>, with form validation. Input components together with form validation components provide built-in support to manage invalid inputs:
  - Permit the user to provide invalid input and receive validation errors on the associated EditContext.
  - Display validation errors in the UI without interfering with the user entering additional webform data.

Format strings

Data binding works with a single DateTime format string using `@bind:format="{FORMAT STRING}"`, where the `{FORMAT STRING}` placeholder is the format string. Other format expressions, such as currency or number formats, aren't available at this time but might be added in a future release.

`DateBinding.razor`:

razor

```
@page "/date-binding"

<PageTitle>Date Binding</PageTitle>

<h1>Date Binding Example</h1>

<p>
    <label>
```

```
        <code>yyyy-MM-dd</code> format:
        <input @bind="startDate" @bind:format="yyyy-MM-dd" />
    </label>
</p>

<p>
    <code>startDate</code>: @startDate
</p>

@code {
    private DateTime startDate = new(2020, 1, 1);
}
```

In the preceding code, the `<input>` element's field type (`type` attribute) defaults to `text`.

Nullable System.DateTime and System.DateTimeOffset are supported:

C#

```
private DateTime? date;
private DateTimeOffset? dateOffset;
```

Specifying a format for the `date` field type isn't recommended because Blazor has built-in support to format dates. In spite of the recommendation, only use the `yyyy-MM-dd` date format for binding to function correctly if a format is supplied with the `date` field type:

razor

```
<input type="date" @bind="startDate" @bind:format="yyyy-MM-dd">
```

Binding with component parameters

A common scenario is binding a property of a child component to a property in its parent component. This scenario is called a *chained bind* because multiple levels of binding occur simultaneously.

You can't implement chained binds with @bind syntax in a child component. An event handler and value must be specified separately to support updating the property in the parent from the child component. The parent component still leverages @bind syntax to set up data binding with the child component.

The following `ChildBind` component has a `Year` component parameter and an EventCallback<TValue>. By convention, the EventCallback<TValue> for the parameter must be named as the component parameter name with a "`Changed`" suffix. The naming syntax is `{PARAMETER NAME}Changed`, where the `{PARAMETER NAME}` placeholder is the parameter name. In the following example, the EventCallback<TValue> is named `YearChanged`.

EventCallback.InvokeAsync invokes the delegate associated with the binding with the provided argument and dispatches an event notification for the changed property.

`ChildBind.razor`:

razor

```
<div class="card bg-light mt-3" style="width:18rem ">
    <div class="card-body">
        <h3 class="card-title">ChildBind Component</h3>
        <p class="card-text">
            Child <code>Year</code>: @Year
        </p>
        <button @onclick="UpdateYearFromChild">Update Year from
Child</button>
    </div>
</div>

@code {
    [Parameter]
    public int Year { get; set; }

    [Parameter]
    public EventCallback<int> YearChanged { get; set; }

    private async Task UpdateYearFromChild() =>
        await YearChanged.InvokeAsync(Random.Shared.Next(1950, 2021));
}
```

For more information on events and EventCallback<TValue>, see the *EventCallback* section of the ASP.NET Core Blazor event handling article.

In the following `Parent1` component, the `year` field is bound to the `Year` parameter of the child component. The `Year` parameter is bindable because it has a companion `YearChanged` event that matches the type of the `Year` parameter.

`Parent1.razor`:

razor

```razor
@page "/parent-1"

<PageTitle>Parent 1</PageTitle>

<h1>Parent Example 1</h1>

<p>Parent <code>year</code>: @year</p>

<button @onclick="UpdateYear">Update Parent <code>year</code></button>

<ChildBind @bind-Year="year" />

@code {
    private int year = 1979;

    private void UpdateYear() => year = Random.Shared.Next(1950, 2021);
}
```

Component parameter binding can also trigger `@bind:after` events. In the following example, the `YearUpdated` method executes asynchronously after binding the `Year` component parameter.

razor

```razor
<ChildBind @bind-Year="year" @bind-Year:after="YearUpdated" />

@code {
    ...

    private async Task YearUpdated()
    {
        ... = await ...;
    }
}
```

By convention, a property can be bound to a corresponding event handler by including an `@bind-{PROPERTY}:event` attribute assigned to the handler, where the `{PROPERTY}` placeholder is the property. `<ChildBind @bind-Year="year" />` is equivalent to writing:

razor

```razor
<ChildBind @bind-Year="year" @bind-Year:event="YearChanged" />
```

In a more sophisticated and real-world example, the following `PasswordEntry` component:

- Sets an `<input>` element's value to a `password` field.

- Exposes changes of a `Password` property to a parent component with an EventCallback that passes in the current value of the child's `password` field as its argument.
- Uses the `onclick` event to trigger the `ToggleShowPassword` method. For more information, see ASP.NET Core Blazor event handling.

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

`PasswordEntry.razor`:

razor

```razor
<div class="card bg-light mt-3" style="width:22rem ">
    <div class="card-body">
        <h3 class="card-title">Password Component</h3>
        <p class="card-text">
            <label>
                Password:
                <input @oninput="OnPasswordChanged"
                       required
                       type="@(showPassword ? "text" : "password")"
                       value="@password" />
            </label>
        </p>
        <button class="btn btn-primary" @onclick="ToggleShowPassword">
            Show password
        </button>
    </div>
</div>

@code {
    private bool showPassword;
    private string? password;

    [Parameter]
    public string? Password { get; set; }
```

```
    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private async Task OnPasswordChanged(ChangeEventArgs e)
    {
        password = e?.Value?.ToString();

        await PasswordChanged.InvokeAsync(password);
    }

    private void ToggleShowPassword() => showPassword = !showPassword;
}
```

The `PasswordEntry` component is used in another component, such as the following `PasswordBinding` component example.

`PasswordBinding.razor`:

razor

```
@page "/password-binding"

<PageTitle>Password Binding</PageTitle>

<h1>Password Binding Example</h1>

<PasswordEntry @bind-Password="password" />

<p>
    <code>password</code>: @password
</p>

@code {
    private string password = "Not set";
}
```

When the `PasswordBinding` component is initially rendered, the `password` value of `Not set` is displayed in the UI. After initial rendering, the value of `password` reflects changes made to the `Password` component parameter value in the `PasswordEntry` component.

> **Note**
>
> The preceding example binds the password one-way from the child `PasswordEntry` component to the parent `PasswordBinding` component. Two-way binding isn't a requirement in this scenario if the goal is for the app to have a shared password entry component for reuse around the app that merely passes the password to the

parent. For an approach that permits two-way binding without writing directly to the child component's parameter, see the `NestedChild` component example in the Bind across more than two components section of this article.

Perform checks or trap errors in the handler. The following revised `PasswordEntry` component provides immediate feedback to the user if a space is used in the password's value.

`PasswordEntry.razor`:

razor

```razor
<div class="card bg-light mt-3" style="width:22rem ">
    <div class="card-body">
        <h3 class="card-title">Password Component</h3>
        <p class="card-text">
            <label>
                Password:
                <input @oninput="OnPasswordChanged"
                       required
                       type="@(showPassword ? "text" : "password")"
                       value="@password" />
            </label>
            <span class="text-danger">@validationMessage</span>
        </p>
        <button class="btn btn-primary" @onclick="ToggleShowPassword">
            Show password
        </button>
    </div>
</div>

@code {
    private bool showPassword;
    private string? password;
    private string? validationMessage;

    [Parameter]
    public string? Password { get; set; }

    [Parameter]
    public EventCallback<string> PasswordChanged { get; set; }

    private Task OnPasswordChanged(ChangeEventArgs e)
    {
        password = e?.Value?.ToString();

        if (password != null && password.Contains(' '))
        {
            validationMessage = "Spaces not allowed!";

            return Task.CompletedTask;
```

```
        }
        else
        {
            validationMessage = string.Empty;

            return PasswordChanged.InvokeAsync(password);
        }
    }

    private void ToggleShowPassword() => showPassword = !showPassword;
}
```

Bind across more than two components

You can bind parameters through any number of nested components, but you must respect the one-way flow of data:

- Change notifications *flow up the hierarchy*.
- New parameter values *flow down the hierarchy*.

A common and recommended approach is to only store the underlying data in the parent component to avoid any confusion about what state must be updated, as shown in the following example.

`Parent2.razor`:

razor

```
@page "/parent-2"

<PageTitle>Parent 2</PageTitle>

<h1>Parent Example 2</h1>

<p>Parent Message: <b>@parentMessage</b></p>

<p>
    <button @onclick="ChangeValue">Change from Parent</button>
</p>

<NestedChild @bind-ChildMessage="parentMessage" />

@code {
    private string parentMessage = "Initial value set in Parent";

    private void ChangeValue() => parentMessage = $"Set in Parent
{DateTime.Now}";
}
```

In the following `NestedChild` component, the `NestedGrandchild` component:

- Assigns the value of `ChildMessage` to `GrandchildMessage` with `@bind:get` syntax.
- Updates `GrandchildMessage` when `ChildMessageChanged` executes with `@bind:set` syntax.

`NestedChild.razor`:

razor

```
<div class="border rounded m-1 p-1">
    <h2>Child Component</h2>

    <p>Child Message: <b>@ChildMessage</b></p>

    <p>
        <button @onclick="ChangeValue">Change from Child</button>
    </p>

    <NestedGrandchild @bind-GrandchildMessage:get="ChildMessage"
        @bind-GrandchildMessage:set="ChildMessageChanged" />
</div>

@code {
    [Parameter]
    public string? ChildMessage { get; set; }

    [Parameter]
    public EventCallback<string?> ChildMessageChanged { get; set; }

    private async Task ChangeValue() =>
        await ChildMessageChanged.InvokeAsync($"Set in Child
{DateTime.Now}");
}
```

`NestedGrandchild.razor`:

razor

```
<div class="border rounded m-1 p-1">
    <h3>Grandchild Component</h3>

    <p>Grandchild Message: <b>@GrandchildMessage</b></p>

    <p>
        <button @onclick="ChangeValue">Change from Grandchild</button>
    </p>
</div>

@code {
    [Parameter]
```

```
    public string? GrandchildMessage { get; set; }

    [Parameter]
    public EventCallback<string> GrandchildMessageChanged { get; set; }

    private async Task ChangeValue() =>
        await GrandchildMessageChanged.InvokeAsync(
            $"Set in Grandchild {DateTime.Now}");
}
```

For an alternative approach suited to sharing data in memory and across components that aren't necessarily nested, see ASP.NET Core Blazor state management.

Bound field or property expression tree

To facilitate deeper interactions with a binding, Blazor allows you to capture of the expression tree of a bound field or property. This is achieved by defining a property with the field or property name suffixed with `Expression`. For any given field or property named `{FIELD OR PROPERTY NAME}`, the corresponding expression tree property is named `{FIELD OR PROPERTY NAME}Expression`.

The following `ChildParameterExpression` component identifies the `Year` expression's model and field name. A FieldIdentifier, which is used to obtain the model and field name, uniquely identifies a single field that can be edited. This may correspond to a property on a model object or can be any other named value. Use of a parameter's expression is useful when creating custom validation components, which isn't covered by the Microsoft Blazor documentation but is addressed by numerous third-party resources.

`ChildParameterExpression.razor`:

razor

```
@using System.Linq.Expressions

<ul>
    <li>Year model: @yearField.Model</li>
    <li>Year field name: @yearField.FieldName</li>
</ul>

@code {
    private FieldIdentifier yearField;

    [Parameter]
    public int Year { get; set; }

    [Parameter]
```

```
    public EventCallback<int> YearChanged { get; set; }

    [Parameter]
    public Expression<Func<int>> YearExpression { get; set; } = default!;

    protected override void OnInitialized() =>
        yearField = FieldIdentifier.Create(YearExpression);
}
```

`Parent3.razor`:

razor

```
@page "/parent-3"

<PageTitle>Parent 3</PageTitle>

<h1>Parent Example 3</h1>

<p>Parent <code>year</code>: @year</p>

<ChildParameterExpression @bind-Year="year" />

@code {
    private int year = 1979;
}
```

Additional resources

- Parameter change detection and additional guidance on Razor component rendering
- ASP.NET Core Blazor forms overview
- Binding to radio buttons in a form
- Binding InputSelect options to C# object null values
- ASP.NET Core Blazor event handling: EventCallback section
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Razor component lifecycle

Article • 12/06/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains the ASP.NET Core Razor component lifecycle and how to use lifecycle events.

Lifecycle events

The Razor component processes Razor component lifecycle events in a set of synchronous and asynchronous lifecycle methods. The lifecycle methods can be overridden to perform additional operations in components during component initialization and rendering.

This article simplifies component lifecycle event processing in order to clarify complex framework logic and doesn't cover every change that was made over the years. You may need to access the ComponentBase reference source to integrate custom event processing with Blazor's lifecycle event processing. Code comments in the reference source include additional remarks on lifecycle event processing that don't appear in this article or in the API documentation.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

The following simplified diagrams illustrate Razor component lifecycle event processing. The C# methods associated with the lifecycle events are defined with examples in the following sections of this article.

Component lifecycle events:

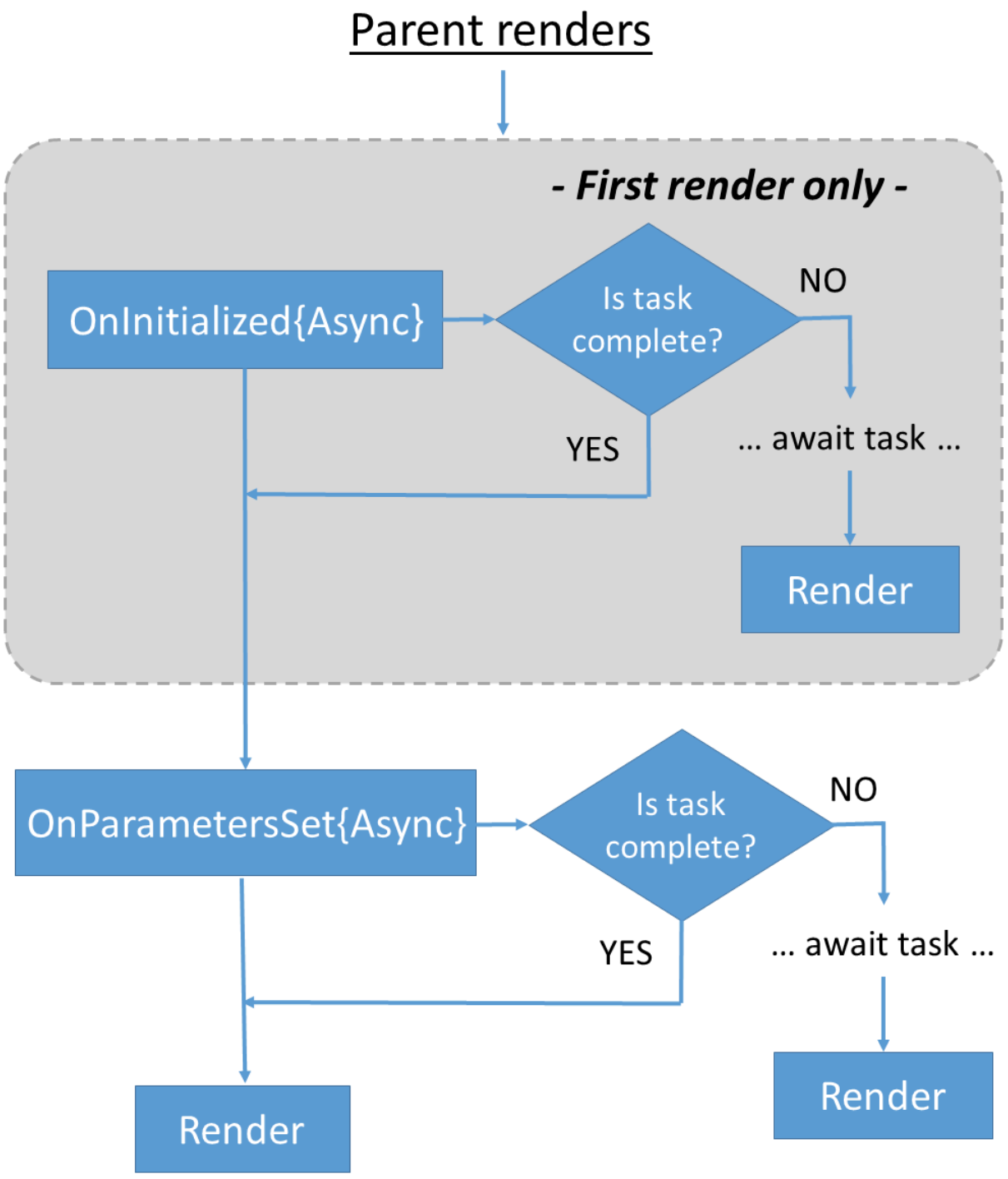
1. If the component is rendering for the first time on a request:

- Create the component's instance.
- Perform property injection.
- Call OnInitialized{Async}. If an incomplete Task is returned, the Task is awaited and then the component is rerendered. The synchronous method is called prior to the asynchronous method.

2. Call OnParametersSet{Async}. If an incomplete Task is returned, the Task is awaited and then the component is rerendered. The synchronous method is called prior to the asynchronous method.
3. Render for all synchronous work and complete Tasks.

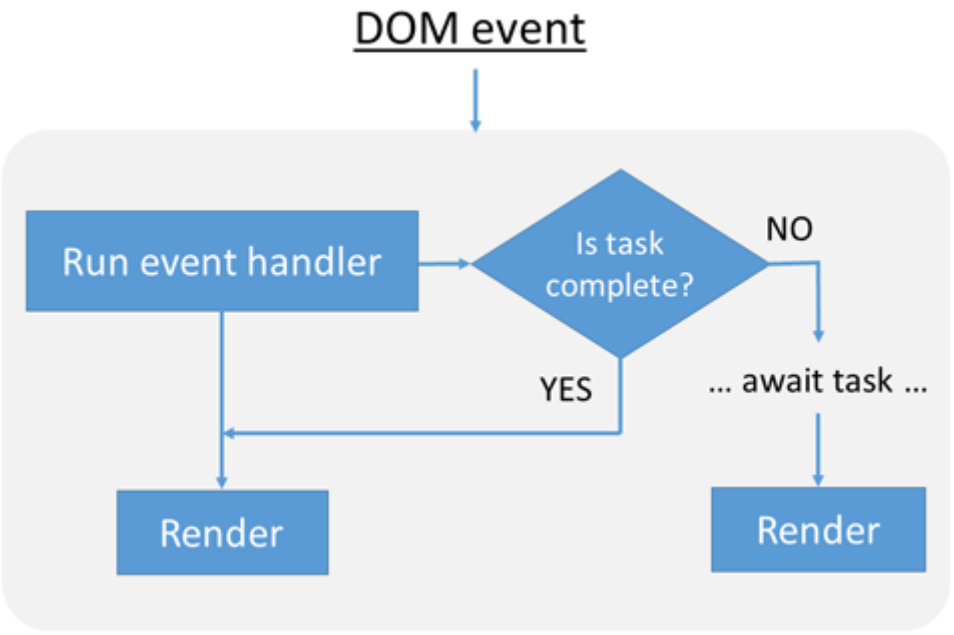
> **Note**
>
> Asynchronous actions performed in lifecycle events might delay component rendering or displaying data. For more information, see the **Handle incomplete asynchronous actions at render** section later in this article.

A parent component renders before its children components because rendering is what determines which children are present. If synchronous parent component initialization is used, the parent initialization is guaranteed to complete first. If asynchronous parent component initialization is used, the completion order of parent and child component initialization can't be determined because it depends on the initialization code running.



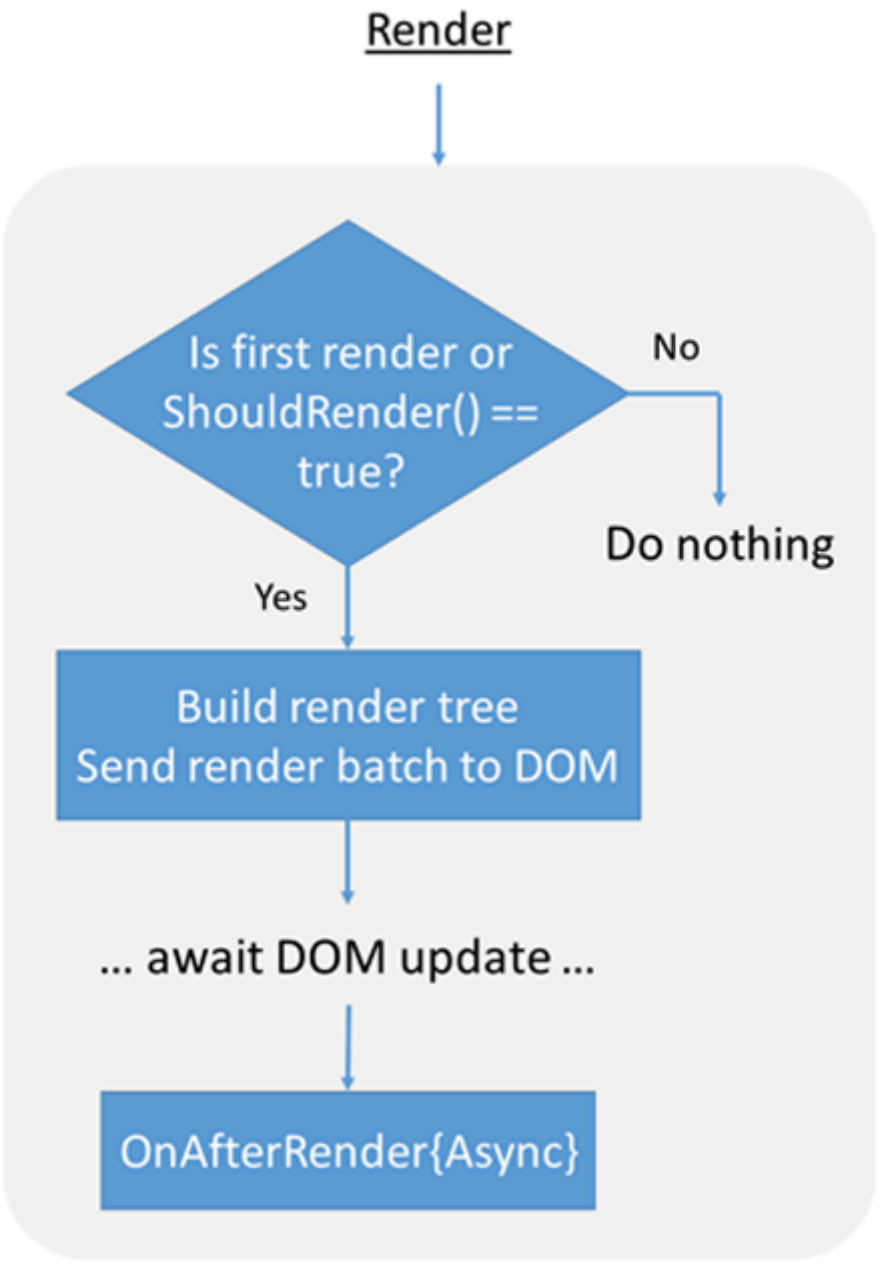
DOM event processing:

1. The event handler is run.
2. If an incomplete Task is returned, the Task is awaited and then the component is rerendered.
3. Render for all synchronous work and complete Tasks.



The `Render` lifecycle:

1. Avoid further rendering operations on the component when both of the following conditions are met:
    - It is not the first render.
    - ShouldRender returns `false`.
2. Build the render tree diff (difference) and render the component.
3. Await the DOM to update.
4. Call OnAfterRender{Async}. The synchronous method is called prior to the asynchronous method.



Developer calls to StateHasChanged result in a rerender. For more information, see ASP.NET Core Razor component rendering.

Quiescence during prerendering

In server-side Blazor apps, prerendering waits for *quiescence*, which means that a component doesn't render until all of the components in the render tree have finished rendering. Quiescence can lead to noticeable delays in rendering when a component performs long-running tasks during initialization and other lifecycle methods, leading to a poor user experience. For more information, see the Handle incomplete asynchronous actions at render section later in this article.

When parameters are set (`SetParametersAsync`)

SetParametersAsync sets parameters supplied by the component's parent in the render tree or from route parameters.

The method's ParameterView parameter contains the set of component parameter values for the component each time SetParametersAsync is called. By overriding the SetParametersAsync method, developer code can interact directly with ParameterView's parameters.

The default implementation of SetParametersAsync sets the value of each property with the [Parameter] or [CascadingParameter] attribute that has a corresponding value in the ParameterView. Parameters that don't have a corresponding value in ParameterView are left unchanged.

Generally, your code should call the base class method (`await base.SetParametersAsync(parameters);`) when overriding SetParametersAsync. In advanced scenarios, developer code can interpret the incoming parameters' values in any way required by not invoking the base class method. For example, there's no requirement to assign the incoming parameters to the properties of the class. However, you must refer to the ComponentBase reference source when structuring your code without calling the base class method because it calls other lifecycle methods and triggers rendering in a complex fashion.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

If you want to rely on the initialization and rendering logic of ComponentBase.SetParametersAsync but not process incoming parameters, you have the option of passing an empty ParameterView to the base class method:

```
C#

await base.SetParametersAsync(ParameterView.Empty);
```

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable and IAsyncDisposable section.

In the following example, ParameterView.TryGetValue assigns the `Param` parameter's value to `value` if parsing a route parameter for `Param` is successful. When `value` isn't `null`, the value is displayed by the component.

Although route parameter matching is case insensitive, TryGetValue only matches case-sensitive parameter names in the route template. The following example requires the use of `/{Param?}` in the route template in order to get the value with TryGetValue, not `/{param?}`. If `/{param?}` is used in this scenario, TryGetValue returns `false` and `message` isn't set to either `message` string.

`SetParamsAsync.razor`:

razor

```razor
@page "/set-params-async/{Param?}"

<PageTitle>Set Parameters Async</PageTitle>

<h1>Set Parameters Async Example</h1>

<p>@message</p>

@code {
    private string message = "Not set";

    [Parameter]
    public string? Param { get; set; }

    public override async Task SetParametersAsync(ParameterView parameters)
    {
        if (parameters.TryGetValue<string>(nameof(Param), out var value))
        {
            if (value is null)
            {
                message = "The value of 'Param' is null.";
            }
            else
            {
                message = $"The value of 'Param' is {value}.";
            }
        }

        await base.SetParametersAsync(parameters);
    }
}
```

Component initialization (`OnInitialized{Async}`)

OnInitialized and OnInitializedAsync are used exclusively to initialize a component for the entire lifetime of the component instance. Parameter values and parameter value changes shouldn't affect the initialization performed in these methods. For example, loading static options into a dropdown list that doesn't change for the lifetime of the component and that isn't dependent on parameter values is performed in one of these lifecycle methods. If parameter values or changes in parameter values affect component state, use OnParametersSet{Async} instead.

These methods are invoked when the component is initialized after having received its initial parameters in SetParametersAsync. The synchronous method is called prior to the asynchronous method.

If synchronous parent component initialization is used, the parent initialization is guaranteed to complete before child component initialization. If asynchronous parent component initialization is used, the completion order of parent and child component initialization can't be determined because it depends on the initialization code running.

For a synchronous operation, override OnInitialized:

`OnInit.razor`:

razor

```razor
@page "/on-init"

<PageTitle>On Initialized</PageTitle>

<h1>On Initialized Example</h1>

<p>@message</p>

@code {
    private string? message;

    protected override void OnInitialized() =>
        message = $"Initialized at {DateTime.Now}";
}
```

To perform an asynchronous operation, override OnInitializedAsync and use the await operator:

C#

```
protected override async Task OnInitializedAsync()
{
    await ...
}
```

If a custom base class is used with custom initialization logic, call OnInitializedAsync on the base class:

C#

```
protected override async Task OnInitializedAsync()
{
    await ...

    await base.OnInitializedAsync();
}
```

It isn't necessary to call ComponentBase.OnInitializedAsync unless a custom base class is used with custom logic. For more information, see the Base class lifecycle methods section.

Blazor apps that prerender their content on the server call OnInitializedAsync *twice*:

- Once when the component is initially rendered statically as part of the page.
- A second time when the browser renders the component.

To prevent developer code in OnInitializedAsync from running twice when prerendering, see the Stateful reconnection after prerendering section. The content in the section focuses on Blazor Web Apps and stateful SignalR *reconnection*. To preserve state during the execution of initialization code while prerendering, see Prerender ASP.NET Core Razor components.

While a Blazor app is prerendering, certain actions, such as calling into JavaScript (JS interop), aren't possible. Components may need to render differently when prerendered. For more information, see the Prerendering with JavaScript interop section.

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable IAsyncDisposable section.

Use *streaming rendering* with static server-side rendering (static SSR) or prerendering to improve the user experience for components that perform long-running asynchronous tasks in OnInitializedAsync to fully render. For more information, see the following resources:

- Handle incomplete asynchronous actions at render (this article)

- ASP.NET Core Razor component rendering

After parameters are set (`OnParametersSet{Async}`)

OnParametersSet or OnParametersSetAsync are called:

- After the component is initialized in OnInitialized or OnInitializedAsync.
- When the parent component rerenders and supplies:
  - Known or primitive immutable types when at least one parameter has changed.
  - Complex-typed parameters. The framework can't know whether the values of a complex-typed parameter have mutated internally, so the framework always treats the parameter set as changed when one or more complex-typed parameters are present.

  For more information on rendering conventions, see ASP.NET Core Razor component rendering.

The synchronous method is called prior to the asynchronous method.

The methods can be invoked even if the parameter values haven't changed. This behavior underscores the need for developers to implement additional logic within the methods to check whether parameter values have indeed changed before re-initializing data or state dependent on those parameters.

For the following example component, navigate to the component's page at a URL:

- With a start date that's received by `StartDate`: `/on-parameters-set/2021-03-19`
- Without a start date, where `StartDate` is assigned a value of the current local time: `/on-parameters-set`

> **Note**
>
> In a component route, it isn't possible to both constrain a DateTime parameter with the route constraint datetime and make the parameter optional. Therefore, the following `OnParamsSet` component uses two @page directives to handle routing with and without a supplied date segment in the URL.

`OnParamsSet.razor`:

```
razor
```

```
@page "/on-params-set"
@page "/on-params-set/{StartDate:datetime}"

<PageTitle>On Parameters Set</PageTitle>

<h1>On Parameters Set Example</h1>

<p>
    Pass a datetime in the URI of the browser's address bar.
    For example, add <code>/1-1-2024</code> to the address.
</p>

<p>@message</p>

@code {
    private string? message;

    [Parameter]
    public DateTime StartDate { get; set; }

    protected override void OnParametersSet()
    {
        if (StartDate == default)
        {
            StartDate = DateTime.Now;

            message = $"No start date in URL. Default value applied " +
                $"(StartDate: {StartDate}).";
        }
        else
        {
            message = $"The start date in the URL was used " +
                $"(StartDate: {StartDate}).";
        }
    }
}
```

Asynchronous work when applying parameters and property values must occur during the OnParametersSetAsync lifecycle event:

C#

```
protected override async Task OnParametersSetAsync()
{
    await ...
}
```

If a custom base class is used with custom initialization logic, call OnParametersSetAsync on the base class:

C#

```
protected override async Task OnParametersSetAsync()
{
    await ...

    await base.OnParametersSetAsync();
}
```

It isn't necessary to call ComponentBase.OnParametersSetAsync unless a custom base class is used with custom logic. For more information, see the Base class lifecycle methods section.

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable IAsyncDisposable section.

For more information on route parameters and constraints, see ASP.NET Core Blazor routing and navigation.

For an example of implementing `SetParametersAsync` manually to improve performance in some scenarios, see ASP.NET Core Blazor performance best practices.

After component render (`OnAfterRender{Async}`)

OnAfterRender and OnAfterRenderAsync are invoked after a component has rendered interactively and the UI has finished updating (for example, after elements are added to the browser DOM). Element and component references are populated at this point. Use this stage to perform additional initialization steps with the rendered content, such as JS interop calls that interact with the rendered DOM elements. The synchronous method is called prior to the asynchronous method.

These methods aren't invoked during prerendering or static server-side rendering (static SSR) on the server because those processes aren't attached to a live browser DOM and are already complete before the DOM is updated.

For OnAfterRenderAsync, the component doesn't automatically rerender after the completion of any returned `Task` to avoid an infinite render loop.

The `firstRender` parameter for OnAfterRender and OnAfterRenderAsync:

- Is set to `true` the first time that the component instance is rendered.
- Can be used to ensure that initialization work is only performed once.

`AfterRender.razor`:

razor

```razor
@page "/after-render"
@inject ILogger<AfterRender> Logger

<PageTitle>After Render</PageTitle>

<h1>After Render Example</h1>

<p>
    <button @onclick="HandleClick">Log information (and trigger a render)
</button>
</p>

<p>Study logged messages in the console.</p>

@code {
    protected override void OnAfterRender(bool firstRender) =>
        Logger.LogInformation("firstRender = {FirstRender}", firstRender);

    private void HandleClick() => Logger.LogInformation("HandleClick
called");
}
```

The `AfterRender.razor` sample produces following output to console when the page is loaded and the button is selected:

> OnAfterRender: firstRender = True
> HandleClick called
> OnAfterRender: firstRender = False

Asynchronous work immediately after rendering must occur during the OnAfterRenderAsync lifecycle event:

C#

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    ...
}
```

If a custom base class is used with custom initialization logic, call OnAfterRenderAsync on the base class:

C#

```
protected override async Task OnAfterRenderAsync(bool firstRender)
{
    ...

    await base.OnAfterRenderAsync(firstRender);
}
```

It isn't necessary to call ComponentBase.OnAfterRenderAsync unless a custom base class is used with custom logic. For more information, see the Base class lifecycle methods section.

Even if you return a Task from OnAfterRenderAsync, the framework doesn't schedule a further render cycle for your component once that task completes. This is to avoid an infinite render loop. This is different from the other lifecycle methods, which schedule a further render cycle once a returned Task completes.

OnAfterRender and OnAfterRenderAsync *aren't called during the prerendering process on the server*. The methods are called when the component is rendered interactively after prerendering. When the app prerenders:

1. The component executes on the server to produce some static HTML markup in the HTTP response. During this phase, OnAfterRender and OnAfterRenderAsync aren't called.
2. When the Blazor script (`blazor.{server|webassembly|web}.js`) starts in the browser, the component is restarted in an interactive rendering mode. After a component is restarted, OnAfterRender and OnAfterRenderAsync **are** called because the app isn't in the prerendering phase any longer.

If event handlers are provided in developer code, unhook them on disposal. For more information, see the Component disposal with IDisposable IAsyncDisposable section.

Base class lifecycle methods

When overriding Blazor's lifecycle methods, it isn't necessary to call base class lifecycle methods for ComponentBase. However, a component should call an overridden base class lifecycle method in the following situations:

- When overriding ComponentBase.SetParametersAsync, `await base.SetParametersAsync(parameters);` is usually invoked because the base class method calls other lifecycle methods and triggers rendering in a complex fashion. For more information, see the When parameters are set (SetParametersAsync) section.

- If the base class method contains logic that must be executed. Library consumers usually call base class lifecycle methods when inheriting a base class because library base classes often have custom lifecycle logic to execute. If the app uses a base class from a library, consult the library's documentation for guidance.

In the following example, `base.OnInitialized();` is called to ensure that the base class's `OnInitialized` method is executed. Without the call, `BlazorRocksBase2.OnInitialized` doesn't execute.

`BlazorRocks2.razor`:

razor

```
@page "/blazor-rocks-2"
@inherits BlazorRocksBase2
@inject ILogger<BlazorRocks2> Logger

<PageTitle>Blazor Rocks!</PageTitle>

<h1>Blazor Rocks! Example 2</h1>

<p>
    @BlazorRocksText
</p>

@code {
    protected override void OnInitialized()
    {
        Logger.LogInformation("Initialization code of BlazorRocks2
executed!");

        base.OnInitialized();
    }
}
```

`BlazorRocksBase2.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample;

public class BlazorRocksBase2 : ComponentBase
{
    [Inject]
    private ILogger<BlazorRocksBase2> Logger { get; set; } = default!;

    public string BlazorRocksText { get; set; } = "Blazor rocks the
browser!";
```

```
    protected override void OnInitialized() =>
        Logger.LogInformation("Initialization code of BlazorRocksBase2
executed!");
}
```

State changes (`StateHasChanged`)

StateHasChanged notifies the component that its state has changed. When applicable, calling StateHasChanged enqueues a rerender that occurs when the app's main thread is free.

StateHasChanged is called automatically for EventCallback methods. For more information on event callbacks, see ASP.NET Core Blazor event handling.

For more information on component rendering and when to call StateHasChanged, including when to invoke it with ComponentBase.InvokeAsync, see ASP.NET Core Razor component rendering.

Handle incomplete asynchronous actions at render

Asynchronous actions performed in lifecycle events might not have completed before the component is rendered. Objects might be `null` or incompletely populated with data while the lifecycle method is executing. Provide rendering logic to confirm that objects are initialized. Render placeholder UI elements (for example, a loading message) while objects are `null`.

In the following `Slow` component, OnInitializedAsync is overridden to asynchronously execute a long-running task. While `isLoading` is `true`, a loading message is displayed to the user. After the `Task` returned by OnInitializedAsync completes, the component is rerendered with the updated state, showing the "`Finished!`" message.

`Slow.razor`:

razor

```
@page "/slow"

<h2>Slow Component</h2>

@if (isLoading)
{
```

```
    <div><em>Loading...</em></div>
}
else
{
    <div>Finished!</div>
}

@code {
    private bool isLoading = true;

    protected override async Task OnInitializedAsync()
    {
        await LoadDataAsync();
        isLoading = false;
    }

    private Task LoadDataAsync()
    {
        return Task.Delay(10000);
    }
}
```

The preceding component uses an `isLoading` variable to display the loading message. A similar approach is used for a component that loads data into a collection and checks if the collection is `null` to present the loading message. The following example checks the `movies` collection for `null` to either display the loading message or display the collection of movies:

razor

```
@if (movies == null)
{
    <p><em>Loading...</em></p>
}
else
{
    @* display movies *@
}

@code {
    private Movies[]? movies;

    protected override async Task OnInitializedAsync()
    {
        movies = await GetMovies();
    }
}
```

Prerendering waits for *quiescence*, which means that a component doesn't render until all of the components in the render tree have finished rendering. This means that a

loading message doesn't display while a child component's OnInitializedAsync method is executing a long-running task during prerendering. To demonstrate this behavior, place the preceding `Slow` component into a test app's `Home` component:

```razor
@page "/"

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app.

<SlowComponent />
```

> **Note**
>
> Although the examples in this section discuss the **OnInitializedAsync** lifecycle method, other lifecycle methods that execute during prerendering may delay final rendering of a component. Only **OnAfterRender{Async}** isn't executed during prerendering and is immune to delays due to quiescence.

During prerendering, the `Home` component doesn't render until the `Slow` component is rendered, which takes ten seconds. The UI is blank during this ten-second period, and there's no loading message. After prerendering, the `Home` component renders, and the `Slow` component's loading message is displayed. After ten more seconds, the `Slow` component finally displays the finished message.

As the preceding demonstration illustrates, quiescence during prerendering results in a poor user experience. To improve the user experience, begin by implementing streaming rendering to avoid waiting for the asynchronous task to complete while prerendering.

Add the [StreamRendering] attribute to the `Slow` component (use `[StreamRendering(true)]` in .NET 8):

```razor
@attribute [StreamRendering]
```

When the `Home` component is prerendering, the `Slow` component is quickly rendered with its loading message. The `Home` component doesn't wait for ten seconds for the `Slow` component to finish rendering. However, the finished message displayed at the

end of prerendering is replaced by the loading message while the component finally renders, which is another ten-second delay. This occurs because the `Slow` component is rendering twice, along with `LoadDataAsync` executing twice. When a component is accessing resources, such as services and databases, double execution of service calls and database queries creates undesirable load on the app's resources.

To address the double rendering of the loading message and the re-execution of service and database calls, persist prerendered state with PersistentComponentState for final rendering of the component, as seen in the following updates to the `Slow` component:

razor

```
@page "/slow"
@attribute [StreamRendering]
@implements IDisposable
@inject PersistentComponentState ApplicationState

<h2>Slow Component</h2>

@if (data is null)
{
    <div><em>Loading...</em></div>
}
else
{
    <div>@data</div>
}

@code {
    private string? data;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistData);

        if (!ApplicationState.TryTakeFromJson<string>("data", out var
restored))
        {
            data = await LoadDataAsync();
        }
        else
        {
            data = restored!;
        }
    }

    private Task PersistData()
    {
        ApplicationState.PersistAsJson("data", data);
```

```
        return Task.CompletedTask;
    }

    private async Task<string> LoadDataAsync()
    {
        await Task.Delay(10000);
        return "Finished!";
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

By combining streaming rendering with persistent component state:

- Services and databases only require a single call for component initialization.
- Components render their UIs quickly with loading messages during long-running tasks for the best user experience.

For more information, see the following resources:

- ASP.NET Core Razor component rendering
- Prerender ASP.NET Core Razor components.

Handle errors

For information on handling errors during lifecycle method execution, see Handle errors in ASP.NET Core Blazor apps.

Stateful reconnection after prerendering

When prerendering on the server, a component is initially rendered statically as part of the page. Once the browser establishes a SignalR connection back to the server, the component is rendered *again* and interactive. If the OnInitialized{Async} lifecycle method for initializing the component is present, the method is executed *twice*:

- When the component is prerendered statically.
- After the server connection has been established.

This can result in a noticeable change in the data displayed in the UI when the component is finally rendered. To avoid this behavior, pass in an identifier to cache the state during prerendering and to retrieve the state after prerendering.

The following code demonstrates a `WeatherForecastService` that avoids the change in data display due to prerendering. The awaited Delay (`await Task.Delay(...)`) simulates a short delay before returning data from the `GetForecastAsync` method.

Add IMemoryCache services with AddMemoryCache on the service collection in the app's `Program` file:

C#

```
builder.Services.AddMemoryCache();
```

`WeatherForecastService.cs`:

C#

```
using Microsoft.Extensions.Caching.Memory;

namespace BlazorSample;

public class WeatherForecastService(IMemoryCache memoryCache)
{
    private static readonly string[] summaries =
    [
        "Freezing", "Bracing", "Chilly", "Cool", "Mild",
        "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    ];

    public IMemoryCache MemoryCache { get; } = memoryCache;

    public Task<WeatherForecast[]?> GetForecastAsync(DateOnly startDate)
    {
        return MemoryCache.GetOrCreateAsync(startDate, async e =>
        {
            e.SetOptions(new MemoryCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow =
                    TimeSpan.FromSeconds(30)
            });

            await Task.Delay(TimeSpan.FromSeconds(10));

            return Enumerable.Range(1, 5).Select(index => new
WeatherForecast
            {
                Date = startDate.AddDays(index),
                TemperatureC = Random.Shared.Next(-20, 55),
                Summary = summaries[Random.Shared.Next(summaries.Length)]
            }).ToArray();
        });
```

```
    }
}
```

For more information on the RenderMode, see ASP.NET Core Blazor SignalR guidance.

The content in this section focuses on Blazor Web Apps and stateful SignalR *reconnection*. To preserve state during the execution of initialization code while prerendering, see Prerender ASP.NET Core Razor components.

Prerendering with JavaScript interop

This section applies to server-side apps that prerender Razor components. Prerendering is covered in Prerender ASP.NET Core Razor components.

> **Note**
>
> Internal navigation for **interactive routing** in Blazor Web Apps doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests. If the app adopts interactive routing, perform a full page reload for component examples that demonstrate prerendering behavior. For more information, see **Prerender ASP.NET Core Razor components**.

During prerendering, calling into JavaScript (JS) isn't possible. The following example demonstrates how to use JS interop as part of a component's initialization logic in a way that's compatible with prerendering.

The following `scrollElementIntoView` function:

- Scrolls to the passed element with scrollIntoView.
- Returns the element's `top` property value from the getBoundingClientRect method.

JavaScript

```
window.scrollElementIntoView = (element) => {
  element.scrollIntoView();
  return element.getBoundingClientRect().top;
}
```

Where IJSRuntime.InvokeAsync calls the JS function in component code, the ElementReference is only used in OnAfterRenderAsync and not in any earlier lifecycle method because there's no HTML DOM element until after the component is rendered.

StateHasChanged (reference source) is called to enqueue rerendering of the component with the new state obtained from the JS interop call (for more information, see ASP.NET Core Razor component rendering). An infinite loop isn't created because StateHasChanged is only called when `scrollPosition` is `null`.

`PrerenderedInterop.razor`:

razor

```razor
@page "/prerendered-interop"
@using Microsoft.AspNetCore.Components
@using Microsoft.JSInterop
@inject IJSRuntime JS

<PageTitle>Prerendered Interop</PageTitle>

<h1>Prerendered Interop Example</h1>

<div @ref="divElement" style="margin-top:2000px">
    Set value via JS interop call: <strong>@scrollPosition</strong>
</div>

@code {
    private ElementReference divElement;
    private double? scrollPosition;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender && scrollPosition is null)
        {
            scrollPosition = await JS.InvokeAsync<double>(
                "scrollElementIntoView", divElement);

            StateHasChanged();
        }
    }
}
```

The preceding example pollutes the client with a global function. For a better approach in production apps, see JavaScript isolation in JavaScript modules.

Component disposal with `IDisposable` and `IAsyncDisposable`

If a component implements IDisposable or IAsyncDisposable, the framework calls for resource disposal when the component is removed from the UI. Don't rely on the exact timing of when these methods are executed. For example, IAsyncDisposable can be

triggered before or after an asynchronous Task awaited in OnInitalizedAsync is called or completes. Also, object disposal code shouldn't assume that objects created during initialization or other lifecycle methods exist.

Components shouldn't need to implement IDisposable and IAsyncDisposable simultaneously. If both are implemented, the framework only executes the asynchronous overload.

Developer code must ensure that IAsyncDisposable implementations don't take a long time to complete.

Disposal of JavaScript interop object references

Examples throughout the JavaScript (JS) interop articles demonstrate typical object disposal patterns:

- When calling JS from .NET, as described in Call JavaScript functions from .NET methods in ASP.NET Core Blazor, dispose any created IJSObjectReference/IJSInProcessObjectReference/JSObjectReference either from .NET or from JS to avoid leaking JS memory.
- When calling .NET from JS, as described in Call .NET methods from JavaScript functions in ASP.NET Core Blazor, dispose of a created DotNetObjectReference either from .NET or from JS to avoid leaking .NET memory.

JS interop object references are implemented as a map keyed by an identifier on the side of the JS interop call that creates the reference. When object disposal is initiated from either the .NET or JS side, Blazor removes the entry from the map, and the object can be garbage collected as long as no other strong reference to the object is present.

At a minimum, always dispose objects created on the .NET side to avoid leaking .NET managed memory.

DOM cleanup tasks during component disposal

For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

For guidance on JSDisconnectedException when a circuit is disconnected, see ASP.NET Core Blazor JavaScript interoperability (JS interop). For general JavaScript interop error handling guidance, see the *JavaScript interop* section in Handle errors in ASP.NET Core Blazor apps.

Synchronous `IDisposable`

For synchronous disposal tasks, use IDisposable.Dispose.

The following component:

- Implements IDisposable with the @implements Razor directive.
- Disposes of `obj`, which is a type that implements IDisposable.
- A null check is performed because `obj` is created in a lifecycle method (not shown).

razor

```
@implements IDisposable

...

@code {
    ...

    public void Dispose()
    {
        obj?.Dispose();
    }
}
```

If a single object requires disposal, a lambda can be used to dispose of the object when Dispose is called. The following example appears in the ASP.NET Core Razor component rendering article and demonstrates the use of a lambda expression for the disposal of a Timer.

`TimerDisposal1.razor`:

razor

```
@page "/timer-disposal-1"
@using System.Timers
@implements IDisposable

<PageTitle>Timer Disposal 1</PageTitle>

<h1>Timer Disposal Example 1</h1>

<p>Current count: @currentCount</p>

@code {
    private int currentCount = 0;
    private Timer timer = new(1000);

    protected override void OnInitialized()
    {
```

```
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
    }

    private void OnTimerCallback()
    {
        _ = InvokeAsync(() =>
        {
            currentCount++;
            StateHasChanged();
        });
    }

    public void Dispose() => timer.Dispose();
}
```

> **Note**
>
> In the preceding example, the call to StateHasChanged is wrapped by a call to ComponentBase.InvokeAsync because the callback is invoked outside of Blazor's synchronization context. For more information, see ASP.NET Core Razor component rendering.

If the object is created in a lifecycle method, such as OnInitialized{Async}, check for `null` before calling `Dispose`.

`TimerDisposal2.razor`:

razor

```
@page "/timer-disposal-2"
@using System.Timers
@implements IDisposable

<PageTitle>Timer Disposal 2</PageTitle>

<h1>Timer Disposal Example 2</h1>

<p>Current count: @currentCount</p>

@code {
    private int currentCount = 0;
    private Timer? timer;

    protected override void OnInitialized()
    {
        timer = new Timer(1000);
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
```

```
    }

    private void OnTimerCallback()
    {
        _ = InvokeAsync(() =>
        {
            currentCount++;
            StateHasChanged();
        });
    }

    public void Dispose() => timer?.Dispose();
}
```

For more information, see:

- Cleaning up unmanaged resources (.NET documentation)
- Null-conditional operators ?. and ?[]

Asynchronous `IAsyncDisposable`

For asynchronous disposal tasks, use IAsyncDisposable.DisposeAsync.

The following component:

- Implements IAsyncDisposable with the @implements Razor directive.
- Disposes of `obj`, which is an unmanaged type that implements IAsyncDisposable.
- A null check is performed because `obj` is created in a lifecycle method (not shown).

razor

```
@implements IAsyncDisposable

...

@code {
    ...

    public async ValueTask DisposeAsync()
    {
        if (obj is not null)
        {
            await obj.DisposeAsync();
        }
    }
}
```

For more information, see:

- Cleaning up unmanaged resources (.NET documentation)
- Null-conditional operators ?. and ?[]

Assignment of `null` to disposed objects

Usually, there's no need to assign `null` to disposed objects after calling Dispose/DisposeAsync. Rare cases for assigning `null` include the following:

- If the object's type is poorly implemented and doesn't tolerate repeat calls to Dispose/DisposeAsync, assign `null` after disposal to gracefully skip further calls to Dispose/DisposeAsync.
- If a long-lived process continues to hold a reference to a disposed object, assigning `null` allows the garbage collector to free the object in spite of the long-lived process holding a reference to it.

These are unusual scenarios. For objects that are implemented correctly and behave normally, there's no point in assigning `null` to disposed objects. In the rare cases where an object must be assigned `null`, we recommend documenting the reason and seeking a solution that prevents the need to assign `null`.

`StateHasChanged`

> ⓘ Note
>
> Calling StateHasChanged in `Dispose` and `DisposeAsync` isn't supported. StateHasChanged might be invoked as part of tearing down the renderer, so requesting UI updates at that point isn't supported.

Event handlers

Always unsubscribe event handlers from .NET events. The following Blazor form examples show how to unsubscribe an event handler in the `Dispose` method:

- Private field and lambda approach

```razor
@implements IDisposable
```

```
<EditForm ... EditContext="editContext" ...>
    ...
    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    ...

    private EventHandler<FieldChangedEventArgs>? fieldChanged;

    protected override void OnInitialized()
    {
        editContext = new(model);

        fieldChanged = (_, __) =>
        {
            ...
        };

        editContext.OnFieldChanged += fieldChanged;
    }

    public void Dispose()
    {
        editContext.OnFieldChanged -= fieldChanged;
    }
}
```

- Private method approach

razor

```
@implements IDisposable

<EditForm ... EditContext="editContext" ...>
    ...
    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    ...

    protected override void OnInitialized()
    {
        editContext = new(model);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object sender,
FieldChangedEventArgs e)
    {
        ...
    }
```

```
    public void Dispose()
    {
        editContext.OnFieldChanged -= HandleFieldChanged;
    }
}
```

For more information, see the Component disposal with IDisposable and IAsyncDisposable section.

For more information on the EditForm component and forms, see ASP.NET Core Blazor forms overview and the other forms articles in the *Forms* node.

Anonymous functions, methods, and expressions

When anonymous functions, methods, or expressions, are used, it isn't necessary to implement IDisposable and unsubscribe delegates. However, failing to unsubscribe a delegate is a problem **when the object exposing the event outlives the lifetime of the component registering the delegate**. When this occurs, a memory leak results because the registered delegate keeps the original object alive. Therefore, only use the following approaches when you know that the event delegate disposes quickly. When in doubt about the lifetime of objects that require disposal, subscribe a delegate method and properly dispose the delegate as the earlier examples show.

- Anonymous lambda method approach (explicit disposal not required):

```
C#

private void HandleFieldChanged(object sender, FieldChangedEventArgs e)
{
    formInvalid = !editContext.Validate();
    StateHasChanged();
}

protected override void OnInitialized()
{
    editContext = new(starship);
    editContext.OnFieldChanged += (s, e) =>
HandleFieldChanged((editContext)s, e);
}
```

- Anonymous lambda expression approach (explicit disposal not required):

```
C#

private ValidationMessageStore? messageStore;
```

```
[CascadingParameter]
private EditContext? CurrentEditContext { get; set; }

protected override void OnInitialized()
{
    ...

    messageStore = new(CurrentEditContext);

    CurrentEditContext.OnValidationRequested += (s, e) =>
messageStore.Clear();
    CurrentEditContext.OnFieldChanged += (s, e) =>
        messageStore.Clear(e.FieldIdentifier);
}
```

The full example of the preceding code with anonymous lambda expressions appears in the ASP.NET Core Blazor forms validation article.

For more information, see Cleaning up unmanaged resources and the topics that follow it on implementing the `Dispose` and `DisposeAsync` methods.

Disposal during JS interop

Trap JSDisconnectedException in potential cases where loss of Blazor's SignalR circuit prevents JS interop calls and results an unhandled exception.

For more information, see the following resources:

- JavaScript isolation in JavaScript modules
- JavaScript interop calls without a circuit

Cancelable background work

Components often perform long-running background work, such as making network calls (HttpClient) and interacting with databases. It's desirable to stop the background work to conserve system resources in several situations. For example, background asynchronous operations don't automatically stop when a user navigates away from a component.

Other reasons why background work items might require cancellation include:

- An executing background task was started with faulty input data or processing parameters.
- The current set of executing background work items must be replaced with a new set of work items.

- The priority of currently executing tasks must be changed.
- The app must be shut down for server redeployment.
- Server resources become limited, necessitating the rescheduling of background work items.

To implement a cancelable background work pattern in a component:

- Use a CancellationTokenSource and CancellationToken.
- On disposal of the component and at any point cancellation is desired by manually canceling the token, call CancellationTokenSource.Cancel to signal that the background work should be cancelled.
- After the asynchronous call returns, call ThrowIfCancellationRequested on the token.

In the following example:

- `await Task.Delay(10000, cts.Token);` represents long-running asynchronous background work.
- `BackgroundResourceMethod` represents a long-running background method that shouldn't start if the `Resource` is disposed before the method is called.

`BackgroundWork.razor`:

razor

```
@page "/background-work"
@implements IDisposable
@inject ILogger<BackgroundWork> Logger

<PageTitle>Background Work</PageTitle>

<h1>Background Work Example</h1>

<p>
    <button @onclick="LongRunningWork">Trigger long running work</button>
    <button @onclick="Dispose">Trigger Disposal</button>
</p>
<p>Study logged messages in the console.</p>
<p>
    If you trigger disposal within 10 seconds of page load, the
    <code>BackgroundResourceMethod</code> isn't executed.
</p>
<p>
    If disposal occurs after <code>BackgroundResourceMethod</code> is called
but
    before action is taken on the resource, an
<code>ObjectDisposedException</code>
    is thrown by <code>BackgroundResourceMethod</code>, and the resource
isn't
```

```
    processed.
</p>

@code {
    private Resource resource = new();
    private CancellationTokenSource cts = new();
    private IList<string> messages = [];

    protected async Task LongRunningWork()
    {
        Logger.LogInformation("Long running work started");

        await Task.Delay(10000, cts.Token);

        cts.Token.ThrowIfCancellationRequested();
        resource.BackgroundResourceMethod(Logger);
    }

    public void Dispose()
    {
        Logger.LogInformation("Executing Dispose");

        if (!cts.IsCancellationRequested)
        {
            cts.Cancel();
        }

        cts?.Dispose();
        resource?.Dispose();
    }

    private class Resource : IDisposable
    {
        private bool disposed;

        public void BackgroundResourceMethod(ILogger<BackgroundWork> logger)
        {
            logger.LogInformation("BackgroundResourceMethod: Start method");

            if (disposed)
            {
                logger.LogInformation("BackgroundResourceMethod: Disposed");
                throw new ObjectDisposedException(nameof(Resource));
            }

            // Take action on the Resource

            logger.LogInformation("BackgroundResourceMethod: Action on
Resource");
        }

        public void Dispose() => disposed = true;
    }
}
```

Blazor Server reconnection events

The component lifecycle events covered in this article operate separately from server-side reconnection event handlers. When the SignalR connection to the client is lost, only UI updates are interrupted. UI updates are resumed when the connection is re-established. For more information on circuit handler events and configuration, see ASP.NET Core Blazor SignalR guidance.

Additional resources

Handle caught exceptions outside of a Razor component's lifecycle

ASP.NET Core Razor component virtualization

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use component virtualization in ASP.NET Core Blazor apps.

Virtualization

Improve the perceived performance of component rendering using the Blazor framework's built-in virtualization support with the Virtualize<TItem> component. Virtualization is a technique for limiting UI rendering to just the parts that are currently visible. For example, virtualization is helpful when the app must render a long list of items and only a subset of items is required to be visible at any given time.

Use the Virtualize<TItem> component when:

- Rendering a set of data items in a loop.
- Most of the items aren't visible due to scrolling.
- The rendered items are the same size.

When the user scrolls to an arbitrary point in the Virtualize<TItem> component's list of items, the component calculates the visible items to show. Unseen items aren't rendered.

Without virtualization, a typical list might use a C# foreach loop to render each item in a list. In the following example:

- `allFlights` is a collection of airplane flights.
- The `FlightSummary` component displays details about each flight.
- The @key directive attribute preserves the relationship of each `FlightSummary` component to its rendered flight by the flight's `FlightId`.

```
razor
```

```razor
<div style="height:500px;overflow-y:scroll">
    @foreach (var flight in allFlights)
    {
        <FlightSummary @key="flight.FlightId" Details="@flight.Summary" />
    }
</div>
```

If the collection contains thousands of flights, rendering the flights takes a long time and users experience a noticeable UI lag. Most of the flights fall outside of the height of the `<div>` element, so most of them aren't seen.

Instead of rendering the entire list of flights at once, replace the foreach loop in the preceding example with the Virtualize<TItem> component:

- Specify `allFlights` as a fixed item source to Virtualize<TItem>.Items. Only the currently visible flights are rendered by the Virtualize<TItem> component.

  If a non-generic collection supplies the items, for example a collection of DataRow, follow the guidance in the Item provider delegate section to supply the items.

- Specify a context for each flight with the `Context` parameter. In the following example, `flight` is used as the context, which provides access to each flight's members.

razor

```razor
<div style="height:500px;overflow-y:scroll">
    <Virtualize Items="allFlights" Context="flight">
        <FlightSummary @key="flight.FlightId" Details="@flight.Summary" />
    </Virtualize>
</div>
```

If a context isn't specified with the `Context` parameter, use the value of `context` in the item content template to access each flight's members:

razor

```razor
<div style="height:500px;overflow-y:scroll">
    <Virtualize Items="allFlights">
        <FlightSummary @key="context.FlightId" Details="@context.Summary" />
    </Virtualize>
</div>
```

The Virtualize<TItem> component:

- Calculates the number of items to render based on the height of the container and the size of the rendered items.
- Recalculates and rerenders the items as the user scrolls.
- Only fetches the slice of records from an external API that correspond to the currently visible region, including overscan, when `ItemsProvider` is used instead of `Items` (see the Item provider delegate section).

The item content for the Virtualize<TItem> component can include:

- Plain HTML and Razor code, as the preceding example shows.
- One or more Razor components.
- A mix of HTML/Razor and Razor components.

Item provider delegate

If you don't want to load all of the items into memory or the collection isn't a generic ICollection<T>, you can specify an items provider delegate method to the component's Virtualize<TItem>.ItemsProvider parameter that asynchronously retrieves the requested items on demand. In the following example, the `LoadEmployees` method provides the items to the Virtualize<TItem> component:

razor

```
<Virtualize Context="employee" ItemsProvider="LoadEmployees">
    <p>
        @employee.FirstName @employee.LastName has the
        job title of @employee.JobTitle.
    </p>
</Virtualize>
```

The items provider receives an ItemsProviderRequest, which specifies the required number of items starting at a specific start index. The items provider then retrieves the requested items from a database or other service and returns them as an ItemsProviderResult<TItem> along with a count of the total items. The items provider can choose to retrieve the items with each request or cache them so that they're readily available.

A Virtualize<TItem> component can only accept **one item source** from its parameters, so don't attempt to simultaneously use an items provider and assign a collection to `Items`. If both are assigned, an InvalidOperationException is thrown when the component's parameters are set at runtime.

The following example loads employees from an `EmployeeService` (not shown). The `totalEmployees` field would typically be assigned by calling a method on the same service (for example, `EmployeesService.GetEmployeesCountAsync`) elsewhere, such as during component initialization.

C#

```
private async ValueTask<ItemsProviderResult<Employee>> LoadEmployees(
    ItemsProviderRequest request)
{
    var numEmployees = Math.Min(request.Count, totalEmployees -
request.StartIndex);
    var employees = await
EmployeesService.GetEmployeesAsync(request.StartIndex,
        numEmployees, request.CancellationToken);

    return new ItemsProviderResult<Employee>(employees, totalEmployees);
}
```

In the following example, a collection of DataRow is a non-generic collection, so an items provider delegate is used for virtualization:

razor

```
<Virtualize Context="row" ItemsProvider="GetRows">
    ...
</Virtualize>

@code{
    ...

    private ValueTask<ItemsProviderResult<DataRow>>
GetRows(ItemsProviderRequest request) =>
        new(new ItemsProviderResult<DataRow>(
            dataTable.Rows.OfType<DataRow>
().Skip(request.StartIndex).Take(request.Count),
            dataTable.Rows.Count));
}
```

Virtualize<TItem>.RefreshDataAsync instructs the component to rerequest data from its ItemsProvider. This is useful when external data changes. There's usually no need to call RefreshDataAsync when using Items.

RefreshDataAsync updates a Virtualize<TItem> component's data without causing a rerender. If RefreshDataAsync is invoked from a Blazor event handler or component lifecycle method, triggering a render isn't required because a render is automatically triggered at the end of the event handler or lifecycle method. If RefreshDataAsync is

triggered separately from a background task or event, such as in the following `ForecastUpdated` delegate, call StateHasChanged to update the UI at the end of the background task or event:

C#

```
<Virtualize ... @ref="virtualizeComponent">
    ...
</Virtualize>

...

private Virtualize<FetchData>? virtualizeComponent;

protected override void OnInitialized()
{
    WeatherForecastSource.ForecastUpdated += async () =>
    {
        await InvokeAsync(async () =>
        {
            await virtualizeComponent?.RefreshDataAsync();
            StateHasChanged();
        });
    });
}
```

In the preceding example:

- RefreshDataAsync is called first to obtain new data for the Virtualize<TItem> component.
- `StateHasChanged` is called to rerender the component.

Placeholder

Because requesting items from a remote data source might take some time, you have the option to render a placeholder with item content:

- Use a Placeholder (`<Placeholder>...</Placeholder>`) to display content until the item data is available.
- Use Virtualize<TItem>.ItemContent to set the item template for the list.

razor

```
<Virtualize Context="employee" ItemsProvider="LoadEmployees">
    <ItemContent>
        <p>
            @employee.FirstName @employee.LastName has the
```

```
            job title of @employee.JobTitle.
        </p>
    </ItemContent>
    <Placeholder>
        <p>
            Loading&hellip;
        </p>
    </Placeholder>
</Virtualize>
```

Empty content

Use the EmptyContent parameter to supply content when the component has loaded and either Items is empty or ItemsProviderResult<TItem>.TotalItemCount is zero.

`EmptyContent.razor`:

razor

```
@page "/empty-content"

<PageTitle>Empty Content</PageTitle>

<h1>Empty Content Example</h1>

<Virtualize Items="stringList">
    <ItemContent>
        <p>
            @context
        </p>
    </ItemContent>
    <EmptyContent>
        <p>
            There are no strings to display.
        </p>
    </EmptyContent>
</Virtualize>

@code {
    private List<string>? stringList;

    protected override void OnInitialized() => stringList ??= new();
}
```

Change the `OnInitialized` method lambda to see the component display strings:

C#

```
protected override void OnInitialized() =>
    stringList ??= new() { "Here's a string!", "Here's another string!" };
```

Item size

The height of each item in pixels can be set with Virtualize<TItem>.ItemSize (default: 50). The following example changes the height of each item from the default of 50 pixels to 25 pixels:

razor

```
<Virtualize Context="employee" Items="employees" ItemSize="25">
    ...
</Virtualize>
```

The Virtualize<TItem> component measures the rendering size (height) of individual items *after* the initial render occurs. Use ItemSize to provide an exact item size in advance to assist with accurate initial render performance and to ensure the correct scroll position for page reloads. If the default ItemSize causes some items to render outside of the currently visible view, a second rerender is triggered. To correctly maintain the browser's scroll position in a virtualized list, the initial render must be correct. If not, users might view the wrong items.

Overscan count

Virtualize<TItem>.OverscanCount determines how many additional items are rendered before and after the visible region. This setting helps to reduce the frequency of rendering during scrolling. However, higher values result in more elements rendered in the page (default: 3). The following example changes the overscan count from the default of three items to four items:

razor

```
<Virtualize Context="employee" Items="employees" OverscanCount="4">
    ...
</Virtualize>
```

State changes

When making changes to items rendered by the Virtualize<TItem> component, call StateHasChanged to enqueue re-evaluation and rerendering of the component. For more information, see ASP.NET Core Razor component rendering.

Keyboard scroll support

To allow users to scroll virtualized content using their keyboard, ensure that the virtualized elements or scroll container itself is focusable. If you fail to take this step, keyboard scrolling doesn't work in Chromium-based browsers.

For example, you can use a `tabindex` attribute on the scroll container:

razor

```razor
<div style="height:500px; overflow-y:scroll" tabindex="-1">
    <Virtualize Items="allFlights">
        <div class="flight-info">...</div>
    </Virtualize>
</div>
```

To learn more about the meaning of `tabindex` value `-1`, `0`, or other values, see tabindex (MDN documentation).

Advanced styles and scroll detection

The Virtualize<TItem> component is only designed to support specific element layout mechanisms. To understand which element layouts work correctly, the following explains how `Virtualize` detects which elements should be visible for display in the correct place.

If your source code looks like the following:

razor

```razor
<div style="height:500px; overflow-y:scroll" tabindex="-1">
    <Virtualize Items="allFlights" ItemSize="100">
        <div class="flight-info">Flight @context.Id</div>
    </Virtualize>
</div>
```

At runtime, the Virtualize<TItem> component renders a DOM structure similar to the following:

HTML

```
<div style="height:500px; overflow-y:scroll" tabindex="-1">
    <div style="height:1100px"></div>
    <div class="flight-info">Flight 12</div>
    <div class="flight-info">Flight 13</div>
    <div class="flight-info">Flight 14</div>
    <div class="flight-info">Flight 15</div>
    <div class="flight-info">Flight 16</div>
    <div style="height:3400px"></div>
</div>
```

The actual number of rows rendered and the size of the spacers vary according to your styling and `Items` collection size. However, notice that there are spacer `div` elements injected before and after your content. These serve two purposes:

- To provide an offset before and after your content, causing currently-visible items to appear at the correct location in the scroll range and the scroll range itself to represent the total size of all content.
- To detect when the user is scrolling beyond the current visible range, meaning that different content must be rendered.

> ⓘ **Note**
>
> To learn how to control the spacer HTML element tag, see the Control the spacer element tag name section later in this article.

The spacer elements internally use an Intersection Observer to receive notification when they're becoming visible. `Virtualize` depends on receiving these events.

`Virtualize` works under the following conditions:

- **All rendered content items, including placeholder content, are of identical height.** This makes it possible to calculate which content corresponds to a given scroll position without first fetching every data item and rendering the data into a DOM element.
- **Both the spacers and the content rows are rendered in a single vertical stack with every item filling the entire horizontal width.** In typical use cases, `Virtualize` works with `div` elements. If you're using CSS to create a more advanced layout, bear in mind the following requirements:
  - Scroll container styling requires a `display` with any of the following values:
    - `block` (the default for a `div`).

    - `table-row-group` (the default for a `tbody`).
    - `flex` with `flex-direction` set to `column`. Ensure that immediate children of the Virtualize<TItem> component don't shrink under flex rules. For example, add `.mycontainer > div { flex-shrink: 0 }`.
  - Content row styling requires a `display` with either of the following values:
    - `block` (the default for a `div`).
    - `table-row` (the default for a `tr`).
  - Don't use CSS to interfere with the layout for the spacer elements. The spacer elements have a `display` value of `block`, except if the parent is a table row group, in which case they default to `table-row`. Don't try to influence spacer element width or height, including by causing them to have a border or `content` pseudo-elements.

Any approach that stops the spacers and content elements from rendering as a single vertical stack, or causes the content items to vary in height, prevents correct functioning of the Virtualize<TItem> component.

Root-level virtualization

The Virtualize<TItem> component supports using the document itself as the scroll root, as an alternative to having some other element with `overflow-y: scroll`. In the following example, the `<html>` or `<body>` elements are styled in a component with `overflow-y: scroll`:

razor

```
<HeadContent>
    <style>
        html, body { overflow-y: scroll }
    </style>
</HeadContent>
```

Control the spacer element tag name

If the Virtualize<TItem> component is placed inside an element that requires a specific child tag name, SpacerElement allows you to obtain or set the virtualization spacer tag name. The default value is `div`. For the following example, the Virtualize<TItem> component renders inside a table body element (tbody), so the appropriate child element for a table row (tr) is set as the spacer.

`VirtualizedTable.razor`:

razor

```razor
@page "/virtualized-table"

<PageTitle>Virtualized Table</PageTitle>

<HeadContent>
    <style>
        html, body {
            overflow-y: scroll
        }
    </style>
</HeadContent>

<h1>Virtualized Table Example</h1>

<table id="virtualized-table">
    <thead style="position: sticky; top: 0; background-color: silver">
        <tr>
            <th>Item</th>
            <th>Another column</th>
        </tr>
    </thead>
    <tbody>
        <Virtualize Items="fixedItems" ItemSize="30" SpacerElement="tr">
            <tr @key="context" style="height: 30px;" id="row-@context">
                <td>Item @context</td>
                <td>Another value</td>
            </tr>
        </Virtualize>
    </tbody>
</table>

@code {
    private List<int> fixedItems = Enumerable.Range(0, 1000).ToList();
}
```

In the preceding example, the document root is used as the scroll container, so the `html` and `body` elements are styled with `overflow-y: scroll`. For more information, see the following resources:

- Root-level virtualization section
- Control head content in ASP.NET Core Blazor apps

ASP.NET Core Razor component rendering

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains Razor component rendering in ASP.NET Core Blazor apps, including when to call StateHasChanged to manually trigger a component to render.

Rendering conventions for `ComponentBase`

Components *must* render when they're first added to the component hierarchy by a parent component. This is the only time that a component must render. Components *may* render at other times according to their own logic and conventions.

Razor components inherit from the ComponentBase base class, which contains logic to trigger rerendering at the following times:

- After applying an updated set of parameters from a parent component.
- After applying an updated value for a cascading parameter.
- After notification of an event and invoking one of its own event handlers.
- After a call to its own StateHasChanged method (see ASP.NET Core Razor component lifecycle). For guidance on how to prevent overwriting child component parameters when StateHasChanged is called in a parent component, see Avoid overwriting parameters in ASP.NET Core Blazor.

Components inherited from ComponentBase skip rerenders due to parameter updates if either of the following are true:

- All of the parameters are from a set of known types† or any primitive type that hasn't changed since the previous set of parameters were set.

  †The Blazor framework uses a set of built-in rules and explicit parameter type checks for change detection. These rules and the types are subject to change at any time. For more information, see the ChangeDetection API in the ASP.NET Core reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

- The override of the component's ShouldRender method returns `false` (the default `ComponentBase` implementation always returns `true`).

Control the rendering flow

In most cases, ComponentBase conventions result in the correct subset of component rerenders after an event occurs. Developers aren't usually required to provide manual logic to tell the framework which components to rerender and when to rerender them. The overall effect of the framework's conventions is that the component receiving an event rerenders itself, which recursively triggers rerendering of descendant components whose parameter values may have changed.

For more information on the performance implications of the framework's conventions and how to optimize an app's component hierarchy for rendering, see ASP.NET Core Blazor performance best practices.

Streaming rendering

Use *streaming rendering* with static server-side rendering (static SSR) or prerendering to stream content updates on the response stream and improve the user experience for components that perform long-running asynchronous tasks to fully render.

For example, consider a component that makes a long-running database query or web API call to render data when the page loads. Normally, asynchronous tasks executed as part of rendering a server-side component must complete before the rendered response is sent, which can delay loading the page. Any significant delay in rendering the page harms the user experience. To improve the user experience, streaming rendering initially renders the entire page quickly with placeholder content while asynchronous operations execute. After the operations are complete, the updated content is sent to the client on the same response connection and patched into the DOM.

Streaming rendering requires the server to avoid buffering the output. The response data must flow to the client as the data is generated. For hosts that enforce buffering, streaming rendering degrades gracefully, and the page loads without streaming rendering.

To stream content updates when using static server-side rendering (static SSR) or prerendering, apply the [StreamRendering] attribute in .NET 9 or later (use `[StreamRendering(true)]` in .NET 8) to the component. Streaming rendering must be explicitly enabled because streamed updates may cause content on the page to shift. Components without the attribute automatically adopt streaming rendering if the parent component uses the feature. Pass `false` to the attribute in a child component to disable the feature at that point and further down the component subtree. The attribute is functional when applied to components supplied by a Razor class library.

The following example is based on the `Weather` component in an app created from the Blazor Web App project template. The call to Task.Delay simulates retrieving weather data asynchronously. The component initially renders placeholder content ("`Loading...`") without waiting for the asynchronous delay to complete. When the asynchronous delay completes and the weather data content is generated, the content is streamed to the response and patched into the weather forecast table.

`Weather.razor`:

razor

```razor
@page "/weather"
@attribute [StreamRendering]

...

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        ...
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
```

```
        </tbody>
    </table>
}

@code {
    ...

    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        await Task.Delay(500);

        ...

        forecasts = ...
    }
}
```

Suppress UI refreshing (`ShouldRender`)

ShouldRender is called each time a component is rendered. Override ShouldRender to manage UI refreshing. If the implementation returns `true`, the UI is refreshed.

Even if ShouldRender is overridden, the component is always initially rendered.

`ControlRender.razor`:

razor

```
@page "/control-render"

<PageTitle>Control Render</PageTitle>

<h1>Control Render Example</h1>

<label>
    <input type="checkbox" @bind="shouldRender" />
    Should Render?
</label>

<p>Current count: @currentCount</p>

<p>
    <button @onclick="IncrementCount">Click me</button>
</p>

@code {
    private int currentCount = 0;
    private bool shouldRender = true;
```

```
    protected override bool ShouldRender() => shouldRender;

    private void IncrementCount() => currentCount++;
}
```

For more information on performance best practices pertaining to ShouldRender, see ASP.NET Core Blazor performance best practices.

StateHasChanged

Calling StateHasChanged enqueues a rerender to occur when the app's main thread is free.

Components are enqueued for rendering, and they aren't enqueued again if there's already a pending rerender. If a component calls StateHasChanged five times in a row in a loop, the component only renders once. This behavior is encoded in ComponentBase, which checks first if it has queued a rerender before enqueuing an additional one.

A component can render multiple times during the same cycle, which commonly occurs when a component has children that interact with each other:

- A parent component renders several children.
- Child components render and trigger an update on the parent.
- A parent component rerenders with new state.

This design allows for StateHasChanged to be called when necessary without the risk of introducing unnecessary rendering. You can always take control of this behavior in individual components by implementing IComponent directly and manually handling when the component renders.

Consider the following `IncrementCount` method that increments a count, calls StateHasChanged, and increments the count again:

C#

```
private void IncrementCount()
{
    currentCount++;
    StateHasChanged();
    currentCount++;
}
```

Stepping through the code in the debugger, you might think that the count updates in the UI for the first `currentCount++` execution immediately after StateHasChanged is

called. However, the UI doesn't show an updated count at that point due to the synchronous processing taking place for this method's execution. There's no opportunity for the renderer to render the component until after the event handler is finished. The UI displays increases for both `currentCount++` executions *in a single render*.

If you await something between the `currentCount++` lines, the awaited call gives the renderer a chance to render. This has led to some developers calling Delay with a one millisecond delay in their components to allow a render to occur, but we don't recommend arbitrarily slowing down an app to enqueue a render.

The best approach is to await Task.Yield, which forces the component to process code asynchronously and render during the current batch with a second render in a separate batch after the yielded task runs the continuation.

Consider the following revised `IncrementCount` method, which updates the UI twice because the render enqueued by StateHasChanged is performed when the task is yielded with the call to Task.Yield:

C#

```
private async Task IncrementCount()
{
    currentCount++;
    StateHasChanged();
    await Task.Yield();
    currentCount++;
}
```

Be careful not to call StateHasChanged unnecessarily, which is a common mistake that imposes unnecessary rendering costs. Code shouldn't need to call StateHasChanged when:

- Routinely handling events, whether synchronously or asynchronously, since ComponentBase triggers a render for most routine event handlers.
- Implementing typical lifecycle logic, such as OnInitialized or OnParametersSetAsync, whether synchronously or asynchronously, since ComponentBase triggers a render for typical lifecycle events.

However, it might make sense to call StateHasChanged in the cases described in the following sections of this article:

- An asynchronous handler involves multiple asynchronous phases
- Receiving a call from something external to the Blazor rendering and event handling system

- To render component outside the subtree that is rerendered by a particular event

An asynchronous handler involves multiple asynchronous phases

Due to the way that tasks are defined in .NET, a receiver of a Task can only observe its final completion, not intermediate asynchronous states. Therefore, ComponentBase can only trigger rerendering when the Task is first returned and when the Task finally completes. The framework can't know to rerender a component at other intermediate points, such as when an IAsyncEnumerable<T> returns data in a series of intermediate Tasks. If you want to rerender at intermediate points, call StateHasChanged at those points.

Consider the following `CounterState1` component, which updates the count four times each time the `IncrementCount` method executes:

- Automatic renders occur after the first and last increments of `currentCount`.
- Manual renders are triggered by calls to StateHasChanged when the framework doesn't automatically trigger rerenders at intermediate processing points where `currentCount` is incremented.

`CounterState1.razor`:

razor

```
@page "/counter-state-1"

<PageTitle>Counter State 1</PageTitle>

<h1>Counter State Example 1</h1>

<p>
    Current count: @currentCount
</p>

<p>
    <button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
</p>

@code {
    private int currentCount = 0;

    private async Task IncrementCount()
    {
        currentCount++;
        // Renders here automatically
```

```
        await Task.Delay(1000);
        currentCount++;
        StateHasChanged();

        await Task.Delay(1000);
        currentCount++;
        StateHasChanged();

        await Task.Delay(1000);
        currentCount++;
        // Renders here automatically
    }
}
```

Receiving a call from something external to the Blazor rendering and event handling system

ComponentBase only knows about its own lifecycle methods and Blazor-triggered events. ComponentBase doesn't know about other events that may occur in code. For example, any C# events raised by a custom data store are unknown to Blazor. In order for such events to trigger rerendering to display updated values in the UI, call StateHasChanged.

Consider the following `CounterState2` component that uses System.Timers.Timer to update a count at a regular interval and calls StateHasChanged to update the UI:

- `OnTimerCallback` runs outside of any Blazor-managed rendering flow or event notification. Therefore, `OnTimerCallback` must call StateHasChanged because Blazor isn't aware of the changes to `currentCount` in the callback.
- The component implements IDisposable, where the Timer is disposed when the framework calls the `Dispose` method. For more information, see ASP.NET Core Razor component lifecycle.

Because the callback is invoked outside of Blazor's synchronization context, the component must wrap the logic of `OnTimerCallback` in ComponentBase.InvokeAsync to move it onto the renderer's synchronization context. This is equivalent to marshalling to the UI thread in other UI frameworks. StateHasChanged can only be called from the renderer's synchronization context and throws an exception otherwise:

> System.InvalidOperationException: 'The current thread is not associated with the Dispatcher. Use InvokeAsync() to switch execution to the Dispatcher when triggering rendering or component state.'

`CounterState2.razor`:

razor

```razor
@page "/counter-state-2"
@using System.Timers
@implements IDisposable

<PageTitle>Counter State 2</PageTitle>

<h1>Counter State Example 2</h1>

<p>
    This counter demonstrates <code>Timer</code> disposal.
</p>

<p>
    Current count: @currentCount
</p>

@code {
    private int currentCount = 0;
    private Timer timer = new(1000);

    protected override void OnInitialized()
    {
        timer.Elapsed += (sender, eventArgs) => OnTimerCallback();
        timer.Start();
    }

    private void OnTimerCallback()
    {
        _ = InvokeAsync(() =>
        {
            currentCount++;
            StateHasChanged();
        });
    }

    public void Dispose() => timer.Dispose();
}
```

To render a component outside the subtree that's rerendered by a particular event

The UI might involve:

1. Dispatching an event to one component.
2. Changing some state.

3. Rerendering a completely different component that isn't a descendant of the component receiving the event.

One way to deal with this scenario is to provide a *state management* class, often as a dependency injection (DI) service, injected into multiple components. When one component calls a method on the state manager, the state manager raises a C# event that's then received by an independent component.

For approaches to manage state, see the following resources:

- Bind across more than two components using data bindings.
- Pass data across a component hierarchy using cascading values and parameters.
- Server-side in-memory state container service (client-side equivalent) section of the *State management* article.

For the state manager approach, C# events are outside the Blazor rendering pipeline. Call StateHasChanged on other components you wish to rerender in response to the state manager's events.

The state manager approach is similar to the earlier case with System.Timers.Timer in the previous section. Since the execution call stack typically remains on the renderer's synchronization context, calling InvokeAsync isn't normally required. Calling InvokeAsync is only required if the logic escapes the synchronization context, such as calling ContinueWith on a Task or awaiting a Task with ConfigureAwait(false). For more information, see the Receiving a call from something external to the Blazor rendering and event handling system section.

WebAssembly loading progress indicator for Blazor Web Apps

A loading progress indicator isn't present in an app created from the Blazor Web App project template. A new loading progress indicator feature is planned for a future release of .NET. In the meantime, an app can adopt custom code to create a loading progress indicator. For more information, see ASP.NET Core Blazor startup.

ASP.NET Core Blazor templated components

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Robert Haken

This article explains how templated components can accept one or more UI templates as parameters, which can then be used as part of the component's rendering logic.

Templated components

Templated components are components that receive one or more UI templates as parameters, which can be utilized in the rendering logic of the component. By using templated components, you can create higher-level components that are more reusable. Examples include:

- A table component that allows a user to specify templates for the table's header, rows, and footer.
- A list component that allows a user to specify a template for rendering items in a list.
- A navigation bar component that allows a user to specify a template for start content and navigation links.

A templated component is defined by specifying one or more component parameters of type RenderFragment or RenderFragment<TValue>. A render fragment represents a segment of UI to render. RenderFragment<TValue> takes a type parameter that can be specified when the render fragment is invoked.

> **Note**
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

Often, templated components are generically typed, as the following `TemplatedNavBar` component (`TemplatedNavBar.razor`) demonstrates. The generic type (`<T>`) in the

following example is used to render IReadOnlyList<T> values, which in this case is a list of pets for a component that displays a navigation bar with links to a pet detail component.

`TemplatedNavBar.razor`:

```
@typeparam TItem

<nav class="navbar navbar-expand navbar-light bg-light">
    <div class="container justify-content-start">
        @StartContent
        <div class="navbar-nav">
            @foreach (var item in Items)
            {
                @ItemTemplate(item)
            }
        </div>
    </div>
</nav>

@code {
    [Parameter]
    public RenderFragment? StartContent { get; set; }

    [Parameter, EditorRequired]
    public RenderFragment<TItem> ItemTemplate { get; set; } = default!;

    [Parameter, EditorRequired]
    public IReadOnlyList<TItem> Items { get; set; } = default!;
}
```

When using a templated component, the template parameters can be specified using child elements that match the names of the parameters. In the following example, `<StartContent>...</StartContent>` and `<ItemTemplate>...</ItemTemplate>` supply RenderFragment<TValue> templates for `StartContent` and `ItemTemplate` of the `TemplatedNavBar` component.

Specify the `Context` attribute on the component element when you want to specify the content parameter name for implicit child content (without any wrapping child element). In the following example, the `Context` attribute appears on the `TemplatedNavBar` element and applies to all RenderFragment<TValue> template parameters.

`Pets1.razor`:

```
@page "/pets-1"

<PageTitle>Pets 1</PageTitle>
```

```
<h1>Pets Example 1</h1>

<TemplatedNavBar Items="pets" Context="pet">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate>
        <NavLink href="@($"/pet-detail/{pet.PetId}?ReturnUrl=%2Fpets-1")"
class="nav-link">
            @pet.Name
        </NavLink>
    </ItemTemplate>
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
        new Pet { PetId = 3, Name = "K-9" }
    };

    private class Pet
    {
        public int PetId { get; set; }
        public string? Name { get; set; }
    }
}
```

Alternatively, you can change the parameter name using the `Context` attribute on the RenderFragment<TValue> child element. In the following example, the `Context` is set on `ItemTemplate` rather than `TemplatedNavBar`.

`Pets2.razor`:

```
@page "/pets-2"

<PageTitle>Pets 2</PageTitle>

<h1>Pets Example 2</h1>

<TemplatedNavBar Items="pets">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate Context="pet">
        <NavLink href="@($"/pet-detail/{pet.PetId}?ReturnUrl=%2Fpets-2")"
class="nav-link">
            @pet.Name
        </NavLink>
    </ItemTemplate>
```

```
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
        new Pet { PetId = 3, Name = "K-9" }
    };

    private class Pet
    {
        public int PetId { get; set; }
        public string? Name { get; set; }
    }
}
```

Component arguments of type RenderFragment<TValue> have an implicit parameter named `context`, which can be used. In the following example, `Context` isn't set. `@context.{PROPERTY}` supplies pet values to the template, where `{PROPERTY}` is a `Pet` property:

`Pets3.razor`:

```
@page "/pets-3"

<PageTitle>Pets 3</PageTitle>

<h1>Pets Example 3</h1>

<TemplatedNavBar Items="pets">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate>
        <NavLink href="@($"/pet-detail/{context.PetId}?ReturnUrl=%2Fpets-
3")" class="nav-link">
            @context.Name
        </NavLink>
    </ItemTemplate>
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
        new Pet { PetId = 3, Name = "K-9" }
    };

    private class Pet
```

```
    {
        public int PetId { get; set; }
        public string? Name { get; set; }
    }
}
```

When using generic-typed components, the type parameter is inferred if possible. However, you can explicitly specify the type with an attribute that has a name matching the type parameter, which is `TItem` in the preceding example:

`Pets4.razor`:

```
@page "/pets-4"

<PageTitle>Pets 4</PageTitle>

<h1>Pets Example 4</h1>

<TemplatedNavBar Items="pets" Context="pet" TItem="Pet">
    <StartContent>
        <a href="/" class="navbar-brand">PetsApp</a>
    </StartContent>
    <ItemTemplate>
        <NavLink href="@($"/pet-detail/{pet.PetId}?ReturnUrl=%2Fpets-4")"
class="nav-link">
            @pet.Name
        </NavLink>
    </ItemTemplate>
</TemplatedNavBar>

@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 2, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 4, Name = "Salem Saberhagen" },
        new Pet { PetId = 7, Name = "K-9" }
    };

    private class Pet
    {
        public int PetId { get; set; }
        public string? Name { get; set; }
    }
}
```

The example provided in the `TemplatedNavBar` component (`TemplatedNavBar.razor`) assumes that the `Items` collection doesn't change after the initial render; or that if it does change, maintaining the state of components/elements used in `ItemTemplate` isn't

necessary. For templated components where such usage can't be anticipated, see the Preserve relationships with @key section.

Preserve relationships with `@key`

Templated components are often used to render collections of items, such as tables or lists. In such general scenarios, we can't assume that the user will avoid stateful components/elements in the item template definition or that there won't be additional changes to the `Items` collection. For such templated components, it's necessary to preserve the relationships with the `@key` directive attribute.

> ⓘ Note
>
> For more information on the `@key` directive attribute, see Retain element, component, and model relationships in ASP.NET Core Blazor.

The following `TableTemplate` component (`TableTemplate.razor`) demonstrates a templated component that preserves relationships with `@key`.

`TableTemplate.razor`:

```
@typeparam TItem

<table class="table">
    <thead>
        <tr>@TableHeader</tr>
    </thead>
    <tbody>
        @foreach (var item in Items)
        {
            <tr @key="@item">@RowTemplate(item)</tr>
        }
    </tbody>
</table>

@code {
    [Parameter]
    public RenderFragment? TableHeader { get; set; }

    [Parameter, EditorRequired]
    public RenderFragment<TItem> RowTemplate { get; set; } = default!;

    [Parameter, EditorRequired]
    public IReadOnlyList<TItem> Items { get; set; } = default!;
}
```

Consider the following `Pets5` component (`Pets5.razor`), which demonstrates the importance of keying data to preserve model relationships. In the component, each iteration of adding a pet in OnAfterRenderAsync results in Blazor rerendering the `TableTemplate` component.

`Pets5.razor`:

```
@page "/pets-5"

<PageTitle>Pets 5</PageTitle>

<h1>Pets Example 5</h1>

<TableTemplate Items="pets" Context="pet">
    <TableHeader>
        <th>ID</th>
        <th>Name</th>
    </TableHeader>
    <RowTemplate>
        <td><input value="@pet.PetId" /></td>
        <td><input value="@pet.Name" /></td>
    </RowTemplate>
</TableTemplate>


@code {
    private List<Pet> pets = new()
    {
        new Pet { PetId = 1, Name = "Mr. Bigglesworth" },
        new Pet { PetId = 2, Name = "Salem Saberhagen" },
        new Pet { PetId = 3, Name = "K-9" }
    };

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        // Insert new pet every 5 seconds
        if (pets.Count < 10)
        {
            await Task.Delay(5000);
            pets.Insert(0, new Pet() { PetId = pets.Count + 1, Name = "<new
pet>" });
            StateHasChanged();
        }
    }

    private class Pet
    {
        public int PetId { get; set; }
        public string? Name { get; set; }
    }
}
```

This demonstration allows you to:

- Select an `<input>` from among several rendered table rows.
- Study the behavior of the page's focus as the pets collection automatically grows.

Without using the `@key` directive attribute in the `TableTemplate` component, the page's focus remains on the same index position (row) of the table, causing the focus to shift each time a pet is added. To demonstrate this, remove the `@key` directive attribute and value, restart the app, and attempt to modify a field value as items are added.

Additional resources

- ASP.NET Core Blazor performance best practices
- Retain element, component, and model relationships in ASP.NET Core Blazor
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor CSS isolation

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Dave Brock

This article explains how CSS isolation scopes CSS to Razor components, which can simplify CSS and avoid collisions with other components or libraries.

Isolate CSS styles to individual pages, views, and components to reduce or avoid:

- Dependencies on global styles that can be challenging to maintain.
- Style conflicts in nested content.

Enable CSS isolation

To define component-specific styles, create a `.razor.css` file matching the name of the `.razor` file for the component in the same folder. The `.razor.css` file is a *scoped CSS file*.

For an `Example` component in an `Example.razor` file, create a file alongside the component named `Example.razor.css`. The `Example.razor.css` file must reside in the same folder as the `Example` component (`Example.razor`). The "`Example`" base name of the file is **not** case-sensitive.

`Example.razor`:

```razor
@page "/example"

<h1>Scoped CSS Example</h1>
```

`Example.razor.css`:

```css
```

```
h1 {
    color: brown;
    font-family: Tahoma, Geneva, Verdana, sans-serif;
}
```

The styles defined in `Example.razor.css` are only applied to the rendered output of the `Example` component. CSS isolation is applied to HTML elements in the matching Razor file. Any `h1` CSS declarations defined elsewhere in the app don't conflict with the `Example` component's styles.

> ⓘ Note
>
> In order to guarantee style isolation when bundling occurs, importing CSS in Razor code blocks isn't supported.

CSS isolation bundling

CSS isolation occurs at build time. Blazor rewrites CSS selectors to match markup rendered by the component. The rewritten CSS styles are bundled and produced as a static asset. The stylesheet is referenced inside the `<head>` tag (location of <head> content). The following `<link>` element is added to an app created from the Blazor project templates:

HTML

```
<link href="{ASSEMBLY NAME}.styles.css" rel="stylesheet">
```

The `{ASSEMBLY NAME}` placeholder is the project's assembly name.

Within the bundled file, each component is associated with a scope identifier. For each styled component, an HTML attribute is appended with the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. The identifier is unique for each app. In the rendered `Counter` component, Blazor appends a scope identifier to the `h1` element:

HTML

```
<h1 b-3xxtam6d07>
```

The `{ASSEMBLY NAME}.styles.css` file uses the scope identifier to group a style declaration with its component. The following example provides the style for the preceding `<h1>` element:

```css
/* /Components/Pages/Counter.razor.rz.scp.css */
h1[b-3xxtam6d07] {
    color: brown;
}
```

At build time, a project bundle is created with the convention `obj/{CONFIGURATION}/{TARGET FRAMEWORK}/scopedcss/projectbundle/{ASSEMBLY NAME}.bundle.scp.css`, where the placeholders are:

- `{CONFIGURATION}`: The app's build configuration (for example, `Debug`, `Release`).
- `{TARGET FRAMEWORK}`: The target framework (for example, `net6.0`).
- `{ASSEMBLY NAME}`: The app's assembly name (for example, `BlazorSample`).

Child component support

CSS isolation only applies to the component you associate with the format `{COMPONENT NAME}.razor.css`, where the `{COMPONENT NAME}` placeholder is usually the component name. To apply changes to a child component, use the `::deep` pseudo-element to any descendant elements in the parent component's `.razor.css` file. The `::deep` pseudo-element selects elements that are *descendants* of an element's generated scope identifier.

The following example shows a parent component called `Parent` with a child component called `Child`.

`Parent.razor`:

```razor
@page "/parent"

<div>
    <h1>Parent component</h1>

    <Child />
</div>
```

`Child.razor`:

```razor
<h1>Child Component</h1>
```

Update the `h1` declaration in `Parent.razor.css` with the `::deep` pseudo-element to signify the `h1` style declaration must apply to the parent component and its children.

`Parent.razor.css`:

```css
::deep h1 {
    color: red;
}
```

The `h1` style now applies to the `Parent` and `Child` components without the need to create a separate scoped CSS file for the child component.

The `::deep` pseudo-element only works with descendant elements. The following markup applies the `h1` styles to components as expected. The parent component's scope identifier is applied to the `div` element, so the browser knows to inherit styles from the parent component.

`Parent.razor`:

```razor
<div>
    <h1>Parent</h1>

    <Child />
</div>
```

However, excluding the `div` element removes the descendant relationship. In the following example, the style is **not** applied to the child component.

`Parent.razor`:

```razor
<h1>Parent</h1>
```

```
<Child />
```

The `::deep` pseudo-element affects where the scope attribute is applied to the rule. When you define a CSS rule in a scoped CSS file, the scope is applied to the right most element. For example: `div > a` is transformed to `div > a[b-{STRING}]`, where the `{STRING}` placeholder is a ten-character string generated by the framework (for example, `b-3xxtam6d07`). If you instead want the rule to apply to a different selector, the `::deep` pseudo-element allows you do so. For example, `div ::deep > a` is transformed to `div[b-{STRING}] > a` (for example, `div[b-3xxtam6d07] > a`).

The ability to attach the `::deep` pseudo-element to any HTML element allows you to create scoped CSS styles that affect elements rendered by other components when you can determine the structure of the rendered HTML tags. For a component that renders an hyperlink tag (`<a>`) inside another component, ensure the component is wrapped in a `div` (or any other element) and use the rule `::deep > a` to create a style that's only applied to that component when the parent component renders.

> **ⓘ Important**
>
> Scoped CSS only applies to ***HTML elements*** and not to Razor components or Tag Helpers, including elements with a Tag Helper applied, such as `<input asp-for="..." />`.

CSS preprocessor support

CSS preprocessors are useful for improving CSS development by utilizing features such as variables, nesting, modules, mixins, and inheritance. While CSS isolation doesn't natively support CSS preprocessors such as Sass or Less, integrating CSS preprocessors is seamless as long as preprocessor compilation occurs before Blazor rewrites the CSS selectors during the build process. Using Visual Studio for example, configure existing preprocessor compilation as a **Before Build** task in the Visual Studio Task Runner Explorer.

Many third-party NuGet packages, such as AspNetCore.SassCompiler, can compile SASS/SCSS files at the beginning of the build process before CSS isolation occurs.

CSS isolation configuration

CSS isolation is designed to work out-of-the-box but provides configuration for some advanced scenarios, such as when there are dependencies on existing tools or workflows.

Customize scope identifier format

Scope identifiers use the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. To customize the scope identifier format, update the project file to a desired pattern:

XML

```
<ItemGroup>
  <None Update="Components/Pages/Example.razor.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

In the preceding example, the CSS generated for `Example.razor.css` changes its scope identifier from `b-{STRING}` to `custom-scope-identifier`.

Use scope identifiers to achieve inheritance with scoped CSS files. In the following project file example, a `BaseComponent.razor.css` file contains common styles across components. A `DerivedComponent.razor.css` file inherits these styles.

XML

```
<ItemGroup>
  <None Update="Components/Pages/BaseComponent.razor.css" CssScope="custom-scope-identifier" />
  <None Update="Components/Pages/DerivedComponent.razor.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

Use the wildcard (`*`) operator to share scope identifiers across multiple files:

XML

```
<ItemGroup>
  <None Update="Components/Pages/*.razor.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

Change base path for static web assets

The `scoped.styles.css` file is generated at the root of the app. In the project file, use the <StaticWebAssetBasePath> property to change the default path. The following example places the `scoped.styles.css` file, and the rest of the app's assets, at the `_content` path:

XML

```
<PropertyGroup>
  <StaticWebAssetBasePath>_content/$(PackageId)</StaticWebAssetBasePath>
</PropertyGroup>
```

Disable automatic bundling

To opt out of how Blazor publishes and loads scoped files at runtime, use the `DisableScopedCssBundling` property. When using this property, it means other tools or processes are responsible for taking the isolated CSS files from the `obj` directory and publishing and loading them at runtime:

XML

```
<PropertyGroup>
  <DisableScopedCssBundling>true</DisableScopedCssBundling>
</PropertyGroup>
```

Disable CSS isolation

Disable CSS isolation for a project by setting the `<ScopedCssEnabled>` property to `false` in the app's project file:

XML

```
<ScopedCssEnabled>false</ScopedCssEnabled>
```

Razor class library (RCL) support

Isolated styles for components in a NuGet package or Razor class library (RCL) are automatically bundled:

- The app uses CSS imports to reference the RCL's bundled styles. For a class library named `ClassLib` and a Blazor app with a `BlazorSample.styles.css` stylesheet, the RCL's stylesheet is imported at the top of the app's stylesheet:

```css
@import '_content/ClassLib/ClassLib.bundle.scp.css';
```

- The RCL's bundled styles aren't published as a static web asset of the app that consumes the styles.

For more information on RCLs, see the following articles:

- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- Reusable Razor UI in class libraries with ASP.NET Core

Additional resources

- Razor Pages CSS isolation
- MVC CSS isolation

Dynamically-rendered ASP.NET Core Razor components

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Dave Brock

Use the built-in DynamicComponent component to render components by type.

Dynamic components

A DynamicComponent is useful for rendering components without iterating through possible types or using conditional logic. For example, DynamicComponent can render a component based on a user selection from a dropdown list.

In the following example:

- `componentType` specifies the type.
- `parameters` specifies component parameters to pass to the `componentType` component.

razor

```
<DynamicComponent Type="componentType" Parameters="parameters" />

@code {
    private Type componentType = ...;
    private IDictionary<string, object> parameters = ...;
}
```

For more information on passing parameter values, see the Pass parameters section later in this article.

Example

In the following example, a Razor component renders a component based on the user's selection from a dropdown list of four possible values.

Expand table

User spaceflight carrier selection	Shared Razor component to render
Rocket Lab®	`RocketLab.razor`
SpaceX®	`SpaceX.razor`
ULA®	`UnitedLaunchAlliance.razor`
Virgin Galactic®	`VirginGalactic.razor`

`RocketLab.razor`:

razor

```razor
<h2>Rocket Lab®</h2>

<p>
    Rocket Lab is a registered trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>
```

`SpaceX.razor`:

razor

```razor
<h2>SpaceX®</h2>

<p>
    SpaceX is a registered trademark of
    <a href="https://www.spacex.com/">Space Exploration Technologies Corp.
</a>
</p>
```

`UnitedLaunchAlliance.razor`:

razor

```razor
<h2>United Launch Alliance®</h2>

<p>
    United Launch Alliance and ULA are registered trademarks of
    <a href="https://www.ulalaunch.com/">United Launch Alliance, LLC</a>.
</p>
```

`VirginGalactic.razor`:

```razor
<h2>Virgin Galactic®</h2>

<p>
    Virgin Galactic is a registered trademark of
    <a href="https://www.virgingalactic.com/">Galactic Enterprises, LLC</a>.
</p>
```

`DynamicComponent1.razor`:

```razor
@page "/dynamic-component-1"

<PageTitle>Dynamic Component 1</PageTitle>

<h1>Dynamic Component Example 1</h1>

<p>
    <label>
        Select your transport:
        <select @onchange="OnDropdownChange">
            <option value="">Select a value</option>
            @foreach (var entry in components.Keys)
            {
                <option value="@entry">@entry</option>
            }
        </select>
    </label>
</p>

@if (selectedType is not null)
{
    <div class="border border-primary my-1 p-1">
        <DynamicComponent Type="selectedType" />
    </div>
}

@code {
    private readonly Dictionary<string, Type> components = new()
    {
        ["Rocket Lab"] = typeof(RocketLab),
        ["SpaceX"] = typeof(SpaceX),
        ["ULA"] = typeof(UnitedLaunchAlliance),
        ["Virgin Galactic"] = typeof(VirginGalactic)
    };
    private Type? selectedType;

    private void OnDropdownChange(ChangeEventArgs e)
```

```
    {
        if ((e.Value is string dropdownValue) &&
            !String.IsNullOrWhiteSpace(dropdownValue))
        {
            selectedType = components[dropdownValue];
        }
        else
        {
            selectedType = null;
        }
    }
}
```

In the preceding example:

- An Dictionary<TKey,TValue> is used to manage components to be displayed.
- Names serve as the dictionary keys and are provided as selection options.
- Component types are stored as dictionary values using the typeof operator.

Pass parameters

If dynamically-rendered components have component parameters, pass them into the DynamicComponent as an `IDictionary<string, object>`. The `string` is the name of the parameter, and the `object` is the parameter's value.

The following example configures a component metadata object (`ComponentMetadata`) to supply parameter values to dynamically-rendered components based on the type name. The example is just one of several approaches that you can adopt. Parameter data can also be provided from a web API, a database, or a method. The only requirement is that the approach returns an `IDictionary<string, object>`.

`ComponentMetadata.cs`:

C#

```
namespace BlazorSample;

public class ComponentMetadata
{
    public required Type Type { get; init; }
    public required string Name { get; init; }
    public Dictionary<string, object> Parameters { get; } = [];
}
```

The following `RocketLabWithWindowSeat` component (`RocketLabWithWindowSeat.razor`) has been updated from the preceding example to include a component parameter

named `WindowSeat` to specify if the passenger prefers a window seat on their flight:

`RocketLabWithWindowSeat.razor`:

razor

```razor
<h2>Rocket Lab®</h2>

<p>
    User selected a window seat: @WindowSeat
</p>

<p>
    Rocket Lab is a trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>

@code {
    [Parameter]
    public bool WindowSeat { get; set; }
}
```

In the following example:

- Only the `RocketLabWithWindowSeat` component's parameter for a window seat (`WindowSeat`) receives the value of the **`Window Seat`** checkbox.
- Component names are used as dictionary keys using the nameof operator, which returns component names as constant strings.
- The dynamically-rendered components are shared components:
  - Shown in this article section: `RocketLabWithWindowSeat` (`RocketLabWithWindowSeat.razor`)
  - Components shown in the Example section earlier in this article:
    - `SpaceX` (`SpaceX.razor`)
    - `UnitedLaunchAlliance` (`UnitedLaunchAlliance.razor`)
    - `VirginGalactic` (`VirginGalactic.razor`)

`DynamicComponent2.razor`:

razor

```razor
@page "/dynamic-component-2"

<PageTitle>Dynamic Component 2</PageTitle>

<h1>Dynamic Component Example 2</h1>

<p>
```

```
    <label>
        <input type="checkbox" @bind="windowSeat"
            @bind:after="HandleWindowSeatChanged" />
        Window Seat (Rocket Lab only)
    </label>
</p>

<p>
    <label>
        Select your transport:
        <select @onchange="OnDropdownChange">
            <option value="">Select a value</option>
            @foreach (var c in components)
            {
                <option value="@c.Key">@c.Value.Name</option>
            }
        </select>
    </label>
</p>

@if (selectedComponent is not null)
{
    <div class="border border-primary my-1 p-1">
        <DynamicComponent Type="selectedComponent.Type"
            Parameters="selectedComponent.Parameters" />
    </div>
}

@code {
    private Dictionary<string, ComponentMetadata> components =
        new()
        {
            [nameof(RocketLabWithWindowSeat)] = new ComponentMetadata()
            {
                Type = typeof(RocketLabWithWindowSeat),
                Name = "Rocket Lab with Window Seat",
                Parameters = { [nameof(RocketLabWithWindowSeat.WindowSeat)]
= false }
            },
            [nameof(VirginGalactic)] = new ComponentMetadata()
            {
                Type = typeof(VirginGalactic),
                Name = "Virgin Galactic"
            },
            [nameof(UnitedLaunchAlliance)] = new ComponentMetadata()
            {
                Type = typeof(UnitedLaunchAlliance),
                Name = "ULA"
            },
            [nameof(SpaceX)] = new ComponentMetadata()
            {
                Type = typeof(SpaceX),
                Name = "SpaceX"
            }
        };
```

```
    private ComponentMetadata? selectedComponent;
    private bool windowSeat;

    private void HandleWindowSeatChanged()
    {
        components[nameof(RocketLabWithWindowSeat)]
            .Parameters[nameof(RocketLabWithWindowSeat.WindowSeat)] =
windowSeat;
    }

    private void OnDropdownChange(ChangeEventArgs e)
    {
        if ((e.Value is string dropdownValue) &&
!String.IsNullOrWhiteSpace(dropdownValue))
        {
            selectedComponent = components[dropdownValue];
        }
        else
        {
            selectedComponent = null;
        }
    }
}
```

Event callbacks (`EventCallback`)

Event callbacks (EventCallback) can be passed to a DynamicComponent in its parameter dictionary.

`ComponentMetadata.cs`:

C#

```
namespace BlazorSample;

public class ComponentMetadata
{
    public required Type Type { get; init; }
    public required string Name { get; init; }
    public Dictionary<string, object> Parameters { get; } = [];
}
```

Implement an event callback parameter (EventCallback) within each dynamically-rendered component.

`RocketLab2.razor`:

razor

```razor
<h2>Rocket Lab®</h2>

<p>
    Rocket Lab is a registered trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>

<button @onclick="OnClickCallback">
    Trigger a Parent component method
</button>

@code {
    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

`SpaceX2.razor`:

```razor
<h2>SpaceX®</h2>

<p>
    SpaceX is a registered trademark of
    <a href="https://www.spacex.com/">Space Exploration Technologies Corp.
</a>
</p>

<button @onclick="OnClickCallback">
    Trigger a Parent component method
</button>

@code {
    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

`UnitedLaunchAlliance2.razor`:

```razor
<h2>United Launch Alliance®</h2>

<p>
    United Launch Alliance and ULA are registered trademarks of
    <a href="https://www.ulalaunch.com/">United Launch Alliance, LLC</a>.
</p>

<button @onclick="OnClickCallback">
    Trigger a Parent component method
</button>
```

```
@code {
    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

`VirginGalactic2.razor`:

razor

```
<h2>Virgin Galactic®</h2>

<p>
    Virgin Galactic is a registered trademark of
    <a href="https://www.virgingalactic.com/">Galactic Enterprises, LLC</a>.
</p>

<button @onclick="OnClickCallback">
    Trigger a Parent component method
</button>

@code {
    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

In the following parent component example, the `ShowDTMessage` method assigns a string with the current time to `message`, and the value of `message` is rendered.

The parent component passes the callback method, `ShowDTMessage` in the parameter dictionary:

- The `string` key is the callback method's name, `OnClickCallback`.
- The `object` value is created by EventCallbackFactory.Create for the parent callback method, `ShowDTMessage`. Note that the this keyword isn't supported in C# field initialization, so a C# property is used for the parameter dictionary.

`DynamicComponent3.razor`:

razor

```
@page "/dynamic-component-3"

<PageTitle>Dynamic Component 3</PageTitle>

<h1>Dynamic Component Example 3</h1>

<p>
    <label>
```

```
        Select your transport:
        <select @onchange="OnDropdownChange">
            <option value="">Select a value</option>
            @foreach (var c in Components)
            {
                <option value="@c.Key">@c.Value.Name</option>
            }
        </select>
    </label>
</p>

@if (selectedComponent is not null)
{
    <div class="border border-primary my-1 p-1">
        <DynamicComponent Type="selectedComponent.Type"
            Parameters="selectedComponent.Parameters" />
    </div>
}

<p>
    @message
</p>

@code {
    private ComponentMetadata? selectedComponent;
    private string? message;

    private Dictionary<string, ComponentMetadata> Components =>
        new()
        {
            [nameof(RocketLab2)] = new ComponentMetadata()
            {
                Type = typeof(RocketLab2),
                Name = "Rocket Lab",
                Parameters = { [nameof(RocketLab2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
ShowDTMessage) }
            },
            [nameof(VirginGalactic2)] = new ComponentMetadata()
            {
                Type = typeof(VirginGalactic2),
                Name = "Virgin Galactic",
                Parameters = { [nameof(VirginGalactic2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
ShowDTMessage) }
            },
            [nameof(UnitedLaunchAlliance2)] = new ComponentMetadata()
            {
                Type = typeof(UnitedLaunchAlliance2),
                Name = "ULA",
                Parameters = {
[nameof(UnitedLaunchAlliance2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
ShowDTMessage) }
            },
```

```
            [nameof(SpaceX2)] = new ComponentMetadata()
            {
                Type = typeof(SpaceX2),
                Name = "SpaceX",
                Parameters = { [nameof(SpaceX2.OnClickCallback)] =
                    EventCallback.Factory.Create<MouseEventArgs>(this,
ShowDTMessage) }
            }
        };

    private void OnDropdownChange(ChangeEventArgs e)
    {
        if ((e.Value is string dropdownValue) &&
!String.IsNullOrWhiteSpace(dropdownValue))
        {
            selectedComponent = Components[dropdownValue];
        }
        else
        {
            selectedComponent = null;
        }
    }

    private void ShowDTMessage(MouseEventArgs e) =>
        message = $"The current DT is: {DateTime.Now}.";
}
```

Avoid catch-all parameters

Avoid the use of catch-all parameters. If catch-all parameters are used, every explicit parameter on DynamicComponent effectively is a reserved word that you can't pass to a dynamic child. Any new parameters passed to DynamicComponent are a breaking change, as they start shadowing child component parameters that happen to have the same name. It's unlikely that the caller always knows a fixed set of parameter names to pass to all possible dynamic children.

Access the dynamically-created component instance

Use the Instance property to access the dynamically-created component instance.

Create an interface to describe the dynamically-created component instance with any methods and properties that you need to access from the parent component when the component is dynamically loaded. The following example specifies a `Log` method for implementation in components.

`Interfaces/ILoggable.cs`:

```C#
using Microsoft.AspNetCore.Components;

namespace BlazorSample.Interfaces;

public interface ILoggable : IComponent
{
    public void Log();
}
```

Each component definition implements the interface. The following example is a modified Rocket Lab® component from the Example section that logs a string via its `Log` method.

`RocketLab3.razor`:

```razor
@using BlazorSample.Interfaces
@implements ILoggable
@inject ILogger<RocketLab3> Logger

<h2>Rocket Lab®</h2>

<p>
    Rocket Lab is a registered trademark of
    <a href="https://www.rocketlabusa.com/">Rocket Lab USA Inc.</a>
</p>

@code {
    public void Log() => Logger.LogInformation("Woot! I logged this call!");
}
```

The remaining three shared components (`VirginGalactic3`, `UnitedLaunchAlliance3`, `SpaceX3`) receive similar treatment:

- The following directives are added to the components, where the `{COMPONENT TYPE}` placeholder is the component type:

  ```razor
  @using BlazorSample.Interfaces
  @implements ILoggable
  @inject ILogger<{COMPONENT TYPE}> Logger
  ```

- Each component implements a `Log` method. The log category written by the logger includes the fully-qualified name of the component type when LogInformation is called:

  razor

  ```
  @code {
      public void Log()
      {
          Logger.LogInformation("Woot! I logged this call!");
      }
  }
  ```

The parent component casts the dynamically-loaded component instance as an `ILoggable` to access members of the interface. In the following example, the loaded component's `Log` method is called when a button is selected in the UI:

razor

```
...
@using BlazorSample.Interfaces

...

<DynamicComponent Type="..." @ref="dc" />

...

<button @onclick="LogFromLoadedComponent">Log from loaded component</button>

@code {
    private DynamicComponent? dc;

    ...

    private void LogFromLoadedComponent() => (dc?.Instance as
ILoggable)?.Log();
}
```

For a working demonstration of the preceding example, see the DynamicComponent4 component in the Blazor sample app.

Trademarks

Rocket Lab is a registered trademark of Rocket Lab USA Inc. SpaceX is a registered trademark of Space Exploration Technologies Corp. United Launch Alliance and ULA

are registered trademarks of United Launch Alliance, LLC. Virgin Galactic is a registered trademark of Galactic Enterprises, LLC.

Additional resources

- ASP.NET Core Blazor event handling
- DynamicComponent

ASP.NET Core Blazor `QuickGrid` component

Article • 12/20/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

The QuickGrid component is a Razor component for quickly and efficiently displaying data in tabular form. `QuickGrid` provides a simple and convenient data grid component for common grid rendering scenarios and serves as a reference architecture and performance baseline for building data grid components. `QuickGrid` is highly optimized and uses advanced techniques to achieve optimal rendering performance.

Package

Add a package reference for the Microsoft.AspNetCore.Components.QuickGrid package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Sample app

For various `QuickGrid` demonstrations, see the **QuickGrid for Blazor** sample app. The demo site is hosted on GitHub Pages. The site loads fast thanks to static prerendering using the community-maintained BlazorWasmPrerendering.Build GitHub project.

`QuickGrid` implementation

To implement a `QuickGrid` component:

- Specify tags for the `QuickGrid` component in Razor markup (`<QuickGrid>...</QuickGrid>`).
- Name a queryable source of data for the grid. Use ***either*** of the following data sources:
  - Items: A nullable `IQueryable<TGridItem>`, where `TGridItem` is the type of data represented by each row in the grid.
  - ItemsProvider: A callback that supplies data for the grid.
- Class: An optional CSS class name. If provided, the class name is included in the `class` attribute of the rendered table.
- Theme: A theme name (default value: `default`). This affects which styling rules match the table.
- Virtualize: If true, the grid is rendered with virtualization. This is normally used in conjunction with scrolling and causes the grid to fetch and render only the data around the current scroll viewport. This can greatly improve the performance when scrolling through large data sets. If you use Virtualize, you should supply a value for ItemSize and must ensure that every row renders with a constant height. Generally, it's preferable not to use Virtualize if the amount of data rendered is small or if you're using pagination.
- ItemSize: Only applicable when using Virtualize. ItemSize defines an expected height in pixels for each row, allowing the virtualization mechanism to fetch the correct number of items to match the display size and to ensure accurate scrolling.
- ItemKey: Optionally defines a value for `@key` on each rendered row. Typically, this is used to specify a unique identifier, such as a primary key value, for each data item. This allows the grid to preserve the association between row elements and data items based on their unique identifiers, even when the `TGridItem` instances are replaced by new copies (for example, after a new query against the underlying data store). If not set, the `@key` is the `TGridItem` instance.
- Pagination: Optionally links this `TGridItem` instance with a PaginationState model, causing the grid to fetch and render only the current page of data. This is normally used in conjunction with a Paginator component or some other UI logic that displays and updates the supplied PaginationState instance.
- In the `QuickGrid` child content (RenderFragment), specify PropertyColumn<TGridItem,TProp>s, which represent `TGridItem` columns whose cells display values:
  - Property: Defines the value to be displayed in this column's cells.
  - Format: Optionally specifies a format string for the value. Using Format requires the `TProp` type to implement IFormattable.
  - Sortable: Indicates whether the data should be sortable by this column. The default value may vary according to the column type. For example, a TemplateColumn<TGridItem> is sorted if any SortBy parameter is specified.

- InitialSortDirection: Indicates the sort direction if IsDefaultSortColumn is `true`.
- IsDefaultSortColumn: Indicates whether this column should be sorted by default.
- PlaceholderTemplate: If specified, virtualized grids use this template to render cells whose data hasn't been loaded.
- HeaderTemplate: An optional template for this column's header cell. If not specified, the default header template includes the Title, along with any applicable sort indicators and options buttons.
- Title: Title text for the column. The title is rendered automatically if HeaderTemplate isn't used.

For example, add the following component to render a grid.

For Blazor Web Apps, the `QuickGrid` component must adopt an interactive render mode to enable interactive features, such as paging and sorting.

`PromotionGrid.razor`:

razor

```
@page "/promotion-grid"
@using Microsoft.AspNetCore.Components.QuickGrid

<PageTitle>Promotion Grid</PageTitle>

<h1>Promotion Grid Example</h1>

<QuickGrid Items="people">
    <PropertyColumn Property="@(p => p.PersonId)" Sortable="true" />
    <PropertyColumn Property="@(p => p.Name)" Sortable="true" />
    <PropertyColumn Property="@(p => p.PromotionDate)" Format="yyyy-MM-dd"
Sortable="true" />
</QuickGrid>

@code {
    private record Person(int PersonId, string Name, DateOnly
PromotionDate);

    private IQueryable<Person> people = new[]
    {
        new Person(10895, "Jean Martin", new DateOnly(1985, 3, 16)),
        new Person(10944, "António Langa", new DateOnly(1991, 12, 1)),
        new Person(11203, "Julie Smith", new DateOnly(1958, 10, 10)),
        new Person(11205, "Nur Sari", new DateOnly(1922, 4, 27)),
        new Person(11898, "Jose Hernandez", new DateOnly(2011, 5, 3)),
        new Person(12130, "Kenji Sato", new DateOnly(2004, 1, 9)),
    }.AsQueryable();
}
```

Access the component in a browser at the relative path `/promotion-grid`.

There aren't current plans to extend `QuickGrid` with features that full-blown commercial grids tend to offer, for example, hierarchical rows, drag-to-reorder columns, or Excel-like range selections. If you require advanced features that you don't wish to develop on your own, continue using third-party grids.

Sort by column

The `QuickGrid` component can sort items by columns. In Blazor Web Apps, sorting requires the component to adopt an interactive render mode.

Add `Sortable="true"` (Sortable) to the PropertyColumn<TGridItem,TProp> tag:

```razor
<PropertyColumn Property="..." Sortable="true" />
```

In the running app, sort the `QuickGrid` column by selecting the rendered column title.

Page items with a `Paginator` component

The `QuickGrid` component can page data from the data source. In Blazor Web Apps, paging requires the component to adopt an interactive render mode.

Add a PaginationState instance to the component's `@code` block. Set the ItemsPerPage to the number of items to display per page. In the following example, the instance is named `pagination`, and ten items per page is set:

```C#
PaginationState pagination = new PaginationState { ItemsPerPage = 10 };
```

Set the `QuickGrid` component's Pagination property to `pagination`:

```razor
<QuickGrid Items="..." Pagination="pagination">
```

To provide a UI for pagination, add a Paginator component above or below the `QuickGrid` component. Set the Paginator.State to `pagination`:

razor

```razor
<Paginator State="pagination" />
```

In the running app, page through the items using a rendered `Paginator` component.

QuickGrid renders additional empty rows to fill in the final page of data when used with a `Paginator` component. In .NET 9 or later, empty data cells (`<td></td>`) are added to the empty rows. The empty rows are intended to facilitate rendering the QuickGrid with stable row height and styling across all pages.

Apply row styles

Apply styles to rows using CSS isolation, which can include styling empty rows for `QuickGrid` components that page data with a Paginator component.

Wrap the `QuickGrid` component in a wrapper block element, for example a `<div>`:

diff

```
+ <div>
    <QuickGrid ...>
        ...
    </QuickGrid>
+ </div>
```

Apply a row style with the `::deep` pseudo-element. In the following example, row height is set to `2em`, including for empty data rows.

`{COMPONENT}.razor.css`:

css

```css
::deep tr {
    height: 2em;
}
```

Alternatively, use the following CSS styling approach:

- Display row cells populated with data.
- Don't display empty row cells, which avoids empty row cell borders from rendering per Bootstrap styling.

`{COMPONENT}.razor.css`:

```css
::deep tr:has(> td:not(:empty)) > td {
    display: table-cell;
}

::deep td:empty {
    display: none;
}
```

For more information on using `::deep` pseudo-elements with CSS isolation, see ASP.NET Core Blazor CSS isolation.

Custom attributes and styles

QuickGrid also supports passing custom attributes and style classes (Class) to the rendered table element:

```razor
<QuickGrid Items="..." custom-attribute="value" Class="custom-class">
```

Entity Framework Core (EF Core) data source

Use the factory pattern to resolve an EF Core database context that provides data to a `QuickGrid` component. For more information on why the factory pattern is recommended, see ASP.NET Core Blazor with Entity Framework Core (EF Core).

A database context factory (IDbContextFactory<TContext>) is injected into the component with the `@inject` directive. The factory approach requires disposal of the database context, so the component implements the IAsyncDisposable interface with the `@implements` directive. The item provider for the `QuickGrid` component is a `DbSet<T>` obtained from the created database context (CreateDbContext) of the injected database context factory.

QuickGrid recognizes EF-supplied IQueryable instances and knows how to resolve queries asynchronously for efficiency.

Add a package reference for the Microsoft.AspNetCore.Components.QuickGrid.EntityFrameworkAdapter NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Call AddQuickGridEntityFrameworkAdapter on the service collection in the `Program` file to register an EF-aware IAsyncQueryExecutor implementation:

C#

```
builder.Services.AddQuickGridEntityFrameworkAdapter();
```

The following example uses an `ExampleTable` DbSet<TEntity> (table) from a `AppDbContext` database context (`context`) as the data source for a `QuickGrid` component:

razor

```
@using Microsoft.AspNetCore.Components.QuickGrid
@using Microsoft.EntityFrameworkCore
@implements IAsyncDisposable
@inject IDbContextFactory<AppDbContext> DbFactory

...

<QuickGrid ... Items="context.ExampleTable" ...>
    ...
</QuickGrid>

@code {
    private AppDbContext context = default!;

    protected override void OnInitialized()
    {
        context = DbFactory.CreateDbContext();
    }

    public async ValueTask DisposeAsync() => await context.DisposeAsync();
}
```

In the code block (`@code`) of the preceding example:

- The `context` field holds the database context, typed as an `AppDbContext`.
- The `OnInitialized` lifecycle method assigns a new database context (CreateDbContext) to the `context` field from the injected factory (`DbFactory`).

- The asynchronous `DisposeAsync` method disposes of the database context when the component is disposed.

You may also use any EF-supported LINQ operator to filter the data before passing it to the Items parameter.

The following example filters movies by a movie title entered in a search box. The database context is `BlazorWebAppMoviesContext`, and the model is `Movie`. The movie's `Title` property is used for the filter operation.

razor

```
@using Microsoft.AspNetCore.Components.QuickGrid
@using Microsoft.EntityFrameworkCore
@implements IAsyncDisposable
@inject IDbContextFactory<BlazorWebAppMoviesContext> DbFactory

...

<p>
    <input type="search" @bind="titleFilter" @bind:event="oninput" />
</p>

<QuickGrid ... Items="FilteredMovies" ...>
    ...
</QuickGrid>

@code {
    private string titleFilter = string.Empty;
    private BlazorWebAppMoviesContext context = default!;

    protected override void OnInitialized()
    {
        context = DbFactory.CreateDbContext();
    }

    private IQueryable<Movie> FilteredMovies =>
        context.Movie.Where(m => m.Title!.Contains(titleFilter));

    public async ValueTask DisposeAsync() => await context.DisposeAsync();
}
```

For a working example, see the following resources:

- Build a Blazor movie database app tutorial
- Blazor movie database sample app : Select the latest version folder in the repository. The sample folder for the tutorial's project is named `BlazorWebAppMovies`.

Display name support

A column title can be assigned using ColumnBase<TGridItem>.Title in the PropertyColumn<TGridItem,TProp>'s tag. In the following movie example, the column is given the name "`Release Date`" for the column's movie release date data:

razor

```
<PropertyColumn Property="movie => movie.ReleaseDate" Title="Release Date"
/>
```

However, managing column titles (names) from bound model properties is usually a better choice for maintaining an app. A model can control the display name of a property with the [Display] attribute. In the following example, the model specifies a movie release date display name of "`Release Date`" for its `ReleaseDate` property:

C#

```
[Display(Name = "Release Date")]
public DateTime ReleaseDate { get; set; }
```

To enable the `QuickGrid` component to use the DisplayAttribute.Name property, subclass PropertyColumn<TGridItem,TProp>, either in the component or in a separate class. Call the GetName method to return the localized DisplayAttribute.Name value if an unlocalized DisplayName ([DisplayName] attribute) doesn't hold the value:

C#

```
public class DisplayNameColumn<TGridItem, TProp> : PropertyColumn<TGridItem,
TProp>
{
    protected override void OnParametersSet()
    {
        if (Title is null && Property.Body is MemberExpression
memberExpression)
        {
            var memberInfo = memberExpression.Member;
            Title =
                memberInfo.GetCustomAttribute<DisplayNameAttribute>
().DisplayName ??
                memberInfo.GetCustomAttribute<DisplayAttribute>().GetName()
??
                memberInfo.Name;
        }

        base.OnParametersSet();
```

```
    }
}
```

Use the subclass in the `QuickGrid` component. In the following example, the preceding `DisplayNameColumn` is used. The name "`Release Date`" is provided by the [Display] attribute in the model, so there's no need to specify a Title:

razor

```
<DisplayNameColumn Property="movie => movie.ReleaseDate" />
```

The [DisplayName] attribute is also supported:

C#

```
[DisplayName("Release Date")]
public DateTime ReleaseDate { get; set; }
```

However, the `[Display]` attribute is recommended because it makes additional properties available. For example, the `[Display]` attribute offers the ability to assign a resource type for localization.

Remote data

In Blazor WebAssembly apps, fetching data from a JSON-based web API on a server is a common requirement. To fetch only the data that's required for the current page/viewport of data and apply sorting or filtering rules on the server, use the ItemsProvider parameter.

ItemsProvider can also be used in a server-side Blazor app if the app is required to query an external endpoint or in other cases where the requirements aren't covered by an IQueryable.

Supply a callback matching the GridItemsProvider<TGridItem> delegate type, where `TGridItem` is the type of data displayed in the grid. The callback is given a parameter of type GridItemsProviderRequest<TGridItem>, which specifies the start index, maximum row count, and sort order of data to return. In addition to returning the matching items, a total item count (`totalItemCount`) is also required for paging and virtualization to function correctly.

The following example obtains data from the public OpenFDA Food Enforcement database.

The GridItemsProvider<TGridItem> converts the GridItemsProviderRequest<TGridItem> into a query against the OpenFDA database. Query parameters are translated into the particular URL format supported by the external JSON API. It's only possible to perform sorting and filtering via sorting and filtering that's supported by the external API. The OpenFDA endpoint doesn't support sorting, so none of the columns are marked as sortable. However, it does support skipping records (`skip` parameter) and limiting the return of records (`limit` parameter), so the component can enable virtualization and scroll quickly through tens of thousands of records.

`FoodRecalls.razor`:

razor

```razor
@page "/food-recalls"
@inject HttpClient Http
@inject NavigationManager Navigation

<PageTitle>Food Recalls</PageTitle>

<h1>OpenFDA Food Recalls</h1>

<div class="grid" tabindex="-1">
    <QuickGrid ItemsProvider="@foodRecallProvider" Virtualize="true">
        <PropertyColumn Title="ID" Property="@(c => c.Event_Id)" />
        <PropertyColumn Property="@(c => c.State)" />
        <PropertyColumn Property="@(c => c.City)" />
        <PropertyColumn Title="Company" Property="@(c => c.Recalling_Firm)"
/>
        <PropertyColumn Property="@(c => c.Status)" />
    </QuickGrid>
</div>

<p>Total: <strong>@numResults results found</strong></p>

@code {
    private GridItemsProvider<FoodRecall>? foodRecallProvider;
    private int numResults;

    protected override async Task OnInitializedAsync()
    {
        foodRecallProvider = async req =>
        {
            var url = Navigation.GetUriWithQueryParameters(
                "https://api.fda.gov/food/enforcement.json",
                new Dictionary<string, object?>
            {
                { "skip", req.StartIndex },
                { "limit", req.Count },
            });

            var response = await
```

```
Http.GetFromJsonAsync<FoodRecallQueryResult>(
                url, req.CancellationToken);

            return GridItemsProviderResult.From(
                items: response!.Results,
                totalItemCount: response!.Meta.Results.Total);
        };

        numResults = (await Http.GetFromJsonAsync<FoodRecallQueryResult>(

"https://api.fda.gov/food/enforcement.json"))!.Meta.Results.Total;
    }
}
```

For more information on calling web APIs, see Call a web API from an ASP.NET Core Blazor app.

`QuickGrid` scaffolder

The `QuickGrid` scaffolder scaffolds Razor components with `QuickGrid` to display data from a database.

The scaffolder generates basic Create, Read, Update, and Delete (CRUD) pages based on an Entity Framework Core data model. You can scaffold individual pages or all of the CRUD pages. You select the model class and the `DbContext`, optionally creating a new `DbContext` if needed.

The scaffolded Razor components are added to the project's in a generated folder named after the model class. The generated `Index` component uses a `QuickGrid` component to display the data. Customize the generated components as needed and enable interactivity to take advantage of interactive features, such as paging, sorting and filtering.

The components produced by the scaffolder require server-side rendering (SSR), so they aren't supported when running on WebAssembly.

Visual Studio

Right-click on the `Components/Pages` folder and select **Add** > **New Scaffolded Item**.

With the **Add New Scaffold Item** dialog open to **Installed** > **Common** > **Blazor** > **Razor Component**, select **Razor Components using Entity Framework (CRUD)**. Select the **Add** button.

CRUD is an acronym for Create, Read, Update, and Delete. The scaffolder produces create, edit, delete, details, and index components for the app.

Complete the **Add Razor Components using Entity Framework (CRUD)** dialog:

- The **Template** dropdown list includes other templates for specifically creating create, edit, delete, details, and list components. This dropdown list comes in handy when you only need to create a specific type of component scaffolded to a model class. Leave the **Template** dropdown list set to **CRUD** to scaffold a full set of components.
- In the **Model class** dropdown list, select the model class. A folder is created for the generated components from the model name (if the model class is named `Movie`, the folder is automatically named `MoviePages`).
- For **DbContext class**, take either of the following approaches:
  - Select an existing DbContext class that you know has a factory provider registration (AddDbContextFactory).
  - Select the + (plus sign) button and use the **Add Data Context** modal dialog to supply a new DbContext class name, which registers the class with a factory provider instead of using the context type directly as a service registration.
- After the model dialog closes, the **Database provider** dropdown list defaults to **SQL Server**. You can select the appropriate provider for the database that you're using. The options include SQL Server, SQLite, PostgreSQL, and Azure Cosmos DB.
- Select **Add**.

For an example use of the `QuickGrid` scaffolder, see Build a Blazor movie database app (Overview).

Integrate ASP.NET Core Razor components with MVC or Razor Pages

Article • 11/18/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Razor components can be integrated into Razor Pages or MVC apps. When the page or view is rendered, components can be prerendered at the same time.

> **Important**
>
> Framework changes across ASP.NET Core releases led to different sets of instructions in this article. Before using this article's guidance, confirm that the document version selector at the top of this article matches the version of ASP.NET Core that you intend to use for your app.

Prerendering can improve Search Engine Optimization (SEO) by rendering content for the initial HTTP response that search engines can use to calculate page rank.

After configuring the project, use the guidance in the following sections depending on the project's requirements:

- For components that are directly routable from user requests. Follow this guidance when visitors should be able to make an HTTP request in their browser for a component with an @page directive.
  - Use routable components in a Razor Pages app
  - Use routable components in an MVC app
- For components that aren't directly routable from user requests, see the Render components from a page or view section. Follow this guidance when the app embeds components into existing pages or views with the Component Tag Helper.

Configuration

Use the following guidance to integrate Razor components into pages or views of an existing Razor Pages or MVC app.

1. Add an imports file to the root folder of the project with the following content. Change the `{APP NAMESPACE}` placeholder to the namespace of the project.

   `_Imports.razor`:

   ```
   razor

   @using System.Net.Http
   @using Microsoft.AspNetCore.Authorization
   @using Microsoft.AspNetCore.Components.Authorization
   @using Microsoft.AspNetCore.Components.Forms
   @using Microsoft.AspNetCore.Components.Routing
   @using Microsoft.AspNetCore.Components.Web
   @using Microsoft.AspNetCore.Components.Web.Virtualization
   @using Microsoft.JSInterop
   @using {APP NAMESPACE}
   ```

2. In the project's layout file (`Pages/Shared/_Layout.cshtml` in Razor Pages apps or `Views/Shared/_Layout.cshtml` in MVC apps):

   - Add the following `<base>` tag and HeadOutlet component Tag Helper to the `<head>` element:

     ```
     CSHTML

     <base href="~/" />
     <component
     type="typeof(Microsoft.AspNetCore.Components.Web.HeadOutlet)"
         render-mode="ServerPrerendered" />
     ```

     The `href` value (the *app base path*) in the preceding example assumes that the app resides at the root URL path (`/`). If the app is a sub-application, follow the guidance in the *App base path* section of the Host and deploy ASP.NET Core Blazor article.

     The HeadOutlet component is used to render head (`<head>`) content for page titles (PageTitle component) and other head elements (HeadContent component) set by Razor components. For more information, see Control head content in ASP.NET Core Blazor apps.

   - Add a `<script>` tag for the `blazor.server.js` script immediately before the `Scripts` render section (`@await RenderSectionAsync(...)`):

     ```
     HTML
     ```

```
<script src="_framework/blazor.server.js"></script>
```

The framework adds the `blazor.server.js` script to the app. There's no need to manually add a `blazor.server.js` script file to the app.

> **Note**
>
> Typically, the layout loads via a `_ViewStart.cshtml` file.

3. Register the Blazor Server services in `Program.cs` where services are registered:

```
C#

builder.Services.AddServerSideBlazor();
```

4. Add the Blazor Hub endpoint to the endpoints of `Program.cs` where routes are mapped. Place the following line after the call to `MapRazorPages` (Razor Pages) or `MapControllerRoute` (MVC):

```
C#

app.MapBlazorHub();
```

5. Integrate components into any page or view. For example, add a `Counter` component to the project's `Shared` folder.

   `Pages/Shared/Counter.razor` (Razor Pages) or `Views/Shared/Counter.razor` (MVC):

```
razor

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
```

```
    }
}
```

Razor Pages:

In the project's `Index` page of a Razor Pages app, add the `Counter` component's namespace and embed the component into the page. When the `Index` page loads, the `Counter` component is prerendered in the page. In the following example, replace the `{APP NAMESPACE}` placeholder with the project's namespace.

`Pages/Index.cshtml`:

CSHTML

```
@page
@using {APP NAMESPACE}.Pages.Shared
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

MVC:

In the project's `Index` view of an MVC app, add the `Counter` component's namespace and embed the component into the view. When the `Index` view loads, the `Counter` component is prerendered in the page. In the following example, replace the `{APP NAMESPACE}` placeholder with the project's namespace.

`Views/Home/Index.cshtml`:

CSHTML

```
@using {APP NAMESPACE}.Views.Shared
@{
    ViewData["Title"] = "Home Page";
}

<component type="typeof(Counter)" render-mode="ServerPrerendered" />
```

For more information, see the Render components from a page or view section.

Use routable components in a Razor Pages app

This section pertains to adding components that are directly routable from user requests.

To support routable Razor components in Razor Pages apps:

1. Follow the guidance in the Configuration section.
2. Add an `App` component to the project root with the following content.

   `App.razor`:

   ```razor
   @using Microsoft.AspNetCore.Components.Routing

   <Router AppAssembly="typeof(App).Assembly">
       <Found Context="routeData">
           <RouteView RouteData="routeData" />
       </Found>
       <NotFound>
           <PageTitle>Not found</PageTitle>
           <p role="alert">Sorry, there's nothing at this address.</p>
       </NotFound>
   </Router>
   ```

3. Add a `_Host` page to the project with the following content. Replace the `{APP NAMESPACE}` placeholder with the app's namespace.

   `Pages/_Host.cshtml`:

   ```CSHTML
   @page
   @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

   <component type="typeof(App)" render-mode="ServerPrerendered" />
   ```

   > **Note**
   >
   > The preceding example assumes that the HeadOutlet component and Blazor script (`_framework/blazor.server.js`) are rendered by the app's layout. For more information, see the Configuration section.

   RenderMode configures whether the `App` component:

   - Is prerendered into the page.

- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

For more information on the Component Tag Helper, including passing parameters and RenderMode configuration, see Component Tag Helper in ASP.NET Core.

4. In the `Program.cs` endpoints, add a low-priority route for the `_Host` page as the last endpoint:

```
C#

app.MapFallbackToPage("/_Host");
```

5. Add routable components to the project. The following example is a `RoutableCounter` component based on the `Counter` component in the Blazor project templates.

`Pages/RoutableCounter.razor`:

```
razor

@page "/routable-counter"

<PageTitle>Routable Counter</PageTitle>

<h1>Routable Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

6. Run the project and navigate to the routable `RoutableCounter` component at `/routable-counter`.

For more information on namespaces, see the Component namespaces section.

Use routable components in an MVC app

This section pertains to adding components that are directly routable from user requests.

To support routable Razor components in MVC apps:

1. Follow the guidance in the Configuration section.
2. Add an `App` component to the project root with the following content.

   `App.razor`:

   ```razor
   @using Microsoft.AspNetCore.Components.Routing

   <Router AppAssembly="typeof(App).Assembly">
       <Found Context="routeData">
           <RouteView RouteData="routeData" />
       </Found>
       <NotFound>
           <PageTitle>Not found</PageTitle>
           <p role="alert">Sorry, there's nothing at this address.</p>
       </NotFound>
   </Router>
   ```

3. Add a `_Host` view to the project with the following content. Replace the `{APP NAMESPACE}` placeholder with the app's namespace.

   `Views/Home/_Host.cshtml`:

   ```CSHTML
   @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

   <component type="typeof(App)" render-mode="ServerPrerendered" />
   ```

   > **Note**
   >
   > The preceding example assumes that the HeadOutlet component and Blazor script (`_framework/blazor.server.js`) are rendered by the app's layout. For more information, see the Configuration section.

   RenderMode configures whether the `App` component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

For more information on the Component Tag Helper, including passing parameters and RenderMode configuration, see Component Tag Helper in ASP.NET Core.

4. Add an action to the Home controller.

`Controllers/HomeController.cs`:

```
C#

public IActionResult Blazor()
{
    return View("_Host");
}
```

5. In the `Program.cs` endpoints, add a low-priority route for the controller action that returns the `_Host` view:

```
C#

app.MapFallbackToController("Blazor", "Home");
```

6. Create a `Pages` folder in the MVC app and add routable components. The following example is a `RoutableCounter` component based on the `Counter` component in the Blazor project templates.

`Pages/RoutableCounter.razor`:

```
razor

@page "/routable-counter"

<PageTitle>Routable Counter</PageTitle>

<h1>Routable Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>

@code {
    private int currentCount = 0;
```

```
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

7. Run the project and navigate to the routable `RoutableCounter` component at `/routable-counter`.

For more information on namespaces, see the Component namespaces section.

Render components from a page or view

This section pertains to adding components to pages or views, where the components aren't directly routable from user requests.

To render a component from a page or view, use the Component Tag Helper.

Render stateful interactive components

Stateful interactive components can be added to a Razor page or view.

When the page or view renders:

- The component is prerendered with the page or view.
- The initial component state used for prerendering is lost.
- New component state is created when the SignalR connection is established.

The following Razor page renders a `Counter` component:

CSHTML

```
<h1>Razor Page</h1>

<component type="typeof(Counter)" render-mode="ServerPrerendered"
    param-InitialValue="InitialValue" />

@functions {
    [BindProperty(SupportsGet=true)]
    public int InitialValue { get; set; }
}
```

For more information, see Component Tag Helper in ASP.NET Core.

Render noninteractive components

In the following Razor page, the `Counter` component is statically rendered with an initial value that's specified using a form. Since the component is statically rendered, the component isn't interactive:

CSHTML

```
<h1>Razor Page</h1>

<form>
    <input type="number" asp-for="InitialValue" />
    <button type="submit">Set initial value</button>
</form>

<component type="typeof(Counter)" render-mode="Static"
    param-InitialValue="InitialValue" />

@functions {
    [BindProperty(SupportsGet=true)]
    public int InitialValue { get; set; }
}
```

For more information, see Component Tag Helper in ASP.NET Core.

Component namespaces

When using a custom folder to hold the project's Razor components, add the namespace representing the folder to either the page/view or to the `_ViewImports.cshtml` file. In the following example:

- Components are stored in the `Components` folder of the project.
- The `{APP NAMESPACE}` placeholder is the project's namespace. `Components` represents the name of the folder.

CSHTML

```
@using {APP NAMESPACE}.Components
```

The `_ViewImports.cshtml` file is located in the `Pages` folder of a Razor Pages app or the `Views` folder of an MVC app.

For more information, see ASP.NET Core Razor components.

Persist prerendered state

Without persisting prerendered state, state used during prerendering is lost and must be recreated when the app is fully loaded. If any state is setup asynchronously, the UI may flicker as the prerendered UI is replaced with temporary placeholders and then fully rendered again.

To persist state for prerendered components, use the Persist Component State Tag Helper (reference source). Add the Tag Helper's tag, `<persist-component-state />`, inside the closing `</body>` tag of the `_Host` page in an app that prerenders components.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

In `Pages/_Host.cshtml` of Blazor apps that are `ServerPrerendered` in a Blazor Server app:

CSHTML

```
<body>
    ...

    <persist-component-state />
</body>
```

Decide what state to persist using the PersistentComponentState service. PersistentComponentState.RegisterOnPersisting registers a callback to persist the component state before the app is paused. The state is retrieved when the application resumes.

In the following example:

- The `{TYPE}` placeholder represents the type of data to persist (for example, `WeatherForecast[]`).
- The `{TOKEN}` placeholder is a state identifier string (for example, `fetchdata`).

razor

```
@implements IDisposable
@inject PersistentComponentState ApplicationState
```

```
...

@code {
    private {TYPE} data;
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistData);

        if (!ApplicationState.TryTakeFromJson<{TYPE}>(
            "{TOKEN}", out var restored))
        {
            data = await ...;
        }
        else
        {
            data = restored!;
        }
    }

    private Task PersistData()
    {
        ApplicationState.PersistAsJson("{TOKEN}", data);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

The following example is an updated version of the `FetchData` component based on the Blazor project template. The `WeatherForecastPreserveState` component persists weather forecast state during prerendering and then retrieves the state to initialize the component. The Persist Component State Tag Helper persists the component state after all component invocations.

`Pages/WeatherForecastPreserveState.razor`:

razor

```
@page "/weather-forecast-preserve-state"
@using BlazorSample.Shared
@implements IDisposable
@inject IWeatherForecastService WeatherForecastService
@inject PersistentComponentState ApplicationState
```

```
<PageTitle>Weather Forecast</PageTitle>

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from the server.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[] forecasts = Array.Empty<WeatherForecast>();
    private PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription =
            ApplicationState.RegisterOnPersisting(PersistForecasts);

        if (!ApplicationState.TryTakeFromJson<WeatherForecast[]>(
            "fetchdata", out var restored))
        {
            forecasts =
                await
WeatherForecastService.GetForecastAsync(DateOnly.FromDateTime(DateTime.Now))
;
        }
        else
        {
            forecasts = restored!;
```

```
        }
    }

    private Task PersistForecasts()
    {
        ApplicationState.PersistAsJson("fetchdata", forecasts);

        return Task.CompletedTask;
    }

    void IDisposable.Dispose()
    {
        persistingSubscription.Dispose();
    }
}
```

By initializing components with the same state used during prerendering, any expensive initialization steps are only executed once. The rendered UI also matches the prerendered UI, so no flicker occurs in the browser.

The persisted prerendered state is transferred to the client, where it's used to restore the component state. ASP.NET Core Data Protection ensures that the data is transferred securely in Blazor Server apps.

Prerendered state size and SignalR message size limit

A large prerendered state size may exceed Blazor's SignalR circuit message size limit, which results in the following:

- The SignalR circuit fails to initialize with an error on the client: Circuit host not initialized.
- The reconnection UI on the client appears when the circuit fails. Recovery isn't possible.

To resolve the problem, use ***either*** of the following approaches:

- Reduce the amount of data that you are putting into the prerendered state.
- Increase the SignalR message size limit. ***WARNING***: Increasing the limit may increase the risk of Denial of Service (DoS) attacks.

Additional Blazor Server resources

- State management: Handle prerendering

- Razor component lifecycle subjects that pertain to prerendering
  - Component initialization (OnInitialized{Async})
  - After component render (OnAfterRender{Async})
  - Stateful reconnection after prerendering
  - Prerendering with JavaScript interop
- Authentication and authorization: General aspects
- Handle Errors: Prerendering
- Host and deploy: Blazor Server
- Threat mitigation: Cross-site scripting (XSS)
- OnNavigateAsync is executed *twice* when prerendering: Handle asynchronous navigation events with OnNavigateAsync

Consume ASP.NET Core Razor components from a Razor class library (RCL)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Components can be shared in a Razor class library (RCL) across projects. Include components and static assets in an app from:

- Another project in the solution.
- A referenced .NET library.
- A NuGet package.

Just as components are regular .NET types, components provided by an RCL are normal .NET assemblies.

Create an RCL

Visual Studio

1. Create a new project.
2. In the **Create a new project** dialog, select **Razor Class Library** from the list of ASP.NET Core project templates. Select **Next**.
3. In the **Configure your new project** dialog, provide a project name in the **Project name** field. Examples in this topic use the project name `ComponentLibrary`. Select **Next**.
4. In the **Additional information** dialog, don't select **Support pages and views**. Select **Create**.
5. Add the RCL to a solution:
   a. Open the solution.
   b. Right-click the solution in **Solution Explorer**. Select **Add** > **Existing Project**.
   c. Navigate to the RCL's project file.
   d. Select the RCL's project file (`.csproj`).
6. Add a reference to the RCL from the app:

a. Right-click the app project. Select **Add** > **Project Reference**.
b. Select the RCL project. Select **OK**.

Consume a Razor component from an RCL

To consume components from an RCL in another project, use either of the following approaches:

- Use the full component type name, which includes the RCL's namespace.
- Individual components can be added by name without the RCL's namespace if Razor's @using directive declares the RCL's namespace. Use the following approaches:
  - Add the `@using` directive to individual components.
  - include the `@using` directive in the top-level `_Imports.razor` file to make the library's components available to an entire project. Add the directive to an `_Imports.razor` file at any level to apply the namespace to a single component or set of components within a folder. When an `_Imports.razor` file is used, individual components don't require an `@using` directive for the RCL's namespace.

In the following examples, `ComponentLibrary` is an RCL containing the `Component1` component. The `Component1` component is an example component automatically added to an RCL created from the RCL project template that isn't created to support pages and views.

`Component1.razor` in the `ComponentLibrary` RCL:

razor

```
<div class="my-component">
    This component is defined in the <strong>ComponentLibrary</strong>
package.
</div>
```

In the app that consumes the RCL, reference the `Component1` component using its namespace, as the following example shows.

`ConsumeComponent1.razor`:

razor

```
@page "/consume-component-1"

<h1>Consume component (full namespace example)</h1>

<ComponentLibrary.Component1 />
```

Alternatively, add a @using directive and use the component without its namespace. The following `@using` directive can also appear in any `_Imports.razor` file in or above the current folder.

`ConsumeComponent2.razor`:

razor

```
@page "/consume-component-2"
@using ComponentLibrary

<h1>Consume component (<code>@@using</code> example)</h1>

<Component1 />
```

For library components that use CSS isolation, the component styles are automatically made available to the consuming app. There's no need to manually link or import the library's individual component stylesheets or its bundled CSS file in the app that consumes the library. The app uses CSS imports to reference the RCL's bundled styles. The bundled styles aren't published as a static web asset of the app that consumes the library. For a class library named `ClassLib` and a Blazor app with a `BlazorSample.styles.css` stylesheet, the RCL's stylesheet is imported at the top of the app's stylesheet automatically at build time:

css

```
@import '_content/ClassLib/ClassLib.bundle.scp.css';
```

For the preceding examples, `Component1`'s stylesheet (`Component1.razor.css`) is bundled automatically.

`Component1.razor.css` in the `ComponentLibrary` RCL:

css

```
.my-component {
    border: 2px dashed red;
    padding: 1em;
    margin: 1em 0;
```

```
    background-image: url('background.png');
}
```

The background image is also included from the RCL project template and resides in the `wwwroot` folder of the RCL.

`wwwroot/background.png` in the `ComponentLibrary` RCL:

To provide additional library component styles from stylesheets in the library's `wwwroot` folder, add stylesheet `<link>` tags to the RCL's consumer, as the next example demonstrates.

> **ⓘ Important**
>
> Generally, library components use CSS isolation to bundle and provide component styles. Component styles that rely upon CSS isolation are automatically made available to the app that uses the RCL. There's no need to manually link or import the library's individual component stylesheets or its bundled CSS file in the app that consumes the library. The following example is for providing global stylesheets *outside of CSS isolation*, which usually isn't a requirement for typical apps that consume RCLs.

The following background image is used in the next example. If you implement the example shown in this section, right-click the image to save it locally.

`wwwroot/extra-background.png` in the `ComponentLibrary` RCL:

Add a new stylesheet to the RCL with an `extra-style` class.

`wwwroot/additionalStyles.css` in the `ComponentLibrary` RCL:

```
css

.extra-style {
    border: 2px dashed blue;
    padding: 1em;
    margin: 1em 0;
```

```
    background-image: url('extra-background.png');
}
```

Add a component to the RCL that uses the `extra-style` class.

`ExtraStyles.razor` in the `ComponentLibrary` RCL:

razor

```
<div class="extra-style">
    <p>
        This component is defined in the <strong>ComponentLibrary</strong>
package.
    </p>
</div>
```

Add a page to the app that uses the `ExtraStyles` component from the RCL.

`ConsumeComponent3.razor`:

razor

```
@page "/consume-component-3"
@using ComponentLibrary

<h1>Consume component (<code>additionalStyles.css</code> example)</h1>

<ExtraStyles />
```

Link to the library's stylesheet in the app's `<head>` markup (location of <head> content):

HTML

```
<link href="_content/ComponentLibrary/additionalStyles.css"
rel="stylesheet">
```

Make routable components available from the RCL

To make routable components in the RCL available for direct requests, the RCL's assembly must be disclosed to the app's router.

Open the app's `App` component (`App.razor`). Assign an Assembly collection to the AdditionalAssemblies parameter of the Router component to include the RCL's

assembly. In the following example, the `ComponentLibrary.Component1` component is used to discover the RCL's assembly.

```razor
AdditionalAssemblies="new[] { typeof(ComponentLibrary.Component1).Assembly
}"
```

For more information, see ASP.NET Core Blazor routing and navigation.

Create an RCL with static assets in the `wwwroot` folder

An RCL's static assets are available to any app that consumes the library.

Place static assets in the `wwwroot` folder of the RCL and reference the static assets with the following path in the app: `_content/{PACKAGE ID}/{PATH AND FILE NAME}`. The `{PACKAGE ID}` placeholder is the library's package ID. The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file. The `{PATH AND FILE NAME}` placeholder is path and file name under `wwwroot`. This path format is also used in the app for static assets supplied by NuGet packages added to the RCL.

The following example demonstrates the use of RCL static assets with an RCL named `ComponentLibrary` and a Blazor app that consumes the RCL. The app has a project reference for the `ComponentLibrary` RCL.

The following Jeep® image is used in this section's example. If you implement the example shown in this section, right-click the image to save it locally.

`wwwroot/jeep-yj.png` in the `ComponentLibrary` RCL:



Add the following `JeepYJ` component to the RCL.

`JeepYJ.razor` in the `ComponentLibrary` RCL:

razor

```
<h3>ComponentLibrary.JeepYJ</h3>

<p>
    <img alt="Jeep YJ&reg;" src="_content/ComponentLibrary/jeep-yj.png" />
</p>
```

Add the following `Jeep` component to the app that consumes the `ComponentLibrary` RCL. The `Jeep` component uses:

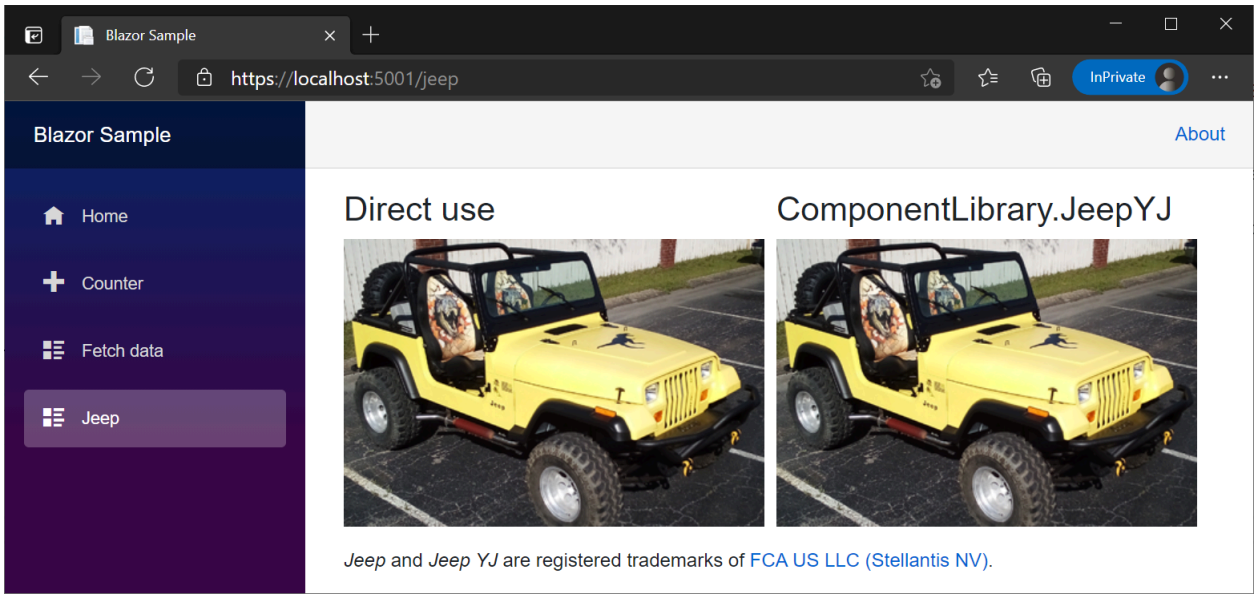
- The Jeep YJ® image from the `ComponentLibrary` RCL's `wwwroot` folder.
- The `JeepYJ` component from the RCL.

`Jeep.razor`:

razor

```
@page "/jeep"
@using ComponentLibrary

<div style="float:left;margin-right:10px">
    <h3>Direct use</h3>

    <p>
        <img alt="Jeep YJ&reg;" src="_content/ComponentLibrary/jeep-yj.png"
/>
    </p>
</div>

<JeepYJ />

<p>
    <em>Jeep</em> and <em>Jeep YJ</em> are registered trademarks of
    <a href="https://www.stellantis.com">FCA US LLC (Stellantis NV)</a>.
</p>
```

Rendered `Jeep` component:



For more information, see Reusable Razor UI in class libraries with ASP.NET Core.

Create an RCL with JavaScript files collocated with components

Collocation of JavaScript (JS) files for Razor components is a convenient way to organize scripts in an app.

Razor components of Blazor apps collocate JS files using the `.razor.js` extension and are publicly addressable using the path to the file in the project:

`{PATH}/{COMPONENT}.razor.js`

- The `{PATH}` placeholder is the path to the component.
- The `{COMPONENT}` placeholder is the component.

When the app is published, the framework automatically moves the script to the web root. Scripts are moved to `bin/Release/{TARGET FRAMEWORK MONIKER}/publish/wwwroot/{PATH}/{COMPONENT}.razor.js`, where the placeholders are:

- `{TARGET FRAMEWORK MONIKER}` is the Target Framework Moniker (TFM).
- `{PATH}` is the path to the component.
- `{COMPONENT}` is the component name.

No change is required to the script's relative URL, as Blazor takes care of placing the JS file in published static assets for you.

This section and the following examples are primarily focused on explaining JS file collocation. The first example demonstrates a collocated JS file with an ordinary JS

function. The second example demonstrates the use of a module to load a function, which is the recommended approach for most production apps. Calling JS from .NET is fully covered in Call JavaScript functions from .NET methods in ASP.NET Core Blazor, where there are further explanations of the Blazor JS API with additional examples. Component disposal, which is present in the second example, is covered in ASP.NET Core Razor component lifecycle.

The following `JsCollocation1` component loads a script via a HeadContent component and calls a JS function with IJSRuntime.InvokeAsync. The `{PATH}` placeholder is the path to the component.

> **Important**
>
> If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component (example: `Components/Pages` in .NET 8 or later or `Pages` in .NET 7 or earlier). In a Blazor Web App (.NET 8 or later), the component requires an interactive render mode applied either globally to the app or to the component definition.

Add the following script after the Blazor script (location of the Blazor start script):

HTML

```
<script src="{PATH}/JsCollocation1.razor.js"></script>
```

`JsCollocation1` component (`{PATH}/JsCollocation1.razor`):

razor

```
@page "/js-collocation-1"
@inject IJSRuntime JS

<PageTitle>JS Collocation 1</PageTitle>

<h1>JS Collocation Example 1</h1>

<button @onclick="ShowPrompt">Call showPrompt1</button>

@if (!string.IsNullOrEmpty(result))
{
    <p>
        Hello @result!
    </p>
}
```

```
@code {
    private string? result;

    public async void ShowPrompt()
    {
        result = await JS.InvokeAsync<string>(
            "showPrompt1", "What's your name?");
        StateHasChanged();
    }
}
```

The collocated JS file is placed next to the `JsCollocation1` component file with the file name `JsCollocation1.razor.js`. In the `JsCollocation1` component, the script is referenced at the path of the collocated file. In the following example, the `showPrompt1` function accepts the user's name from a Window prompt() and returns it to the `JsCollocation1` component for display.

`{PATH}/JsCollocation1.razor.js`:

JavaScript

```
function showPrompt1(message) {
  return prompt(message, 'Type your name here');
}
```

The preceding approach isn't recommended for general use in production apps because the approach pollutes the client with global functions. A better approach for production apps is to use JS modules. The same general principles apply to loading a JS module from a collocated JS file, as the next example demonstrates.

The following `JsCollocation2` component's `OnAfterRenderAsync` method loads a JS module into `module`, which is an IJSObjectReference of the component class. `module` is used to call the `showPrompt2` function. The `{PATH}` placeholder is the path to the component.

> **Important**
>
> If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component. In a Blazor Web App (.NET 8 or later), the component requires an interactive render mode applied either globally to the app or to the component definition.

`JsCollocation2` component (`{PATH}/JsCollocation2.razor`):

razor

```razor
@page "/js-collocation-2"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>JS Collocation 2</PageTitle>

<h1>JS Collocation Example 2</h1>

<button @onclick="ShowPrompt">Call showPrompt2</button>

@if (!string.IsNullOrEmpty(result))
{
    <p>
        Hello @result!
    </p>
}

@code {
    private IJSObjectReference? module;
    private string? result;

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            /*
                Change the {PATH} placeholder in the next line to the path
of
                the collocated JS file in the app. Examples:

                ./Components/Pages/JsCollocation2.razor.js (.NET 8 or later)
                ./Pages/JsCollocation2.razor.js (.NET 7 or earlier)
            */
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./{PATH}/JsCollocation2.razor.js");
        }
    }

    public async void ShowPrompt()
    {
        if (module is not null)
        {
            result = await module.InvokeAsync<string>(
                "showPrompt2", "What's your name?");
            StateHasChanged();
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
```

```
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

In the preceding example, JSDisconnectedException is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try`-`catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

`{PATH}/JsCollocation2.razor.js`:

JavaScript

```
export function showPrompt2(message) {
  return prompt(message, 'Type your name here');
}
```

Use of scripts and modules for collocated JS in a Razor class library (RCL) is only supported for Blazor's JS interop mechanism based on the IJSRuntime interface. If you're implementing JavaScript [JSImport]/[JSExport] interop, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

For scripts or modules provided by a Razor class library (RCL) using IJSRuntime-based JS interop, the following path is used:

`./_content/{PACKAGE ID}/{PATH}/{COMPONENT}.{EXTENSION}.js`

- The path segment for the current directory (`./`) is required in order to create the correct static asset path to the JS file.
- The `{PACKAGE ID}` placeholder is the RCL's package identifier (or library name for a class library referenced by the app).
- The `{PATH}` placeholder is the path to the component. If a Razor component is located at the root of the RCL, the path segment isn't included.
- The `{COMPONENT}` placeholder is the component name.

- The `{EXTENSION}` placeholder matches the extension of component, either `razor` or `cshtml`.

In the following Blazor app example:

- The RCL's package identifier is `AppJS`.
- A module's scripts are loaded for the `JsCollocation3` component (`JsCollocation3.razor`).
- The `JsCollocation3` component is in the `Components/Pages` folder of the RCL.

C#

```
module = await JS.InvokeAsync<IJSObjectReference>("import",
    "./_content/AppJS/Components/Pages/JsCollocation3.razor.js");
```

Client-side browser compatibility analyzer

Client-side apps target the full .NET API surface area, but not all .NET APIs are supported on WebAssembly due to browser sandbox constraints. Unsupported APIs throw PlatformNotSupportedException when running on WebAssembly. A platform compatibility analyzer warns the developer when the app uses APIs that aren't supported by the app's target platforms. For client-side apps, this means checking that APIs are supported in browsers. Annotating .NET framework APIs for the compatibility analyzer is an on-going process, so not all .NET framework API is currently annotated.

Blazor Web Apps that enable Interactive WebAssembly components, Blazor WebAssembly apps, and RCL projects *automatically* enable browser compatibility checks by adding `browser` as a supported platform with the `SupportedPlatform` MSBuild item. Library developers can manually add the `SupportedPlatform` item to a library's project file to enable the feature:

XML

```
<ItemGroup>
  <SupportedPlatform Include="browser" />
</ItemGroup>
```

When authoring a library, indicate that a particular API isn't supported in browsers by specifying `browser` to UnsupportedOSPlatformAttribute:

C#

```
using System.Runtime.Versioning;

...

[UnsupportedOSPlatform("browser")]
private static string GetLoggingDirectory()
{
    ...
}
```

For more information, see Annotating APIs as unsupported on specific platforms (dotnet/designs GitHub repository).

JavaScript isolation in JavaScript modules

Blazor enables JavaScript isolation in standard JavaScript modules. JavaScript isolation provides the following benefits:

- Imported JavaScript no longer pollutes the global namespace.
- Consumers of the library and components aren't required to manually import the related JavaScript.

For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Avoid trimming JavaScript-invokable .NET methods

Runtime relinking trims class instance JavaScript-invokable .NET methods unless they're explicitly preserved. For more information, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor.

Build, pack, and ship to NuGet

Because Razor class libraries that contain Razor components are standard .NET libraries, packing and shipping them to NuGet is no different from packing and shipping any library to NuGet. Packing is performed using the dotnet pack command in a command shell:

.NET CLI

```
dotnet pack
```

Upload the package to NuGet using the dotnet nuget push command in a command shell.

Trademarks

Jeep and *Jeep YJ* are registered trademarks of FCA US LLC (Stellantis NV).

Additional resources

- Reusable Razor UI in class libraries with ASP.NET Core
- Use ASP.NET Core APIs in a class library
- Add an XML Intermediate Language (IL) Trimmer configuration file to a library
- CSS isolation support with Razor class libraries

ASP.NET Core Razor class libraries (RCLs) with static server-side rendering (static SSR)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article provides guidance for component library authors considering support for static server-side rendering (static SSR).

Blazor encourages the development of an ecosystem of open-source and commercial component libraries, formally called *Razor class libraries (RCLs)*. Developers might also create reusable components for sharing components privately across apps within their own companies. Ideally, components are developed for compatibility with as many hosting models and rendering modes as possible. Static SSR introduces additional restrictions that can be more challenging to support than interactive rendering modes.

Understand the capabilities and restrictions of static SSR

Static SSR is a mode in which components run when the server handles an incoming HTTP request. Blazor renders the component as HTML, which is included in the response. Once the response is sent, the server-side component and renderer state is discarded, so all that remains is HTML in the browser.

The benefit of this mode is cheaper, more scalable hosting, because no ongoing server resources are required to hold component state, no ongoing connection must be maintained between browser and server, and no WebAssembly payload is required in the browser.

All existing components can still be used with static SSR. However, the cost of this mode is that event handlers, such as `@onclick`†, can't be run for the following reasons:

- There's no .NET code in the browser to run them.

- The server has immediately discarded any component and renderer state that would be needed to execute event handlers or to rerender the same component instances.

†There's a special exception for the `@onsubmit` event handler for forms, which is always functional, regardless of render mode.

This is equivalent to how components behave during prerendering, before Blazor's SignalR circuit or the .NET WebAssembly runtime is started.

For components whose only role is to produce read-only DOM content, these behaviors for static SSR are completely sufficient. However, library authors must consider what approach to take when including interactive components in their libraries.

Options for component authors

There are three main approaches:

- **Don't use interactive behaviors** (Basic)

  For components whose only role is to produce read-only DOM content, the developer isn't required to take any special action. These components naturally work with any render mode.

  Examples:
  - A "user card" component that loads data corresponding to a person and renders it in a stylized UI with a photo, job title, and other details.
  - A "video" component that acts as a wrapper around the HTML `<video>` element, making it more convenient to use in a Razor component.

- **Require interactive rendering** (Basic)

  You can choose to require that your component is only used with interactive rendering. This limits the applicability of your component, but means that you may freely rely on arbitrary event handlers. Even then, you should still avoid declaring a specific `@rendermode` and permit the app author who consumes your library to select one.

  Examples:
  - A video editing component in which users may splice and re-order segments of video. Even if there was a way to represent these edit operations with plain HTML buttons and form posts, the user experience would be unacceptable without true interactivity.

    - A collaborative document editor that must show the activities of other users in real time.

- **Use interactive behaviors, but design for static SSR and progressive enhancement** (Advanced)

  Many interactive behaviors can be implemented using only HTML capabilities. With a good understanding of HTML and CSS, you can often produce a useful baseline of functionality that works with static SSR. You can still declare event handlers that implement more advanced, optional behaviors, which only work in interactive render modes.

  Examples:
    - A grid component. Under static SSR, the component may only support displaying data and navigating between pages (implemented with `<a>` links). When used with interactive rendering, the component may add live sorting and filtering.
    - A tabset component. As long as navigation between tabs is achieved using `<a>` links and state is held only in URL query parameters, the component can work without `@onclick`.
    - An advanced file upload component. Under static SSR, the component may behave as a native `<input type=file>`. When used with interactive rendering, the component could also display upload progress.
    - A stock ticker. Under static SSR, the component may display the stock quote at the time the HTML was rendered. When used with interactive rendering, the component may then update the stock price in real time.

For any of these strategies, there's also the option of implementing interactive features with JavaScript.

To choose among these approaches, reusable Razor component authors must make a cost/benefit tradeoff. Your component is more useful and has a broader potential user base if it supports all render modes, including static SSR. However, it takes more work to design and implement a component that supports and takes best advantage of each render mode.

When to use the `@rendermode` directive

In most cases, reusable component authors should **not** specify a render mode, even when interactivity is required. This is because the component author doesn't know whether the app enables support for InteractiveServer, InteractiveWebAssembly, or both

with InteractiveAuto. By not specifying a `@rendermode`, the component author leaves the choice to the app developer.

Even if the component author thinks that interactivity is required, there may still be cases where an app author considers it sufficient to use static SSR alone. For example, a map component with drag and zoom interactivity may seem to require interactivity. However, some scenarios may only call for rendering a static map image and avoiding drag/zoom features.

The only reason why a reusable component author should use the `@rendermode` directive on their component is if the implementation is fundamentally coupled to one specific render mode and would certainly cause an error if used in a different mode. Consider a component with a core purpose of interacting directly with the host OS using Windows or Linux-specific APIs. It might be impossible to use such a component on WebAssembly. If so, it's reasonable to declare `@rendermode InteractiveServer` for the component.

Streaming rendering

Reusable Razor components are free to declare `@attribute [StreamRendering]` for streaming rendering ([StreamRendering] attribute API). This results in incremental UI updates during static SSR. Since the same data-loading patterns produce incremental UI updates during interactive rendering, regardless of the presence of the `[StreamRendering]` attribute, the component can behave correctly in all cases. Even in cases where streaming static SSR is suppressed on the server, the component still renders its correct final state.

Using links across render modes

Reusable Razor components may use links and enhanced navigation. HTML `<a>` tags should produce equivalent behaviors with or without an interactive Router component and whether or not enhanced navigation is enabled/disabled at an ancestor level in the DOM.

Using forms across render modes

Reusable Razor components may include forms (either `<form>` or `<EditForm>`), as these can be implemented to work equivalently across both static and interactive render modes.

Consider the following example:

razor

```
<EditForm Enhance FormName="NewProduct" Model="Model"
OnValidSubmit="SaveProduct">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <p><label>Name: <InputText @bind-Value="Item.Name" /></label></p>

    <button type="submit">Submit</button>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Product? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private async Task Save()
    {
        ...
    }
}
```

The Enhance, FormName, and SupplyParameterFromFormAttribute APIs are only used during static SSR and are ignored during interactive rendering. The form works correctly during both interactive and static SSR.

Avoid APIs that are specific to static SSR

HttpContext is only available during static SSR, so don't rely on HttpContext when creating components that work across render modes. The HttpContext API doesn't make sense to use during interactive rendering either because there's no active HTTP request in flight at those times. It's meaningless to think about setting a HTTP status code or writing to the HTTP response.

Reusable components are free to receive an HttpContext when available, as follows:

C#

```
[CascadingParameter]
public HttpContext? Context { get; set; }
```

The value is `null` during interactive rendering and is only set during static SSR.

In many cases, there are better alternatives than using HttpContext. If you need to know the current URL or to perform a redirection, the APIs on NavigationManager work with all render modes. If you need to know the user's authentication state, use Blazor's AuthenticationStateProvider service (AuthenticationStateProvider documentation) over using HttpContext.User.

Use Razor components in JavaScript apps and SPA frameworks

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article covers how to render Razor components from JavaScript, use Blazor custom elements, and generate Angular and React components.

> **Note**
>
> We recommend using the `blazor.server.js` (Blazor Server) and `blazor.webassembly.js` (Blazor WebAssembly) scripts when integrating Razor components into an existing JavaScript app until better support for the `blazor.web.js` (Blazor Web App) script is added in the future. For more information, see RegisterCustomElement stopped working in Blazor 8 (dotnet/aspnetcore #53920).

Angular sample apps

- CustomElementsBlazorSample (Blazor Server) (javiercn/CustomElementsBlazorSample, branch: blazor-server): Blazor Server is supported in .NET 8/9. To migrate this .NET 7 sample, see Migrate from ASP.NET Core 7.0 to 8.0 and Migrate from ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9.
- CustomElementsBlazorSample (Blazor WebAssembly) (javiercn/CustomElementsBlazorSample, branch: blazor-wasm): To migrate this .NET 7 sample, see Migrate from ASP.NET Core 7.0 to 8.0 and Migrate from ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9.

Render Razor components from JavaScript

Razor components can be dynamically-rendered from JavaScript (JS) for existing JS apps.

The example in this section renders the following Razor component into a page via JS.

`Quote.razor`:

razor

```razor
<div class="m-5 p-5">
    <h2>Quote</h2>
    <p>@Text</p>
</div>

@code {
    [Parameter]
    public string? Text { get; set; }
}
```

In the `Program` file, add the namespace for the location of the component.

Call RegisterForJavaScript on the app's root component collection to register a Razor component as a root component for JS rendering.

RegisterForJavaScript includes an overload that accepts the name of a JS function that executes initialization logic (`javaScriptInitializer`). The JS function is called once per component registration immediately after the Blazor app starts and before any components are rendered. This function can be used for integration with JS technologies, such as HTML custom elements or a JS-based SPA framework.

One or more initializer functions can be created and called by different component registrations. The typical use case is to reuse the same initializer function for multiple components, which is expected if the initializer function is configuring integration with custom elements or another JS-based SPA framework.

> **ⓘ Important**
>
> Don't confuse the `javaScriptInitializer` parameter of RegisterForJavaScript with JavaScript initializers. The name of the parameter and the JS initializers feature is coincidental.

The following example demonstrates the dynamic registration of the preceding `Quote` component with "`quote`" as the identifier.

- In a Blazor Server app, modify the call to AddServerSideBlazor in the `Program` file:

  C#

```
builder.Services.AddServerSideBlazor(options =>
{
    options.RootComponents.RegisterForJavaScript<Quote>(identifier:
"quote",
        javaScriptInitializer: "initializeComponent");
});
```

- In a Blazor WebAssembly app, call RegisterForJavaScript on RootComponents in the client-side `Program` file:

C#

```
builder.RootComponents.RegisterForJavaScript<Quote>(identifier:
"quote",
    javaScriptInitializer: "initializeComponent");
```

Attach the initializer function with `name` and `parameters` function parameters to the `window` object. For demonstration purposes, the following `initializeComponent` function logs the name and parameters of the registered component.

`wwwroot/jsComponentInitializers.js`:

JavaScript

```
window.initializeComponent = (name, parameters) => {
  console.log({ name: name, parameters: parameters });
}
```

Render the component from JS into a container element using the registered identifier, passing component parameters as needed.

In the following example:

- The `Quote` component (`quote` identifier) is rendered into the `quoteContainer` element when the `showQuote` function is called.
- A quote string is passed to the component's `Text` parameter.

`wwwroot/scripts.js`:

JavaScript

```
window.showQuote = async () => {
  let targetElement = document.getElementById('quoteContainer');
  await Blazor.rootComponents.add(targetElement, 'quote',
  {
```

```
    text: "Crow: I have my doubts that this movie is actually 'starring' " +
      "anybody. More like, 'camera is generally pointed at.'"
  });
}

const btn = document.querySelector("#showQuoteBtn");
btn.addEventListener("click", showQuote);
```

After the Blazor script is loaded, load the preceding scripts into the JS app:

HTML

```
<script src="_framework/{BLAZOR SCRIPT}"></script>
<script src="jsComponentInitializers.js"></script>
<script src="scripts.js"></script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script.

In HTML, place the target container element (`quoteContainer`). For the demonstration in this section, a button triggers rendering the `Quote` component by calling the `showQuote` JS function:

HTML

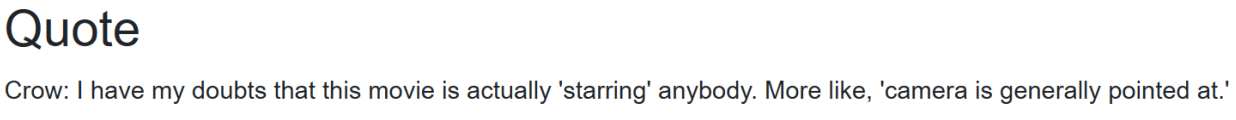
```
<button id="showQuoteBtn">Show Quote</button>

<div id="quoteContainer"></div>
```

On initialization before any components are rendered, the browser's developer tools console logs the `Quote` component's identifier (`name`) and parameters (`parameters`) when `initializeComponent` is called:

Console

```
Object { name: "quote", parameters: (1) […] }
  name: "quote"
  parameters: Array [ {…} ]
    0: Object { name: "Text", type: "string" }
    length: 1
```

When the **Show Quote** button is selected, the `Quote` component is rendered with the quote stored in `Text` displayed:



Quote ©1988-1999 Satellite of Love LLC: *Mystery Science Theater 3000* (Trace Beaulieu (Crow))

> **Note**
>
> `rootComponents.add` returns an instance of the component. Call `dispose` on the instance to release it:
>
> JavaScript
>
> ```
> const rootComponent = await window.Blazor.rootComponents.add(...);
>
> ...
>
> rootComponent.dispose();
> ```

The preceding example dynamically renders the root component when the `showQuote()` JS function is called. To render a root component into a container element when Blazor starts, use a JavaScript initializer to render the component, as the following example demonstrates.

The following example builds on the preceding example, using the `Quote` component, the root component registration in the `Program` file, and the initialization of `jsComponentInitializers.js`. The `showQuote()` function (and the `script.js` file) aren't used.

In HTML, place the target container element, `quoteContainer2` for this example:

HTML

```
<div id="quoteContainer2"></div>
```

Using a JavaScript initializer, add the root component to the target container element.

`wwwroot/{PACKAGE ID/ASSEMBLY NAME}.lib.module.js`:

JavaScript

```
export function afterStarted(blazor) {
  let targetElement = document.getElementById('quoteContainer2');
  blazor.rootComponents.add(targetElement, 'quote',
    {
      text: "Crow: I have my doubts that this movie is actually 'starring' "
+
          "anybody. More like, 'camera is generally pointed at.'"
```

```
    });
}
```

> **Note**
>
> For the call to `rootComponents.add`, use the `blazor` parameter (lowercase `b`) provided by the Blazor start event. Although the registration is valid when using the `Blazor` object (uppercase `B`), the preferred approach is to use the parameter.

For an advanced example with additional features, see the example in the `BasicTestApp` of the ASP.NET Core reference source (`dotnet/aspnetcore` GitHub repository):

- JavaScriptRootComponents.razor
- wwwroot/js/jsRootComponentInitializers.js
- wwwroot/index.html

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Blazor custom elements

Use Blazor custom elements to dynamically render Razor components from other SPA frameworks, such as Angular or React.

Blazor custom elements:

- Use standard HTML interfaces to implement custom HTML elements.
- Eliminate the need to manually manage the state and lifecycle of root Razor components using JavaScript APIs.
- Are useful for gradually introducing Razor components into existing projects written in other SPA frameworks.

Custom elements don't support child content or templated components.

Element name

Per the HTML specification, custom element tag names must adopt kebab case:

- ✗ `mycounter`
- ✗ `MY-COUNTER`
- ✗ `MyCounter`
- ✓ `my-counter`
- ✓ `my-cool-counter`

Package

Add a package reference for Microsoft.AspNetCore.Components.CustomElements to the app's project file.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Example component

The following examples are based on the `Counter` component from the Blazor project template.

`Counter.razor`:

razor

```razor
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount() => currentCount++;
}
```

Blazor Server registration

Take the following steps to register a root component as a custom element in a Blazor Server app.

Add the Microsoft.AspNetCore.Components.Web namespace to the top of the `Program` file:

```
C#

using Microsoft.AspNetCore.Components.Web;
```

Add a namespace for the app's components. In the following example, the app's namespace is `BlazorSample` and the components are located in the `Pages` folder:

```
C#

using BlazorSample.Pages;
```

Modify the call to AddServerSideBlazor. Specify the custom element with RegisterCustomElement on the RootComponents circuit option. The following example registers the `Counter` component with the custom HTML element `my-counter`:

```
C#

builder.Services.AddServerSideBlazor(options =>
{
    options.RootComponents.RegisterCustomElement<Counter>("my-counter");
});
```

Blazor WebAssembly registration

Take the following steps to register a root component as a custom element in a Blazor WebAssembly app.

Add the Microsoft.AspNetCore.Components.Web namespace to the top of the `Program` file:

```
C#

using Microsoft.AspNetCore.Components.Web;
```

Add a namespace for the app's components. In the following example, the app's namespace is `BlazorSample` and the components are located in the `Pages` folder:

```
C#

using BlazorSample.Pages;
```

Call RegisterCustomElement on RootComponents. The following example registers the `Counter` component with the custom HTML element `my-counter`:

```
C#

builder.RootComponents.RegisterCustomElement<Counter>("my-counter");
```

Use the registered custom element

Use the custom element with any web framework. For example, the preceding `my-counter` custom HTML element that renders the app's `Counter` component is used in a React app with the following markup:

```
HTML

<my-counter></my-counter>
```

For a complete example of how to create custom elements with Blazor, see the CustomElementsComponent component in the reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Pass parameters

Pass parameters to your Razor component either as HTML attributes or as JavaScript properties on the DOM element.

The following `Counter` component uses an `IncrementAmount` parameter to set the increment amount of the **Click me** button.

`Counter.razor`:

```razor
@page "/counter"

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    [Parameter]
    public int IncrementAmount { get; set; } = 1;

    private void IncrementCount()
    {
        currentCount += IncrementAmount;
    }
}
```

Render the `Counter` component with the custom element and pass a value to the `IncrementAmount` parameter as an HTML attribute. The attribute name adopts kebab-case syntax (`increment-amount`, not `IncrementAmount`):

```HTML
<my-counter increment-amount="10"></my-counter>
```

Alternatively, you can set the parameter's value as a JavaScript property on the element object. The property name adopts camel case syntax (`incrementAmount`, not `IncrementAmount`):

```JavaScript
const elem = document.querySelector("my-counter");
elem.incrementAmount = 10;
```

You can update parameter values at any time using either attribute or property syntax.

Supported parameter types:

- Using JavaScript property syntax, you can pass objects of any JSON-serializable type.
- Using HTML attributes, you are limited to passing objects of string, boolean, or numerical types.

Generate Angular and React components

Generate framework-specific JavaScript (JS) components from Razor components for web frameworks, such as Angular or React. This capability isn't included with .NET, but is enabled by the support for rendering Razor components from JS. The JS component generation sample on GitHub demonstrates how to generate Angular and React components from Razor components. See the GitHub sample app's `README.md` file for additional information.

> ⚠ **Warning**
>
> The Angular and React component features are currently **experimental, unsupported, and subject to change or be removed at any time**. We welcome your feedback on how well this particular approach meets your requirements.

Render Razor components outside of ASP.NET Core

Article • 11/14/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Razor components can be rendered outside of the context of an HTTP request. You can render Razor components as HTML directly to a string or stream independently of the ASP.NET Core hosting environment. This is convenient for scenarios where you want to generate HTML fragments, such as for generating email content, generating static site content, or for building a content templating engine.

In the following example, a Razor component is rendered to an HTML string from a console app:

In a command shell, create a new console app project:

.NET CLI

```
dotnet new console -o ConsoleApp1
cd ConsoleApp1
```

In a command shell in the `ConsoleApp1` folder, add package references for Microsoft.AspNetCore.Components.Web and Microsoft.Extensions.Logging to the console app:

.NET CLI

```
dotnet add package Microsoft.AspNetCore.Components.Web
dotnet add package Microsoft.Extensions.Logging
```

In the console app's project file (`ConsoleApp1.csproj`), update the console app project to use the Razor SDK:

diff

```
- <Project Sdk="Microsoft.NET.Sdk">
```

```
+ <Project Sdk="Microsoft.NET.Sdk.Razor">
```

Add the following `RenderMessage` component to the project.

`RenderMessage.razor`:

razor

```razor
<h1>Render Message</h1>

<p>@Message</p>

@code {
    [Parameter]
    public string Message { get; set; }
}
```

Update the `Program` file:

- Set up dependency injection (IServiceCollection/BuildServiceProvider) and logging (AddLogging/ILoggerFactory).
- Create an HtmlRenderer and render the `RenderMessage` component by calling RenderComponentAsync.

Any calls to RenderComponentAsync must be made in the context of calling `InvokeAsync` on a component dispatcher. A component dispatcher is available from the HtmlRenderer.Dispatcher property.

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Web;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using ConsoleApp1;

IServiceCollection services = new ServiceCollection();
services.AddLogging();

IServiceProvider serviceProvider = services.BuildServiceProvider();
ILoggerFactory loggerFactory =
serviceProvider.GetRequiredService<ILoggerFactory>();

await using var htmlRenderer = new HtmlRenderer(serviceProvider,
loggerFactory);

var html = await htmlRenderer.Dispatcher.InvokeAsync(async () =>
{
    var dictionary = new Dictionary<string, object?>
```

```
    {
        { "Message", "Hello from the Render Message component!" }
    };

    var parameters = ParameterView.FromDictionary(dictionary);
    var output = await htmlRenderer.RenderComponentAsync<RenderMessage>
(parameters);

    return output.ToHtmlString();
});

Console.WriteLine(html);
```

> **Note**
>
> Pass ParameterView.Empty to RenderComponentAsync when rendering the component without passing parameters.

Alternatively, you can write the HTML to a TextWriter by calling `output.WriteHtmlTo(textWriter)`.

The task returned by RenderComponentAsync completes when the component is fully rendered, including completing any asynchronous lifecycle methods. If you want to observe the rendered HTML earlier, call BeginRenderingComponent instead. Then, wait for the component rendering to complete by awaiting HtmlRootComponent.QuiescenceTask on the returned HtmlRootComponent instance.

ASP.NET Core built-in Razor components

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

The following built-in Razor components are provided by the Blazor framework. For information on non-security-related project template components, see ASP.NET Core Blazor project structure. For information on security-related project template components, see the Security node articles.

- AntiforgeryToken
- AuthorizeView
- CascadingValue
- DataAnnotationsValidator
- DynamicComponent
- Editor<T>
- EditForm
- ErrorBoundary
- FocusOnNavigate
- HeadContent
- HeadOutlet
- InputCheckbox
- InputDate
- InputFile
- InputNumber
- InputRadio
- InputRadioGroup
- InputSelect
- InputText
- InputTextArea
- LayoutComponentBase
- LayoutView
- NavigationLock
- NavLink
- PageTitle

- OwningComponentBase
- Paginator
- QuickGrid
- Router
- RouteView
- SectionContent
- SectionOutlet
- ValidationMessage
- ValidationSummary
- Virtualize

ASP.NET Core Blazor globalization and localization

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to render globalized and localized content to users in different cultures and languages.

Globalization and localization

For globalization, Blazor provides number and date formatting. For localization, Blazor renders content using the .NET Resources system.

A limited set of ASP.NET Core's localization features are supported:

✔ IStringLocalizer and IStringLocalizer<T> are supported in Blazor apps.

✘ IHtmlLocalizer, IViewLocalizer, and Data Annotations localization are ASP.NET Core MVC features and *not supported* in Blazor apps.

This article describes how to use Blazor's globalization and localization features based on:

- The Accept-Language header, which is set by the browser based on a user's language preferences in browser settings.
- A culture set by the app not based on the value of the Accept-Language header. The setting can be static for all users or dynamic based on app logic. When the setting is based on the user's preference, the setting is usually saved for reload on future visits.

For additional general information, see the following resources:

- Globalization and localization in ASP.NET Core
- .NET Fundamentals: Globalization
- .NET Fundamentals: Localization

Often, the terms *language* and *culture* are used interchangeably when dealing with globalization and localization concepts.

In this article, *language* refers to selections made by a user in their browser's settings. The user's language selections are submitted in browser requests in the Accept-Language header. Browser settings usually use the word "language" in the UI.

Culture pertains to members of .NET and Blazor API. For example, a user's request can include the Accept-Language header specifying a *language* from the user's perspective, but the app ultimately sets the CurrentCulture ("culture") property from the language that the user requested. API usually uses the word "culture" in its member names.

The guidance in this article doesn't cover setting the page's HTML language attribute (<html lang="...">), which accessiblity tools use. You can set the value statically by assigning a language to the `lang` attribute of the `<html>` tag or to `document.documentElement.lang` in JavaScript. You can dynamically set the value of `document.documentElement.lang` with JS interop.

> **Note**
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from the article's examples.

Globalization

The @bind attribute directive applies formats and parses values for display based on the user's first preferred language that the app supports. @bind supports the @bind:culture parameter to provide a System.Globalization.CultureInfo for parsing and formatting a value.

The current culture can be accessed from the System.Globalization.CultureInfo.CurrentCulture property.

CultureInfo.InvariantCulture is used for the following field types (`<input type="{TYPE}" />`, where the `{TYPE}` placeholder is the type):

- `date`
- `number`

The preceding field types:

- Are displayed using their appropriate browser-based formatting rules.
- Can't contain free-form text.
- Provide user interaction characteristics based on the browser's implementation.

Blazor provides built-in support to render values in the current culture. Therefore, specifying a culture with @bind:culture isn't recommended when using the `date` and `number` field types.

The following field types have specific formatting requirements and aren't supported by all of the major browsers, so they aren't supported by Blazor:

- `datetime-local`
- `month`
- `week`

For current browser support of the preceding types, see Can I use.

.NET globalization and International Components for Unicode (ICU) support (Blazor WebAssembly)

Blazor WebAssembly uses a reduced globalization API and set of built-in International Components for Unicode (ICU) locales. For more information, see .NET globalization and ICU: ICU on WebAssembly.

To load a custom ICU data file to control the app's locales, see WASM Globalization Icu. Currently, manually building the custom ICU data file is required. .NET tooling to ease the process of creating the file is planned for .NET 10 in November, 2025.

Invariant globalization

This section only applies to client-side Blazor scenarios.

If the app doesn't require localization, configure the app to support the invariant culture, which is generally based on United States English (`en-US`). Using invariant globalization reduces the app's download size and results in faster app startup. Set the `InvariantGlobalization` property to `true` in the app's project file (`.csproj`):

```
XML
```

```
<PropertyGroup>
  <InvariantGlobalization>true</InvariantGlobalization>
</PropertyGroup>
```

Alternatively, configure invariant globalization with the following approaches:

- In `runtimeconfig.json`:

  JSON

  ```
  {
    "runtimeOptions": {
      "configProperties": {
        "System.Globalization.Invariant": true
      }
    }
  }
  ```

- With an environment variable:
  - Key: `DOTNET_SYSTEM_GLOBALIZATION_INVARIANT`
  - Value: `true` or `1`

For more information, see Runtime configuration options for globalization (.NET documentation).

Timezone information

This section only applies to client-side Blazor scenarios.

Adopting invariant globalization only results in using non-localized timezone names. To trim timezone code and data, which reduces the app's download size and results in faster app startup, apply the `<InvariantTimezone>` MSBuild property with a value of `true` in the app's project file:

XML

```
<PropertyGroup>
  <InvariantTimezone>true</InvariantTimezone>
</PropertyGroup>
```

> ⓘ Note

<BlazorEnableTimeZoneSupport> overrides an earlier `<InvariantTimezone>` setting. We recommend removing the `<BlazorEnableTimeZoneSupport>` setting.

Demonstration component

The following `CultureExample1` component can be used to demonstrate Blazor globalization and localization concepts covered by this article.

`CultureExample1.razor`:

razor

```
@page "/culture-example-1"
@using System.Globalization

<h1>Culture Example 1</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Rendered values</h2>

<ul>
    <li><b>Date</b>: @dt</li>
    <li><b>Number</b>: @number.ToString("N2")</li>
</ul>

<h2><code>&lt;input&gt;</code> elements that don't set a <code>type</code>
</h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.CurrentCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input @bind="number" /></label></li>
</ul>

<h2><code>&lt;input&gt;</code> elements that set a <code>type</code></h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.InvariantCulture</code>.
</p>

<ul>
```

```
    <li><label><b>Date:</b> <input type="date" @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input type="number" @bind="number" /></label>
</li>
</ul>

@code {
    private DateTime dt = DateTime.Now;
    private double number = 1999.69;
}
```

The number string format (`N2`) in the preceding example (`.ToString("N2")`) is a standard .NET numeric format specifier. The `N2` format is supported for all numeric types, includes a group separator, and renders up to two decimal places.

Optionally, add a menu item to the navigation in the `NavMenu` component (`NavMenu.razor`) for the `CultureExample1` component.

Dynamically set the culture from the `Accept-Language` header

Add the Microsoft.Extensions.Localization package to the app.

The Accept-Language header is set by the browser and controlled by the user's language preferences in browser settings. In browser settings, a user sets one or more preferred languages in order of preference. The order of preference is used by the browser to set quality values (`q`, 0-1) for each language in the header. The following example specifies United States English, English, and Costa Rican Spanish with a preference for United States English or English:

Accept-Language: en-US,en;q=0.9,es-CR;q=0.8

The app's culture is set by matching the first requested language that matches a supported culture of the app.

In ***client-side development***, set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the client-side app's project file (`.csproj`):

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlo
balizationData>
</PropertyGroup>
```

> **Note**
>
> If the app's specification requires limiting the supported cultures to an explicit list, see the Dynamically set the client-side culture by user preference section of this article.

Apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

Add the following line to the `Program` file where services are registered:

```
C#

builder.Services.AddLocalization();
```

In ***server-side development***, specify the app's supported cultures before any middleware that might check the request culture. Generally, place Request Localization Middleware immediately before calling MapRazorComponents. The following example configures supported cultures for United States English and Costa Rican Spanish:

```
C#

app.UseRequestLocalization(new RequestLocalizationOptions()
    .AddSupportedCultures(new[] { "en-US", "es-CR" })
    .AddSupportedUICultures(new[] { "en-US", "es-CR" }));
```

For information on ordering the Localization Middleware in the middleware pipeline of the `Program` file, see ASP.NET Core Middleware.

Use the `CultureExample1` component shown in the Demonstration component section to study how globalization works. Issue a request with United States English (`en-US`). Switch to Costa Rican Spanish (`es-CR`) in the browser's language settings. Request the webpage again.

> **Note**
>
> Some browsers force you to use the default language setting for both requests and the browser's own UI settings. This can make changing the language back to one that you understand difficult because all of the setting UI screens might end up in a language that you can't read. A browser such as Opera is a good choice for

testing because it permits you to set a default language for webpage requests but leave the browser's settings UI in your language.

When the culture is United States English (`en-US`), the rendered component uses month/day date formatting (`6/7`), 12-hour time (`AM`/`PM`), and comma separators in numbers with a dot for the decimal value (`1,999.69`):

- **Date**: 6/7/2021 6:45:22 AM
- **Number**: 1,999.69

When the culture is Costa Rican Spanish (`es-CR`), the rendered component uses day/month date formatting (`7/6`), 24-hour time, and period separators in numbers with a comma for the decimal value (`1.999,69`):

- **Date**: 7/6/2021 6:49:38
- **Number**: 1.999,69

Statically set the client-side culture

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the app's project file (`.csproj`):

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlo
balizationData>
</PropertyGroup>
```

The app's culture can be set in JavaScript when Blazor starts with the `applicationCulture` Blazor start option. The following example configures the app to launch using the United States English (`en-US`) culture.

Prevent Blazor autostart by adding `autostart="false"` to Blazor's <script> tag:

HTML

```
<script src="{BLAZOR SCRIPT}" autostart="false"></script>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Add the following `<script>` block after Blazor's <script> tag and before the closing `</body>` tag:

Blazor Web App:

HTML

```
<script>
  Blazor.start({
    webAssembly: {
      applicationCulture: 'en-US'
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script>
  Blazor.start({
    applicationCulture: 'en-US'
  });
</script>
```

The value for `applicationCulture` must conform to the BCP-47 language tag format. For more information on Blazor startup, see ASP.NET Core Blazor startup.

An alternative to setting the culture Blazor's start option is to set the culture in C# code. Set CultureInfo.DefaultThreadCurrentCulture and CultureInfo.DefaultThreadCurrentUICulture in the `Program` file to the same culture.

Add the System.Globalization namespace to the `Program` file:

C#

```
using System.Globalization;
```

Add the culture settings before the line that builds and runs the WebAssemblyHostBuilder (`await builder.Build().RunAsync();`):

C#

```
CultureInfo.DefaultThreadCurrentCulture = new CultureInfo("en-US");
CultureInfo.DefaultThreadCurrentUICulture = new CultureInfo("en-US");
```

> **Note**
>
> Currently, Blazor WebAssembly apps only load resources based on DefaultThreadCurrentCulture. For more information, see Blazor WASM only relies on the current culture (current UI culture isn't respected) (dotnet/aspnetcore #56824).

Use the `CultureExample1` component shown in the Demonstration component section to study how globalization works. Issue a request with United States English (`en-US`). Switch to Costa Rican Spanish (`es-CR`) in the browser's language settings. Request the webpage again. When the requested language is Costa Rican Spanish, the app's culture remains United States English (`en-US`).

Statically set the server-side culture

Server-side apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

In the `Program` file:

```
C#

builder.Services.AddLocalization();
```

Specify the static culture in the `Program` file before any middleware that might check the request culture. Generally, place Request Localization Middleware immediately before MapRazorComponents. The following example configures United States English:

```
C#

app.UseRequestLocalization("en-US");
```

The culture value for UseRequestLocalization must conform to the BCP-47 language tag format.

For information on ordering the Localization Middleware in the middleware pipeline of the `Program` file, see ASP.NET Core Middleware.

Use the `CultureExample1` component shown in the Demonstration component section to study how globalization works. Issue a request with United States English (`en-US`). Switch to Costa Rican Spanish (`es-CR`) in the browser's language settings. Request the

webpage again. When the requested language is Costa Rican Spanish, the app's culture remains United States English (`en-US`).

Dynamically set the client-side culture by user preference

Examples of locations where an app might store a user's preference include in browser local storage (common for client-side scenarios), in a localization cookie or database (common for server-side scenarios), or in an external service attached to an external database and accessed by a web API. The following example demonstrates how to use browser local storage.

Add the Microsoft.Extensions.Localization package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the project file:

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlobalizationData>
</PropertyGroup>
```

The app's culture for client-side rendering is set using the Blazor framework's API. A user's culture selection can be persisted in browser local storage.

Provide JS functions after Blazor's <script> tag to get and set the user's culture selection with browser local storage:

HTML

```
<script>
  window.blazorCulture = {
    get: () => window.localStorage['BlazorCulture'],
    set: (value) => window.localStorage['BlazorCulture'] = value
```

```
  };
</script>
```

> **Note**
>
> The preceding example pollutes the client with global functions. For a better approach in production apps, see JavaScript isolation in JavaScript modules.

Add the namespaces for System.Globalization and Microsoft.JSInterop to the top of the `Program` file:

C#

```
using System.Globalization;
using Microsoft.JSInterop;
```

Remove the following line:

diff

```
- await builder.Build().RunAsync();
```

Replace the preceding line with the following code. The code adds Blazor's localization service to the app's service collection with AddLocalization and uses JS interop to call into JS and retrieve the user's culture selection from local storage. If local storage doesn't contain a culture for the user, the code sets a default value of United States English (`en-US`).

C#

```
builder.Services.AddLocalization();

var host = builder.Build();

const string defaultCulture = "en-US";

var js = host.Services.GetRequiredService<IJSRuntime>();
var result = await js.InvokeAsync<string>("blazorCulture.get");
var culture = CultureInfo.GetCultureInfo(result ?? defaultCulture);

if (result == null)
{
    await js.InvokeVoidAsync("blazorCulture.set", defaultCulture);
}

CultureInfo.DefaultThreadCurrentCulture = culture;
```

```
CultureInfo.DefaultThreadCurrentUICulture = culture;

await host.RunAsync();
```

> ⓘ Note
>
> Currently, Blazor WebAssembly apps only load resources based on DefaultThreadCurrentCulture. For more information, see Blazor WASM only relies on the current culture (current UI culture isn't respected) (dotnet/aspnetcore #56824).

The following `CultureSelector` component shows how to perform the following actions:

- Set the user's culture selection into browser local storage via JS interop.
- Reload the component that they requested (`forceLoad: true`), which uses the updated culture.

`CultureSelector.razor`:

razor

```
@using System.Globalization
@inject IJSRuntime JS
@inject NavigationManager Navigation

<p>
    <label>
        Select your locale:
        <select @bind="selectedCulture"
@bind:after="ApplySelectedCultureAsync">
            @foreach (var culture in supportedCultures)
            {
                <option value="@culture">@culture.DisplayName</option>
            }
        </select>
    </label>
</p>

@code
{
    private CultureInfo[] supportedCultures = new[]
    {
        new CultureInfo("en-US"),
        new CultureInfo("es-CR"),
    };

    private CultureInfo? selectedCulture;

    protected override void OnInitialized()
```

```
    {
        selectedCulture = CultureInfo.CurrentCulture;
    }

    private async Task ApplySelectedCultureAsync()
    {
        if (CultureInfo.CurrentCulture != selectedCulture)
        {
            await JS.InvokeVoidAsync("blazorCulture.set",
selectedCulture!.Name);

            Navigation.NavigateTo(Navigation.Uri, forceLoad: true);
        }
    }
}
```

> ⓘ Note
>
> For more information on IJSInProcessRuntime, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Inside the closing tag of the `</main>` element in the `MainLayout` component (`MainLayout.razor`), add the `CultureSelector` component:

razor

```
<article class="bottom-row px-4">
    <CultureSelector />
</article>
```

Use the `CultureExample1` component shown in the Demonstration component section to study how the preceding example works.

Dynamically set the server-side culture by user preference

Examples of locations where an app might store a user's preference include in browser local storage (common for client-side scenarios), in a localization cookie or database (common for server-side scenarios), or in an external service attached to an external database and accessed by a web API. The following example demonstrates how to use a localization cookie.

> ⓘ Note

The following example assumes that the app adopts ***global*** interactivity by specifying the interactive server-side rendering (interactive SSR) on the `Routes` component in the `App` component (`Components/App.razor`):

razor

```
<Routes @rendermode="InteractiveServer" />
```

If the app adopts ***per-page/component*** interactivity, see the remarks at the end of this section to modify the render modes of the example's components.

Add the Microsoft.Extensions.Localization package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Server-side apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

In the `Program` file:

C#

```
builder.Services.AddLocalization();
```

Set the app's default and supported cultures with RequestLocalizationOptions.

Before the call to MapRazorComponents in the request processing pipeline, place the following code:

C#

```
var supportedCultures = new[] { "en-US", "es-CR" };
var localizationOptions = new RequestLocalizationOptions()
    .SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);
```

For information on ordering the Localization Middleware in the middleware pipeline, see ASP.NET Core Middleware.

The following example shows how to set the current culture in a cookie that can be read by the Localization Middleware.

The following namespaces are required for the `App` component:

- System.Globalization
- Microsoft.AspNetCore.Localization

Add the following to the top of the `App` component file (`Components/App.razor`):

razor

```
@using System.Globalization
@using Microsoft.AspNetCore.Localization
```

Add the following `@code` block to the bottom of the `App` component file:

razor

```
@code {
    [CascadingParameter]
    public HttpContext? HttpContext { get; set; }

    protected override void OnInitialized()
    {
        HttpContext?.Response.Cookies.Append(
            CookieRequestCultureProvider.DefaultCookieName,
            CookieRequestCultureProvider.MakeCookieValue(
                new RequestCulture(
                    CultureInfo.CurrentCulture,
                    CultureInfo.CurrentUICulture)));
    }
}
```

For information on ordering the Localization Middleware in the middleware pipeline, see ASP.NET Core Middleware.

If the app isn't configured to process controller actions:

- Add MVC services by calling AddControllers on the service collection in the `Program` file:

  C#

```
builder.Services.AddControllers();
```

- Add controller endpoint routing in the `Program` file by calling MapControllers on the IEndpointRouteBuilder (`app`):

```
C#

app.MapControllers();
```

To provide UI to allow a user to select a culture, use a *redirect-based approach* with a localization cookie. The app persists the user's selected culture via a redirect to a controller. The controller sets the user's selected culture into a cookie and redirects the user back to the original URI. The process is similar to what happens in a web app when a user attempts to access a secure resource, where the user is redirected to a sign-in page and then redirected back to the original resource.

`Controllers/CultureController.cs`:

```
C#

using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;

[Route("[controller]/[action]")]
public class CultureController : Controller
{
    public IActionResult Set(string culture, string redirectUri)
    {
        if (culture != null)
        {
            HttpContext.Response.Cookies.Append(
                CookieRequestCultureProvider.DefaultCookieName,
                CookieRequestCultureProvider.MakeCookieValue(
                    new RequestCulture(culture, culture)));
        }

        return LocalRedirect(redirectUri);
    }
}
```

> ⚠ **Warning**

Use the LocalRedirect action result, as shown in the preceding example, to prevent open redirect attacks. For more information, see Prevent open redirect attacks in ASP.NET Core.

The following `CultureSelector` component shows how to call the `Set` method of the `CultureController` with the new culture. The component is placed in the `Shared` folder for use throughout the app.

`CultureSelector.razor`:

razor

```
@using System.Globalization
@inject IJSRuntime JS
@inject NavigationManager Navigation

<p>
    <label>
        Select your locale:
        <select @bind="selectedCulture"
@bind:after="ApplySelectedCultureAsync">
            @foreach (var culture in supportedCultures)
            {
                <option value="@culture">@culture.DisplayName</option>
            }
        </select>
    </label>
</p>

@code
{
    private CultureInfo[] supportedCultures = new[]
    {
        new CultureInfo("en-US"),
        new CultureInfo("es-CR"),
    };

    private CultureInfo? selectedCulture;

    protected override void OnInitialized()
    {
        selectedCulture = CultureInfo.CurrentCulture;
    }

    private async Task ApplySelectedCultureAsync()
    {
        if (CultureInfo.CurrentCulture != selectedCulture)
        {
            var uri = new Uri(Navigation.Uri)
                .GetComponents(UriComponents.PathAndQuery,
UriFormat.Unescaped);
```

```
            var cultureEscaped = Uri.EscapeDataString(selectedCulture.Name);
            var uriEscaped = Uri.EscapeDataString(uri);

            Navigation.NavigateTo(
                $"Culture/Set?culture={cultureEscaped}&redirectUri=
{uriEscaped}",
                forceLoad: true);
        }
    }
}
```

Add the `CultureSelector` component to the `MainLayout` component. Place the following markup inside the closing `</main>` tag in the `Components/Layout/MainLayout.razor` file:

razor

```
<article class="bottom-row px-4">
    <CultureSelector />
</article>
```

Use the `CultureExample1` component shown in the Demonstration component section to study how the preceding example works.

The preceding example assumes that the app adopts ***global*** interactivity by specifying the Interactive Server render mode on the `Routes` component in the `App` component (`Components/App.razor`):

razor

```
<Routes @rendermode="InteractiveServer" />
```

If the app adopts ***per-page/component*** interactivity, make the following changes:

- Add the Interactive Server render mode to the top of the `CultureExample1` component file (`Components/Pages/CultureExample1.razor`):

  razor

  ```
  @rendermode InteractiveServer
  ```

- In the app's main layout (`Components/Layout/MainLayout.razor`), apply the Interactive Server render mode to the `CultureSelector` component:

  razor

```
<CultureSelector @rendermode="InteractiveServer" />
```

Dynamically set the culture in a Blazor Web App by user preference

This section applies to Blazor Web Apps that adopt Auto (Server and WebAssembly) interactivity.

Examples of locations where an app might store a user's preference include in browser local storage (common for client-side scenarios), in a localization cookie or database (common for server-side scenarios), both local storage and a localization cookie (Blazor Web Apps with server and WebAssembly components), or in an external service attached to an external database and accessed by a web API. The following example demonstrates how to use browser local storage for client-side rendered (CSR) components and a localization cookie for server-side rendered (SSR) components.

Updates to the `.Client` project

Add the Microsoft.Extensions.Localization package to the `.Client` project.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the `.Client` project file:

XML

```
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlobalizationData>
</PropertyGroup>
```

Add the namespaces for System.Globalization and Microsoft.JSInterop to the top of the `.Client` project's `Program` file:

C#

```
using System.Globalization;
using Microsoft.JSInterop;
```

Remove the following line:

diff

```
- await builder.Build().RunAsync();
```

Replace the preceding line with the following code. The code adds Blazor's localization service to the app's service collection with AddLocalization and uses JS interop to call into JS and retrieve the user's culture selection from local storage. If local storage doesn't contain a culture for the user, the code sets a default value of United States English (`en-US`).

C#

```
builder.Services.AddLocalization();

var host = builder.Build();

const string defaultCulture = "en-US";

var js = host.Services.GetRequiredService<IJSRuntime>();
var result = await js.InvokeAsync<string>("blazorCulture.get");
var culture = CultureInfo.GetCultureInfo(result ?? defaultCulture);

if (result == null)
{
    await js.InvokeVoidAsync("blazorCulture.set", defaultCulture);
}

CultureInfo.DefaultThreadCurrentCulture = culture;
CultureInfo.DefaultThreadCurrentUICulture = culture;

await host.RunAsync();
```

> ⓘ **Note**
>
> Currently, Blazor WebAssembly apps only load resources based on DefaultThreadCurrentCulture. For more information, see Blazor WASM only relies on the current culture (current UI culture isn't respected) (dotnet/aspnetcore #56824).

Add the following `CultureSelector` component to the `.Client` project.

The component adopts the following approaches to work for either SSR or CSR components:

- The display name of each available culture in the dropdown list is provided by a dictionary because client-side globalization data include localized text of culture display names that server-side globalization data provides. For example, server-side localization displays `English (United States)` when `en-US` is the culture and `Ingles ()` when a different culture is used. Because localization of the culture display names isn't available with Blazor WebAssembly globalization, the display name for United States English on the client for any loaded culture is just `en-US`. Using a custom dictionary permits the component to at least display full English culture names.
- When user changes the culture, JS interop sets the culture in local browser storage and a controller action updates the localization cookie with the culture. The controller is added to the app later in the Server project updates section.

`Pages/CultureSelector.razor`:

razor

```
@using System.Globalization
@inject IJSRuntime JS
@inject NavigationManager Navigation

<p>
    <label>
        Select your locale:
        <select @bind="@selectedCulture"
@bind:after="ApplySelectedCultureAsync">
            @foreach (var culture in supportedCultures)
            {
                <option value="@culture">@cultureDict[culture.Name]</option>
            }
        </select>
    </label>
</p>

@code
{
    private Dictionary<string, string> cultureDict =
        new()
        {
            { "en-US", "English (United States)" },
            { "es-CR", "Spanish (Costa Rica)" }
        };

    private CultureInfo[] supportedCultures = new[]
```

```
    {
        new CultureInfo("en-US"),
        new CultureInfo("es-CR"),
    };

    private CultureInfo? selectedCulture;

    protected override void OnInitialized()
    {
        selectedCulture = CultureInfo.CurrentCulture;
    }

    private async Task ApplySelectedCultureAsync()
    {
        if (CultureInfo.CurrentCulture != selectedCulture)
        {
            await JS.InvokeVoidAsync("blazorCulture.set",
selectedCulture!.Name);

            var uri = new Uri(Navigation.Uri)
                .GetComponents(UriComponents.PathAndQuery,
UriFormat.Unescaped);
            var cultureEscaped = Uri.EscapeDataString(selectedCulture.Name);
            var uriEscaped = Uri.EscapeDataString(uri);

            Navigation.NavigateTo(
                $"Culture/Set?culture={cultureEscaped}&redirectUri=
{uriEscaped}",
                forceLoad: true);
        }
    }
}
```

> **Note**
>
> For more information on IJSInProcessRuntime, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

In the `.Client` project, place the following `CultureClient` component to study how globalization works for CSR components.

`Pages/CultureClient.razor`:

razor

```
@page "/culture-client"
@rendermode InteractiveWebAssembly
@using System.Globalization

<PageTitle>Culture Client</PageTitle>
```

```
<h1>Culture Client</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Rendered values</h2>

<ul>
    <li><b>Date</b>: @dt</li>
    <li><b>Number</b>: @number.ToString("N2")</li>
</ul>

<h2><code>&lt;input&gt;</code> elements that don't set a <code>type</code>
</h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.CurrentCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input @bind="number" /></label></li>
</ul>

<h2><code>&lt;input&gt;</code> elements that set a <code>type</code></h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.InvariantCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input type="date" @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input type="number" @bind="number" /></label>
</li>
</ul>

@code {
    private DateTime dt = DateTime.Now;
    private double number = 1999.69;
}
```

Server project updates

Add the Microsoft.Extensions.Localization package to the server project.

> **Note**

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Server-side apps are localized using Localization Middleware. Add localization services to the app with AddLocalization.

In the server project's `Program` file where services are registered:

C#

```
builder.Services.AddLocalization();
```

Set the app's default and supported cultures with RequestLocalizationOptions.

Before the call to MapRazorComponents in the request processing pipeline, place the following code:

C#

```
var supportedCultures = new[] { "en-US", "es-CR" };
var localizationOptions = new RequestLocalizationOptions()
    .SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);
```

The following example shows how to set the current culture in a cookie that can be read by the Localization Middleware.

The following namespaces are required for the `App` component:

- System.Globalization
- Microsoft.AspNetCore.Localization

Add the following to the top of the `App` component file (`Components/App.razor`):

razor

```
@using System.Globalization
@using Microsoft.AspNetCore.Localization
```

The app's culture for client-side rendering is set using the Blazor framework's API. A user's culture selection can be persisted in browser local storage for CSR components.

After the Blazor's <script> tag, provide JS functions to get and set the user's culture selection with browser local storage:

HTML

```
<script>
  window.blazorCulture = {
    get: () => window.localStorage['BlazorCulture'],
    set: (value) => window.localStorage['BlazorCulture'] = value
  };
</script>
```

> **Note**
>
> The preceding example pollutes the client with global functions. For a better approach in production apps, see JavaScript isolation in JavaScript modules.

Add the following `@code` block to the bottom of the `App` component file:

razor

```
@code {
    [CascadingParameter]
    public HttpContext? HttpContext { get; set; }

    protected override void OnInitialized()
    {
        HttpContext?.Response.Cookies.Append(
            CookieRequestCultureProvider.DefaultCookieName,
            CookieRequestCultureProvider.MakeCookieValue(
                new RequestCulture(
                    CultureInfo.CurrentCulture,
                    CultureInfo.CurrentUICulture)));
    }
}
```

If the server project isn't configured to process controller actions:

- Add MVC services by calling AddControllers on the service collection in the `Program` file:

  C#

  ```
  builder.Services.AddControllers();
  ```

- Add controller endpoint routing in the `Program` file by calling MapControllers on the IEndpointRouteBuilder (`app`):

  C#

  ```
  app.MapControllers();
  ```

To allow a user to select a culture for SSR components, use a *redirect-based approach* with a localization cookie. The app persists the user's selected culture via a redirect to a controller. The controller sets the user's selected culture into a cookie and redirects the user back to the original URI. The process is similar to what happens in a web app when a user attempts to access a secure resource, where the user is redirected to a sign-in page and then redirected back to the original resource.

`Controllers/CultureController.cs`:

C#

```
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;

[Route("[controller]/[action]")]
public class CultureController : Controller
{
    public IActionResult Set(string culture, string redirectUri)
    {
        if (culture != null)
        {
            HttpContext.Response.Cookies.Append(
                CookieRequestCultureProvider.DefaultCookieName,
                CookieRequestCultureProvider.MakeCookieValue(
                    new RequestCulture(culture, culture)));
        }

        return LocalRedirect(redirectUri);
    }
}
```

> ⚠ **Warning**
>
> Use the LocalRedirect action result, as shown in the preceding example, to prevent open redirect attacks. For more information, see Prevent open redirect attacks in ASP.NET Core.

Add the `CultureSelector` component to the `MainLayout` component. Place the following markup inside the closing `</main>` tag in the `Components/Layout/MainLayout.razor` file:

razor

```razor
<article class="bottom-row px-4">
    <CultureSelector @rendermode="InteractiveAuto" />
</article>
```

Use the `CultureExample1` component shown in the Demonstration component section to study how the preceding example works.

In the server project, place the following `CultureServer` component to study how globalization works for SSR components.

`Components/Pages/CultureServer.razor`:

razor

```razor
@page "/culture-server"
@rendermode InteractiveServer
@using System.Globalization

<PageTitle>Culture Server</PageTitle>

<h1>Culture Server</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Rendered values</h2>

<ul>
    <li><b>Date</b>: @dt</li>
    <li><b>Number</b>: @number.ToString("N2")</li>
</ul>

<h2><code>&lt;input&gt;</code> elements that don't set a <code>type</code>
</h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.CurrentCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input @bind="number" /></label></li>
</ul>
```

```
<h2><code>&lt;input&gt;</code> elements that set a <code>type</code></h2>

<p>
    The following <code>&lt;input&gt;</code> elements use
    <code>CultureInfo.InvariantCulture</code>.
</p>

<ul>
    <li><label><b>Date:</b> <input type="date" @bind="dt" /></label></li>
    <li><label><b>Number:</b> <input type="number" @bind="number" /></label>
</li>
</ul>

@code {
    private DateTime dt = DateTime.Now;
    private double number = 1999.69;
}
```

Add both the `CultureClient` and `CultureServer` components to the sidebar navigation in `Components/Layout/NavMenu.razor`:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="culture-server">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Culture (Server)
    </NavLink>
</div>
<div class="nav-item px-3">
    <NavLink class="nav-link" href="culture-client">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Culture (Client)
    </NavLink>
</div>
```

Interactive Auto components

The guidance in this section also works for components that adopt the Interactive Auto render mode:

razor

```
@rendermode InteractiveAuto
```

Localization

If the app doesn't already support dynamic culture selection, add the Microsoft.Extensions.Localization package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Client-side localization

Set the `BlazorWebAssemblyLoadAllGlobalizationData` property to `true` in the app's project file (`.csproj`):

```xml
<PropertyGroup>

<BlazorWebAssemblyLoadAllGlobalizationData>true</BlazorWebAssemblyLoadAllGlobalizationData>
</PropertyGroup>
```

In the `Program` file, add namespace the namespace for System.Globalization to the top of the file:

```csharp
using System.Globalization;
```

Add Blazor's localization service to the app's service collection with AddLocalization:

```csharp
builder.Services.AddLocalization();
```

Server-side localization

Use Localization Middleware to set the app's culture.

If the app doesn't already support dynamic culture selection:

- Add localization services to the app with AddLocalization.

- Specify the app's default and supported cultures in the `Program` file. The following example configures supported cultures for United States English and Costa Rican Spanish.

```
C#

builder.Services.AddLocalization();
```

Place Request Localization Middleware before any middleware that might check the request culture. Generally, place the middleware immediately before calling MapRazorComponents:

```
C#

var supportedCultures = new[] { "en-US", "es-CR" };
var localizationOptions = new RequestLocalizationOptions()
    .SetDefaultCulture(supportedCultures[0])
    .AddSupportedCultures(supportedCultures)
    .AddSupportedUICultures(supportedCultures);

app.UseRequestLocalization(localizationOptions);
```

For information on ordering the Localization Middleware in the middleware pipeline, see ASP.NET Core Middleware.

If the app should localize resources based on storing a user's culture setting, use a localization culture cookie. Use of a cookie ensures that the WebSocket connection can correctly propagate the culture. If localization schemes are based on the URL path or query string, the scheme might not be able to work with WebSockets, thus fail to persist the culture. Therefore, the recommended approach is to use a localization culture cookie. See the Dynamically set the server-side culture by user preference section of this article to see an example Razor expression that persists the user's culture selection.

Example of localized resources

The example of localized resources in this section works with the prior examples in this article where the app's supported cultures are English (`en`) as a default locale and Spanish (`es`) as a user-selectable or browser-specified alternate locale.

Create a resource file for each locale. In the following example, resources are created for a `Greeting` string in English and Spanish:

- English (`en`): `Hello, World!`

- Spanish (`es`): `¡Hola, Mundo!`

> **Note**
>
> The following resource file can be added in Visual Studio by right-clicking the `Pages` folder and selecting **Add** > **New Item** > **Resources File**. Name the file `CultureExample2.resx`. When the editor appears, provide data for a new entry. Set the **Name** to `Greeting` and **Value** to `Hello, World!`. Save the file.
>
> If using Visual Studio Code, we recommend installing Tim Heuer's ResX Viewer and Editor. Add an empty `CultureExample2.resx` file to the `Pages` folder. The extension automatically takes over managing the file in the UI. Select the **Add New Resource** button. Follow the instructions to add an entry for `Greeting` (key), `Hello, World!` (value), and `None` (comment). Save the file. If you close and re-open the file, you can see the `Greeting` resource.
>
> Tim Heuer's ResX Viewer and Editor isn't owned or maintained by Microsoft and isn't covered by any Microsoft Support Agreement or license.

The following demonstrates a typical resource file. You can manually place resource files into the app's `Pages` folder if you prefer not to use built-in tooling with an integrated development environment (IDE), such as Visual Studio's built-in resource file editor or Visual Studio Code with an extension for creating and editing resource files.

`Pages/CultureExample2.resx`:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <xsd:schema id="root" xmlns=""
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
    <xsd:import namespace="http://www.w3.org/XML/1998/namespace" />
    <xsd:element name="root" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="metadata">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0" />
              </xsd:sequence>
              <xsd:attribute name="name" use="required" type="xsd:string" />
              <xsd:attribute name="type" type="xsd:string" />
              <xsd:attribute name="mimetype" type="xsd:string" />
              <xsd:attribute ref="xml:space" />
```

```
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="assembly">
            <xsd:complexType>
              <xsd:attribute name="alias" type="xsd:string" />
              <xsd:attribute name="name" type="xsd:string" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="data">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
                <xsd:element name="comment" type="xsd:string" minOccurs="0"
msdata:Ordinal="2" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required"
msdata:Ordinal="1" />
              <xsd:attribute name="type" type="xsd:string"
msdata:Ordinal="3" />
              <xsd:attribute name="mimetype" type="xsd:string"
msdata:Ordinal="4" />
              <xsd:attribute ref="xml:space" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="resheader">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required" />
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="version">
    <value>2.0</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <data name="Greeting" xml:space="preserve">
    <value>Hello, World!</value>
```

```
  </data>
</root>
```

> **Note**
>
> The following resource file can be added in Visual Studio by right-clicking the `Pages` folder and selecting **Add** > **New Item** > **Resources File**. Name the file `CultureExample2.es.resx`. When the editor appears, provide data for a new entry. Set the **Name** to `Greeting` and **Value** to `¡Hola, Mundo!`. Save the file.
>
> If using Visual Studio Code, we recommend installing Tim Heuer's ResX Viewer and Editor. Add an empty `CultureExample2.resx` file to the `Pages` folder. The extension automatically takes over managing the file in the UI. Select the **Add New Resource** button. Follow the instructions to add an entry for `Greeting` (key), `¡Hola, Mundo!` (value), and `None` (comment). Save the file. If you close and re-open the file, you can see the `Greeting` resource.

The following demonstrates a typical resource file. You can manually place resource files into the app's `Pages` folder if you prefer not to use built-in tooling with an integrated development environment (IDE), such as Visual Studio's built-in resource file editor or Visual Studio Code with an extension for creating and editing resource files.

`Pages/CultureExample2.es.resx`:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <xsd:schema id="root" xmlns=""
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
    <xsd:import namespace="http://www.w3.org/XML/1998/namespace" />
    <xsd:element name="root" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="metadata">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0" />
              </xsd:sequence>
              <xsd:attribute name="name" use="required" type="xsd:string" />
              <xsd:attribute name="type" type="xsd:string" />
              <xsd:attribute name="mimetype" type="xsd:string" />
              <xsd:attribute ref="xml:space" />
            </xsd:complexType>
          </xsd:element>
```

```
          <xsd:element name="assembly">
            <xsd:complexType>
              <xsd:attribute name="alias" type="xsd:string" />
              <xsd:attribute name="name" type="xsd:string" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="data">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
                <xsd:element name="comment" type="xsd:string" minOccurs="0"
msdata:Ordinal="2" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required"
msdata:Ordinal="1" />
              <xsd:attribute name="type" type="xsd:string"
msdata:Ordinal="3" />
              <xsd:attribute name="mimetype" type="xsd:string"
msdata:Ordinal="4" />
              <xsd:attribute ref="xml:space" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="resheader">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="value" type="xsd:string" minOccurs="0"
msdata:Ordinal="1" />
              </xsd:sequence>
              <xsd:attribute name="name" type="xsd:string" use="required" />
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="version">
    <value>2.0</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter, System.Windows.Forms,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <data name="Greeting" xml:space="preserve">
    <value>¡Hola, Mundo!</value>
  </data>
</root>
```

The following component demonstrates the use of the localized `Greeting` string with IStringLocalizer<T>. The Razor markup `@Loc["Greeting"]` in the following example localizes the string keyed to the `Greeting` value, which is set in the preceding resource files.

Add the namespace for Microsoft.Extensions.Localization to the app's `_Imports.razor` file:

razor

```
@using Microsoft.Extensions.Localization
```

`CultureExample2.razor`:

razor

```
@page "/culture-example-2"
@using System.Globalization
@inject IStringLocalizer<CultureExample2> Loc

<h1>Culture Example 2</h1>

<ul>
    <li><b>CurrentCulture</b>: @CultureInfo.CurrentCulture</li>
    <li><b>CurrentUICulture</b>: @CultureInfo.CurrentUICulture</li>
</ul>

<h2>Greeting</h2>

<p>
    @Loc["Greeting"]
</p>

<p>
    @greeting
</p>

@code {
    private string? greeting;

    protected override void OnInitialized()
    {
        greeting = Loc["Greeting"];
    }
}
```

Optionally, add a menu item for the `CultureExample2` component to the navigation in the `NavMenu` component (`NavMenu.razor`).

WebAssembly culture provider reference source

To further understand how the Blazor framework processes localization, see the WebAssemblyCultureProvider class in the ASP.NET Core reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Shared resources

To create localization shared resources, adopt the following approach.

- Create a dummy class with an arbitrary class name. In the following example:
  - The app uses the `BlazorSample` namespace, and localization assets use the `BlazorSample.Localization` namespace.
  - The dummy class is named `SharedResource`.
  - The class file is placed in a `Localization` folder at the root of the app.

  `Localization/SharedResource.cs`:

  C#

  ```
  namespace BlazorSample.Localization;

  public class SharedResource
  {
  }
  ```

- Create the shared resource files with a **Build Action** of `Embedded resource`. In the following example:

  - The files are placed in the `Localization` folder with the dummy `SharedResource` class (`Localization/SharedResource.cs`).

    - Name the resource files to match the name of the dummy class. The following example files include a default localization file and a file for Spanish (`es`) localization.
    - `Localization/SharedResource.resx`
    - `Localization/SharedResource.es.resx`

    > ⚠ **Warning**
    >
    > When following the approach in this section, you can't simultaneously set LocalizationOptions.ResourcesPath and use IStringLocalizerFactory.Create to load resources.

- To reference the dummy class for an injected IStringLocalizer<T> in a Razor component, either place an @using directive for the localization namespace or include the localization namespace in the dummy class reference. In the following examples:
    - The first example states the `Localization` namespace for the `SharedResource` dummy class with an @using directive.
    - The second example states the `SharedResource` dummy class's namespace explicitly.

    In a Razor component, use ***either*** of the following approaches:

    razor

    ```
    @using Localization
    @inject IStringLocalizer<SharedResource> Loc
    ```

    razor

    ```
    @inject IStringLocalizer<Localization.SharedResource> Loc
    ```

For additional guidance, see Globalization and localization in ASP.NET Core.

Location override using "Sensors" pane in developer tools

When using the location override using the **Sensors** pane in Google Chrome or Microsoft Edge developer tools, the fallback language is reset after prerendering. Avoid

setting the language using the **Sensors** pane when testing. Set the language using the browser's language settings.

For more information, see Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707).

Additional resources

- Set the app base path
- Globalization and localization in ASP.NET Core
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics
- Calling InvokeAsync(StateHasChanged) causes page to fallback to default culture (dotnet/aspnetcore #28521)
- Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707) (Location override using "Sensors" pane)

ASP.NET Core Blazor forms overview

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use forms in Blazor.

Input components and forms

The Blazor framework supports forms and provides built-in input components:

- Bound to an object or model that can use data annotations.
  - HTML forms with the `<form>` element.
  - EditForm components.
- Built-in input components.

> **Note**
>
> Unsupported ASP.NET Core validation features are covered in the Unsupported validation features section.

The Microsoft.AspNetCore.Components.Forms namespace provides:

- Classes for managing form elements, state, and validation.
- Access to built-in Input* components.

A project created from the Blazor project template includes the namespace in the app's `_Imports.razor` file, which makes the namespace available to the app's Razor components.

Standard HTML forms are supported. Create a form using the normal HTML `<form>` tag and specify an `@onsubmit` handler for handling the submitted form request.

`StarshipPlainForm.razor`:

```
razor
```

```
@page "/starship-plain-form"
@inject ILogger<StarshipPlainForm> Logger

<form method="post" @onsubmit="Submit" @formname="starship-plain-form">
    <AntiforgeryToken />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</form>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Id = {Id}", Model?.Id);

    public class Starship
    {
        public string? Id { get; set; }
    }
}
```

In the preceding `StarshipPlainForm` component:

- The form is rendered where the `<form>` element appears. The form is named with the @formname directive attribute, which uniquely identifies the form to the Blazor framework.
- The model is created in the component's `@code` block and held in a public property (`Model`). The `[SupplyParameterFromForm]` attribute indicates that the value of the associated property should be supplied from the form data. Data in the request that matches the property's name is bound to the property.
- The InputText component is an input component for editing string values. The `@bind-Value` directive attribute binds the `Model.Id` model property to the InputText component's Value property.
- The `Submit` method is registered as a handler for the `@onsubmit` callback. The handler is called when the form is submitted by the user.

> **ⓘ Important**

Always use the **@formname** directive attribute with a unique form name.

Blazor enhances page navigation and form handling by intercepting the request in order to apply the response to the existing DOM, preserving as much of the rendered form as possible. The enhancement avoids the need to fully load the page and provides a much smoother user experience, similar to a single-page app (SPA), although the component is rendered on the server. For more information, see ASP.NET Core Blazor routing and navigation.

Streaming rendering is supported for plain HTML forms.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

The preceding example includes antiforgery support by including an AntiforgeryToken component in the form. Antiforgery support is explained further in the Antiforgery support section of this article.

To submit a form based on another element's DOM events, for example `oninput` or `onblur`, use JavaScript to submit the form (submit (MDN documentation)).

Instead of using plain forms in Blazor apps, a form is typically defined with Blazor's built-in form support using the framework's EditForm component. The following Razor component demonstrates typical elements, components, and Razor code to render a webform using an EditForm component.

`Starship1.razor`:

razor

```razor
@page "/starship-1"
@inject ILogger<Starship1> Logger

<EditForm Model="Model" OnSubmit="Submit" FormName="Starship1">
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
```

```
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Id = {Id}", Model?.Id);

    public class Starship
    {
        public string? Id { get; set; }
    }
}
```

In the preceding `Starship1` component:

- The EditForm component is rendered where the `<EditForm>` element appears. The form is named with the @formname directive attribute, which uniquely identifies the form to the Blazor framework.
- The model is created in the component's `@code` block and held in a public property (`Model`). The property is assigned to the EditForm.Model parameter. The `[SupplyParameterFromForm]` attribute indicates that the value of the associated property should be supplied from the form data. Data in the request that matches the property's name is bound to the property.
- The InputText component is an input component for editing string values. The `@bind-Value` directive attribute binds the `Model.Id` model property to the InputText component's Value property.
- The `Submit` method is registered as a handler for the OnSubmit callback. The handler is called when the form is submitted by the user.

> **ⓘ Important**
>
> Always use the **@formname** directive attribute with a unique form name.

Blazor enhances page navigation and form handling for EditForm components. For more information, see ASP.NET Core Blazor routing and navigation.

Streaming rendering is supported for EditForm.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

†For more information on property binding, see ASP.NET Core Blazor data binding.

In the next example, the preceding component is modified to create the form in the `Starship2` component:

- OnSubmit is replaced with OnValidSubmit, which processes assigned event handler if the form is valid when submitted by the user.
- A ValidationSummary component is added to display validation messages when the form is invalid on form submission.
- The data annotations validator (DataAnnotationsValidator component†) attaches validation support using data annotations:
  - If the `<input>` form field is left blank when the **`Submit`** button is selected, an error appears in the validation summary (ValidationSummary component‡) ("`The Id field is required.`") and `Submit` is **not** called.
  - If the `<input>` form field contains more than ten characters when the **`Submit`** button is selected, an error appears in the validation summary ("`Id is too long.`"). `Submit` is **not** called.
  - If the `<input>` form field contains a valid value when the **`Submit`** button is selected, `Submit` is called.

†The DataAnnotationsValidator component is covered in the Validator component section. ‡The ValidationSummary component is covered in the Validation Summary and Validation Message components section.

`Starship2.razor`:

razor

```razor
@page "/starship-2"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship2> Logger

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship2">
    <DataAnnotationsValidator />
    <ValidationSummary />
```

```
    <label>
        Identifier:
        <InputText @bind-Value="Model!.Id" />
    </label>
    <button type="submit">Submit</button>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Id = {Id}", Model?.Id);

    public class Starship
    {
        [Required]
        [StringLength(10, ErrorMessage = "Id is too long.")]
        public string? Id { get; set; }
    }
}
```

Handle form submission

The EditForm provides the following callbacks for handling form submission:

- Use OnValidSubmit to assign an event handler to run when a form with valid fields is submitted.
- Use OnInvalidSubmit to assign an event handler to run when a form with invalid fields is submitted.
- Use OnSubmit to assign an event handler to run regardless of the form fields' validation status. The form is validated by calling EditContext.Validate in the event handler method. If Validate returns `true`, the form is valid.

Clear a form or field

Reset a form by clearing its model back its default state, which can be performed inside or outside of an EditForm's markup:

razor

```
<button @onclick="ClearForm">Clear form</button>

...
```

```
private void ClearForm() => Model = new();
```

Alternatively, use an explicit Razor expression:

razor

```
<button @onclick="@(() => Model = new())">Clear form</button>
```

Reset a field by clearing its model value back to its default state:

razor

```
<button @onclick="ResetId">Reset Identifier</button>

...

private void ResetId() => Model!.Id = string.Empty;
```

Alternatively, use an explicit Razor expression:

razor

```
<button @onclick="@(() => Model!.Id = string.Empty)">Reset
Identifier</button>
```

There's no need to call StateHasChanged in the preceding examples because StateHasChanged is automatically called by the Blazor framework to rerender the component after an event handler is invoked. If an event handler isn't used to invoke the methods that clear a form or field, then developer code should call StateHasChanged to rerender the component.

Antiforgery support

Antiforgery services are automatically added to Blazor apps when AddRazorComponents is called in the `Program` file.

The app uses Antiforgery Middleware by calling UseAntiforgery in its request processing pipeline in the `Program` file. UseAntiforgery is called after the call to UseRouting. If there are calls to UseRouting and UseEndpoints, the call to UseAntiforgery must go between them. A call to UseAntiforgery must be placed after calls to UseAuthentication and UseAuthorization.

The AntiforgeryToken component renders an antiforgery token as a hidden field, and the `[RequireAntiforgeryToken]` attribute enables antiforgery protection. If an antiforgery check fails, a 400 - Bad Request response is thrown and the form isn't processed.

For forms based on EditForm, the AntiforgeryToken component and `[RequireAntiforgeryToken]` attribute are automatically added to provide antiforgery protection.

For forms based on the HTML `<form>` element, manually add the AntiforgeryToken component to the form:

razor

```razor
<form method="post" @onsubmit="Submit" @formname="starshipForm">
    <AntiforgeryToken />
    <input id="send" type="submit" value="Send" />
</form>

@if (submitted)
{
    <p>Form submitted!</p>
}

@code{
    private bool submitted = false;

    private void Submit() => submitted = true;
}
```

> ⚠ **Warning**
>
> For forms based on either **EditForm** or the HTML `<form>` element, antiforgery protection can be disabled by passing `required: false` to the `[RequireAntiforgeryToken]` attribute. The following example disables antiforgery and is ***not recommended*** for public apps:
>
> razor
>
> ```razor
> @using Microsoft.AspNetCore.Antiforgery
> @attribute [RequireAntiforgeryToken(required: false)]
> ```

For more information, see ASP.NET Core Blazor authentication and authorization.

Mitigate overposting attacks

Statically-rendered server-side forms, such as those typically used in components that create and edit records in a database with a form model, can be vulnerable to an *overposting* attack, also known as a *mass assignment* attack. An overposting attack occurs when a malicious user issues an HTML form POST to the server that processes data for properties that aren't part of the rendered form and that the developer doesn't wish to allow users to modify. The term "overposting" literally means that the malicious user has *over*-POSTed with the form.

Overposting isn't a concern when the model doesn't include restricted properties for create and update operations. However, it's important to keep overposting in mind when working with static SSR-based Blazor forms that you maintain.

To mitigate overposting, we recommend using a separate view model/data transfer object (DTO) for the form and database with create (insert) and update operations. When the form is submitted, only properties of the view model/DTO are used by the component and C# code to modify the database. Any extra data included by a malicious user is discarded, so the malicious user is prevented from conducting an overposting attack.

Enhanced form handling

Enhance navigation for form POST requests with the Enhance parameter for EditForm forms or the `data-enhance` attribute for HTML forms (`<form>`):

razor

```
<EditForm ... Enhance ...>
    ...
</EditForm>
```

HTML

```
<form ... data-enhance ...>
    ...
</form>
```

❌ You can't set enhanced navigation on a form's ancestor element to enable enhanced form handling.

HTML

```
<div ... data-enhance ...>
    <form ...>
        <!-- NOT enhanced -->
```

```
    </form>
</div>
```

Enhanced form posts only work with Blazor endpoints. Posting an enhanced form to non-Blazor endpoint results in an error.

To disable enhanced form handling:

- For an EditForm, remove the Enhance parameter from the form element (or set it to `false`: `Enhance="false"`).
- For an HTML `<form>`, remove the `data-enhance` attribute from form element (or set it to `false`: `data-enhance="false"`).

Blazor's enhanced navigation and form handing may undo dynamic changes to the DOM if the updated content isn't part of the server rendering. To preserve the content of an element, use the `data-permanent` attribute.

In the following example, the content of the `<div>` element is updated dynamically by a script when the page loads:

HTML

```
<div data-permanent>
    ...
</div>
```

To disable enhanced navigation and form handling globally, see ASP.NET Core Blazor startup.

For guidance on using the `enhancedload` event to listen for enhanced page updates, see ASP.NET Core Blazor routing and navigation.

Examples

Examples don't adopt enhanced form handling for form POST requests, but all of the examples can be updated to adopt the enhanced features by following the guidance in the Enhanced form handling section.

To demonstrate how forms work with data annotations validation, example components rely on System.ComponentModel.DataAnnotations API. If you wish to avoid an extra line of code in components that use data annotations, make the namespace available throughout the app's components with the imports file (`_Imports.razor`):

```
```

razor

```
@using System.ComponentModel.DataAnnotations
```

Form examples reference aspects of the Star Trek universe. Star Trek is a copyright ©1966-2023 of CBS Studios and Paramount.

Client-side validation requires a circuit

In Blazor Web Apps, client-side validation requires an active Blazor SignalR circuit. Client-side validation isn't available to forms in components that have adopted static server-side rendering (static SSR). Forms that adopt static SSR are validated on the server after the form is submitted.

Unsupported validation features

All of the data annotation built-in validators are supported in Blazor except for the [Remote] validation attribute.

jQuery validation isn't supported in Razor components. We recommend any of the following approaches:

- Follow the guidance in ASP.NET Core Blazor forms validation for either:
  - Server-side validation in a Blazor Web App that adopts an interactive render mode.
  - Client-side validation in a standalone Blazor Web Assembly app.
- Use native HTML validation attributes (see Client-side form validation (MDN documentation)).
- Adopt a third-party validation JavaScript library.

For statically-rendered forms on the server, a new mechanism for client-side validation is under consideration for .NET 10 in late 2025. For more information, see Create server rendered forms with client validation using Blazor without a circuit (dotnet/aspnetcore #51040).

Additional resources

- ASP.NET Core Blazor file uploads
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)
- ASP.NET Core GitHub repository (dotnet/aspnetcore) forms test assets

ASP.NET Core Blazor input components

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes Blazor's built-in input components.

Input components

The Blazor framework provides built-in input components to receive and validate user input. The built-in input components in the following table are supported in an EditForm with an EditContext.

The components in the table are also supported outside of a form in Razor component markup. Inputs are validated when they're changed and when a form is submitted.

Expand table

Input component	Rendered as...
InputCheckbox	`<input type="checkbox">`
InputDate<TValue>	`<input type="date">`
InputFile	`<input type="file">`
InputNumber<TValue>	`<input type="number">`
InputRadio<TValue>	`<input type="radio">`
InputRadioGroup<TValue>	Group of child InputRadio<TValue>
InputSelect<TValue>	`<select>`
InputText	`<input>`
InputTextArea	`<textarea>`

For more information on the InputFile component, see ASP.NET Core Blazor file uploads.

All of the input components, including EditForm, support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the rendered HTML element.

Input components provide default behavior for validating when a field is changed:

- For input components in a form with an EditContext, the default validation behavior includes updating the field CSS class to reflect the field's state as valid or invalid with validation styling of the underlying HTML element.
- For controls that don't have an EditContext, the default validation reflects the valid or invalid state but doesn't provide validation styling to the underlying HTML element.

Some components include useful parsing logic. For example, InputDate<TValue> and InputNumber<TValue> handle unparseable values gracefully by registering unparseable values as validation errors. Types that can accept null values also support nullability of the target field (for example, `int?` for a nullable integer).

For more information on the InputFile component, see ASP.NET Core Blazor file uploads.

Example form

The following `Starship` type, which is used in several of this article's examples and examples in other *Forms* node articles, defines a diverse set of properties with data annotations:

- `Id` is required because it's annotated with the RequiredAttribute. `Id` requires a value of at least one character but no more than 16 characters using the StringLengthAttribute.
- `Description` is optional because it isn't annotated with the RequiredAttribute.
- `Classification` is required.
- The `MaximumAccommodation` property defaults to zero but requires a value from one to 100,000 per its RangeAttribute.
- `IsValidatedDesign` requires that the property have a `true` value, which matches a selected state when the property is bound to a checkbox in the UI (`<input type="checkbox">`).
- `ProductionDate` is a DateTime and required.

`Starship.cs`:

```
C#
```

```
using System.ComponentModel.DataAnnotations;

namespace BlazorSample;

public class Starship
{
    [Required]
    [StringLength(16, ErrorMessage = "Identifier too long (16 character
limit).")]
    public string? Id { get; set; }

    public string? Description { get; set; }

    [Required]
    public string? Classification { get; set; }

    [Range(1, 100000, ErrorMessage = "Accommodation invalid (1-100000).")]
    public int MaximumAccommodation { get; set; }

    [Required]
    [Range(typeof(bool), "true", "true", ErrorMessage = "Approval
required.")]
    public bool IsValidatedDesign { get; set; }

    [Required]
    public DateTime ProductionDate { get; set; }
}
```

The following form accepts and validates user input using:

- The properties and validation defined in the preceding `Starship` model.
- Several of Blazor's built-in input components.

When the model property for the ship's classification (`Classification`) is set, the option matching the model is checked. For example, `checked="@(Model!.Classification == "Exploration")"` for the classification of an exploration ship. The reason for explicitly setting the checked option is that the value of a `<select>` element is only present in the browser. If the form is rendered on the server after it's submitted, any state from the client is overridden with state from the server, which doesn't ordinarily mark an option as checked. By setting the checked option from the model property, the classification always reflects the model's state. This preserves the classification selection across form submissions that result in the form rerendering on the server. In situations where the form isn't rerendered on the server, such as when the Interactive Server render mode is applied directly to the component, explicit assignment of the checked option from the model isn't necessary because Blazor preserves the state for the `<select>` element on the client.

`Starship3.razor`:

razor

```razor
@page "/starship-3"
@inject ILogger<Starship3> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship3">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="Model!.Description" />
        </label>
    </div>
    <div>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="Model!.Classification">
                <option value="">
                    Select classification ...
                </option>
                <option checked="@(Model!.Classification == "Exploration")"
                    value="Exploration">
                    Exploration
                </option>
                <option checked="@(Model!.Classification == "Diplomacy")"
                    value="Diplomacy">
                    Diplomacy
                </option>
                <option checked="@(Model!.Classification == "Defense")"
                    value="Defense">
                    Defense
                </option>
            </InputSelect>
        </label>
    </div>
    <div>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="Model!.MaximumAccommodation" />
        </label>
    </div>
```

```
    <div>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="Model!.IsValidatedDesign" />
        </label>
    </div>
    <div>
        <label>
            Production Date:
            <InputDate @bind-Value="Model!.ProductionDate" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() =>
        Model ??= new() { ProductionDate = DateTime.UtcNow };

    private void Submit()
    {
        Logger.LogInformation("Id = {Id} Description = {Description} " +
            "Classification = {Classification} MaximumAccommodation = " +
            "{MaximumAccommodation} IsValidatedDesign = " +
            "{IsValidatedDesign} ProductionDate = {ProductionDate}",
            Model?.Id, Model?.Description, Model?.Classification,
            Model?.MaximumAccommodation, Model?.IsValidatedDesign,
            Model?.ProductionDate);
    }
}
```

The EditForm in the preceding example creates an EditContext based on the assigned `Starship` instance (`Model="..."`) and handles a valid form. The next example demonstrates how to assign an EditContext to a form and validate when the form is submitted.

In the following example:

- A shortened version of the earlier `Starfleet Starship Database` form (`Starship3` component) is used that only accepts a value for the starship's Id. The other `Starship` properties receive valid default values when an instance of the `Starship` type is created.
- The `Submit` method executes when the **`Submit`** button is selected.
- The form is validated by calling EditContext.Validate in the `Submit` method.
- Logging is executed depending on the validation result.

`Starship4.razor`:

razor

```razor
@page "/starship-4"
@inject ILogger<Starship4> Logger

<EditForm EditContext="editContext" OnSubmit="Submit" FormName="Starship4">
    <DataAnnotationsValidator />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??=
            new()
                {
                    Id = "NCC-1701",
                    Classification = "Exploration",
                    MaximumAccommodation = 150,
                    IsValidatedDesign = true,
                    ProductionDate = new DateTime(2245, 4, 11)
                };
        editContext = new(Model);
    }

    private void Submit()
    {
        if (editContext != null && editContext.Validate())
        {
            Logger.LogInformation("Submit: Form is valid");
        }
        else
        {
            Logger.LogInformation("Submit: Form is INVALID");
        }
    }
}
```

> ⓘ Note
>
> Changing the EditContext after it's assigned is **not** supported.

Multiple option selection with the `InputSelect` component

Binding supports multiple option selection with the InputSelect<TValue> component. The @onchange event provides an array of the selected options via event arguments (ChangeEventArgs). The value must be bound to an array type, and binding to an array type makes the multiple attribute optional on the InputSelect<TValue> tag.

In the following example, the user must select at least two starship classifications but no more than three classifications.

`Starship5.razor`:

razor

```
@page "/starship-5"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship5> Logger

<h1>Bind Multiple <code>InputSelect</code> Example</h1>

<EditForm EditContext="editContext" OnValidSubmit="Submit"
FormName="Starship5">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Select classifications (Minimum: 2, Maximum: 3):
            <InputSelect @bind-Value="Model!.SelectedClassification">
                <option
value="@Classification.Exploration">Exploration</option>
                <option value="@Classification.Diplomacy">Diplomacy</option>
                <option value="@Classification.Defense">Defense</option>
                <option value="@Classification.Research">Research</option>
            </InputSelect>
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@if (Model?.SelectedClassification?.Length > 0)
```

```
{
    <div>@string.Join(", ", Model.SelectedClassification)</div>
}

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model = new();
        editContext = new(Model);
    }

    private void Submit() => Logger.LogInformation("Submit: Processing
form");

    private class Starship
    {
        [Required]
        [MinLength(2, ErrorMessage = "Select at least two
classifications.")]
        [MaxLength(3, ErrorMessage = "Select no more than three
classifications.")]
        public Classification[]? SelectedClassification { get; set; } =
            new[] { Classification.None };
    }

    private enum Classification { None, Exploration, Diplomacy, Defense,
Research }
}
```

For information on how empty strings and `null` values are handled in data binding, see the Binding InputSelect options to C# object null values section.

Binding `InputSelect` options to C# object `null` values

For information on how empty strings and `null` values are handled in data binding, see ASP.NET Core Blazor data binding.

Display name support

Several built-in components support display names with the InputBase<TValue>.DisplayName parameter.

In the `Starfleet Starship Database` form (`Starship3` component) of the Example form section, the production date of a new starship doesn't specify a display name:

razor

```
<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate" />
</label>
```

If the field contains an invalid date when the form is submitted, the error message doesn't display a friendly name. The field name, "`ProductionDate`" doesn't have a space between "`Production`" and "`Date`" when it appears in the validation summary:

> The ProductionDate field must be a date.

Set the DisplayName property to a friendly name with a space between the words "`Production`" and "`Date`":

razor

```
<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate"
        DisplayName="Production Date" />
</label>
```

The validation summary displays the friendly name when the field's value is invalid:

> The Production Date field must be a date.

Error message template support

InputDate<TValue> and InputNumber<TValue> support error message templates:

- InputDate<TValue>.ParsingErrorMessage
- InputNumber<TValue>.ParsingErrorMessage

In the `Starfleet Starship Database` form (`Starship3` component) of the Example form section with a friendly display name assigned, the `Production Date` field produces an error message using the following default error message template:

css

```
The {0} field must be a date.
```

The position of the `{0}` placeholder is where the value of the DisplayName property appears when the error is displayed to the user.

razor

```
<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate"
        DisplayName="Production Date" />
</label>
```

> The Production Date field must be a date.

Assign a custom template to ParsingErrorMessage to provide a custom message:

razor

```
<label>
    Production Date:
    <InputDate @bind-Value="Model!.ProductionDate"
        DisplayName="Production Date"
        ParsingErrorMessage="The {0} field has an incorrect date value." />
</label>
```

> The Production Date field has an incorrect date value.

ASP.NET Core Blazor forms binding

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use binding in Blazor forms.

`EditForm` / `EditContext` model

An EditForm creates an EditContext based on the assigned object as a cascading value for other components in the form. The EditContext tracks metadata about the edit process, including which form fields have been modified and the current validation messages. Assigning to either an EditForm.Model or an EditForm.EditContext can bind a form to data.

Model binding

Assignment to EditForm.Model:

razor

```
<EditForm ... Model="Model" ...>
    ...
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();
}
```

Context binding

Assignment to EditForm.EditContext:

razor

```
<EditForm ... EditContext="editContext" ...>
    ...
</EditForm>

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??= new();
        editContext = new(Model);
    }
}
```

Assign **either** an EditContext **or** a Model to an EditForm. If both are assigned, a runtime error is thrown.

Supported types

Binding supports:

- Primitive types
- Collections
- Complex types
- Recursive types
- Types with constructors
- Enums

You can also use the [DataMember] and [IgnoreDataMember] attributes to customize model binding. Use these attributes to rename properties, ignore properties, and mark properties as required.

Additional binding options

Additional model binding options are available from RazorComponentsServiceOptions when calling AddRazorComponents:

- MaxFormMappingCollectionSize: Maximum number of elements allowed in a form collection.
- MaxFormMappingRecursionDepth: Maximum depth allowed when recursively mapping form data.

- MaxFormMappingErrorCount: Maximum number of errors allowed when mapping form data.
- MaxFormMappingKeySize: Maximum size of the buffer used to read form data keys.

The following demonstrates the default values assigned by the framework:

C#

```
builder.Services.AddRazorComponents(options =>
{
    options.FormMappingUseCurrentCulture = true;
    options.MaxFormMappingCollectionSize = 1024;
    options.MaxFormMappingErrorCount = 200;
    options.MaxFormMappingKeySize = 1024 * 2;
    options.MaxFormMappingRecursionDepth = 64;
}).AddInteractiveServerComponents();
```

Form names

Use the FormName parameter to assign a form name. Form names must be unique to bind model data. The following form is named `RomulanAle`:

razor

```
<EditForm ... FormName="RomulanAle" ...>
    ...
</EditForm>
```

Supplying a form name:

- Is required for all forms that are submitted by statically-rendered server-side components.
- Isn't required for forms that are submitted by interactively-rendered components, which includes forms in Blazor WebAssembly apps and components with an interactive render mode. However, we recommend supplying a unique form name for every form to prevent runtime form posting errors if interactivity is ever dropped for a form.

The form name is only checked when the form is posted to an endpoint as a traditional HTTP POST request from a statically-rendered server-side component. The framework doesn't throw an exception at the point of rendering a form, but only at the point that an HTTP POST arrives and doesn't specify a form name.

There's an unnamed (empty string) form scope above the app's root component, which suffices when there are no form name collisions in the app. If form name collisions are possible, such as when including a form from a library and you have no control of the form name used by the library's developer, provide a form name scope with the FormMappingScope component in the Blazor Web App's main project.

In the following example, the `HelloFormFromLibrary` component has a form named `Hello` and is in a library.

`HelloFormFromLibrary.razor`:

razor

```
<EditForm FormName="Hello" Model="this" OnSubmit="Submit">
    <InputText @bind-Value="Name" />
    <button type="submit">Submit</button>
</EditForm>

@if (submitted)
{
    <p>Hello @Name from the library's form!</p>
}

@code {
    bool submitted = false;

    [SupplyParameterFromForm]
    private string? Name { get; set; }

    private void Submit() => submitted = true;
}
```

The following `NamedFormsWithScope` component uses the library's `HelloFormFromLibrary` component and also has a form named `Hello`. The FormMappingScope component's scope name is `ParentContext` for any forms supplied by the `HelloFormFromLibrary` component. Although both of the forms in this example have the form name (`Hello`), the form names don't collide and events are routed to the correct form for form POST events.

`NamedFormsWithScope.razor`:

razor

```
@page "/named-forms-with-scope"

<div>Hello form from a library</div>

<FormMappingScope Name="ParentContext">
```

```
    <HelloFormFromLibrary />
</FormMappingScope>

<div>Hello form using the same form name</div>

<EditForm FormName="Hello" Model="this" OnSubmit="Submit">
    <InputText @bind-Value="Name" />
    <button type="submit">Submit</button>
</EditForm>

@if (submitted)
{
    <p>Hello @Name from the app form!</p>
}

@code {
    bool submitted = false;

    [SupplyParameterFromForm]
    private string? Name { get; set; }

    private void Submit() => submitted = true;
}
```

Supply a parameter from the form (`[SupplyParameterFromForm]`)

The `[SupplyParameterFromForm]` attribute indicates that the value of the associated property should be supplied from the form data for the form. Data in the request that matches the name of the property is bound to the property. Inputs based on `InputBase<TValue>` generate form value names that match the names Blazor uses for model binding. Unlike component parameter properties (`[Parameter]`), properties annotated with `[SupplyParameterFromForm]` aren't required to be marked `public`.

You can specify the following form binding parameters to the [SupplyParameterFromForm] attribute:

- Name: Gets or sets the name for the parameter. The name is used to determine the prefix to use to match the form data and decide whether or not the value needs to be bound.
- FormName: Gets or sets the name for the handler. The name is used to match the parameter to the form by form name to decide whether or not the value needs to be bound.

The following example independently binds two forms to their models by form name.

`Starship6.razor`:

razor

```
@page "/starship-6"
@inject ILogger<Starship6> Logger

<EditForm Model="Model1" OnSubmit="Submit1" FormName="Holodeck1">
    <div>
        <label>
            Holodeck 1 Identifier:
            <InputText @bind-Value="Model1!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

<EditForm Model="Model2" OnSubmit="Submit2" FormName="Holodeck2">
    <div>
        <label>
            Holodeck 2 Identifier:
            <InputText @bind-Value="Model2!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm(FormName = "Holodeck1")]
    private Holodeck? Model1 { get; set; }

    [SupplyParameterFromForm(FormName = "Holodeck2")]
    private Holodeck? Model2 { get; set; }

    protected override void OnInitialized()
    {
        Model1 ??= new();
        Model2 ??= new();
    }

    private void Submit1() => Logger.LogInformation("Submit1: Id={Id}",
Model1?.Id);

    private void Submit2() => Logger.LogInformation("Submit2: Id={Id}",
Model2?.Id);

    public class Holodeck
    {
        public string? Id { get; set; }
```

```
    }
}
```

Nest and bind forms

The following guidance demonstrates how to nest and bind child forms.

The following ship details class (`ShipDetails`) holds a description and length for a subform.

`ShipDetails.cs`:

C#

```
namespace BlazorSample;

public class ShipDetails
{
    public string? Description { get; set; }
    public int? Length { get; set; }
}
```

The following `Ship` class names an identifier (`Id`) and includes the ship details.

`Ship.cs`:

C#

```
namespace BlazorSample
{
    public class Ship
    {
        public string? Id { get; set; }
        public ShipDetails Details { get; set; } = new();
    }
}
```

The following subform is used for editing values of the `ShipDetails` type. This is implemented by inheriting Editor<T> at the top of the component. Editor<T> ensures that the child component generates the correct form field names based on the model (`T`), where `T` in the following example is `ShipDetails`.

`StarshipSubform.razor`:

razor

```
@inherits Editor<ShipDetails>

<div>
    <label>
        Description:
        <InputText @bind-Value="Value!.Description" />
    </label>
</div>
<div>
    <label>
        Length:
        <InputNumber @bind-Value="Value!.Length" />
    </label>
</div>
```

The main form is bound to the `Ship` class. The `StarshipSubform` component is used to edit ship details, bound as `Model!.Details`.

`Starship7.razor`:

razor

```
@page "/starship-7"
@inject ILogger<Starship7> Logger

<EditForm Model="Model" OnSubmit="Submit" FormName="Starship7">
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <StarshipSubform @bind-Value="Model!.Details" />
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Ship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() =>
        Logger.LogInformation("Id = {Id} Desc = {Description} Length =
{Length}",
            Model?.Id, Model?.Details?.Description, Model?.Details?.Length);
}
```

Initialize form data with static SSR

When a component adopts static SSR, the OnInitialized{Async} lifecycle method and the OnParametersSet{Async} lifecycle method fire when the component is initially rendered and on every form POST to the server. To initialize form model values, confirm if the model already has data before assigning new model values in `OnParametersSet{Async}`, as the following example demonstrates.

`StarshipInit.razor`:

razor

```razor
@page "/starship-init"
@inject ILogger<StarshipInit> Logger

<EditForm Model="Model" OnValidSubmit="Submit" FormName="StarshipInit">
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    protected override void OnParametersSet()
    {
        if (Model!.Id == default)
        {
            LoadData();
        }
    }

    private void LoadData()
    {
        Model!.Id = "Set by LoadData";
    }

    private void Submit()
    {
        Logger.LogInformation("Id = {Id}", Model?.Id);
    }
```

```
    public class Starship
    {
        public string? Id { get; set; }
    }
}
```

Advanced form mapping error scenarios

The framework instantiates and populates the FormMappingContext for a form, which is the context associated with a given form's mapping operation. Each mapping scope (defined by a FormMappingScope component) instantiates FormMappingContext. Each time a `[SupplyParameterFromForm]` asks the context for a value, the framework populates the FormMappingContext with the attempted value and any mapping errors.

Developers aren't expected to interact with FormMappingContext directly, as it's mainly a source of data for InputBase<TValue>, EditContext, and other internal implementations to show mapping errors as validation errors. In advanced custom scenarios, developers can access FormMappingContext directly as a `[CascadingParameter]` to write custom code that consumes the attempted values and mapping errors.

Custom input components

For custom input processing scenarios, the following subsections demonstrate custom input components:

- Input component based on InputBase<T>: The component inherits from InputBase<TValue>, which provides a base implementation for binding, callbacks, and validation. Components that inherit from InputBase<TValue> must be used in a Blazor form (EditForm).
- Input component with full developer control: The component takes full control of input processing. The component's code must manage binding, callbacks, and validation. The component can be used inside or outside of a Blazor form.

We recommend that you derive your custom input components from InputBase<TValue> unless specific requirements prevent you from doing so. The InputBase<TValue> class is actively maintained by the ASP.NET Core team, ensuring it stays up-to-date with the latest Blazor features and framework changes.

Input component based on `InputBase<T>`

The following example component:

- Inherits from InputBase<TValue>. Components that inherit from InputBase<TValue> must be used in a Blazor form (EditForm).
- Takes boolean input from a checkbox.
- Sets the background color of its container `<div>` based on the checkbox's state, which occurs when the `AfterChange` method executes after binding (`@bind:after`).
- Is required to override the base class's `TryParseValueFromString` method but doesn't process string input data because a checkbox doesn't provide string data. Example implementations of `TryParseValueFromString` for other types of input components that process string input are available in the ASP.NET Core reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

`EngineeringApprovalInputDerived.razor`:

razor

```
@using System.Diagnostics.CodeAnalysis
@inherits InputBase<bool>

<div class="@divCssClass">
    <label>
        Engineering Approval:
        <input @bind="CurrentValue" @bind:after="AfterChange"
class="@CssClass"
            type="checkbox" />
    </label>
</div>

@code {
    private string? divCssClass;

    private void AfterChange()
    {
        divCssClass = CurrentValue ? "bg-success text-white" : null;
    }

    protected override bool TryParseValueFromString(
```

```
        string? value, out bool result,
        [NotNullWhen(false)] out string? validationErrorMessage)
            => throw new NotSupportedException(
                "This component does not parse string inputs. " +
                $"Bind to the '{nameof(CurrentValue)}' property, " +
                $"not '{nameof(CurrentValueAsString)}'.");
}
```

To use the preceding component in the starship example form (Starship3.razor/Starship.cs), replace the `<div>` block for the engineering approval field with an `EngineeringApprovalInputDerived` component instance bound to the model's `IsValidatedDesign` property:

diff

```
- <div>
-     <label>
-         Engineering Approval:
-         <InputCheckbox @bind-Value="Model!.IsValidatedDesign" />
-     </label>
- </div>
+ <EngineeringApprovalInputDerived @bind-Value="Model!.IsValidatedDesign" />
```

If the component that inherits from InputBase<TValue> is ever statically rendered, assign the InputBase<TValue>.NameAttributeValue property to the `name` attribute of `<input>` elements:

razor

```
<input @bind="CurrentValue" @bind:after="AfterChange" class="@CssClass"
    type="checkbox" name="@NameAttributeValue" />
```

The preceding assignment isn't necessary if the component is guaranteed to always render interactively.

Input component with full developer control

The following example component:

- Doesn't inherit from InputBase<TValue>. The component takes full control of input processing, including binding, callbacks, and validation. The component can be used inside or outside of a Blazor form (EditForm).
- Takes boolean input from a checkbox.
- Changes the background color if the checkbox is checked.

Code in the component includes:

- The `Value` property is used with two-way binding to get or set the value of the input. `ValueChanged` is the callback that updates the bound value.

- When used in a Blazor form:
  - The EditContext is a cascading value.
  - `fieldCssClass` styles the field based on the result of EditContext validation.
  - `ValueExpression` is an expression (`Expression<Func<T>>`) assigned by the framework that identifies the bound value.
  - FieldIdentifier uniquely identifies a single field that can be edited, usually corresponding to a model property. The field identifier is created with the expression that identifies the bound value (`ValueExpression`).

- In the `OnChange` event handler:
  - The value of the checkbox input is obtained from InputFileChangeEventArgs.
  - The background color and text color of the container `<div>` element are set.
  - EventCallback.InvokeAsync invokes the delegate associated with the binding and dispatches an event notification to consumers that the value has changed.
  - If the component is used in an EditForm (the `EditContext` property isn't `null`), EditContext.NotifyFieldChanged is called to trigger validation.

`EngineeringApprovalInputStandalone.razor`:

razor

```
@using System.Globalization
@using System.Linq.Expressions

<div class="@divCssClass">
    <label>
        Engineering Approval:
        <input class="@fieldCssClass" @onchange="OnChange" type="checkbox"
            value="@Value" />
    </label>
</div>

@code {
    private string? divCssClass;
    private FieldIdentifier fieldIdentifier;
    private string? fieldCssClass =>
EditContext?.FieldCssClass(fieldIdentifier);

    [CascadingParameter]
    private EditContext? EditContext { get; set; }

    [Parameter]
```

```
    public bool? Value { get; set; }

    [Parameter]
    public EventCallback<bool> ValueChanged { get; set; }

    [Parameter]
    public Expression<Func<bool>>? ValueExpression { get; set; }

    protected override void OnInitialized()
    {
        fieldIdentifier = FieldIdentifier.Create(ValueExpression!);
    }

    private async Task OnChange(ChangeEventArgs args)
    {
        BindConverter.TryConvertToBool(args.Value,
CultureInfo.CurrentCulture,
            out var value);

        divCssClass = value ? "bg-success text-white" : null;

        await ValueChanged.InvokeAsync(value);
        EditContext?.NotifyFieldChanged(fieldIdentifier);
    }
}
```

To use the preceding component in the starship example form (Starship3.razor/Starship.cs), replace the `<div>` block for the engineering approval field with a `EngineeringApprovalInputStandalone` component instance bound to the model's `IsValidatedDesign` property:

diff

```
- <div>
-     <label>
-         Engineering Approval:
-         <InputCheckbox @bind-Value="Model!.IsValidatedDesign" />
-     </label>
- </div>
+ <EngineeringApprovalInputStandalone @bind-Value="Model!.IsValidatedDesign"
/>
```

The `EngineeringApprovalInputStandalone` component is also functional outside of an EditForm:

razor

```
<EngineeringApprovalInputStandalone @bind-Value="ValidDesign" />

<div>
```

```
    <b>ValidDesign:</b> @ValidDesign
</div>

@code {
    private bool ValidDesign { get; set; }
}
```

Radio buttons

The example in this section is based on the `Starfleet Starship Database` form (`Starship3` component) of the Example form section of this article.

Add the following enum types to the app. Create a new file to hold them or add them to the `Starship.cs` file.

C#

```
public class ComponentEnums
{
    public enum Manufacturer { SpaceX, NASA, ULA, VirginGalactic, Unknown }
    public enum Color { ImperialRed, SpacecruiserGreen, StarshipBlue,
VoyagerOrange }
    public enum Engine { Ion, Plasma, Fusion, Warp }
}
```

Make the `ComponentEnums` class accessible to the:

- `Starship` model in `Starship.cs` (for example, `using static ComponentEnums;`).
- `Starfleet Starship Database` form (`Starship3.razor`) (for example, `@using static ComponentEnums`).

Use InputRadio<TValue> components with the InputRadioGroup<TValue> component to create a radio button group. In the following example, properties are added to the `Starship` model described in the Example form section of the *Input components* article:

C#

```
[Required]
[Range(typeof(Manufacturer), nameof(Manufacturer.SpaceX),
    nameof(Manufacturer.VirginGalactic), ErrorMessage = "Pick a
manufacturer.")]
public Manufacturer Manufacturer { get; set; } = Manufacturer.Unknown;

[Required, EnumDataType(typeof(Color))]
public Color? Color { get; set; } = null;
```

```
[Required, EnumDataType(typeof(Engine))]
public Engine? Engine { get; set; } = null;
```

Update the `Starfleet Starship Database` form (`Starship3` component) of the Example form section of the *Input components* article. Add the components to produce:

- A radio button group for the ship manufacturer.
- A nested radio button group for engine and ship color.

> ⓘ Note
>
> Nested radio button groups aren't often used in forms because they can result in a disorganized layout of form controls that may confuse users. However, there are cases when they make sense in UI design, such as in the following example that pairs recommendations for two user inputs, ship engine and ship color. One engine and one color are required by the form's validation. The form's layout uses nested InputRadioGroup<TValue>s to pair engine and color recommendations. However, the user can combine any engine with any color to submit the form.

> ⓘ Note
>
> Be sure to make the `ComponentEnums` class available to the component for the following example:
>
> razor
>
> ```
> @using static ComponentEnums
> ```

razor

```
<fieldset>
    <legend>Manufacturer</legend>
    <InputRadioGroup @bind-Value="Model!.Manufacturer">
        @foreach (var manufacturer in Enum.GetValues<Manufacturer>())
        {
            <div>
                <label>
                    <InputRadio Value="manufacturer" />
                    @manufacturer
                </label>
            </div>
        }
    </InputRadioGroup>
</fieldset>
```

```
<fieldset>
    <legend>Engine and Color</legend>
    <p>
        Engine and color pairs are recommended, but any
        combination of engine and color is allowed.
    </p>
    <InputRadioGroup Name="engine" @bind-Value="Model!.Engine">
        <InputRadioGroup Name="color" @bind-Value="Model!.Color">
            <div style="margin-bottom:5px">
                <div>
                    <label>
                        <InputRadio Name="engine" Value="Engine.Ion" />
                        Ion
                    </label>
                </div>
                <div>
                    <label>
                        <InputRadio Name="color" Value="Color.ImperialRed"
/>
                        Imperial Red
                    </label>
                </div>
            </div>
            <div style="margin-bottom:5px">
                <div>
                    <label>
                        <InputRadio Name="engine" Value="Engine.Plasma" />
                        Plasma
                    </label>
                </div>
                <div>
                    <label>
                        <InputRadio Name="color"
Value="Color.SpacecruiserGreen" />
                        Spacecruiser Green
                    </label>
                </div>
            </div>
            <div style="margin-bottom:5px">
                <div>
                    <label>
                        <InputRadio Name="engine" Value="Engine.Fusion" />
                        Fusion
                    </label>
                </div>
                <div>
                    <label>
                        <InputRadio Name="color" Value="Color.StarshipBlue"
/>
                        Starship Blue
                    </label>
                </div>
            </div>
            <div style="margin-bottom:5px">
```

```
                <div>
                    <label>
                        <InputRadio Name="engine" Value="Engine.Warp" />
                        Warp
                    </label>
                </div>
                <div>
                    <label>
                        <InputRadio Name="color" Value="Color.VoyagerOrange"
/>
                        Voyager Orange
                    </label>
                </div>
            </div>
        </InputRadioGroup>
    </InputRadioGroup>
</fieldset>
```

> **Note**
>
> If `Name` is omitted, InputRadio<TValue> components are grouped by their most recent ancestor.

If you implemented the preceding Razor markup in the `Starship3` component of the Example form section of the *Input components* article, update the logging for the `Submit` method:

C#

```
Logger.LogInformation("Id = {Id} Description = {Description} " +
    "Classification = {Classification} MaximumAccommodation = " +
    "{MaximumAccommodation} IsValidatedDesign = " +
    "{IsValidatedDesign} ProductionDate = {ProductionDate} " +
    "Manufacturer = {Manufacturer}, Engine = {Engine}, " +
    "Color = {Color}",
    Model?.Id, Model?.Description, Model?.Classification,
    Model?.MaximumAccommodation, Model?.IsValidatedDesign,
    Model?.ProductionDate, Model?.Manufacturer, Model?.Engine,
    Model?.Color);
```

ASP.NET Core Blazor forms validation

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use validation in Blazor forms.

Form validation

In basic form validation scenarios, an EditForm instance can use declared EditContext and ValidationMessageStore instances to validate form fields. A handler for the OnValidationRequested event of the EditContext executes custom validation logic. The handler's result updates the ValidationMessageStore instance.

Basic form validation is useful in cases where the form's model is defined within the component hosting the form, either as members directly on the component or in a subclass. Use of a validator component is recommended where an independent model class is used across several components.

In Blazor Web Apps, client-side validation requires an active Blazor SignalR circuit. Client-side validation isn't available to forms in components that have adopted static server-side rendering (static SSR). Forms that adopt static SSR are validated on the server after the form is submitted.

In the following component, the `HandleValidationRequested` handler method clears any existing validation messages by calling ValidationMessageStore.Clear before validating the form.

`Starship8.razor`:

razor

```
@page "/starship-8"
@implements IDisposable
@inject ILogger<Starship8> Logger

<h2>Holodeck Configuration</h2>

<EditForm EditContext="editContext" OnValidSubmit="Submit"
FormName="Starship8">
```

```
    <div>
        <label>
            <InputCheckbox @bind-Value="Model!.Subsystem1" />
            Safety Subsystem
        </label>
    </div>
    <div>
        <label>
            <InputCheckbox @bind-Value="Model!.Subsystem2" />
            Emergency Shutdown Subsystem
        </label>
    </div>
    <div>
        <ValidationMessage For="() => Model!.Options" />
    </div>
    <div>
        <button type="submit">Update</button>
    </div>
</EditForm>

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Holodeck? Model { get; set; }

    private ValidationMessageStore? messageStore;

    protected override void OnInitialized()
    {
        Model ??= new();
        editContext = new(Model);
        editContext.OnValidationRequested += HandleValidationRequested;
        messageStore = new(editContext);
    }

    private void HandleValidationRequested(object? sender,
        ValidationRequestedEventArgs args)
    {
        messageStore?.Clear();

        // Custom validation logic
        if (!Model!.Options)
        {
            messageStore?.Add(() => Model.Options, "Select at least one.");
        }
    }

    private void Submit() => Logger.LogInformation("Submit: Processing
form");

    public class Holodeck
    {
        public bool Subsystem1 { get; set; }
        public bool Subsystem2 { get; set; }
```

```
        public bool Options => Subsystem1 || Subsystem2;
    }

    public void Dispose()
    {
        if (editContext is not null)
        {
            editContext.OnValidationRequested -= HandleValidationRequested;
        }
    }
}
```

Data Annotations Validator component and custom validation

The DataAnnotationsValidator component attaches data annotations validation to a cascaded EditContext. Enabling data annotations validation requires the DataAnnotationsValidator component. To use a different validation system than data annotations, use a custom implementation instead of the DataAnnotationsValidator component. The framework implementations for DataAnnotationsValidator are available for inspection in the reference source:

- DataAnnotationsValidator
- AddDataAnnotationsValidation.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Blazor performs two types of validation:

- *Field validation* is performed when the user tabs out of a field. During field validation, the DataAnnotationsValidator component associates all reported validation results with the field.
- *Model validation* is performed when the user submits the form. During model validation, the DataAnnotationsValidator component attempts to determine the field based on the member name that the validation result reports. Validation

results that aren't associated with an individual member are associated with the model rather than a field.

Validator components

Validator components support form validation by managing a ValidationMessageStore for a form's EditContext.

The Blazor framework provides the DataAnnotationsValidator component to attach validation support to forms based on validation attributes (data annotations). You can create custom validator components to process validation messages for different forms on the same page or the same form at different steps of form processing (for example, client validation followed by server validation). The validator component example shown in this section, `CustomValidation`, is used in the following sections of this article:

- Business logic validation with a validator component
- Server validation with a validator component

Of the data annotation built-in validators, only the [Remote] validation attribute isn't supported in Blazor.

> **Note**
>
> Custom data annotation validation attributes can be used instead of custom validator components in many cases. Custom attributes applied to the form's model activate with the use of the **DataAnnotationsValidator** component. When used with server validation, any custom attributes applied to the model must be executable on the server. For more information, see the **Custom validation attributes** section.

Create a validator component from ComponentBase:

- The form's EditContext is a cascading parameter of the component.
- When the validator component is initialized, a new ValidationMessageStore is created to maintain a current list of form errors.
- The message store receives errors when developer code in the form's component calls the `DisplayErrors` method. The errors are passed to the `DisplayErrors` method in a Dictionary<string, List<string>>. In the dictionary, the key is the name of the form field that has one or more errors. The value is the error list.
- Messages are cleared when any of the following have occurred:

- Validation is requested on the EditContext when the OnValidationRequested event is raised. All of the errors are cleared.
- A field changes in the form when the OnFieldChanged event is raised. Only the errors for the field are cleared.
- The `ClearErrors` method is called by developer code. All of the errors are cleared.

Update the namespace in the following class to match your app's namespace.

`CustomValidation.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.Forms;

namespace BlazorSample;

public class CustomValidation : ComponentBase
{
    private ValidationMessageStore? messageStore;

    [CascadingParameter]
    private EditContext? CurrentEditContext { get; set; }

    protected override void OnInitialized()
    {
        if (CurrentEditContext is null)
        {
            throw new InvalidOperationException(
                $"{nameof(CustomValidation)} requires a cascading " +
                $"parameter of type {nameof(EditContext)}. " +
                $"For example, you can use {nameof(CustomValidation)} " +
                $"inside an {nameof(EditForm)}.");
        }

        messageStore = new(CurrentEditContext);

        CurrentEditContext.OnValidationRequested += (s, e) =>
            messageStore?.Clear();
        CurrentEditContext.OnFieldChanged += (s, e) =>
            messageStore?.Clear(e.FieldIdentifier);
    }

    public void DisplayErrors(Dictionary<string, List<string>> errors)
    {
        if (CurrentEditContext is not null)
        {
            foreach (var err in errors)
            {
                messageStore?.Add(CurrentEditContext.Field(err.Key),
err.Value);
```

```
            }

            CurrentEditContext.NotifyValidationStateChanged();
        }
    }

    public void ClearErrors()
    {
        messageStore?.Clear();
        CurrentEditContext?.NotifyValidationStateChanged();
    }
}
```

> **ⓘ Important**
>
> Specifying a namespace is **required** when deriving from ComponentBase. Failing to specify a namespace results in a build error:
>
> > Tag helpers cannot target tag name '<global namespace>.{CLASS NAME}' because it contains a ' ' character.
>
> The `{CLASS NAME}` placeholder is the name of the component class. The custom validator example in this section specifies the example namespace `BlazorSample`.

> **ⓘ Note**
>
> Anonymous lambda expressions are registered event handlers for OnValidationRequested and OnFieldChanged in the preceding example. It isn't necessary to implement IDisposable and unsubscribe the event delegates in this scenario. For more information, see ASP.NET Core Razor component lifecycle.

Business logic validation with a validator component

For general business logic validation, use a validator component that receives form errors in a dictionary.

Basic validation is useful in cases where the form's model is defined within the component hosting the form, either as members directly on the component or in a subclass. Use of a validator component is recommended where an independent model class is used across several components.

In the following example:

- A shortened version of the `Starfleet Starship Database` form (`Starship3` component) of the Example form section of the *Input components* article is used that only accepts the starship's classification and description. Data annotation validation isn't triggered on form submission because the DataAnnotationsValidator component isn't included in the form.
- The `CustomValidation` component from the Validator components section of this article is used.
- The validation requires a value for the ship's description (`Description`) if the user selects the "`Defense`" ship classification (`Classification`).

When validation messages are set in the component, they're added to the validator's ValidationMessageStore and shown in the EditForm's validation summary.

`Starship9.razor`:

razor

```
@page "/starship-9"
@inject ILogger<Starship9> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship9">
    <CustomValidation @ref="customValidation" />
    <ValidationSummary />
    <div>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="Model!.Classification">
                <option value="">
                    Select classification ...
                </option>
                <option checked="@(Model!.Classification == "Exploration")"
                    value="Exploration">
                    Exploration
                </option>
                <option checked="@(Model!.Classification == "Diplomacy")"
                    value="Diplomacy">
                    Diplomacy
                </option>
                <option checked="@(Model!.Classification == "Defense")"
                    value="Defense">
                    Defense
                </option>
            </InputSelect>
        </label>
```

```
    </div>
    <div>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="Model!.Description" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    private CustomValidation? customValidation;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() =>
        Model ??= new() { ProductionDate = DateTime.UtcNow };

    private void Submit()
    {
        customValidation?.ClearErrors();

        var errors = new Dictionary<string, List<string>>();

        if (Model!.Classification == "Defense" &&
                string.IsNullOrEmpty(Model.Description))
        {
            errors.Add(nameof(Model.Description),
                new() { "For a 'Defense' ship classification, " +
                "'Description' is required." });
        }

        if (errors.Any())
        {
            customValidation?.DisplayErrors(errors);
        }
        else
        {
            Logger.LogInformation("Submit called: Processing the form");
        }
    }
}
```

> **Note**
>
> As an alternative to using validation components, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server

validation, the attributes must be executable on the server. For more information, see the Custom validation attributes section.

Server validation with a validator component

This section is focused on Blazor Web App scenarios, but the approach for any type of app that uses server validation with web API adopts the same general approach.

Server validation is supported in addition to client validation:

- Process client validation in the form with the DataAnnotationsValidator component.
- When the form passes client validation (OnValidSubmit is called), send the EditContext.Model to a backend server API for form processing.
- Process model validation on the server.
- The server API includes both the built-in framework data annotations validation and custom validation logic supplied by the developer. If validation passes on the server, process the form and send back a success status code (200 - OK). If validation fails, return a failure status code (400 - Bad Request) and the field validation errors.
- Either disable the form on success or display the errors.

Basic validation is useful in cases where the form's model is defined within the component hosting the form, either as members directly on the component or in a subclass. Use of a validator component is recommended where an independent model class is used across several components.

The following example is based on:

- A Blazor Web App with Interactive WebAssembly components created from the Blazor Web App project template.
- The `Starship` model (`Starship.cs`) of the Example form section of the *Input components* article.
- The `CustomValidation` component shown in the Validator components section.

Place the `Starship` model (`Starship.cs`) into a shared class library project so that both the client and server projects can use the model. Add or update the namespace to match the namespace of the shared app (for example, `namespace BlazorSample.Shared`). Since the model requires data annotations, confirm that the shared class library uses the shared framework or add the System.ComponentModel.Annotations package to the shared project.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the main project of the Blazor Web App, add a controller to process starship validation requests and return failed validation messages. Update the namespaces in the last `using` statement for the shared class library project and the `namespace` for the controller class. In addition to client and server data annotations validation, the controller validates that a value is provided for the ship's description (`Description`) if the user selects the `Defense` ship classification (`Classification`).

The validation for the `Defense` ship classification only occurs on the server in the controller because the upcoming form doesn't perform the same validation client-side when the form is submitted to the server. Server validation without client validation is common in apps that require private business logic validation of user input on the server. For example, private information from data stored for a user might be required to validate user input. Private data obviously can't be sent to the client for client validation.

> **Note**
>
> The `StarshipValidation` controller in this section uses Microsoft Identity 2.0. The Web API only accepts tokens for users that have the "`API.Access`" scope for this API. Additional customization is required if the API's scope name is different from `API.Access`.
>
> For more information on security, see:
>
> - ASP.NET Core Blazor authentication and authorization (and the other articles in the Blazor *Security and Identity* node)
> - Microsoft identity platform documentation

`Controllers/StarshipValidation.cs`:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using BlazorSample.Shared;
```

```
namespace BlazorSample.Server.Controllers;

[Authorize]
[ApiController]
[Route("[controller]")]
public class StarshipValidationController(
    ILogger<StarshipValidationController> logger)
    : ControllerBase
{
    static readonly string[] scopeRequiredByApi = new[] { "API.Access" };

    [HttpPost]
    public async Task<IActionResult> Post(Starship model)
    {
        HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);

        try
        {
            if (model.Classification == "Defense" &&
                string.IsNullOrEmpty(model.Description))
            {
                ModelState.AddModelError(nameof(model.Description),
                    "For a 'Defense' ship " +
                    "classification, 'Description' is required.");
            }
            else
            {
                logger.LogInformation("Processing the form asynchronously");

                // async ...

                return Ok(ModelState);
            }
        }
        catch (Exception ex)
        {
            logger.LogError("Validation Error: {Message}", ex.Message);
        }

        return BadRequest(ModelState);
    }
}
```

Confirm or update the namespace of the preceding controller (`BlazorSample.Server.Controllers`) to match the app's controllers' namespace.

When a model binding validation error occurs on the server, an ApiController (ApiControllerAttribute) normally returns a default bad request response with a ValidationProblemDetails. The response contains more data than just the validation

errors, as shown in the following example when all of the fields of the `Starfleet Starship Database` form aren't submitted and the form fails validation:

JSON

```
{
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Id": ["The Id field is required."],
    "Classification": ["The Classification field is required."],
    "IsValidatedDesign": ["This form disallows unapproved ships."],
    "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
  }
}
```

> **Note**
>
> To demonstrate the preceding JSON response, you must either disable the form's client validation to permit empty field form submission or use a tool to send a request directly to the server API, such as Firefox Browser Developer.

If the server API returns the preceding default JSON response, it's possible for the client to parse the response in developer code to obtain the children of the `errors` node for forms validation error processing. It's inconvenient to write developer code to parse the file. Parsing the JSON manually requires producing a Dictionary<string, List<string>> of errors after calling ReadFromJsonAsync. Ideally, the server API should only return the validation errors, as the following example shows:

JSON

```
{
  "Id": ["The Id field is required."],
  "Classification": ["The Classification field is required."],
  "IsValidatedDesign": ["This form disallows unapproved ships."],
  "MaximumAccommodation": ["Accommodation invalid (1-100000)."]
}
```

To modify the server API's response to make it only return the validation errors, change the delegate that's invoked on actions that are annotated with ApiControllerAttribute in the `Program` file. For the API endpoint (`/StarshipValidation`), return a BadRequestObjectResult with the ModelStateDictionary. For any other API endpoints, preserve the default behavior by returning the object result with a new ValidationProblemDetails.

Add the Microsoft.AspNetCore.Mvc namespace to the top of the `Program` file in the main project of the Blazor Web App:

C#

```
using Microsoft.AspNetCore.Mvc;
```

In the `Program` file, add or update the following AddControllersWithViews extension method and add the following call to ConfigureApiBehaviorOptions:

C#

```
builder.Services.AddControllersWithViews()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            if (context.HttpContext.Request.Path == "/StarshipValidation")
            {
                return new BadRequestObjectResult(context.ModelState);
            }
            else
            {
                return new BadRequestObjectResult(
                    new ValidationProblemDetails(context.ModelState));
            }
        };
    });
```

If you're adding controllers to the main project of the Blazor Web App for the first time, map controller endpoints when you place the preceding code that registers services for controllers. The following example uses default controller routes:

C#

```
app.MapDefaultControllerRoute();
```

> ⓘ Note
>
> The preceding example explicitly registers controller services by calling AddControllersWithViews to automatically mitigate Cross-Site Request Forgery (XSRF/CSRF) attacks. If you merely use AddControllers, antiforgery isn't enabled automatically.

For more information on controller routing and validation failure error responses, see the following resources:

- Routing to controller actions in ASP.NET Core
- Handle errors in ASP.NET Core controller-based web APIs

In the `.Client` project, add the `CustomValidation` component shown in the Validator components section. Update the namespace to match the app (for example, `namespace BlazorSample.Client`).

In the `.Client` project, the `Starfleet Starship Database` form is updated to show server validation errors with help of the `CustomValidation` component. When the server API returns validation messages, they're added to the `CustomValidation` component's ValidationMessageStore. The errors are available in the form's EditContext for display by the form's validation summary.

In the following component, update the namespace of the shared project (`@using BlazorSample.Shared`) to the shared project's namespace. Note that the form requires authorization, so the user must be signed into the app to navigate to the form.

`Starship10.razor`:

> **Note**
>
> Forms based on EditForm automatically enable antiforgery support. The controller should use AddControllersWithViews to register controller services and automatically enable antiforgery support for the web API.

razor

```razor
@page "/starship-10"
@rendermode InteractiveWebAssembly
@using System.Net
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using BlazorSample.Shared
@attribute [Authorize]
@inject HttpClient Http
@inject ILogger<Starship10> Logger

<h1>Starfleet Starship Database</h1>

<h2>New Ship Entry Form</h2>

<EditForm FormName="Starship10" Model="Model" OnValidSubmit="Submit">
```

```
    <DataAnnotationsValidator />
    <CustomValidation @ref="customValidation" />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" disabled="@disabled" />
        </label>
    </div>
    <div>
        <label>
            Description (optional):
            <InputTextArea @bind-Value="Model!.Description"
                disabled="@disabled" />
        </label>
    </div>
    <div>
        <label>
            Primary Classification:
            <InputSelect @bind-Value="Model!.Classification"
disabled="@disabled">
                <option value="">Select classification ...</option>
                <option value="Exploration">Exploration</option>
                <option value="Diplomacy">Diplomacy</option>
                <option value="Defense">Defense</option>
            </InputSelect>
        </label>
    </div>
    <div>
        <label>
            Maximum Accommodation:
            <InputNumber @bind-Value="Model!.MaximumAccommodation"
                disabled="@disabled" />
        </label>
    </div>
    <div>
        <label>
            Engineering Approval:
            <InputCheckbox @bind-Value="Model!.IsValidatedDesign"
                disabled="@disabled" />
        </label>
    </div>
    <div>
        <label>
            Production Date:
            <InputDate @bind-Value="Model!.ProductionDate"
disabled="@disabled" />
        </label>
    </div>
    <div>
        <button type="submit" disabled="@disabled">Submit</button>
    </div>
    <div style="@messageStyles">
        @message
    </div>
```

```
</EditForm>

@code {
    private CustomValidation? customValidation;
    private bool disabled;
    private string? message;
    private string messageStyles = "visibility:hidden";

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() =>
        Model ??= new() { ProductionDate = DateTime.UtcNow };

    private async Task Submit(EditContext editContext)
    {
        customValidation?.ClearErrors();

        try
        {
            var response = await Http.PostAsJsonAsync<Starship>(
                "StarshipValidation", (Starship)editContext.Model);

            var errors = await response.Content
                .ReadFromJsonAsync<Dictionary<string, List<string>>>() ??
                new Dictionary<string, List<string>>();

            if (response.StatusCode == HttpStatusCode.BadRequest &&
                errors.Any())
            {
                customValidation?.DisplayErrors(errors);
            }
            else if (!response.IsSuccessStatusCode)
            {
                throw new HttpRequestException(
                    $"Validation failed. Status Code:
{response.StatusCode}");
            }
            else
            {
                disabled = true;
                messageStyles = "color:green";
                message = "The form has been processed.";
            }
        }
        catch (AccessTokenNotAvailableException ex)
        {
            ex.Redirect();
        }
        catch (Exception ex)
        {
            Logger.LogError("Form processing error: {Message}", ex.Message);
            disabled = true;
            messageStyles = "color:red";
            message = "There was an error processing the form.";
```

```
        }
    }
}
```

The `.Client` project of a Blazor Web App must also register an HttpClient for HTTP POST requests to a backend web API controller. Confirm or add the following to the `.Client` project's `Program` file:

C#

```
builder.Services.AddScoped(sp =>
    new HttpClient { BaseAddress = new
Uri(builder.HostEnvironment.BaseAddress) });
```

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the app and is typically derived from the `<base>` tag's `href` value in the host page.

> **Note**
>
> As an alternative to the use of a validation component, data annotation validation attributes can be used. Custom attributes applied to the form's model activate with the use of the DataAnnotationsValidator component. When used with server validation, the attributes must be executable on the server. For more information, see the Custom validation attributes section.

`InputText` based on the input event

Use the InputText component to create a custom component that uses the `oninput` event (input) instead of the `onchange` event (change). Use of the `input` event triggers field validation on each keystroke.

The following `CustomInputText` component inherits the framework's `InputText` component and sets event binding to the `oninput` event (input).

`CustomInputText.razor`:

razor

```
@inherits InputText
```

```
<input @attributes="AdditionalAttributes"
       class="@CssClass"
       @bind="CurrentValueAsString"
       @bind:event="oninput" />
```

The `CustomInputText` component can be used anywhere InputText is used. The following component uses the shared `CustomInputText` component.

`Starship11.razor`:

razor

```
@page "/starship-11"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship11> Logger

<EditForm Model="Model" OnValidSubmit="Submit" FormName="Starship11">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <CustomInputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

<div>
    CurrentValue: @Model?.Id
</div>

@code {
    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized() => Model ??= new();

    private void Submit() => Logger.LogInformation("Submit: Processing
form");

    public class Starship
    {
        [Required]
        [StringLength(10, ErrorMessage = "Id is too long.")]
        public string? Id { get; set; }
    }
}
```

Validation Summary and Validation Message components

The ValidationSummary component summarizes all validation messages, which is similar to the Validation Summary Tag Helper:

```razor
<ValidationSummary />
```

Output validation messages for a specific model with the `Model` parameter:

```razor
<ValidationSummary Model="Model" />
```

The ValidationMessage<TValue> component displays validation messages for a specific field, which is similar to the Validation Message Tag Helper. Specify the field for validation with the For attribute and a lambda expression naming the model property:

```razor
<ValidationMessage For="@(() => Model!.MaximumAccommodation)" />
```

The ValidationMessage<TValue> and ValidationSummary components support arbitrary attributes. Any attribute that doesn't match a component parameter is added to the generated `<div>` or `<ul>` element.

Control the style of validation messages in the app's stylesheet (`wwwroot/css/app.css` or `wwwroot/css/site.css`). The default `validation-message` class sets the text color of validation messages to red:

```css
.validation-message {
    color: red;
}
```

Determine if a form field is valid

Use EditContext.IsValid to determine if a field is valid without obtaining validation messages.

❌ Supported, but not recommended:

```
C#

var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();
```

✔ Recommended:

```
C#

var isValid = editContext.IsValid(fieldIdentifier);
```

Custom validation attributes

To ensure that a validation result is correctly associated with a field when using a custom validation attribute, pass the validation context's MemberName when creating the ValidationResult.

`CustomValidator.cs`:

```
C#

using System;
using System.ComponentModel.DataAnnotations;

public class CustomValidator : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        ...

        return new ValidationResult("Validation message to user.",
            new[] { validationContext.MemberName });
    }
}
```

Inject services into custom validation attributes through the ValidationContext. The following example demonstrates a salad chef form that validates user input with dependency injection (DI).

The `SaladChef` class indicates the approved starship ingredient list for a Ten Forward salad.

`SaladChef.cs`:

```
C#

namespace BlazorSample;

public class SaladChef
{
    public string[] SaladToppers = { "Horva", "Kanda Root", "Krintar",
"Plomeek",
        "Syto Bean" };
}
```

Register `SaladChef` in the app's DI container in the `Program` file:

```
C#

builder.Services.AddTransient<SaladChef>();
```

The `IsValid` method of the following `SaladChefValidatorAttribute` class obtains the `SaladChef` service from DI to check the user's input.

`SaladChefValidatorAttribute.cs`:

```
C#

using System.ComponentModel.DataAnnotations;

namespace BlazorSample;

public class SaladChefValidatorAttribute : ValidationAttribute
{
    protected override ValidationResult? IsValid(object? value,
        ValidationContext validationContext)
    {
        var saladChef = validationContext.GetRequiredService<SaladChef>();

        if (saladChef.SaladToppers.Contains(value?.ToString()))
        {
            return ValidationResult.Success;
        }

        return new ValidationResult("Is that a Vulcan salad topper?! " +
            "The following toppers are available for a Ten Forward salad: "
+
            string.Join(", ", saladChef.SaladToppers));
```

```
    }
}
```

The following component validates user input by applying the `SaladChefValidatorAttribute` (`[SaladChefValidator]`) to the salad ingredient string (`SaladIngredient`).

`Starship12.razor`:

razor

```razor
@page "/starship-12"
@inject SaladChef SaladChef

<EditForm Model="this" autocomplete="off" FormName="Starship12">
    <DataAnnotationsValidator />
    <div>
        <label>
            Salad topper (@saladToppers):
            <input @bind="SaladIngredient" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
    <ul>
        @foreach (var message in context.GetValidationMessages())
        {
            <li class="validation-message">@message</li>
        }
    </ul>
</EditForm>

@code {
    private string? saladToppers;

    [SaladChefValidator]
    public string? SaladIngredient { get; set; }

    protected override void OnInitialized() =>
        saladToppers ??= string.Join(", ", SaladChef.SaladToppers);
}
```

Custom validation CSS class attributes

Custom validation CSS class attributes are useful when integrating with CSS frameworks, such as Bootstrap.

To specify custom validation CSS class attributes, start by providing CSS styles for custom validation. In the following example, valid (`validField`) and invalid (`invalidField`) styles are specified.

Add the following CSS classes to the app's stylesheet:

```css
.validField {
    border-color: lawngreen;
}

.invalidField {
    background-color: tomato;
}
```

Create a class derived from FieldCssClassProvider that checks for field validation messages and applies the appropriate valid or invalid style.

`CustomFieldClassProvider.cs`:

```C#
using Microsoft.AspNetCore.Components.Forms;

public class CustomFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = editContext.IsValid(fieldIdentifier);

        return isValid ? "validField" : "invalidField";
    }
}
```

Set the `CustomFieldClassProvider` class as the Field CSS Class Provider on the form's EditContext instance with SetFieldCssClassProvider.

`Starship13.razor`:

```razor
@page "/starship-13"
@using System.ComponentModel.DataAnnotations
@inject ILogger<Starship13> Logger

<EditForm EditContext="editContext" OnValidSubmit="Submit"
```

```
FormName="Starship13">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

@code {
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??= new();
        editContext = new(Model);
        editContext.SetFieldCssClassProvider(new
CustomFieldClassProvider());
    }

    private void Submit() => Logger.LogInformation("Submit: Processing
form");

    public class Starship
    {
        [Required]
        [StringLength(10, ErrorMessage = "Id is too long.")]
        public string? Id { get; set; }
    }
}
```

The preceding example checks the validity of all form fields and applies a style to each field. If the form should only apply custom styles to a subset of the fields, make `CustomFieldClassProvider` apply styles conditionally. The following `CustomFieldClassProvider2` example only applies a style to the `Name` field. For any fields with names not matching `Name`, `string.Empty` is returned, and no style is applied. Using reflection, the field is matched to the model member's property or field name, not an `id` assigned to the HTML entity.

`CustomFieldClassProvider2.cs`:

```
C#
```

```
using Microsoft.AspNetCore.Components.Forms;

public class CustomFieldClassProvider2 : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        if (fieldIdentifier.FieldName == "Name")
        {
            var isValid = editContext.IsValid(fieldIdentifier);

            return isValid ? "validField" : "invalidField";
        }

        return string.Empty;
    }
}
```

> **Note**
>
> Matching the field name in the preceding example is case sensitive, so a model property member designated "`Name`" must match a conditional check on "`Name`":
>
> - ✔ `fieldId.FieldName == "Name"`
> - ✘ `fieldId.FieldName == "name"`
> - ✘ `fieldId.FieldName == "NAME"`
> - ✘ `fieldId.FieldName == "nAmE"`

Add an additional property to `Model`, for example:

C#

```
[StringLength(10, ErrorMessage = "Description is too long.")]
public string? Description { get; set; }
```

Add the `Description` to the `CustomValidationForm` component's form:

razor

```
<InputText @bind-Value="Model!.Description" />
```

Update the EditContext instance in the component's `OnInitialized` method to use the new Field CSS Class Provider:

C#

```
editContext?.SetFieldCssClassProvider(new CustomFieldClassProvider2());
```

Because a CSS validation class isn't applied to the `Description` field, it isn't styled. However, field validation runs normally. If more than 10 characters are provided, the validation summary indicates the error:

> Description is too long.

In the following example:

- The custom CSS style is applied to the `Name` field.
- Any other fields apply logic similar to Blazor's default logic and using Blazor's default field CSS validation styles, `modified` with `valid` or `invalid`. Note that for the default styles, you don't need to add them to the app's stylesheet if the app is based on a Blazor project template. For apps not based on a Blazor project template, the default styles can be added to the app's stylesheet:

  CSS

  ```
  .valid.modified:not([type=checkbox]) {
      outline: 1px solid #26b050;
  }

  .invalid {
      outline: 1px solid red;
  }
  ```

`CustomFieldClassProvider3.cs`:

C#

```
using Microsoft.AspNetCore.Components.Forms;

public class CustomFieldClassProvider3 : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext,
        in FieldIdentifier fieldIdentifier)
    {
        var isValid = editContext.IsValid(fieldIdentifier);

        if (fieldIdentifier.FieldName == "Name")
        {
            return isValid ? "validField" : "invalidField";
        }
```

```
        else
        {
            if (editContext.IsModified(fieldIdentifier))
            {
                return isValid ? "modified valid" : "modified invalid";
            }
            else
            {
                return isValid ? "valid" : "invalid";
            }
        }
    }
}
```

Update the EditContext instance in the component's `OnInitialized` method to use the preceding Field CSS Class Provider:

C#

```
editContext.SetFieldCssClassProvider(new CustomFieldClassProvider3());
```

Using `CustomFieldClassProvider3`:

- The `Name` field uses the app's custom validation CSS styles.
- The `Description` field uses logic similar to Blazor's logic and Blazor's default field CSS validation styles.

Class-level validation with `IValidatableObject`

Class-level validation with IValidatableObject (API documentation) is supported for Blazor form models. IValidatableObject validation only executes when the form is submitted and only if all other validation succeeds.

Blazor data annotations validation package

The Microsoft.AspNetCore.Components.DataAnnotations.Validation is a package that fills validation experience gaps using the DataAnnotationsValidator component. The package is currently *experimental*.

> ⚠ **Warning**
>
> The Microsoft.AspNetCore.Components.DataAnnotations.Validation package has a latest version of *release candidate* at NuGet.org. Continue to use the

experimental release candidate package at this time. Experimental features are provided for the purpose of exploring feature viability and may not ship in a stable version. Watch the Announcements GitHub repository, the dotnet/aspnetcore GitHub repository, or this topic section for further updates.

Nested models, collection types, and complex types

Blazor provides support for validating form input using data annotations with the built-in DataAnnotationsValidator. However, the DataAnnotationsValidator only validates top-level properties of the model bound to the form that aren't collection- or complex-type properties.

To validate the bound model's entire object graph, including collection- and complex-type properties, use the `ObjectGraphDataAnnotationsValidator` provided by the *experimental* Microsoft.AspNetCore.Components.DataAnnotations.Validation package:

razor

```
<EditForm ...>
    <ObjectGraphDataAnnotationsValidator />
    ...
</EditForm>
```

Annotate model properties with `[ValidateComplexType]`. In the following model classes, the `ShipDescription` class contains additional data annotations to validate when the model is bound to the form:

`Starship.cs`:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

public class Starship
{
    ...

    [ValidateComplexType]
    public ShipDescription ShipDescription { get; set; } = new();

    ...
}
```

`ShipDescription.cs`:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

public class ShipDescription
{
    [Required]
    [StringLength(40, ErrorMessage = "Description too long (40 char).")]
    public string? ShortDescription { get; set; }

    [Required]
    [StringLength(240, ErrorMessage = "Description too long (240 char).")]
    public string? LongDescription { get; set; }
}
```

Enable the submit button based on form validation

To enable and disable the submit button based on form validation, the following example:

- Uses a shortened version of the earlier `Starfleet Starship Database` form (`Starship3` component) of the Example form section of the *Input components* article that only accepts a value for the ship's Id. The other `Starship` properties receive valid default values when an instance of the `Starship` type is created.
- Uses the form's EditContext to assign the model when the component is initialized.
- Validates the form in the context's OnFieldChanged callback to enable and disable the submit button.
- Implements IDisposable and unsubscribes the event handler in the `Dispose` method. For more information, see ASP.NET Core Razor component lifecycle.

> **Note**
>
> When assigning to the EditForm.EditContext, don't also assign an EditForm.Model to the EditForm.

`Starship14.razor`:

razor

```
@page "/starship-14"
@implements IDisposable
@inject ILogger<Starship14> Logger

<EditForm EditContext="editContext" OnValidSubmit="Submit"
FormName="Starship14">
    <DataAnnotationsValidator />
    <ValidationSummary />
    <div>
        <label>
            Identifier:
            <InputText @bind-Value="Model!.Id" />
        </label>
    </div>
    <div>
        <button type="submit" disabled="@formInvalid">Submit</button>
    </div>
</EditForm>

@code {
    private bool formInvalid = false;
    private EditContext? editContext;

    [SupplyParameterFromForm]
    private Starship? Model { get; set; }

    protected override void OnInitialized()
    {
        Model ??=
            new()
                {
                    Id = "NCC-1701",
                    Classification = "Exploration",
                    MaximumAccommodation = 150,
                    IsValidatedDesign = true,
                    ProductionDate = new DateTime(2245, 4, 11)
                };
        editContext = new(Model);
        editContext.OnFieldChanged += HandleFieldChanged;
    }

    private void HandleFieldChanged(object? sender, FieldChangedEventArgs e)
    {
        if (editContext is not null)
        {
            formInvalid = !editContext.Validate();
            StateHasChanged();
        }
    }

    private void Submit() => Logger.LogInformation("Submit: Processing
form");

    public void Dispose()
```

```
    {
        if (editContext is not null)
        {
            editContext.OnFieldChanged -= HandleFieldChanged;
        }
    }
}
```

If a form isn't preloaded with valid values and you wish to disable the `Submit` button on form load, set `formInvalid` to `true`.

A side effect of the preceding approach is that a validation summary (ValidationSummary component) is populated with invalid fields after the user interacts with any one field. Address this scenario in either of the following ways:

- Don't use a ValidationSummary component on the form.
- Make the ValidationSummary component visible when the submit button is selected (for example, in a `Submit` method).

razor

```
<EditForm ... EditContext="editContext" OnValidSubmit="Submit" ...>
    <DataAnnotationsValidator />
    <ValidationSummary style="@displaySummary" />

    ...

    <button type="submit" disabled="@formInvalid">Submit</button>
</EditForm>

@code {
    private string displaySummary = "display:none";

    ...

    private void Submit()
    {
        displaySummary = "display:block";
    }
}
```

Troubleshoot ASP.NET Core Blazor forms

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article provides troubleshooting guidance for Blazor forms.

Large form payloads and the SignalR message size limit

This section only applies to Blazor Web Apps, Blazor Server apps, and hosted Blazor WebAssembly solutions that implement SignalR.

If form processing fails because the component's form payload has exceeded the maximum incoming SignalR message size permitted for hub methods, the form can adopt streaming JS interop without increasing the message size limit. For more information on the size limit and the error thrown, see ASP.NET Core Blazor SignalR guidance.

In the following example a text area (`<textarea>`) is used with streaming JS interop to move up to 50,000 bytes of data to the server.

Add a JavaScript (JS) `getText` function to the app:

```
window.getText = (elem) => {
  const textValue = elem.value;
  const utf8Encoder = new TextEncoder();
  const encodedTextValue = utf8Encoder.encode(textValue);
  return encodedTextValue;
};
```

For information on where to place JS in a Blazor app, see JavaScript location in ASP.NET Core Blazor apps.

Due to security considerations, zero-length streams aren't permitted for streaming JS Interop. Therefore, the following `StreamFormData` component traps a JSException and returns an empty string if the text area is blank when the form is submitted.

`StreamFormData.razor`:

```
@page "/stream-form-data"
@inject IJSRuntime JS
@inject ILogger<StreamFormData> Logger

<h1>Stream form data with JS interop</h1>

<EditForm FormName="StreamFormData" Model="this" OnSubmit="Submit">
    <div>
        <label>
            &lt;textarea&gt; value streamed for assignment to
            <code>TextAreaValue (&lt;= 50,000 characters)</code>:
            <textarea @ref="largeTextArea" />
        </label>
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</EditForm>

<div>
    Length: @TextAreaValue?.Length
</div>

@code {
    private ElementReference largeTextArea;

    public string? TextAreaValue { get; set; }

    protected override void OnInitialized() =>
        TextAreaValue ??= string.Empty;

    private async Task Submit()
    {
        TextAreaValue = await GetTextAsync();

        Logger.LogInformation("TextAreaValue length: {Length}",
            TextAreaValue.Length);
    }

    public async Task<string> GetTextAsync()
    {
        try
        {
            var streamRef =
                await JS.InvokeAsync<IJSStreamReference>("getText",
largeTextArea);
            var stream = await streamRef.OpenReadStreamAsync(maxAllowedSize:
50_000);
            var streamReader = new StreamReader(stream);

            return await streamReader.ReadToEndAsync();
        }
```

```
        catch (JSException jsException)
        {
            if (jsException.InnerException is
                    ArgumentOutOfRangeException outOfRangeException &&
                outOfRangeException.ActualValue is not null &&
                outOfRangeException.ActualValue is long actualLength &&
                actualLength == 0)
            {
                return string.Empty;
            }

            throw;
        }
    }
}
```

EditForm parameter error

> InvalidOperationException: EditForm requires a Model parameter, or an EditContext parameter, but not both.

Confirm that the EditForm assigns a Model **or** an EditContext. Don't use both for the same form.

When assigning to Model, confirm that the model type is instantiated.

Connection disconnected

> Error: Connection disconnected with error 'Error: Server returned an error on close: Connection closed with an error.'.

> System.IO.InvalidDataException: The maximum message size of 32768B was exceeded. The message size can be configured in AddHubOptions.

For more information and guidance, see the following resources:

- Large form payloads and the SignalR message size limit
- ASP.NET Core Blazor SignalR guidance

ASP.NET Core Blazor file uploads

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to upload files in Blazor with the InputFile component.

File uploads

> **Warning**
>
> Always follow security best practices when permitting users to upload files. For more information, see Upload files in ASP.NET Core.

Use the InputFile component to read browser file data into .NET code. The InputFile component renders an HTML `<input>` element of type `file` for single file uploads. Add the `multiple` attribute to permit the user to upload multiple files at once.

File selection isn't cumulative when using an InputFile component or its underlying HTML <input type="file">, so you can't add files to an existing file selection. The component always replaces the user's initial file selection, so file references from prior selections aren't available.

The following InputFile component executes the `LoadFiles` method when the OnChange (change) event occurs. An InputFileChangeEventArgs provides access to the selected file list and details about each file:

razor

```
<InputFile OnChange="LoadFiles" multiple />

@code {
    private void LoadFiles(InputFileChangeEventArgs e)
    {
        ...
    }
}
```

Rendered HTML:

```HTML
<input multiple="" type="file" _bl_2="">
```

> **Note**
>
> In the preceding example, the `<input>` element's `_bl_2` attribute is used for Blazor's internal processing.

To read data from a user-selected file, call IBrowserFile.OpenReadStream on the file and read from the returned stream. For more information, see the File streams section.

OpenReadStream enforces a maximum size in bytes of its Stream. Reading one file or multiple files larger than 500 KB results in an exception. This limit prevents developers from accidentally reading large files into memory. The `maxAllowedSize` parameter of OpenReadStream can be used to specify a larger size if required.

If you need access to a Stream that represents the file's bytes, use IBrowserFile.OpenReadStream. Avoid reading the incoming file stream directly into memory all at once. For example, don't copy all of the file's bytes into a MemoryStream or read the entire stream into a byte array all at once. These approaches can result in degraded app performance and potential Denial of Service (DoS) risk, especially for server-side components. Instead, consider adopting either of the following approaches:

- Copy the stream directly to a file on disk without reading it into memory. Note that Blazor apps executing code on the server aren't able to access the client's file system directly.
- Upload files from the client directly to an external service. For more information, see the Upload files to an external service section.

In the following examples, `browserFile` represents the uploaded file and implements IBrowserFile. Working implementations for IBrowserFile are shown in the file upload components later in this article.

✔ The following approach is **recommended** because the file's Stream is provided directly to the consumer, a FileStream that creates the file at the provided path:

```C#
await using FileStream fs = new(path, FileMode.Create);
```

```
await browserFile.OpenReadStream().CopyToAsync(fs);
```

✔ The following approach is **recommended** for Microsoft Azure Blob Storage because the file's Stream is provided directly to UploadBlobAsync:

```
C#

await blobContainerClient.UploadBlobAsync(
    trustedFileName, browserFile.OpenReadStream());
```

✘ The following approach is **NOT recommended** because the file's Stream content is read into a String in memory (`reader`):

```
C#

var reader =
    await new StreamReader(browserFile.OpenReadStream()).ReadToEndAsync();
```

✘ The following approach is **NOT recommended** for Microsoft Azure Blob Storage because the file's Stream content is copied into a MemoryStream in memory (`memoryStream`) before calling UploadBlobAsync:

```
C#

var memoryStream = new MemoryStream();
await browserFile.OpenReadStream().CopyToAsync(memoryStream);
await blobContainerClient.UploadBlobAsync(
    trustedFileName, memoryStream));
```

A component that receives an image file can call the BrowserFileExtensions.RequestImageFileAsync convenience method on the file to resize the image data within the browser's JavaScript runtime before the image is streamed into the app. Use cases for calling RequestImageFileAsync are most appropriate for Blazor WebAssembly apps.

Autofac Inversion of Control (IoC) container users

If you're using the Autofac Inversion of Control (IoC) container instead of the built-in ASP.NET Core dependency injection container, set DisableImplicitFromServicesParameters to `true` in the server-side circuit handler hub

options. For more information, see FileUpload: Did not receive any data in the allotted time (dotnet/aspnetcore #38842).

File size read and upload limits

Server-side or client-side, there's no file read or upload size limit specifically for the InputFile component. However, client-side Blazor reads the file's bytes into a single JavaScript array buffer when marshalling the data from JavaScript to C#, which is limited to 2 GB or to the device's available memory. Large file uploads (> 250 MB) may fail for client-side uploads using the InputFile component. For more information, see the following discussions:

- The Blazor InputFile Component should handle chunking when the file is uploaded (dotnet/runtime #84685)
- Request Streaming upload via http handler (dotnet/runtime #36634)

For large client-side file uploads that fail when attempting to use the InputFile component, we recommend chunking large files with a custom component using multiple HTTP range requests instead of using the InputFile component.

Work is currently scheduled for .NET 9 (late 2024) to address the client-side file size upload limitation.

Examples

The following examples demonstrate multiple file upload in a component. InputFileChangeEventArgs.GetMultipleFiles allows reading multiple files. Specify the maximum number of files to prevent a malicious user from uploading a larger number of files than the app expects. InputFileChangeEventArgs.File allows reading the first and only file if the file upload doesn't support multiple files.

InputFileChangeEventArgs is in the Microsoft.AspNetCore.Components.Forms namespace, which is typically one of the namespaces in the app's `_Imports.razor` file. When the namespace is present in the `_Imports.razor` file, it provides API member access to the app's components.

Namespaces in the `_Imports.razor` file aren't applied to C# files (`.cs`). C# files require an explicit using directive at the top of the class file:

```
razor
```

```
using Microsoft.AspNetCore.Components.Forms;
```

For testing file upload components, you can create test files of any size with PowerShell:

PowerShell

```
$out = new-object byte[] {SIZE}; (new-object Random).NextBytes($out);
[IO.File]::WriteAllBytes('{PATH}', $out)
```

In the preceding command:

- The `{SIZE}` placeholder is the size of the file in bytes (for example, `2097152` for a 2 MB file).
- The `{PATH}` placeholder is the path and file with file extension (for example, `D:/test_files/testfile2MB.txt`).

Server-side file upload example

To use the following code, create a `Development/unsafe_uploads` folder at the root of the app running in the `Development` environment.

Because the example uses the app's environment as part of the path where files are saved, additional folders are required if other environments are used in testing and production. For example, create a `Staging/unsafe_uploads` folder for the `Staging` environment. Create a `Production/unsafe_uploads` folder for the `Production` environment.

> **⚠ Warning**
>
> The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see Upload files in ASP.NET Core.

`FileUpload1.razor`:

razor

```
@page "/file-upload-1"
@using System
```

```
@using System.IO
@using Microsoft.AspNetCore.Hosting
@inject ILogger<FileUpload1> Logger
@inject IWebHostEnvironment Environment

<PageTitle>File Upload 1</PageTitle>

<h1>File Upload Example 1</h1>

<p>
    <label>
        Max file size:
        <input type="number" @bind="maxFileSize" />
    </label>
</p>

<p>
    <label>
        Max allowed files:
        <input type="number" @bind="maxAllowedFiles" />
    </label>
</p>

<p>
    <label>
        Upload up to @maxAllowedFiles of up to @maxFileSize bytes:
        <InputFile OnChange="LoadFiles" multiple />
    </label>
</p>

@if (isLoading)
{
    <p>Uploading...</p>
}
else
{
    <ul>
        @foreach (var file in loadedFiles)
        {
            <li>
                <ul>
                    <li>Name: @file.Name</li>
                    <li>Last modified: @file.LastModified.ToString()</li>
                    <li>Size (bytes): @file.Size</li>
                    <li>Content type: @file.ContentType</li>
                </ul>
            </li>
        }
    </ul>
}

@code {
    private List<IBrowserFile> loadedFiles = new();
    private long maxFileSize = 1024 * 15;
    private int maxAllowedFiles = 3;
```

```
    private bool isLoading;

    private async Task LoadFiles(InputFileChangeEventArgs e)
    {
        isLoading = true;
        loadedFiles.Clear();

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            try
            {
                var trustedFileName = Path.GetRandomFileName();
                var path = Path.Combine(Environment.ContentRootPath,
                    Environment.EnvironmentName, "unsafe_uploads",
                    trustedFileName);

                await using FileStream fs = new(path, FileMode.Create);
                await file.OpenReadStream(maxFileSize).CopyToAsync(fs);

                loadedFiles.Add(file);

                Logger.LogInformation(
                    "Unsafe Filename: {UnsafeFilename} File saved:
{Filename}",
                    file.Name, trustedFileName);
            }
            catch (Exception ex)
            {
                Logger.LogError("File: {Filename} Error: {Error}",
                    file.Name, ex.Message);
            }
        }

        isLoading = false;
    }
}
```

Client-side file upload example

The following example processes file bytes and doesn't send files to a destination outside of the app. For an example of a Razor component that sends a file to a server or service, see the following sections:

- Upload files to a server with client-side rendering (CSR)
- Upload files to an external service

The component assumes that the Interactive WebAssembly render mode (`InteractiveWebAssembly`) is inherited from a parent component or applied globally to the app.

razor

```razor
@page "/file-upload-1"
@inject ILogger<FileUpload1> Logger

<PageTitle>File Upload 1</PageTitle>

<h1>File Upload Example 1</h1>

<p>
    <label>
        Max file size:
        <input type="number" @bind="maxFileSize" />
    </label>
</p>

<p>
    <label>
        Max allowed files:
        <input type="number" @bind="maxAllowedFiles" />
    </label>
</p>

<p>
    <label>
        Upload up to @maxAllowedFiles of up to @maxFileSize bytes:
        <InputFile OnChange="LoadFiles" multiple />
    </label>
</p>

@if (isLoading)
{
    <p>Uploading...</p>
}
else
{
    <ul>
        @foreach (var file in loadedFiles)
        {
            <li>
                <ul>
                    <li>Name: @file.Name</li>
                    <li>Last modified: @file.LastModified.ToString()</li>
                    <li>Size (bytes): @file.Size</li>
                    <li>Content type: @file.ContentType</li>
                </ul>
            </li>
        }
    </ul>
}

@code {
    private List<IBrowserFile> loadedFiles = new();
    private long maxFileSize = 1024 * 15;
```

```
    private int maxAllowedFiles = 3;
    private bool isLoading;

    private void LoadFiles(InputFileChangeEventArgs e)
    {
        isLoading = true;
        loadedFiles.Clear();

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            try
            {
                loadedFiles.Add(file);
            }
            catch (Exception ex)
            {
                Logger.LogError("File: {FileName} Error: {Error}",
                    file.Name, ex.Message);
            }
        }

        isLoading = false;
    }
}
```

IBrowserFile returns metadata exposed by the browser as properties. Use this metadata for preliminary validation.

- Name
- Size
- LastModified
- ContentType

Never trust the values of the preceding properties, especially the Name property for display in the UI. Treat all user-supplied data as a significant security risk to the app, server, and network. For more information, see Upload files in ASP.NET Core.

Upload files to a server with server-side rendering

This section applies to Interactive Server components in Blazor Web Apps or Blazor Server apps.

The following example demonstrates uploading files from a server-side app to a backend web API controller in a separate app, possibly on a separate server.

In the server-side app's `Program` file, add IHttpClientFactory and related services that allow the app to create HttpClient instances:

```
C#

builder.Services.AddHttpClient();
```

For more information, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

For the examples in this section:

- The web API runs at the URL: `https://localhost:5001`
- The server-side app runs at the URL: `https://localhost:5003`

For testing, the preceding URLs are configured in the projects' `Properties/launchSettings.json` files.

The following `UploadResult` class maintains the result of an uploaded file. When a file fails to upload on the server, an error code is returned in `ErrorCode` for display to the user. A safe file name is generated on the server for each file and returned to the client in `StoredFileName` for display. Files are keyed between the client and server using the unsafe/untrusted file name in `FileName`.

`UploadResult.cs`:

```
C#

public class UploadResult
{
    public bool Uploaded { get; set; }
    public string? FileName { get; set; }
    public string? StoredFileName { get; set; }
    public int ErrorCode { get; set; }
}
```

A security best practice for production apps is to avoid sending error messages to clients that might reveal sensitive information about an app, server, or network. Providing detailed error messages can aid a malicious user in devising attacks on an app, server, or network. The example code in this section only sends back an error code number (`int`) for display by the component client-side if a server-side error occurs. If a user requires assistance with a file upload, they provide the error code to support personnel for support ticket resolution without ever knowing the exact cause of the error.

The following `LazyBrowserFileStream` class defines a custom stream type that lazily calls OpenReadStream just before the first bytes of the stream are requested. The stream isn't transmitted from the browser to the server until reading the stream begins in .NET.

`LazyBrowserFileStream.cs`:

C#

```
using Microsoft.AspNetCore.Components.Forms;
using System.Diagnostics.CodeAnalysis;

namespace BlazorSample;

internal sealed class LazyBrowserFileStream(IBrowserFile file, int
maxAllowedSize)
    : Stream
{
    private readonly IBrowserFile file = file;
    private readonly int maxAllowedSize = maxAllowedSize;
    private Stream? underlyingStream;
    private bool isDisposed;

    public override bool CanRead => true;

    public override bool CanSeek => false;

    public override bool CanWrite => false;

    public override long Length => file.Size;

    public override long Position
    {
        get => underlyingStream?.Position ?? 0;
        set => throw new NotSupportedException();
    }

    public override void Flush() => underlyingStream?.Flush();

    public override Task<int> ReadAsync(byte[] buffer, int offset, int
count,
        CancellationToken cancellationToken)
    {
        EnsureStreamIsOpen();

        return underlyingStream.ReadAsync(buffer, offset, count,
cancellationToken);
    }

    public override ValueTask<int> ReadAsync(Memory<byte> buffer,
        CancellationToken cancellationToken = default)
    {
        EnsureStreamIsOpen();
        return underlyingStream.ReadAsync(buffer, cancellationToken);
```

```
    }

    [MemberNotNull(nameof(underlyingStream))]
    private void EnsureStreamIsOpen() =>
        underlyingStream ??= file.OpenReadStream(maxAllowedSize);

    protected override void Dispose(bool disposing)
    {
        if (isDisposed)
        {
            return;
        }

        underlyingStream?.Dispose();
        isDisposed = true;

        base.Dispose(disposing);
    }

    public override int Read(byte[] buffer, int offset, int count)
        => throw new NotSupportedException();

    public override long Seek(long offset, SeekOrigin origin)
        => throw new NotSupportedException();

    public override void SetLength(long value)
        => throw new NotSupportedException();

    public override void Write(byte[] buffer, int offset, int count)
        => throw new NotSupportedException();
}
```

The following `FileUpload2` component:

- Permits users to upload files from the client.
- Displays the untrusted/unsafe file name provided by the client in the UI. The untrusted/unsafe file name is automatically HTML-encoded by Razor for safe display in the UI.

> ⚠ **Warning**
>
> **Don't trust file names supplied by clients** for:
>
> - Saving the file to a file system or service.
> - Display in UIs that don't encode file names automatically or via developer code.

For more information on security considerations when uploading files to a server, see Upload files in ASP.NET Core.

`FileUpload2.razor`:

razor

```
@page "/file-upload-2"
@using System.Net.Http.Headers
@using System.Text.Json
@inject IHttpClientFactory ClientFactory
@inject ILogger<FileUpload2> Logger

<PageTitle>File Upload 2</PageTitle>

<h1>File Upload Example 2</h1>

<p>
    This example requires a backend server API to function. For more
information,
    see the <em>Upload files to a server</em> section
    of the <em>ASP.NET Core Blazor file uploads</em> article.
</p>

<p>
    <label>
        Upload up to @maxAllowedFiles files:
        <InputFile OnChange="OnInputFileChange" multiple />
    </label>
</p>

@if (files.Any())
{
    <div class="card">
        <div class="card-body">
            <ul>
                @foreach (var file in files)
                {
                    <li>
                        File: @file.Name
                        <br>
                        @if (FileUpload(uploadResults, file.Name, Logger,
                           out var result))
                        {
                            <span>
                                Stored File Name: @result.StoredFileName
                            </span>
                        }
                        else
                        {
                            <span>
                                There was an error uploading the file
```

```
                                (Error: @result.ErrorCode).
                            </span>
                        }
                    </li>
                }
            </ul>
        </div>
    </div>
}

@code {
    private List<File> files = new();
    private List<UploadResult> uploadResults = new();
    private int maxAllowedFiles = 3;
    private bool shouldRender;

    protected override bool ShouldRender() => shouldRender;

    private async Task OnInputFileChange(InputFileChangeEventArgs e)
    {
        shouldRender = false;
        int maxFileSize = 1024 * 15;
        var upload = false;

        using var content = new MultipartFormDataContent();

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            if (uploadResults.SingleOrDefault(
                f => f.FileName == file.Name) is null)
            {
                try
                {
                    files.Add(new() { Name = file.Name });

                    var stream = new LazyBrowserFileStream(file,
maxFileSize);
                    var fileContent = new StreamContent(stream);

                    fileContent.Headers.ContentType =
                        new MediaTypeHeaderValue(file.ContentType);

                    content.Add(
                        content: fileContent,
                        name: "\"files\"",
                        fileName: file.Name);

                    upload = true;
                }
                catch (Exception ex)
                {
                    Logger.LogInformation(
                        "{FileName} not uploaded (Err: 6): {Message}",
                        file.Name, ex.Message);
```

```
                    uploadResults.Add(
                        new()
                        {
                            FileName = file.Name,
                            ErrorCode = 6,
                            Uploaded = false
                        });
                }
            }
        }

        if (upload)
        {
            var client = ClientFactory.CreateClient();

            var response =
                await client.PostAsync("https://localhost:5001/Filesave",
                content);

            if (response.IsSuccessStatusCode)
            {
                var options = new JsonSerializerOptions
                {
                    PropertyNameCaseInsensitive = true,
                };

                using var responseStream =
                    await response.Content.ReadAsStreamAsync();

                var newUploadResults = await JsonSerializer
                    .DeserializeAsync<IList<UploadResult>>(responseStream,
options);

                if (newUploadResults is not null)
                {
                    uploadResults =
uploadResults.Concat(newUploadResults).ToList();
                }
            }
        }

        shouldRender = true;
    }

    private static bool FileUpload(IList<UploadResult> uploadResults,
        string? fileName, ILogger<FileUpload2> logger, out UploadResult
result)
    {
        result = uploadResults.SingleOrDefault(f => f.FileName == fileName)
?? new();

        if (!result.Uploaded)
        {
            logger.LogInformation("{FileName} not uploaded (Err: 5)",
fileName);
```

```
            result.ErrorCode = 5;
        }

        return result.Uploaded;
    }

    private class File
    {
        public string? Name { get; set; }
    }
}
```

If the component limits file uploads to a single file at a time or if the component only adopts interactive client-side rendering (CSR, `InteractiveWebAssembly`), the component can avoid the use of the `LazyBrowserFileStream` and use a Stream. The following demonstrates the changes for the `FileUpload2` component:

diff

```
- var stream = new LazyBrowserFileStream(file, maxFileSize);
- var fileContent = new StreamContent(stream);
+ var fileContent = new StreamContent(file.OpenReadStream(maxFileSize));
```

Remove the `LazyBrowserFileStream` class (`LazyBrowserFileStream.cs`), as it isn't used.

The following controller in the web API project saves uploaded files from the client.

> ⓘ **Important**
>
> The controller in this section is intended for use in a separate web API project from the Blazor app. The web API should mitigate Cross-Site Request Forgery (XSRF/CSRF) attacks if file upload users are authenticated.

To use the following code, create a `Development/unsafe_uploads` folder at the root of the web API project for the app running in the `Development` environment.

Because the example uses the app's environment as part of the path where files are saved, additional folders are required if other environments are used in testing and production. For example, create a `Staging/unsafe_uploads` folder for the `Staging` environment. Create a `Production/unsafe_uploads` folder for the `Production` environment.

> ⚠ **Warning**

The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see Upload files in ASP.NET Core.

`Controllers/FilesaveController.cs`:

C#

```
using System.Net;
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("[controller]")]
public class FilesaveController(
    IHostEnvironment env, ILogger<FilesaveController> logger)
    : ControllerBase
{
    [HttpPost]
    public async Task<ActionResult<IList<UploadResult>>> PostFile(
        [FromForm] IEnumerable<IFormFile> files)
    {
        var maxAllowedFiles = 3;
        long maxFileSize = 1024 * 15;
        var filesProcessed = 0;
        var resourcePath = new Uri($"{Request.Scheme}://{Request.Host}/");
        List<UploadResult> uploadResults = [];

        foreach (var file in files)
        {
            var uploadResult = new UploadResult();
            string trustedFileNameForFileStorage;
            var untrustedFileName = file.FileName;
            uploadResult.FileName = untrustedFileName;
            var trustedFileNameForDisplay =
                WebUtility.HtmlEncode(untrustedFileName);

            if (filesProcessed < maxAllowedFiles)
            {
                if (file.Length == 0)
                {
                    logger.LogInformation("{FileName} length is 0 (Err: 1)",
                        trustedFileNameForDisplay);
                    uploadResult.ErrorCode = 1;
                }
                else if (file.Length > maxFileSize)
                {
                    logger.LogInformation("{FileName} of {Length} bytes is "
+
                        "larger than the limit of {Limit} bytes (Err: 2)",
```

```
                        trustedFileNameForDisplay, file.Length,
maxFileSize);
                    uploadResult.ErrorCode = 2;
                }
                else
                {
                    try
                    {
                        trustedFileNameForFileStorage =
Path.GetRandomFileName();
                        var path = Path.Combine(env.ContentRootPath,
                            env.EnvironmentName, "unsafe_uploads",
                            trustedFileNameForFileStorage);

                        await using FileStream fs = new(path,
FileMode.Create);
                        await file.CopyToAsync(fs);

                        logger.LogInformation("{FileName} saved at {Path}",
                            trustedFileNameForDisplay, path);
                        uploadResult.Uploaded = true;
                        uploadResult.StoredFileName =
trustedFileNameForFileStorage;
                    }
                    catch (IOException ex)
                    {
                        logger.LogError("{FileName} error on upload (Err:
3): {Message}",
                            trustedFileNameForDisplay, ex.Message);
                        uploadResult.ErrorCode = 3;
                    }
                }

                filesProcessed++;
            }
            else
            {
                logger.LogInformation("{FileName} not uploaded because the "
+
                    "request exceeded the allowed {Count} of files (Err:
4)",
                    trustedFileNameForDisplay, maxAllowedFiles);
                uploadResult.ErrorCode = 4;
            }

            uploadResults.Add(uploadResult);
        }

        return new CreatedResult(resourcePath, uploadResults);
    }
}
```

In the preceding code, GetRandomFileName is called to generate a secure file name. Never trust the file name provided by the browser, as a cyberattacker may choose an existing file name that overwrites an existing file or send a path that attempts to write outside of the app.

The server app must register controller services and map controller endpoints. For more information, see Routing to controller actions in ASP.NET Core.

Upload files to a server with client-side rendering (CSR)

This section applies to client-side rendered (CSR) components in Blazor Web Apps or Blazor WebAssembly apps.

The following example demonstrates uploading files to a backend web API controller in a separate app, possibly on a separate server, from a component in a Blazor Web App that adopts CSR or a component in a Blazor WebAssembly app.

The following `UploadResult` class maintains the result of an uploaded file. When a file fails to upload on the server, an error code is returned in `ErrorCode` for display to the user. A safe file name is generated on the server for each file and returned to the client in `StoredFileName` for display. Files are keyed between the client and server using the unsafe/untrusted file name in `FileName`.

`UploadResult.cs`:

C#

```
public class UploadResult
{
    public bool Uploaded { get; set; }
    public string? FileName { get; set; }
    public string? StoredFileName { get; set; }
    public int ErrorCode { get; set; }
}
```

> ⓘ Note
>
> The preceding `UploadResult` class can be shared between client- and server-based projects. When client and server projects share the class, add an import to each project's `_Imports.razor` file for the shared project. For example:
>
> razor

```
@using BlazorSample.Shared
```

The following `FileUpload2` component:

- Permits users to upload files from the client.
- Displays the untrusted/unsafe file name provided by the client in the UI. The untrusted/unsafe file name is automatically HTML-encoded by Razor for safe display in the UI.

A security best practice for production apps is to avoid sending error messages to clients that might reveal sensitive information about an app, server, or network. Providing detailed error messages can aid a malicious user in devising attacks on an app, server, or network. The example code in this section only sends back an error code number (`int`) for display by the component client-side if a server-side error occurs. If a user requires assistance with a file upload, they provide the error code to support personnel for support ticket resolution without ever knowing the exact cause of the error.

> ⚠ **Warning**
>
> **Don't trust file names supplied by clients** for:
>
> - Saving the file to a file system or service.
> - Display in UIs that don't encode file names automatically or via developer code.
>
> For more information on security considerations when uploading files to a server, see Upload files in ASP.NET Core.

In the Blazor Web App main project, add IHttpClientFactory and related services in the project's `Program` file:

C#

```
builder.Services.AddHttpClient();
```

The `HttpClient` services must be added to the main project because the client-side component is prerendered on the server. If you disable prerendering for the following component, you aren't required to provide the `HttpClient` services in the main app and don't need to add the preceding line to the main project.

For more information on adding `HttpClient` services to an ASP.NET Core app, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

The client project (`.Client`) of a Blazor Web App must also register an HttpClient for HTTP POST requests to a backend web API controller. Confirm or add the following to the client project's `Program` file:

C#

```
builder.Services.AddScoped(sp =>
    new HttpClient { BaseAddress = new
Uri(builder.HostEnvironment.BaseAddress) });
```

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the app and is typically derived from the `<base>` tag's `href` value in the host page. If you're calling an external web API, set the URI to the web API's base address.

Specify the Interactive WebAssembly render mode attribute at the top of the following component in a Blazor Web App:

razor

```
@rendermode InteractiveWebAssembly
```

`FileUpload2.razor`:

razor

```
@page "/file-upload-2"
@using System.Linq
@using System.Net.Http.Headers
@inject HttpClient Http
@inject ILogger<FileUpload2> Logger

<PageTitle>File Upload 2</PageTitle>

<h1>File Upload Example 2</h1>

<p>
    <label>
        Upload up to @maxAllowedFiles files:
        <InputFile OnChange="OnInputFileChange" multiple />
    </label>
</p>
```

```
@if (files.Count > 0)
{
    <div class="card">
        <div class="card-body">
            <ul>
                @foreach (var file in files)
                {
                    <li>
                        File: @file.Name
                        <br>
                        @if (FileUpload(uploadResults, file.Name, Logger,
                           out var result))
                        {
                            <span>
                                Stored File Name: @result.StoredFileName
                            </span>
                        }
                        else
                        {
                            <span>
                                There was an error uploading the file
                                (Error: @result.ErrorCode).
                            </span>
                        }
                    </li>
                }
            </ul>
        </div>
    </div>
}

@code {
    private List<File> files = new();
    private List<UploadResult> uploadResults = new();
    private int maxAllowedFiles = 3;
    private bool shouldRender;

    protected override bool ShouldRender() => shouldRender;

    private async Task OnInputFileChange(InputFileChangeEventArgs e)
    {
        shouldRender = false;
        long maxFileSize = 1024 * 15;
        var upload = false;

        using var content = new MultipartFormDataContent();

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            if (uploadResults.SingleOrDefault(
                f => f.FileName == file.Name) is null)
            {
                try
                {
                    files.Add(new() { Name = file.Name });
```

```
                    var fileContent = new
StreamContent(file.OpenReadStream(maxFileSize));

                    fileContent.Headers.ContentType =
                        new MediaTypeHeaderValue(file.ContentType);

                    content.Add(
                        content: fileContent,
                        name: "\"files\"",
                        fileName: file.Name);

                    upload = true;
                }
                catch (Exception ex)
                {
                    Logger.LogInformation(
                        "{FileName} not uploaded (Err: 6): {Message}",
                        file.Name, ex.Message);

                    uploadResults.Add(
                        new()
                        {
                            FileName = file.Name,
                            ErrorCode = 6,
                            Uploaded = false
                        });
                }
            }
        }

        if (upload)
        {
            var response = await Http.PostAsync("/Filesave", content);

            var newUploadResults = await response.Content
                .ReadFromJsonAsync<IList<UploadResult>>();

            if (newUploadResults is not null)
            {
                uploadResults =
uploadResults.Concat(newUploadResults).ToList();
            }
        }

        shouldRender = true;
    }

    private static bool FileUpload(IList<UploadResult> uploadResults,
        string? fileName, ILogger<FileUpload2> logger, out UploadResult
result)
    {
        result = uploadResults.SingleOrDefault(f => f.FileName == fileName)
?? new();
```

```
        if (!result.Uploaded)
        {
            logger.LogInformation("{FileName} not uploaded (Err: 5)",
fileName);
            result.ErrorCode = 5;
        }

        return result.Uploaded;
    }

    private class File
    {
        public string? Name { get; set; }
    }
}
```

The following controller in the server-side project saves uploaded files from the client.

To use the following code, create a `Development/unsafe_uploads` folder at the root of the server-side project for the app running in the `Development` environment.

Because the example uses the app's environment as part of the path where files are saved, additional folders are required if other environments are used in testing and production. For example, create a `Staging/unsafe_uploads` folder for the `Staging` environment. Create a `Production/unsafe_uploads` folder for the `Production` environment.

> ⚠ **Warning**
>
> The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see Upload files in ASP.NET Core.

In the following example, update the shared project's namespace to match the shared project if a shared project is supplying the `UploadResult` class.

`Controllers/FilesaveController.cs`:

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
```

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using BlazorSample.Shared;

[ApiController]
[Route("[controller]")]
public class FilesaveController(
    IHostEnvironment env, ILogger<FilesaveController> logger)
    : ControllerBase
{
    [HttpPost]
    public async Task<ActionResult<IList<UploadResult>>> PostFile(
        [FromForm] IEnumerable<IFormFile> files)
    {
        var maxAllowedFiles = 3;
        long maxFileSize = 1024 * 15;
        var filesProcessed = 0;
        var resourcePath = new Uri($"{Request.Scheme}://{Request.Host}/");
        List<UploadResult> uploadResults = [];

        foreach (var file in files)
        {
            var uploadResult = new UploadResult();
            string trustedFileNameForFileStorage;
            var untrustedFileName = file.FileName;
            uploadResult.FileName = untrustedFileName;
            var trustedFileNameForDisplay =
                WebUtility.HtmlEncode(untrustedFileName);

            if (filesProcessed < maxAllowedFiles)
            {
                if (file.Length == 0)
                {
                    logger.LogInformation("{FileName} length is 0 (Err: 1)",
                        trustedFileNameForDisplay);
                    uploadResult.ErrorCode = 1;
                }
                else if (file.Length > maxFileSize)
                {
                    logger.LogInformation("{FileName} of {Length} bytes is "
+
                        "larger than the limit of {Limit} bytes (Err: 2)",
                        trustedFileNameForDisplay, file.Length,
maxFileSize);
                    uploadResult.ErrorCode = 2;
                }
                else
                {
                    try
                    {
                        trustedFileNameForFileStorage =
Path.GetRandomFileName();
                        var path = Path.Combine(env.ContentRootPath,
```

```
                            env.EnvironmentName, "unsafe_uploads",
                            trustedFileNameForFileStorage);

                        await using FileStream fs = new(path,
FileMode.Create);
                        await file.CopyToAsync(fs);

                        logger.LogInformation("{FileName} saved at {Path}",
                            trustedFileNameForDisplay, path);
                        uploadResult.Uploaded = true;
                        uploadResult.StoredFileName =
trustedFileNameForFileStorage;
                    }
                    catch (IOException ex)
                    {
                        logger.LogError("{FileName} error on upload (Err:
3): {Message}",
                            trustedFileNameForDisplay, ex.Message);
                        uploadResult.ErrorCode = 3;
                    }
                }

                filesProcessed++;
            }
            else
            {
                logger.LogInformation("{FileName} not uploaded because the "
+
                    "request exceeded the allowed {Count} of files (Err:
4)",
                    trustedFileNameForDisplay, maxAllowedFiles);
                uploadResult.ErrorCode = 4;
            }

            uploadResults.Add(uploadResult);
        }

        return new CreatedResult(resourcePath, uploadResults);
    }
}
```

In the preceding code, GetRandomFileName is called to generate a secure file name. Never trust the file name provided by the browser, as a cyberattacker may choose an existing file name that overwrites an existing file or send a path that attempts to write outside of the app.

The server app must register controller services and map controller endpoints. For more information, see Routing to controller actions in ASP.NET Core.

Cancel a file upload

A file upload component can detect when a user has cancelled an upload by using a CancellationToken when calling into the IBrowserFile.OpenReadStream or StreamReader.ReadAsync.

Create a CancellationTokenSource for the `InputFile` component. At the start of the `OnInputFileChange` method, check if a previous upload is in progress.

If a file upload is in progress:

- Call Cancel on the previous upload.
- Create a new CancellationTokenSource for the next upload and pass the CancellationTokenSource.Token to OpenReadStream or ReadAsync.

Upload files server-side with progress

The following example demonstrates how to upload files in a server-side app with upload progress displayed to the user.

To use the following example in a test app:

- **Create a folder to save uploaded files for the `Development` environment: `Development/unsafe_uploads`.**
- Configure the maximum file size (`maxFileSize`, 15 KB in the following example) and maximum number of allowed files (`maxAllowedFiles`, 3 in the following example).
- Set the buffer to a different value (10 KB in the following example), if desired, for increased granularity in progress reporting. We don't recommended using a buffer larger than 30 KB due to performance and security concerns.

> **⚠ Warning**
>
> The example saves files without scanning their contents, and the guidance in this article doesn't take into account additional security best practices for uploaded files. On staging and production systems, disable execute permission on the upload folder and scan files with an anti-virus/anti-malware scanner API immediately after upload. For more information, see Upload files in ASP.NET Core.

`FileUpload3.razor`:

razor

```
@page "/file-upload-3"
@inject ILogger<FileUpload3> Logger
@inject IWebHostEnvironment Environment
```

```
<PageTitle>File Upload 3</PageTitle>

<h1>File Upload Example 3</h1>

<p>
    <label>
        Max file size:
        <input type="number" @bind="maxFileSize" />
    </label>
</p>

<p>
    <label>
        Max allowed files:
        <input type="number" @bind="maxAllowedFiles" />
    </label>
</p>

<p>
    <label>
        Upload up to @maxAllowedFiles of up to @maxFileSize bytes:
        <InputFile OnChange="LoadFiles" multiple />
    </label>
</p>

@if (isLoading)
{
    <p>Progress: @string.Format("{0:P0}", progressPercent)</p>
}
else
{
    <ul>
        @foreach (var file in loadedFiles)
        {
            <li>
                <ul>
                    <li>Name: @file.Name</li>
                    <li>Last modified: @file.LastModified.ToString()</li>
                    <li>Size (bytes): @file.Size</li>
                    <li>Content type: @file.ContentType</li>
                </ul>
            </li>
        }
    </ul>
}

@code {
    private List<IBrowserFile> loadedFiles = new();
    private long maxFileSize = 1024 * 15;
    private int maxAllowedFiles = 3;
    private bool isLoading;
    private decimal progressPercent;

    private async Task LoadFiles(InputFileChangeEventArgs e)
```

```
    {
        isLoading = true;
        loadedFiles.Clear();
        progressPercent = 0;

        foreach (var file in e.GetMultipleFiles(maxAllowedFiles))
        {
            try
            {
                var trustedFileName = Path.GetRandomFileName();
                var path = Path.Combine(Environment.ContentRootPath,
                    Environment.EnvironmentName, "unsafe_uploads",
trustedFileName);

                await using FileStream writeStream = new(path,
FileMode.Create);
                using var readStream = file.OpenReadStream(maxFileSize);
                var bytesRead = 0;
                var totalRead = 0;
                var buffer = new byte[1024 * 10];

                while ((bytesRead = await readStream.ReadAsync(buffer)) !=
0)
                {
                    totalRead += bytesRead;
                    await writeStream.WriteAsync(buffer, 0, bytesRead);
                    progressPercent = Decimal.Divide(totalRead, file.Size);
                    StateHasChanged();
                }

                loadedFiles.Add(file);

                Logger.LogInformation(
                    "Unsafe Filename: {UnsafeFilename} File saved:
{Filename}",
                    file.Name, trustedFileName);
            }
            catch (Exception ex)
            {
                Logger.LogError("File: {FileName} Error: {Error}",
                    file.Name, ex.Message);
            }
        }

        isLoading = false;
    }
}
```

For more information, see the following API resources:

- FileStream: Provides a Stream for a file, supporting both synchronous and asynchronous read and write operations.

- FileStream.ReadAsync: The preceding `FileUpload3` component reads the stream asynchronously with ReadAsync. Reading a stream synchronously with Read isn't supported in Razor components.

File streams

With server interactivity, file data is streamed over the SignalR connection into .NET code on the server as the file is read.

For a WebAssembly-rendered component, file data is streamed directly into the .NET code within the browser.

Upload image preview

For an image preview of uploading images, start by adding an `InputFile` component with a component reference and an `OnChange` handler:

razor

```
<InputFile @ref="inputFile" OnChange="ShowPreview" />
```

Add an image element with an element reference, which serves as the placeholder for the image preview:

razor

```
<img @ref="previewImageElem" />
```

Add the associated references:

razor

```
@code {
    private InputFile? inputFile;
    private ElementReference previewImageElem;
}
```

In JavaScript, add a function called with an HTML input and img element that performs the following:

- Extracts the selected file.
- Creates an object URL with createObjectURL.

- Sets an event listener to revoke the object URL with revokeObjectURL after the image is loaded, so memory isn't leaked.
- Sets the `img` element's source to display the image.

JavaScript

```
window.previewImage = (inputElem, imgElem) => {
  const url = URL.createObjectURL(inputElem.files[0]);
  imgElem.addEventListener('load', () => URL.revokeObjectURL(url), { once:
true });
  imgElem.src = url;
}
```

Finally, use an injected IJSRuntime to add the `OnChange` handler that calls the JavaScript function:

razor

```
@inject IJSRuntime JS

...

@code {
    ...

    private async Task ShowPreview() => await JS.InvokeVoidAsync(
        "previewImage", inputFile!.Element, previewImageElem);
}
```

The preceding example is for uploading a single image. The approach can be expanded to support `multiple` images.

The following `FileUpload4` component shows the complete example.

`FileUpload4.razor`:

razor

```
@page "/file-upload-4"
@inject IJSRuntime JS

<h1>File Upload Example</h1>

<InputFile @ref="inputFile" OnChange="ShowPreview" />

<img style="max-width:200px;max-height:200px" @ref="previewImageElem" />

@code {
    private InputFile? inputFile;
```

```
    private ElementReference previewImageElem;

    private async Task ShowPreview() => await JS.InvokeVoidAsync(
        "previewImage", inputFile!.Element, previewImageElem);
}
```

Upload files to an external service

Instead of an app handling file upload bytes and the app's server receiving uploaded files, clients can directly upload files to an external service. The app can safely process the files from the external service on demand. This approach hardens the app and its server against malicious attacks and potential performance problems.

Consider an approach that uses Azure Files, Azure Blob Storage, or a third-party service with the following potential benefits:

- Upload files from the client directly to an external service with a JavaScript client library or REST API. For example, Azure offers the following client libraries and APIs:
  - Azure Storage File Share client library
  - Azure Files REST API
  - Azure Storage Blob client library for JavaScript
  - Blob service REST API
- Authorize user uploads with a user-delegated shared-access signature (SAS) token generated by the app (server-side) for each client file upload. For example, Azure offers the following SAS features:
  - Azure Storage File Share client library for JavaScript: with SAS Token
  - Azure Storage Blob client library for JavaScript: with SAS Token
- Provide automatic redundancy and file share backup.
- Limit uploads with quotas. Note that Azure Blob Storage's quotas are set at the account level, not the container level. However, Azure Files quotas are at the file share level and might provide better control over upload limits. For more information, see the Azure documents linked earlier in this list.
- Secure files with server-side encryption (SSE).

For more information on Azure Blob Storage and Azure Files, see the Azure Storage documentation.

Server-side SignalR message size limit

File uploads may fail even before they start, when Blazor retrieves data about the files that exceeds the maximum SignalR message size.

SignalR defines a message size limit that applies to every message Blazor receives, and the InputFile component streams files to the server in messages that respect the configured limit. However, the first message, which indicates the set of files to upload, is sent as a unique single message. The size of the first message may exceed the SignalR message size limit. The issue isn't related to the size of the files, it's related to the number of files.

The logged error is similar to the following:

> Error: Connection disconnected with error 'Error: Server returned an error on close: Connection closed with an error.'. e.log @ blazor.server.js:1

When uploading files, reaching the message size limit on the first message is rare. If the limit is reached, the app can configure HubOptions.MaximumReceiveMessageSize with a larger value.

For more information on SignalR configuration and how to set MaximumReceiveMessageSize, see ASP.NET Core Blazor SignalR guidance.

Maximum parallel invocations per client hub setting

Blazor relies on MaximumParallelInvocationsPerClient set to 1, which is the default value.

Increasing the value leads to a high probability that `CopyTo` operations throw `System.InvalidOperationException: 'Reading is not allowed after reader was completed.'`. For more information, see MaximumParallelInvocationsPerClient > 1 breaks file upload in Blazor Server mode (dotnet/aspnetcore #53951).

Troubleshoot

The line that calls IBrowserFile.OpenReadStream throws a System.TimeoutException:

> System.TimeoutException: Did not receive any data in the allotted time.

Possible causes:

- Using the Autofac Inversion of Control (IoC) container instead of the built-in ASP.NET Core dependency injection container. To resolve the issue, set DisableImplicitFromServicesParameters to `true` in the server-side circuit handler

hub options. For more information, see FileUpload: Did not receive any data in the allotted time (dotnet/aspnetcore #38842).

- Not reading the stream to completion. This isn't a framework issue. Trap the exception and investigate it further in your local environment/network.
- Using server-side rendering and calling OpenReadStream on multiple files before reading them to completion. To resolve the issue, use the `LazyBrowserFileStream` class and approach described in the Upload files to a server with server-side rendering section of this article.

Additional resources

- ASP.NET Core Blazor file downloads
- Upload files in ASP.NET Core
- ASP.NET Core Blazor forms overview
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor file downloads

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to download files in Blazor apps.

File downloads

This article covers approaches for the following scenarios, where a file shouldn't be opened by a browser but downloaded and saved on the client:

- Stream file content to a raw binary data buffer on the client: Typically, this approach is used for relatively small files (< 250 MB).
- Download a file via a URL without streaming: Usually, this approach is used for relatively large files (> 250 MB).

When downloading files from a different origin than the app, Cross-Origin Resource Sharing (CORS) considerations apply. For more information, see the Cross-Origin Resource Sharing (CORS) section.

Security considerations

Use caution when providing users with the ability to download files from a server. Cyberattackers may execute Denial of Service (DoS) attacks, API exploitation attacks, or attempt to compromise networks and servers in other ways.

Security steps that reduce the likelihood of a successful attack are:

- Download files from a dedicated file download area on the server, preferably from a non-system drive. Using a dedicated location makes it easier to impose security restrictions on downloadable files. Disable execute permissions on the file download area.
- Client-side security checks are easy to circumvent by malicious users. Always perform client-side security checks on the server, too.
- Don't receive files from users or other untrusted sources and then make the files available for immediate download without performing security checks on the files.

For more information, see Upload files in ASP.NET Core.

Download from a stream

This section applies to files that are typically up to 250 MB in size.

The recommended approach for downloading relatively small files (< 250 MB) is to stream file content to a raw binary data buffer on the client with JavaScript (JS) interop. This approach is effective for components that adopt an interactive render mode but not components that adopt static server-side rendering (static SSR).

> ⚠ **Warning**
>
> The approach in this section reads the file's content into a **JS ArrayBuffer**. This approach loads the entire file into the client's memory, which can impair performance. To download relatively large files (>= 250 MB), we recommend following the guidance in the **Download from a URL** section.

The following `downloadFileFromStream` JS function:

- Reads the provided stream into an ArrayBuffer.
- Creates a Blob to wrap the `ArrayBuffer`.
- Creates an object URL to serve as the file's download address.
- Creates an HTMLAnchorElement (`<a>` element).
- Assigns the file's name (`fileName`) and URL (`url`) for the download.
- Triggers the download by firing a click event on the anchor element.
- Removes the anchor element.
- Revokes the object URL (`url`) by calling URL.revokeObjectURL. **This is an important step to ensure memory isn't leaked on the client.**

HTML

```
<script>
  window.downloadFileFromStream = async (fileName, contentStreamReference)
=> {
    const arrayBuffer = await contentStreamReference.arrayBuffer();
    const blob = new Blob([arrayBuffer]);
    const url = URL.createObjectURL(blob);
    const anchorElement = document.createElement('a');
    anchorElement.href = url;
    anchorElement.download = fileName ?? '';
    anchorElement.click();
    anchorElement.remove();
    URL.revokeObjectURL(url);
```

```
  }
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

The following component:

- Uses native byte-streaming interop to ensure efficient transfer of the file to the client.
- Has a method named `GetFileStream` to retrieve a Stream for the file that's downloaded to clients. Alternative approaches include retrieving a file from storage or generating a file dynamically in C# code. For this demonstration, the app creates a 50 KB file of random data from a new byte array (`new byte[]`). The bytes are wrapped with a MemoryStream to serve as the example's dynamically-generated binary file.
- The `DownloadFileFromStream` method:
  - Retrieves the Stream from `GetFileStream`.
  - Specifies a file name when file is saved on the user's machine. The following example names the file `quote.txt`.
  - Wraps the Stream in a DotNetStreamReference, which allows streaming the file data to the client.
  - Invokes the `downloadFileFromStream` JS function to accept the data on the client.

`FileDownload1.razor`:

razor

```razor
@page "/file-download-1"
@using System.IO
@inject IJSRuntime JS

<PageTitle>File Download 1</PageTitle>

<h1>File Download Example 1</h1>

<button @onclick="DownloadFileFromStream">
    Download File From Stream
</button>

@code {
    private Stream GetFileStream()
    {
```

```
        var randomBinaryData = new byte[50 * 1024];
        var fileStream = new MemoryStream(randomBinaryData);

        return fileStream;
    }

    private async Task DownloadFileFromStream()
    {
        var fileStream = GetFileStream();
        var fileName = "log.bin";

        using var streamRef = new DotNetStreamReference(stream: fileStream);

        await JS.InvokeVoidAsync("downloadFileFromStream", fileName,
streamRef);
    }
}
```

For a component in a server-side app that must return a Stream for a physical file, the component can call File.OpenRead, as the following example demonstrates:

C#

```
private Stream GetFileStream() => File.OpenRead(@"{PATH}");
```

In the preceding example, the `{PATH}` placeholder is the path to the file. The `@` prefix indicates that the string is a *verbatim string literal*, which permits the use of backslashes (`\`) in a Windows OS path and embedded double-quotes (`""`) for a single quote in the path. Alternatively, avoid the string literal (`@`) and use either of the following approaches:

- Use escaped backslashes (`\\`) and quotes (`\"`).
- Use forward slashes (`/`) in the path, which are supported across platforms in ASP.NET Core apps, and escaped quotes (`\"`).

Download from a URL

This section applies to files that are relatively large, typically 250 MB or larger.

The recommended approach for downloading relatively large files (>= 250 MB) with interactively-rendered components or files of any size for statically-rendered components is to use JS to trigger an anchor element with the file's name and URL.

The example in this section uses a download file named `quote.txt`, which is placed in a folder named `files` in the app's web root (`wwwroot` folder). The use of the `files` folder

is only for demonstration purposes. You can organize downloadable files in any folder layout within the web root (`wwwroot` folder) that you prefer, including serving the files directly from the `wwwroot` folder.

`wwwroot/files/quote.txt`:

text

```
When victory is ours, we'll wipe every trace of the Thals and their city
from the face of this land. We will avenge the deaths of all Kaleds who've
fallen in the cause of right and justice and build a peace which will be a
monument to their sacrifice. Our battle cry will be "Total extermination of
the Thals!"

- General Ravon (Guy Siner, http://guysiner.com/)
  Dr. Who: Genesis of the Daleks (https://www.bbc.co.uk/programmes/p00vd5g2)
  Copyright 1975 BBC (https://www.bbc.co.uk/)
```

The following `triggerFileDownload` JS function:

- Creates an HTMLAnchorElement (`<a>` element).
- Assigns the file's name (`fileName`) and URL (`url`) for the download.
- Triggers the download by firing a click event on the anchor element.
- Removes the anchor element.

HTML

```
<script>
  window.triggerFileDownload = (fileName, url) => {
    const anchorElement = document.createElement('a');
    anchorElement.href = url;
    anchorElement.download = fileName ?? '';
    anchorElement.click();
    anchorElement.remove();
  }
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

The following example component downloads the file from the same origin that the app uses. If the file download is attempted from a different origin, configure Cross-Origin

Resource Sharing (CORS). For more information, see the Cross-Origin Resource Sharing (CORS) section.

`FileDownload2.razor`:

razor

```razor
@page "/file-download-2"
@inject IJSRuntime JS

<PageTitle>File Download 2</PageTitle>

<h1>File Download Example 2</h1>

<button @onclick="DownloadFileFromURL">
    Download File From URL
</button>

@code {
    private async Task DownloadFileFromURL()
    {
        var fileName = "quote.txt";
        var fileURL = "/files/quote.txt";
        await JS.InvokeVoidAsync("triggerFileDownload", fileName, fileURL);
    }
}
```

For interactive components, the button in the preceding example calls the `DownloadFileFromURL` handler to invoke the JavaScript (JS) function `triggerFileDownload`.

If the component adopts static server-side rendering (static SSR), add an event handler for the button (addEventListener (MDN documentation)) to call `triggerFileDownload` following the guidance in ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR).

Cross-Origin Resource Sharing (CORS)

Without taking further steps to enable Cross-Origin Resource Sharing (CORS) for files that don't have the same origin as the app, downloading files won't pass CORS checks made by the browser.

For more information on CORS with ASP.NET Core apps and other Microsoft products and services that host files for download, see the following resources:

- Enable Cross-Origin Requests (CORS) in ASP.NET Core
- Using Azure CDN with CORS (Azure documentation)

- Cross-Origin Resource Sharing (CORS) support for Azure Storage (REST documentation)
- Core Cloud Services - Set up CORS for your website and storage assets (Learn module)
- IIS CORS module Configuration Reference (IIS documentation)

Additional resources

- ASP.NET Core Blazor static files
- ASP.NET Core Blazor JavaScript interoperability (JS interop)
- JavaScript location in ASP.NET Core Blazor apps
- ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)
- <a>: The Anchor element: Security and privacy (MDN documentation)
- ASP.NET Core Blazor file uploads
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor JavaScript interoperability (JS interop)

Article • 11/19/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

A Blazor app can invoke JavaScript (JS) functions from .NET methods and .NET methods from JS functions. These scenarios are called *JavaScript interoperability* (*JS interop*).

Further JS interop guidance is provided in the following articles:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

> **Note**
>
> JavaScript `[JSImport]`/`[JSExport]` interop API is available for client-side components in ASP.NET Core in .NET 7 or later.
>
> For more information, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

JavaScript interop abstractions and features package

The @microsoft/dotnet-js-interop package (npmjs.com) (Microsoft.JSInterop NuGet package) provides abstractions and features for interop between .NET and JavaScript (JS) code. Reference source is available in the dotnet/aspnetcore GitHub repository (/src/JSInterop folder). For more information, see the GitHub repository's `README.md` file.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To

select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Additional resources for writing JS interop scripts in TypeScript:

- TypeScript
- Tutorial: Create an ASP.NET Core app with TypeScript in Visual Studio
- Manage npm packages in Visual Studio

Interaction with the DOM

Only mutate the DOM with JavaScript (JS) when the object doesn't interact with Blazor. Blazor maintains representations of the DOM and interacts directly with DOM objects. If an element rendered by Blazor is modified externally using JS directly or via JS Interop, the DOM may no longer match Blazor's internal representation, which can result in undefined behavior. Undefined behavior may merely interfere with the presentation of elements or their functions but may also introduce security risks to the app or server.

This guidance not only applies to your own JS interop code but also to any JS libraries that the app uses, including anything provided by a third-party framework, such as Bootstrap JS and jQuery.

In a few documentation examples, JS interop is used to mutate an element *purely for demonstration purposes* as part of an example. In those cases, a warning appears in the text.

For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

JavaScript class with a field of type function

A JavaScript class with a field of type function isn't supported by Blazor JS interop. Use Javascript functions in classes.

❌ `GreetingHelpers.sayHello` in the following class as a field of type function isn't discovered by Blazor's JS interop and can't be executed from C# code:

JavaScript

```
export class GreetingHelpers {
  sayHello = function() {
    ...
```

```
  }
}
```

✔ `GreetingHelpers.sayHello` in the following class as a function is supported:

JavaScript

```
export class GreetingHelpers {
  sayHello() {
    ...
  }
}
```

Arrow functions are also supported:

JavaScript

```
export class GreetingHelpers {
  sayHello = () => {
    ...
  }
}
```

Avoid inline event handlers

A JavaScript function can be invoked directly from an inline event handler. In the following example, `alertUser` is a JavaScript function called when the button is selected by the user:

HTML

```
<button onclick="alertUser">Click Me!</button>
```

However, the use of inline event handlers is a *poor design choice* for calling JavaScript functions:

- Mixing HTML markup and JavaScript code often leads to unmaintainable code.
- Inline event handler execution may be blocked by a Content Security Policy (CSP) (MDN documentation).

We recommend avoiding inline event handlers in favor of approaches that assign handlers in JavaScript with addEventListener, as the following example demonstrates:

`AlertUser.razor.js`:

JavaScript

```javascript
export function alertUser() {
  alert('The button was selected!');
}

export function addHandlers() {
  const btn = document.getElementById("btn");
  btn.addEventListener("click", alertUser);
}
```

`AlertUser.razor`:

razor

```razor
@page "/alert-user"
@implements IAsyncDisposable
@inject IJSRuntime JS

<h1>Alert User</h1>

<p>
    <button id="btn">Click Me!</button>
</p>

@code {
    private IJSObjectReference? module;

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/Pages/AlertUser.razor.js");

            await module.InvokeVoidAsync("addHandlers");
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
```

```
    }
}
```

In the preceding example, JSDisconnectedException is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try`-`catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

For more information, see the following resources:

- JavaScript location in ASP.NET Core Blazor apps
- Introduction to events (MDN documentation)

Asynchronous JavaScript calls

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side rendering models. When adopting server-side rendering, JS interop calls must be asynchronous because they're sent over a network connection. For apps that exclusively adopt client-side rendering, synchronous JS interop calls are supported.

For more information, see the following articles:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Object serialization

Blazor uses System.Text.Json for serialization with the following requirements and default behaviors:

- Types must have a default constructor, get/set accessors must be public, and fields are never serialized.
- Global default serialization isn't customizable to avoid breaking existing component libraries, impacts on performance and security, and reductions in reliability.
- Serializing .NET member names results in lowercase JSON key names.
- JSON is deserialized as JsonElement C# instances, which permit mixed casing. Internal casting for assignment to C# model properties works as expected in spite

of any case differences between JSON key names and C# property names.
- Complex framework types, such as KeyValuePair, might be trimmed away by the IL Trimmer on publish and not present for JS interop. We recommend creating custom types for types that the IL Trimmer trims away.
- Blazor always relies on reflection for JSON serialization, including when using C# source generation. Setting `JsonSerializerIsReflectionEnabledByDefault` to `false` in the app's project file results in an error when serialization is attempted.

JsonConverter API is available for custom serialization. Properties can be annotated with a [JsonConverter] attribute to override default serialization for an existing data type.

For more information, see the following resources in the .NET documentation:

- JSON serialization and deserialization (marshalling and unmarshalling) in .NET
- How to customize property names and values with System.Text.Json
- How to write custom converters for JSON serialization (marshalling) in .NET

Blazor supports optimized byte array JS interop that avoids encoding/decoding byte arrays into Base64. The app can apply custom serialization and pass the resulting bytes. For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

DOM cleanup tasks during component disposal

Don't execute JS interop code for DOM cleanup tasks during component disposal. Instead, use the MutationObserver pattern in JavaScript (JS) on the client for the following reasons:

- The component may have been removed from the DOM by the time your cleanup code executes in `Dispose{Async}`.
- During server-side rendering, the Blazor renderer may have been disposed by the framework by the time your cleanup code executes in `Dispose{Async}`.

The MutationObserver pattern allows you to run a function when an element is removed from the DOM.

In the following example, the `DOMCleanup` component:

- Contains a `<div>` with an `id` of `cleanupDiv`. The `<div>` element is removed from the DOM along with the rest of the component's DOM markup when the component is removed from the DOM.
- Loads the `DOMCleanup` JS class from the `DOMCleanup.razor.js` file and calls its `createObserver` function to set up the `MutationObserver` callback. These tasks are

accomplished in the OnAfterRenderAsync lifecycle method.

`DOMCleanup.razor`:

razor

```razor
@page "/dom-cleanup"
@implements IAsyncDisposable
@inject IJSRuntime JS

<h1>DOM Cleanup Example</h1>

<div id="cleanupDiv"></div>

@code {
    private IJSObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>(
                "import", "./Components/Pages/DOMCleanup.razor.js");

            await module.InvokeVoidAsync("DOMCleanup.createObserver");
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

In the preceding example, JSDisconnectedException is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try`-`catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

In the following example, the `MutationObserver` callback is executed each time a DOM change occurs. Execute your cleanup code when the `if` statement confirms that the target element (`cleanupDiv`) was removed (`if (targetRemoved) { ... }`). It's important to disconnect and delete the `MutationObserver` to avoid a memory leak after your cleanup code executes.

`DOMCleanup.razor.js` placed side-by-side with the preceding `DOMCleanup` component:

JavaScript

```
export class DOMCleanup {
  static observer;

  static createObserver() {
    const target = document.querySelector('#cleanupDiv');

    this.observer = new MutationObserver(function (mutations) {
      const targetRemoved = mutations.some(function (mutation) {
        const nodes = Array.from(mutation.removedNodes);
        return nodes.indexOf(target) !== -1;
      });

      if (targetRemoved) {
        // Cleanup resources here
        // ...

        // Disconnect and delete MutationObserver
        this.observer && this.observer.disconnect();
        delete this.observer;
      }
    });

    this.observer.observe(target.parentNode, { childList: true });
  }
}

window.DOMCleanup = DOMCleanup;
```

The preceding approaches attach the `MutationObserver` to `target.parentNode`, which works until `parentNode` itself is removed from the DOM. This is a common scenario, for example, when navigating to a new page, which causes the entire page component to be removed from the DOM. In such cases, any child components observing changes within the page aren't cleaned up properly.

Don't assume that observing `document.body`, instead of `target.parentNode`, is a better target. Observing `document.body` has performance implications because callback logic is executed for *all* DOM updates, whether or not they have anything to do with your element. Use either of the following approaches:

- In cases where you can identify a suitable ancestor node to observe, use `MutationObserver` with it. Ideally, this ancestor is scoped to the changes that you want to observe, rather than `document.body`.
- Instead of using `MutationObserver`, consider using a custom element and disconnectedCallback. The event always fires when your custom element is disconnected, no matter where it resides in the DOM relative to the DOM change.

JavaScript interop calls without a circuit

This section only applies to server-side apps.

JavaScript (JS) interop calls can't be issued after Blazor's SignalR circuit is disconnected. Without a circuit during component disposal or at any other time that a circuit doesn't exist, the following method calls fail and log a message that the circuit is disconnected as a JSDisconnectedException:

- JS interop method calls
  - IJSRuntime.InvokeAsync
  - JSRuntimeExtensions.InvokeAsync
  - JSRuntimeExtensions.InvokeVoidAsync
- `Dispose`/`DisposeAsync` calls on any IJSObjectReference.

In order to avoid logging JSDisconnectedException or to log custom information, catch the exception in a try-catch statement.

For the following component disposal example:

- The server-side component implements IAsyncDisposable.
- `module` is an IJSObjectReference for a JS module.
- JSDisconnectedException is caught and not logged.
- Optionally, you can log custom information in the `catch` statement at whatever log level you prefer. The following example doesn't log custom information because it assumes the developer doesn't care about when or where circuits are disconnected during component disposal.

C#

```
async ValueTask IAsyncDisposable.DisposeAsync()
{
    try
    {
        if (module is not null)
        {
            await module.DisposeAsync();
```

```
            }
        }
        catch (JSDisconnectedException)
        {
        }
    }
```

If you must clean up your own JS objects or execute other JS code on the client after a circuit is lost in a server-side Blazor app, use the MutationObserver pattern in JS on the client. The MutationObserver pattern allows you to run a function when an element is removed from the DOM.

For more information, see the following articles:

- Handle errors in ASP.NET Core Blazor apps: The *JavaScript interop* section discusses error handling in JS interop scenarios.
- ASP.NET Core Razor component lifecycle: The *Component disposal with* `IDisposable` *and* `IAsyncDisposable` section describes how to implement disposal patterns in Razor components.

Cached JavaScript files

JavaScript (JS) files and other static assets aren't generally cached on clients during development in the Development environment. During development, static asset requests include the Cache-Control header with a value of no-cache or max-age with a value of zero (`0`).

During production in the Production environment, JS files are usually cached by clients.

To disable client-side caching in browsers, developers usually adopt one of the following approaches:

- Disable caching when the browser's developer tools console is open. Guidance can be found in the developer tools documentation of each browser maintainer:
  - Chrome DevTools
  - Microsoft Edge Developer Tools overview
- Perform a manual browser refresh of any webpage of the Blazor app to reload JS files from the server. ASP.NET Core's HTTP Caching Middleware always honors a valid no-cache Cache-Control header sent by a client.

For more information, see:

- ASP.NET Core Blazor environments
- Response caching in ASP.NET Core

Size limits on JavaScript interop calls

This section only applies to interactive components in server-side apps. For client-side components, the framework doesn't impose a limit on the size of JavaScript (JS) interop inputs and outputs.

For interactive components in server-side apps, JS interop calls passing data from the client to the server are limited in size by the maximum incoming SignalR message size permitted for hub methods, which is enforced by HubOptions.MaximumReceiveMessageSize (default: 32 KB). JS to .NET SignalR messages larger than MaximumReceiveMessageSize throw an error. The framework doesn't impose a limit on the size of a SignalR message from the hub to a client. For more information on the size limit, error messages, and guidance on dealing with message size limits, see ASP.NET Core Blazor SignalR guidance.

Determine where the app is running

If it's relevant for the app to know where code is running for JS interop calls, use OperatingSystem.IsBrowser to determine if the component is executing in the context of browser on WebAssembly.

JavaScript location in ASP.NET Core Blazor apps

Article • 01/16/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Load JavaScript (JS) code using any of the following approaches:

- Load a script in <head> markup (*Not generally recommended*)
- Load a script in <body> markup
- Load a script from an external JavaScript file (.js) collocated with a component
- Load a script from an external JavaScript file (.js)
- Inject a script before or after Blazor starts

Inline JavaScript isn't recommended for Blazor apps. We recommend using JS collocation combined with JS modules.

Location of `<script>` tags

Only place a `<script>` tag in a component file (`.razor`) if the component is guaranteed to adopt static server-side rendering (static SSR) because the `<script>` tag can't be updated dynamically. Placing a `<script>` tag in a component file doesn't produce a compile-time warning or error, but script loading behavior might not match your expectations in components that adopt an interactive render mode.

> **Note**
>
> Documentation examples usually place scripts in a `<script>` tag or load global scripts from external files. These approaches pollute the client with global functions. For production apps, we recommend placing JS into separate JS modules that can be imported when needed. For more information, see the JavaScript isolation in JavaScript modules section.

Load a script in `<head>` markup

The approach in this section isn't generally recommended.

Place the JavaScript (JS) tags (`<script>...</script>`) in the <head> element markup:

HTML

```
<head>
    ...

    <script>
      window.jsMethod = (methodParameter) => {
        ...
      };
    </script>
</head>
```

Loading JS from the `<head>` isn't the best approach for the following reasons:

- JS interop may fail if the script depends on Blazor. We recommend loading scripts using one of the other approaches, not via the `<head>` markup.
- The page may become interactive slower due to the time it takes to parse the JS in the script.

In component markup, scripts can be loaded via a HeadContent component with the usual caveat that the approach slows down page load on the client, which we recommend avoiding. When a script is loaded with a `HeadContent` component in a Blazor Server app, Blazor WebAssembly app, or a Blazor Web App using an interactive render mode (interactive SSR, CSR), navigating away from the component's page removes the `<script>` tag from the rendered `<head>` content but doesn't unload the script's JavaScript code, including event handlers that the script registers, exposed variables, and methods that the script provides. Only Blazor Web Apps using static SSR unload JavaScript code when the user navigates away from the page. Generally, you're better off adding `<script>` tags to the physical `<head>` content, unless you explicitly desire to keep such script references in the components that use them and don't mind that the code isn't unloaded on navigation events.

Load a script in `<body>` markup

Place the JavaScript tags (`<script>...</script>`) inside the closing </body> element after the Blazor script reference:

HTML

```
<body>
    ...

    <script src="{BLAZOR SCRIPT}"></script>
    <script>
      window.jsMethod = (methodParameter) => {
        ...
      };
    </script>
</body>
```

In the preceding example, the `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.

Load a script from an external JavaScript file (`.js`) collocated with a component

Collocation of JavaScript (JS) files for Razor components is a convenient way to organize scripts in an app.

Razor components of Blazor apps collocate JS files using the `.razor.js` extension and are publicly addressable using the path to the file in the project:

`{PATH}/{COMPONENT}.razor.js`

- The `{PATH}` placeholder is the path to the component.
- The `{COMPONENT}` placeholder is the component.

When the app is published, the framework automatically moves the script to the web root. Scripts are moved to `bin/Release/{TARGET FRAMEWORK MONIKER}/publish/wwwroot/{PATH}/{COMPONENT}.razor.js`, where the placeholders are:

- `{TARGET FRAMEWORK MONIKER}` is the Target Framework Moniker (TFM).
- `{PATH}` is the path to the component.
- `{COMPONENT}` is the component name.

No change is required to the script's relative URL, as Blazor takes care of placing the JS file in published static assets for you.

This section and the following examples are primarily focused on explaining JS file collocation. The first example demonstrates a collocated JS file with an ordinary JS function. The second example demonstrates the use of a module to load a function, which is the recommended approach for most production apps. Calling JS from .NET is

fully covered in Call JavaScript functions from .NET methods in ASP.NET Core Blazor, where there are further explanations of the Blazor JS API with additional examples. Component disposal, which is present in the second example, is covered in ASP.NET Core Razor component lifecycle.

The following `JsCollocation1` component loads a script via a HeadContent component and calls a JS function with IJSRuntime.InvokeAsync. The `{PATH}` placeholder is the path to the component.

> **ⓘ Important**
>
> If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component (example: `Components/Pages` in .NET 8 or later or `Pages` in .NET 7 or earlier). In a Blazor Web App (.NET 8 or later), the component requires an interactive render mode applied either globally to the app or to the component definition.

Add the following script after the Blazor script (location of the Blazor start script):

HTML

```
<script src="{PATH}/JsCollocation1.razor.js"></script>
```

`JsCollocation1` component (`{PATH}/JsCollocation1.razor`):

razor

```
@page "/js-collocation-1"
@inject IJSRuntime JS

<PageTitle>JS Collocation 1</PageTitle>

<h1>JS Collocation Example 1</h1>

<button @onclick="ShowPrompt">Call showPrompt1</button>

@if (!string.IsNullOrEmpty(result))
{
    <p>
        Hello @result!
    </p>
}

@code {
    private string? result;
```

```
    public async Task ShowPrompt()
    {
        result = await JS.InvokeAsync<string>(
            "showPrompt1", "What's your name?");
        StateHasChanged();
    }
}
```

The collocated JS file is placed next to the `JsCollocation1` component file with the file name `JsCollocation1.razor.js`. In the `JsCollocation1` component, the script is referenced at the path of the collocated file. In the following example, the `showPrompt1` function accepts the user's name from a Window prompt() and returns it to the `JsCollocation1` component for display.

`{PATH}/JsCollocation1.razor.js`:

JavaScript

```
function showPrompt1(message) {
  return prompt(message, 'Type your name here');
}
```

The preceding approach isn't recommended for general use in production apps because the approach pollutes the client with global functions. A better approach for production apps is to use JS modules. The same general principles apply to loading a JS module from a collocated JS file, as the next example demonstrates.

The following `JsCollocation2` component's `OnAfterRenderAsync` method loads a JS module into `module`, which is an IJSObjectReference of the component class. `module` is used to call the `showPrompt2` function. The `{PATH}` placeholder is the path to the component.

> **ⓘ Important**
>
> If you use the following code for a demonstration in a test app, change the `{PATH}` placeholder to the path of the component. In a Blazor Web App (.NET 8 or later), the component requires an interactive render mode applied either globally to the app or to the component definition.

`JsCollocation2` component (`{PATH}/JsCollocation2.razor`):

razor

```razor
@page "/js-collocation-2"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>JS Collocation 2</PageTitle>

<h1>JS Collocation Example 2</h1>

<button @onclick="ShowPrompt">Call showPrompt2</button>

@if (!string.IsNullOrEmpty(result))
{
    <p>
        Hello @result!
    </p>
}

@code {
    private IJSObjectReference? module;
    private string? result;

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            /*
                Change the {PATH} placeholder in the next line to the path
of
                the collocated JS file in the app. Examples:

                ./Components/Pages/JsCollocation2.razor.js (.NET 8 or later)
                ./Pages/JsCollocation2.razor.js (.NET 7 or earlier)
            */
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./{PATH}/JsCollocation2.razor.js");
        }
    }

    public async void ShowPrompt()
    {
        if (module is not null)
        {
            result = await module.InvokeAsync<string>(
                "showPrompt2", "What's your name?");
            StateHasChanged();
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
```

```
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

In the preceding example, JSDisconnectedException is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try`-`catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

`{PATH}/JsCollocation2.razor.js`:

JavaScript

```
export function showPrompt2(message) {
  return prompt(message, 'Type your name here');
}
```

> **Important**
>
> Don't place a `<script>` tag for `JsCollocation2.razor.js` after the **Blazor script** because the module is loaded and cached automatically when the **dynamic import()** is invoked.

Use of scripts and modules for collocated JS in a Razor class library (RCL) is only supported for Blazor's JS interop mechanism based on the IJSRuntime interface. If you're implementing JavaScript [JSImport]/[JSExport] interop, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

For scripts or modules provided by a Razor class library (RCL) using IJSRuntime-based JS interop, the following path is used:

`./_content/{PACKAGE ID}/{PATH}/{COMPONENT}.{EXTENSION}.js`

- The path segment for the current directory (`./`) is required in order to create the correct static asset path to the JS file.

- The `{PACKAGE ID}` placeholder is the RCL's package identifier (or library name for a class library referenced by the app).
- The `{PATH}` placeholder is the path to the component. If a Razor component is located at the root of the RCL, the path segment isn't included.
- The `{COMPONENT}` placeholder is the component name.
- The `{EXTENSION}` placeholder matches the extension of component, either `razor` or `cshtml`.

In the following Blazor app example:

- The RCL's package identifier is `AppJS`.
- A module's scripts are loaded for the `JsCollocation3` component (`JsCollocation3.razor`).
- The `JsCollocation3` component is in the `Components/Pages` folder of the RCL.

C#

```
module = await JS.InvokeAsync<IJSObjectReference>("import",
    "./_content/AppJS/Components/Pages/JsCollocation3.razor.js");
```

For more information on RCLs, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Load a script from an external JavaScript file (`.js`)

Place the JavaScript (JS) tags (`<script>...</script>`) with a script source (`src`) path inside the closing </body> element after the Blazor script reference:

HTML

```
<body>
    ...

    <script src="{BLAZOR SCRIPT}"></script>
    <script src="{SCRIPT PATH AND FILE NAME (.js)}"></script>
</body>
```

For the placeholders in the preceding example:

- The `{BLAZOR SCRIPT}` placeholder is the Blazor script path and file name. For the location of the script, see ASP.NET Core Blazor project structure.
- The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and script file name under `wwwroot`.

In the following example of the preceding `<script>` tag, the `scripts.js` file is in the `wwwroot/js` folder of the app:

HTML

```
<script src="js/scripts.js"></script>
```

You can also serve scripts directly from the `wwwroot` folder if you prefer not to keep all of your scripts in a separate folder under `wwwroot`:

HTML

```
<script src="scripts.js"></script>
```

When the external JS file is supplied by a Razor class library, specify the JS file using its stable static web asset path: `_content/{PACKAGE ID}/{SCRIPT PATH AND FILE NAME (.js)}`:

- The `{PACKAGE ID}` placeholder is the library's package ID. The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file.
- The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and file name under `wwwroot`.

HTML

```
<body>
    ...

    <script src="{BLAZOR SCRIPT}"></script>
    <script src="_content/{PACKAGE ID}/{SCRIPT PATH AND FILE NAME (.js)}">
</script>
</body>
```

In the following example of the preceding `<script>` tag:

- The Razor class library has an assembly name of `ComponentLibrary`, and a `<PackageId>` isn't specified in the library's project file.
- The `scripts.js` file is in the class library's `wwwroot` folder.

HTML

```
<script src="_content/ComponentLibrary/scripts.js"></script>
```

For more information, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Inject a script before or after Blazor starts

To ensure scripts load before or after Blazor starts, use a JavaScript initializer. For more information and examples, see ASP.NET Core Blazor startup.

JavaScript isolation in JavaScript modules

Blazor enables JavaScript (JS) isolation in standard JS modules (ECMAScript specification).

JS isolation provides the following benefits:

- Imported JS no longer pollutes the global namespace.
- Consumers of a library and components aren't required to import the related JS.

In server-side scenarios, always trap JSDisconnectedException in case loss of Blazor's SignalR circuit prevents a JS interop call from disposing a module, which results in an unhandled exception. Blazor WebAssembly apps don't use a SignalR connection during JS interop, so there's no need to trap JSDisconnectedException in Blazor WebAssembly apps for module disposal.

For more information, see the following resources:

- JavaScript isolation in JavaScript modules
- JavaScript interop calls without a circuit

Call JavaScript functions from .NET methods in ASP.NET Core Blazor

Article • 10/18/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to invoke JavaScript (JS) functions from .NET.

For information on how to call .NET methods from JS, see Call .NET methods from JavaScript functions in ASP.NET Core Blazor.

Invoke JS functions

IJSRuntime is registered by the Blazor framework. To call into JS from .NET, inject the IJSRuntime abstraction and call one of the following methods:

- IJSRuntime.InvokeAsync
- JSRuntimeExtensions.InvokeAsync
- JSRuntimeExtensions.InvokeVoidAsync

For the preceding .NET methods that invoke JS functions:

- The function identifier (`String`) is relative to the global scope (`window`). To call `window.someScope.someFunction`, the identifier is `someScope.someFunction`. There's no need to register the function before it's called.
- Pass any number of JSON-serializable arguments in `Object[]` to a JS function.
- The cancellation token (`CancellationToken`) propagates a notification that operations should be canceled.
- `TimeSpan` represents a time limit for a JS operation.
- The `TValue` return type must also be JSON serializable. `TValue` should match the .NET type that best maps to the JSON type returned.
- A JS Promise is returned for `InvokeAsync` methods. `InvokeAsync` unwraps the Promise and returns the value awaited by the Promise.

For Blazor apps with prerendering enabled, which is the default for server-side apps, calling into JS isn't possible during prerendering. For more information, see the

Prerendering section.

The following example is based on TextDecoder, a JS-based decoder. The example demonstrates how to invoke a JS function from a C# method that offloads a requirement from developer code to an existing JS API. The JS function accepts a byte array from a C# method, decodes the array, and returns the text to the component for display.

HTML

```
<script>
  window.convertArray = (win1251Array) => {
    var win1251decoder = new TextDecoder('windows-1251');
    var bytes = new Uint8Array(win1251Array);
    var decodedArray = win1251decoder.decode(bytes);
    return decodedArray;
  };
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

The following component:

- Invokes the `convertArray` JS function with InvokeAsync when selecting a button (**`Convert Array`**).
- After the JS function is called, the passed array is converted into a string. The string is returned to the component for display (`text`).

`CallJs1.razor`:

razor

```
@page "/call-js-1"
@inject IJSRuntime JS

<PageTitle>Call JS 1</PageTitle>

<h1>Call JS Example 1</h1>

<p>
    <button @onclick="ConvertArray">Convert Array</button>
</p>

<p>
```

```
    @text
</p>

<p>
    Quote ©2005 <a href="https://www.uphe.com">Universal Pictures</a>:
    <a href="https://www.uphe.com/movies/serenity-2005">Serenity</a><br>
    <a href="https://www.imdb.com/name/nm0472710/">David Krumholtz on
IMDB</a>
</p>

@code {
    private MarkupString text;

    private uint[] quoteArray =
        new uint[]
        {
            60, 101, 109, 62, 67, 97, 110, 39, 116, 32, 115, 116, 111, 112,
32,
            116, 104, 101, 32, 115, 105, 103, 110, 97, 108, 44, 32, 77, 97,
            108, 46, 60, 47, 101, 109, 62, 32, 45, 32, 77, 114, 46, 32, 85,
110,
            105, 118, 101, 114, 115, 101, 10, 10,
        };

    private async Task ConvertArray() =>
        text = new(await JS.InvokeAsync<string>("convertArray",
quoteArray));
}
```

JavaScript API restricted to user gestures

This section applies to server-side components.

Some browser JavaScript (JS) APIs can only be executed in the context of a user gesture, such as using the Fullscreen API (MDN documentation). These APIs can't be called through the JS interop mechanism in server-side components because UI event handling is performed asynchronously and generally no longer in the context of the user gesture. The app must handle the UI event completely in JavaScript, so use `onclick` instead of Blazor's `@onclick` directive attribute.

Invoke JavaScript functions without reading a returned value (`InvokeVoidAsync`)

Use InvokeVoidAsync when:

- .NET isn't required to read the result of a JavaScript (JS) call.

- JS functions return void(0)/void 0 or undefined.

Provide a `displayTickerAlert1` JS function. The function is called with InvokeVoidAsync and doesn't return a value:

HTML

```
<script>
  window.displayTickerAlert1 = (symbol, price) => {
    alert(`${symbol}: $${price}!`);
  };
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

Component (`.razor`) example (`InvokeVoidAsync`)

`TickerChanged` calls the `handleTickerChanged1` method in the following component.

`CallJs2.razor`:

razor

```
@page "/call-js-2"
@inject IJSRuntime JS

<PageTitle>Call JS 2</PageTitle>

<h1>Call JS Example 2</h1>

<p>
    <button @onclick="SetStock">Set Stock</button>
</p>

@if (stockSymbol is not null)
{
    <p>@stockSymbol price: @price.ToString("c")</p>
}

@code {
    private string? stockSymbol;
    private decimal price;

    private async Task SetStock()
    {
```

```
            stockSymbol =
                $"{(char)('A' + Random.Shared.Next(0, 26))}" +
                $"{(char)('A' + Random.Shared.Next(0, 26))}";
            price = Random.Shared.Next(1, 101);
            await JS.InvokeVoidAsync("displayTickerAlert1", stockSymbol, price);
        }
    }
```

Class (`.cs`) example (`InvokeVoidAsync`)

`JsInteropClasses1.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;

public class JsInteropClasses1(IJSRuntime js) : IDisposable
{
    private readonly IJSRuntime js = js;

    public async ValueTask TickerChanged(string symbol, decimal price) =>
        await js.InvokeVoidAsync("displayTickerAlert1", symbol, price);

    // Calling SuppressFinalize(this) prevents derived types that introduce
    // a finalizer from needing to re-implement IDisposable.
    public void Dispose() => GC.SuppressFinalize(this);
}
```

`TickerChanged` calls the `handleTickerChanged1` method in the following component.

`CallJs3.razor`:

razor

```
@page "/call-js-3"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call JS 3</PageTitle>

<h1>Call JS Example 3</h1>

<p>
    <button @onclick="SetStock">Set Stock</button>
</p>

@if (stockSymbol is not null)
{
```

```
    <p>@stockSymbol price: @price.ToString("c")</p>
}

@code {
    private string? stockSymbol;
    private decimal price;
    private JsInteropClasses1? jsClass;

    protected override void OnInitialized() => jsClass = new(JS);

    private async Task SetStock()
    {
        if (jsClass is not null)
        {
            stockSymbol =
                $"{(char)('A' + Random.Shared.Next(0, 26))}" +
                $"{(char)('A' + Random.Shared.Next(0, 26))}";
            price = Random.Shared.Next(1, 101);
            await jsClass.TickerChanged(stockSymbol, price);
        }
    }

    public void Dispose() => jsClass?.Dispose();
}
```

Invoke JavaScript functions and read a returned value (`InvokeAsync`)

Use InvokeAsync when .NET should read the result of a JavaScript (JS) call.

Provide a `displayTickerAlert2` JS function. The following example returns a string for display by the caller:

HTML

```
<script>
  window.displayTickerAlert2 = (symbol, price) => {
    if (price < 20) {
      alert(`${symbol}: $${price}!`);
      return "User alerted in the browser.";
    } else {
      return "User NOT alerted.";
    }
  };
</script>
```

> ⓘ Note

For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

Component (`.razor`) example (`InvokeAsync`)

`TickerChanged` calls the `handleTickerChanged2` method and displays the returned string in the following component.

`CallJs4.razor`:

razor

```
@page "/call-js-4"
@inject IJSRuntime JS

<PageTitle>Call JS 4</PageTitle>

<h1>Call JS Example 4</h1>

<p>
    <button @onclick="SetStock">Set Stock</button>
</p>

@if (stockSymbol is not null)
{
    <p>@stockSymbol price: @price.ToString("c")</p>
}

@if (result is not null)
{
    <p>@result</p>
}

@code {
    private string? stockSymbol;
    private decimal price;
    private string? result;

    private async Task SetStock()
    {
        stockSymbol =
            $"{(char)('A' + Random.Shared.Next(0, 26))}" +
            $"{(char)('A' + Random.Shared.Next(0, 26))}";
        price = Random.Shared.Next(1, 101);
        var interopResult =
            await JS.InvokeAsync<string>("displayTickerAlert2", stockSymbol,
price);
        result = $"Result of TickerChanged call for {stockSymbol} at " +
            $"{price.ToString("c")}: {interopResult}";
```

```
    }
}
```

Class (`.cs`) example (`InvokeAsync`)

`JsInteropClasses2.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;

public class JsInteropClasses2(IJSRuntime js) : IDisposable
{
    private readonly IJSRuntime js = js;

    public async ValueTask<string> TickerChanged(string symbol, decimal
price) =>
        await js.InvokeAsync<string>("displayTickerAlert2", symbol, price);

    // Calling SuppressFinalize(this) prevents derived types that introduce
    // a finalizer from needing to re-implement IDisposable.
    public void Dispose() => GC.SuppressFinalize(this);
}
```

`TickerChanged` calls the `handleTickerChanged2` method and displays the returned string in the following component.

`CallJs5.razor`:

razor

```
@page "/call-js-5"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call JS 5</PageTitle>

<h1>Call JS Example 5</h1>

<p>
    <button @onclick="SetStock">Set Stock</button>
</p>

@if (stockSymbol is not null)
{
    <p>@stockSymbol price: @price.ToString("c")</p>
}
```

```razor
@if (result is not null)
{
    <p>@result</p>
}

@code {
    private string? stockSymbol;
    private decimal price;
    private JsInteropClasses2? jsClass;
    private string? result;

    protected override void OnInitialized() => jsClass = new(JS);

    private async Task SetStock()
    {
        if (jsClass is not null)
        {
            stockSymbol =
                $"{(char)('A' + Random.Shared.Next(0, 26))}" +
                $"{(char)('A' + Random.Shared.Next(0, 26))}";
            price = Random.Shared.Next(1, 101);
            var interopResult = await jsClass.TickerChanged(stockSymbol,
price);
            result = $"Result of TickerChanged call for {stockSymbol} at " +
                $"{price.ToString("c")}: {interopResult}";
        }
    }

    public void Dispose() => jsClass?.Dispose();
}
```

Dynamic content generation scenarios

For dynamic content generation with BuildRenderTree, use the `[Inject]` attribute:

razor

```
[Inject]
IJSRuntime JS { get; set; }
```

Prerendering

This section applies to server-side apps that prerender Razor components. Prerendering is covered in Prerender ASP.NET Core Razor components.

ⓘ Note

> Internal navigation for **interactive routing** in Blazor Web Apps doesn't involve requesting new page content from the server. Therefore, prerendering doesn't occur for internal page requests. If the app adopts interactive routing, perform a full page reload for component examples that demonstrate prerendering behavior. For more information, see Prerender ASP.NET Core Razor components.

During prerendering, calling into JavaScript (JS) isn't possible. The following example demonstrates how to use JS interop as part of a component's initialization logic in a way that's compatible with prerendering.

The following `scrollElementIntoView` function:

- Scrolls to the passed element with scrollIntoView.
- Returns the element's `top` property value from the getBoundingClientRect method.

JavaScript

```
window.scrollElementIntoView = (element) => {
  element.scrollIntoView();
  return element.getBoundingClientRect().top;
}
```

Where IJSRuntime.InvokeAsync calls the JS function in component code, the ElementReference is only used in OnAfterRenderAsync and not in any earlier lifecycle method because there's no HTML DOM element until after the component is rendered.

StateHasChanged (reference source) is called to enqueue rerendering of the component with the new state obtained from the JS interop call (for more information, see ASP.NET Core Razor component rendering). An infinite loop isn't created because StateHasChanged is only called when `scrollPosition` is `null`.

`PrerenderedInterop.razor`:

razor

```
@page "/prerendered-interop"
@using Microsoft.AspNetCore.Components
@using Microsoft.JSInterop
@inject IJSRuntime JS

<PageTitle>Prerendered Interop</PageTitle>

<h1>Prerendered Interop Example</h1>

<div @ref="divElement" style="margin-top:2000px">
```

```
    Set value via JS interop call: <strong>@scrollPosition</strong>
</div>

@code {
    private ElementReference divElement;
    private double? scrollPosition;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender && scrollPosition is null)
        {
            scrollPosition = await JS.InvokeAsync<double>(
                "scrollElementIntoView", divElement);

            StateHasChanged();
        }
    }
}
```

The preceding example pollutes the client with a global function. For a better approach in production apps, see JavaScript isolation in JavaScript modules.

Synchronous JS interop in client-side components

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from .NET to JavaScript in a client-side component, cast IJSRuntime to IJSInProcessRuntime to make the JS interop call:

razor

```
@inject IJSRuntime JS

...

@code {
```

```
    protected override void HandleSomeEvent()
    {
        var jsInProcess = (IJSInProcessRuntime)JS;
        var value = jsInProcess.Invoke<string>
("javascriptFunctionIdentifier");
    }
}
```

When working with IJSObjectReference in ASP.NET Core 5.0 or later client-side components, you can use IJSInProcessObjectReference synchronously instead. IJSInProcessObjectReference implements IAsyncDisposable/IDisposable and should be disposed for garbage collection to prevent a memory leak, as the following example demonstrates:

razor

```
@inject IJSRuntime JS
@implements IAsyncDisposable

...

@code {
    ...
    private IJSInProcessObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSInProcessObjectReference>
("import",
            "./scripts.js");
        }
    }

    ...

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
}
```

In the preceding example, a JSDisconnectedException isn't trapped during module disposal because there's no Blazor-SignalR circuit in a Blazor WebAssembly app to lose. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

JavaScript location

Load JavaScript (JS) code using any of approaches described by the article on JavaScript location:

- Load a script in <head> markup (*Not generally recommended*)
- Load a script in <body> markup
- Load a script from an external JavaScript file (.js) collocated with a component
- Load a script from an external JavaScript file (.js)
- Inject a script before or after Blazor starts

For information on isolating scripts in JS modules, see the JavaScript isolation in JavaScript modules section.

> ⚠ **Warning**
>
> Only place a `<script>` tag in a component file (`.razor`) if the component is guaranteed to adopt static server-side rendering (static SSR) because the `<script>` tag can't be updated dynamically.

JavaScript isolation in JavaScript modules

Blazor enables JavaScript (JS) isolation in standard JavaScript modules (ECMAScript specification). JavaScript module loading works the same way in Blazor as it does for other types of web apps, and you're free to customize how modules are defined in your app. For a guide on how to use JavaScript modules, see MDN Web Docs: JavaScript modules.

JS isolation provides the following benefits:

- Imported JS no longer pollutes the global namespace.
- Consumers of a library and components aren't required to import the related JS.

Dynamic import with the import() operator is supported with ASP.NET Core and Blazor:

JavaScript

```
if ({CONDITION}) import("/additionalModule.js");
```

In the preceding example, the `{CONDITION}` placeholder represents a conditional check to determine if the module should be loaded.

For browser compatibility, see Can I use: JavaScript modules: dynamic import.

In server-side scenarios, JS interop calls can't be issued after Blazor's SignalR circuit is disconnected. Without a circuit during component disposal or at any other time that a circuit doesn't exist, the following method calls fail and log a message that the circuit is disconnected as a JSDisconnectedException:

- JS interop method calls
  - IJSRuntime.InvokeAsync
  - JSRuntimeExtensions.InvokeAsync
  - JSRuntimeExtensions.InvokeVoidAsync)
- `Dispose`/`DisposeAsync` calls on any IJSObjectReference.

In order to avoid logging JSDisconnectedException or to log custom information in server-side Blazor, catch the exception in a try-catch statement.

For the following component disposal example:

- The server-side component implements IAsyncDisposable.
- `module` is an IJSObjectReference for a JS module.
- JSDisconnectedException is caught and not logged.
- Optionally, you can log custom information in the `catch` statement at whatever log level you prefer. The following example doesn't log custom information. The code assumes that the developer doesn't care about when or where circuits are disconnected during component disposal.

C#

```
async ValueTask IAsyncDisposable.DisposeAsync()
{
    try
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
    catch (JSDisconnectedException)
    {
    }
}
```

If you must clean up your own JS objects or execute other JS code on the client after a circuit is lost in a server-side Blazor app, use the MutationObserver pattern in JS on the client. The `MutationObserver` pattern allows you to execute JS code when an element is removed from the DOM.

For more information, see the following articles:

- ASP.NET Core Blazor JavaScript interoperability (JS interop): Includes a code example of the `MutationObserver` pattern.
- Handle errors in ASP.NET Core Blazor apps: The *JavaScript interop* section discusses error handling in JS interop scenarios.
- ASP.NET Core Razor component lifecycle: The *Component disposal with `IDisposable` and `IAsyncDisposable`* section describes how to implement disposal patterns in Razor components.

The following JS module exports a JS function for showing a browser window prompt. Place the following JS code in an external JS file.

`wwwroot/scripts.js`:

JavaScript

```
export function showPrompt(message) {
  return prompt(message, 'Type anything here');
}
```

Add the preceding JS module to an app or class library as a static web asset in the `wwwroot` folder and then import the module into the .NET code by calling InvokeAsync on the IJSRuntime instance.

IJSRuntime imports the module as an IJSObjectReference, which represents a reference to a JS object from .NET code. Use the IJSObjectReference to invoke exported JS functions from the module.

`CallJs6.razor`:

razor

```
@page "/call-js-6"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call JS 6</PageTitle>

<h1>Call JS Example 6</h1>

<p>
    <button @onclick="TriggerPrompt">Trigger browser window prompt</button>
</p>

<p>
    @result
```

```
</p>

@code {
    private IJSObjectReference? module;
    private string? result;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./scripts.js");
        }
    }

    private async Task TriggerPrompt() => result = await Prompt("Provide
text");

    public async ValueTask<string?> Prompt(string message) =>
        module is not null ?
            await module.InvokeAsync<string>("showPrompt", message) : null;

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

In the preceding example:

- By convention, the `import` identifier is a special identifier used specifically for importing a JS module.
- Specify the module's external JS file using its stable static web asset path: `./{SCRIPT PATH AND FILE NAME (.js)}`, where:
  - The path segment for the current directory (`./`) is required in order to create the correct static asset path to the JS file.
  - The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and file name under `wwwroot`.
- Disposes the IJSObjectReference for garbage collection in IAsyncDisposable.DisposeAsync.

Dynamically importing a module requires a network request, so it can only be achieved asynchronously by calling InvokeAsync.

`IJSInProcessObjectReference` represents a reference to a JS object whose functions can be invoked synchronously in client-side components. For more information, see the Synchronous JS interop in client-side components section.

> **Note**
>
> When the external JS file is supplied by a Razor class library, specify the module's JS file using its stable static web asset path: `./_content/{PACKAGE ID}/{SCRIPT PATH AND FILE NAME (.js)}`:
>
> - The path segment for the current directory (`./`) is required in order to create the correct static asset path to the JS file.
> - The `{PACKAGE ID}` placeholder is the library's package ID. The package ID defaults to the project's assembly name if `<PackageId>` isn't specified in the project file. In the following example, the library's assembly name is `ComponentLibrary` and the library's project file doesn't specify `<PackageId>`.
> - The `{SCRIPT PATH AND FILE NAME (.js)}` placeholder is the path and file name under `wwwroot`. In the following example, the external JS file (`script.js`) is placed in the class library's `wwwroot` folder.
> - `module` is a private nullable IJSObjectReference of the component class (`private IJSObjectReference? module;`).
>
> C#
>
> ```
> module = await js.InvokeAsync<IJSObjectReference>(
>     "import", "./_content/ComponentLibrary/scripts.js");
> ```
>
> For more information, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Throughout the Blazor documentation, examples use the `.js` file extension for module files, not the newer .mjs file extension (RFC 9239). Our documentation continues to use the `.js` file extension for the same reasons the Mozilla Foundation's documentation continues to use the `.js` file extension. For more information, see Aside — .mjs versus .js (MDN documentation).

Capture references to elements

Some JavaScript (JS) interop scenarios require references to HTML elements. For example, a UI library may require an element reference for initialization, or you might need to call command-like APIs on an element, such as `click` or `play`.

Capture references to HTML elements in a component using the following approach:

- Add an `@ref` attribute to the HTML element.
- Define a field of type ElementReference whose name matches the value of the `@ref` attribute.

The following example shows capturing a reference to the `username` `<input>` element:

razor

```razor
<input @ref="username" ... />

@code {
    private ElementReference username;
}
```

> ⚠ **Warning**
>
> Only use an element reference to mutate the contents of an empty element that doesn't interact with Blazor. This scenario is useful when a third-party API supplies content to the element. Because Blazor doesn't interact with the element, there's no possibility of a conflict between Blazor's representation of the element and the DOM.
>
> In the following example, it's *dangerous* to mutate the contents of the unordered list (`ul`) using `MyList` via JS interop because Blazor interacts with the DOM to populate this element's list items (`<li>`) from the `Todos` object:
>
> razor
>
> ```razor
> <ul @ref="MyList">
>     @foreach (var item in Todos)
>     {
>         <li>@item.Text</li>
>     }
> </ul>
> ```

Using the `MyList` element reference to merely read DOM content or trigger an event is supported.

If JS interop *mutates the contents* of element `MyList` and Blazor attempts to apply diffs to the element, the diffs won't match the DOM. Modifying the contents of the list via JS interop with the `MyList` element reference is ***not supported***.

For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

An ElementReference is passed through to JS code via JS interop. The JS code receives an HTMLElement instance, which it can use with normal DOM APIs. For example, the following code defines a .NET extension method (`TriggerClickEvent`) that enables sending a mouse click to an element.

The JS function `clickElement` creates a click event on the passed HTML element (`element`):

JavaScript

```
window.interopFunctions = {
  clickElement : function (element) {
    element.click();
  }
}
```

To call a JS function that doesn't return a value, use JSRuntimeExtensions.InvokeVoidAsync. The following code triggers a client-side click event by calling the preceding JS function with the captured ElementReference:

razor

```
@inject IJSRuntime JS

<button @ref="exampleButton">Example Button</button>

<button @onclick="TriggerClick">
    Trigger click event on <code>Example Button</code>
</button>

@code {
    private ElementReference exampleButton;

    public async Task TriggerClick()
    {
        await JS.InvokeVoidAsync(
            "interopFunctions.clickElement", exampleButton);
```

```
    }
}
```

To use an extension method, create a static extension method that receives the IJSRuntime instance:

C#

```
public static async Task TriggerClickEvent(this ElementReference elementRef,
    IJSRuntime js)
{
    await js.InvokeVoidAsync("interopFunctions.clickElement", elementRef);
}
```

The `clickElement` method is called directly on the object. The following example assumes that the `TriggerClickEvent` method is available from the `JsInteropClasses` namespace:

razor

```
@inject IJSRuntime JS
@using JsInteropClasses

<button @ref="exampleButton">Example Button</button>

<button @onclick="TriggerClick">
    Trigger click event on <code>Example Button</code>
</button>

@code {
    private ElementReference exampleButton;

    public async Task TriggerClick()
    {
        await exampleButton.TriggerClickEvent(JS);
    }
}
```

ⓘ Important

The `exampleButton` variable is only populated after the component is rendered. If an unpopulated ElementReference is passed to JS code, the JS code receives a value of `null`. To manipulate element references after the component has finished rendering, use the OnAfterRenderAsync or OnAfterRender component lifecycle methods.

When working with generic types and returning a value, use ValueTask<TResult>:

C#

```
public static ValueTask<T> GenericMethod<T>(
        this ElementReference elementRef, IJSRuntime js) =>
    js.InvokeAsync<T>("{JAVASCRIPT FUNCTION}", elementRef);
```

The `{JAVASCRIPT FUNCTION}` placeholder is the JS function identifier.

`GenericMethod` is called directly on the object with a type. The following example assumes that the `GenericMethod` is available from the `JsInteropClasses` namespace:

razor

```
@inject IJSRuntime JS
@using JsInteropClasses

<input @ref="username" />

<button @onclick="OnClickMethod">Do something generic</button>

<p>
    returnValue: @returnValue
</p>

@code {
    private ElementReference username;
    private string? returnValue;

    private async Task OnClickMethod()
    {
        returnValue = await username.GenericMethod<string>(JS);
    }
}
```

Reference elements across components

An ElementReference can't be passed between components because:

- The instance is only guaranteed to exist after the component is rendered, which is during or after a component's OnAfterRender/OnAfterRenderAsync method executes.
- An ElementReference is a struct, which can't be passed as a component parameter.

For a parent component to make an element reference available to other components, the parent component can:

- Allow child components to register callbacks.
- Invoke the registered callbacks during the OnAfterRender event with the passed element reference. Indirectly, this approach allows child components to interact with the parent's element reference.

HTML

```
<style>
    .red { color: red }
</style>
```

HTML

```
<script>
  function setElementClass(element, className) {
    var myElement = element;
    myElement.classList.add(className);
  }
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

`CallJs7.razor` (parent component):

razor

```
@page "/call-js-7"

<PageTitle>Call JS 7</PageTitle>

<h1>Call JS Example 7</h1>

<h2 @ref="title">Hello, world!</h2>

Welcome to your new app.

<SurveyPrompt Parent="this" Title="How is Blazor working for you?" />
```

`CallJs7.razor.cs`:

C#

```
using Microsoft.AspNetCore.Components;

namespace BlazorSample.Pages;

public partial class CallJs7 :
    ComponentBase, IObservable<ElementReference>, IDisposable
{
    private bool disposing;
    private readonly List<IObserver<ElementReference>> subscriptions = [];
    private ElementReference title;

    protected override void OnAfterRender(bool firstRender)
    {
        base.OnAfterRender(firstRender);

        foreach (var subscription in subscriptions)
        {
            subscription.OnNext(title);
        }
    }

    public void Dispose()
    {
        disposing = true;

        foreach (var subscription in subscriptions)
        {
            try
            {
                subscription.OnCompleted();
            }
            catch (Exception)
            {
            }
        }

        subscriptions.Clear();

        // The following prevents derived types that introduce a
        // finalizer from needing to re-implement IDisposable.
        GC.SuppressFinalize(this);
    }

    public IDisposable Subscribe(IObserver<ElementReference> observer)
    {
        if (disposing)
        {
            throw new InvalidOperationException("Parent being disposed");
        }

        subscriptions.Add(observer);

        return new Subscription(observer, this);
    }
```

```
    private class Subscription(IObserver<ElementReference> observer,
        CallJs7 self) : IDisposable
    {
        public IObserver<ElementReference> Observer { get; } = observer;
        public CallJs7 Self { get; } = self;

        public void Dispose() => Self.subscriptions.Remove(Observer);
    }
}
```

In the preceding example, the namespace of the app is `BlazorSample`. If testing the code locally, update the namespace.

`SurveyPrompt.razor` (child component):

razor

```
<div class="alert alert-secondary mt-4">
    <span class="oi oi-pencil me-2" aria-hidden="true"></span>
    <strong>@Title</strong>

    <span class="text-nowrap">
        Please take our
        <a target="_blank" class="font-weight-bold link-dark"
href="https://go.microsoft.com/fwlink/?linkid=2186158">brief survey</a>
    </span>
    and tell us what you think.
</div>

@code {
    // Demonstrates how a parent component can supply parameters
    [Parameter]
    public string? Title { get; set; }
}
```

`SurveyPrompt.razor.cs`:

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.JSInterop;

namespace BlazorSample.Components;

public partial class SurveyPrompt :
    ComponentBase, IObserver<ElementReference>, IDisposable
{
    private IDisposable? subscription = null;

    [Parameter]
```

```
    public IObservable<ElementReference>? Parent { get; set; }

    [Inject]
    public IJSRuntime? JS {get; set;}

    protected override void OnParametersSet()
    {
        base.OnParametersSet();

        subscription?.Dispose();
        subscription = Parent?.Subscribe(this);
    }

    public void OnCompleted() => subscription = null;

    public void OnError(Exception error) => subscription = null;

    public void OnNext(ElementReference value) =>
        _ = (JS?.InvokeAsync<object>("setElementClass", [value, "red"]));

    public void Dispose()
    {
        subscription?.Dispose();

        // The following prevents derived types that introduce a
        // finalizer from needing to re-implement IDisposable.
        GC.SuppressFinalize(this);
    }
}
```

In the preceding example, the namespace of the app is `BlazorSample` with shared components in the `Shared` folder. If testing the code locally, update the namespace.

Harden JavaScript interop calls

This section only applies to Interactive Server components, but client-side components may also set JS interop timeouts if conditions warrant it.

In server-side apps with server interactivity, JavaScript (JS) interop may fail due to networking errors and should be treated as unreliable. Blazor apps use a one minute timeout for JS interop calls. If an app can tolerate a more aggressive timeout, set the timeout using one of the following approaches.

Set a global timeout in the `Program.cs` with CircuitOptions.JSInteropDefaultCallTimeout:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents(options =>
```

```
        options.JSInteropDefaultCallTimeout = {TIMEOUT});
```

The `{TIMEOUT}` placeholder is a TimeSpan (for example, `TimeSpan.FromSeconds(80)`).

Set a per-invocation timeout in component code. The specified timeout overrides the global timeout set by JSInteropDefaultCallTimeout:

```
C#

var result = await JS.InvokeAsync<string>("{ID}", {TIMEOUT}, new[] { "Arg1"
});
```

In the preceding example:

- The `{TIMEOUT}` placeholder is a TimeSpan (for example, `TimeSpan.FromSeconds(80)`).
- The `{ID}` placeholder is the identifier for the function to invoke. For example, the value `someScope.someFunction` invokes the function `window.someScope.someFunction`.

Although a common cause of JS interop failures are network failures with server-side components, per-invocation timeouts can be set for JS interop calls for client-side components. Although no Blazor-SignalR circuit exists for a client-side component, JS interop calls might fail for other reasons that apply.

For more information on resource exhaustion, see Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering.

Avoid circular object references

Objects that contain circular references can't be serialized on the client for either:

- .NET method calls.
- JavaScript method calls from C# when the return type has circular references.

JavaScript libraries that render UI

Sometimes you may wish to use JavaScript (JS) libraries that produce visible user interface elements within the browser DOM. At first glance, this might seem difficult because Blazor's diffing system relies on having control over the tree of DOM elements and runs into errors if some external code mutates the DOM tree and invalidates its mechanism for applying diffs. This isn't a Blazor-specific limitation. The same challenge occurs with any diff-based UI framework.

Fortunately, it's straightforward to embed externally-generated UI within a Razor component UI reliably. The recommended technique is to have the component's code (`.razor` file) produce an empty element. As far as Blazor's diffing system is concerned, the element is always empty, so the renderer does not recurse into the element and instead leaves its contents alone. This makes it safe to populate the element with arbitrary externally-managed content.

The following example demonstrates the concept. Within the `if` statement when `firstRender` is `true`, interact with `unmanagedElement` outside of Blazor using JS interop. For example, call an external JS library to populate the element. Blazor leaves the element's contents alone until this component is removed. When the component is removed, the component's entire DOM subtree is also removed.

razor

```razor
<h1>Hello! This is a Razor component rendered at @DateTime.Now</h1>

<div @ref="unmanagedElement"></div>

@code {
    private ElementReference unmanagedElement;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            ...
        }
    }
}
```

Consider the following example that renders an interactive map using open-source Mapbox APIs.

The following JS module is placed into the app or made available from a Razor class library.

> **Note**
>
> To create the Mapbox map, obtain an access token from Mapbox Sign in and provide it where the `{ACCESS TOKEN}` appears in the following code.

`wwwroot/mapComponent.js`:

JavaScript

```
import 'https://api.mapbox.com/mapbox-gl-js/v1.12.0/mapbox-gl.js';

mapboxgl.accessToken = '{ACCESS TOKEN}';

export function addMapToElement(element) {
  return new mapboxgl.Map({
    container: element,
    style: 'mapbox://styles/mapbox/streets-v11',
    center: [-74.5, 40],
    zoom: 9
  });
}

export function setMapCenter(map, latitude, longitude) {
  map.setCenter([longitude, latitude]);
}
```

To produce correct styling, add the following stylesheet tag to the host HTML page.

Add the following `<link>` element to the `<head>` element markup (location of <head> content):

HTML

```
<link href="https://api.mapbox.com/mapbox-gl-js/v1.12.0/mapbox-gl.css"
    rel="stylesheet" />
```

`CallJs8.razor`:

razor

```
@page "/call-js-8"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call JS 8</PageTitle>

<HeadContent>
    <link href="https://api.mapbox.com/mapbox-gl-js/v1.12.0/mapbox-gl.css"
        rel="stylesheet" />
</HeadContent>

<h1>Call JS Example 8</h1>

<div @ref="mapElement" style='width:400px;height:300px'></div>

<button @onclick="() => ShowAsync(51.454514, -2.587910)">Show Bristol,
UK</button>
<button @onclick="() => ShowAsync(35.6762, 139.6503)">Show Tokyo,
Japan</button>
```

```
@code
{
    private ElementReference mapElement;
    private IJSObjectReference? mapModule;
    private IJSObjectReference? mapInstance;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            mapModule = await JS.InvokeAsync<IJSObjectReference>(
                "import", "./mapComponent.js");
            mapInstance = await mapModule.InvokeAsync<IJSObjectReference>(
                "addMapToElement", mapElement);
        }
    }

    private async Task ShowAsync(double latitude, double longitude)
    {
        if (mapModule is not null && mapInstance is not null)
        {
            await mapModule.InvokeVoidAsync("setMapCenter", mapInstance,
                latitude, longitude);
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (mapInstance is not null)
        {
            try
            {
                await mapInstance.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }

        if (mapModule is not null)
        {
            try
            {
                await mapModule.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

The preceding example produces an interactive map UI. The user:

- Can drag to scroll or zoom.
- Select buttons to jump to predefined locations.



In the preceding example:

- The `<div>` with `@ref="mapElement"` is left empty as far as Blazor is concerned. The `mapbox-gl.js` script can safely populate the element and modify its contents. Use this technique with any JS library that renders UI. You can embed components from a third-party JS SPA framework inside Razor components, as long as they don't try to reach out and modify other parts of the page. It is **not** safe for external JS code to modify elements that Blazor does not regard as empty.
- When using this approach, bear in mind the rules about how Blazor retains or destroys DOM elements. The component safely handles button click events and updates the existing map instance because DOM elements are retained where possible. If you were rendering a list of map elements from inside a `@foreach` loop, you want to use `@key` to ensure the preservation of component instances. Otherwise, changes in the list data could cause component instances to retain the state of previous instances in an undesirable manner. For more information, see how to use the @key directive attribute to preserve the relationship among elements, components, and model objects.

- The example encapsulates JS logic and dependencies within a JavaScript module and loads the module dynamically using the `import` identifier. For more information, see JavaScript isolation in JavaScript modules.

Byte array support

Blazor supports optimized byte array JavaScript (JS) interop that avoids encoding/decoding byte arrays into Base64. The following example uses JS interop to pass a byte array to JavaScript.

Provide a `receiveByteArray` JS function. The function is called with InvokeVoidAsync and doesn't return a value:

HTML

```
<script>
  window.receiveByteArray = (bytes) => {
    let utf8decoder = new TextDecoder();
    let str = utf8decoder.decode(bytes);
    return str;
  };
</script>
```

> ⓘ Note
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

`CallJs9.razor`:

razor

```
@page "/call-js-9"
@inject IJSRuntime JS

<h1>Call JS Example 9</h1>

<p>
    <button @onclick="SendByteArray">Send Bytes</button>
</p>

<p>
    @result
</p>

<p>
```

```
    Quote &copy;2005 <a href="https://www.uphe.com">Universal Pictures</a>:
    <a href="https://www.uphe.com/movies/serenity-2005">Serenity</a><br>
    <a href="https://www.imdb.com/name/nm0821612/">Jewel Staite on IMDB</a>
</p>

@code {
    private string? result;

    private async Task SendByteArray()
    {
        var bytes = new byte[] { 0x45, 0x76, 0x65, 0x72, 0x79, 0x74, 0x68,
0x69,
            0x6e, 0x67, 0x27, 0x73, 0x20, 0x73, 0x68, 0x69, 0x6e, 0x79,
0x2c,
            0x20, 0x43, 0x61, 0x70, 0x74, 0x69, 0x61, 0x6e, 0x2e, 0x20,
0x4e,
            0x6f, 0x74, 0x20, 0x74, 0x6f, 0x20, 0x66, 0x72, 0x65, 0x74, 0x2e
};

        result = await JS.InvokeAsync<string>("receiveByteArray", bytes);
    }
}
```

Stream from .NET to JavaScript

Blazor supports streaming data directly from .NET to JavaScript (JS). Streams are created using a DotNetStreamReference.

DotNetStreamReference represents a .NET stream and uses the following parameters:

- `stream`: The stream sent to JS.
- `leaveOpen`: Determines if the stream is left open after transmission. If a value isn't provided, `leaveOpen` defaults to `false`.

In JS, use an array buffer or a readable stream to receive the data:

- Using an ArrayBuffer:

  JavaScript

  ```
  async function streamToJavaScript(streamRef) {
    const data = await streamRef.arrayBuffer();
  }
  ```

- Using a ReadableStream:

  JavaScript

```
async function streamToJavaScript(streamRef) {
  const stream = await streamRef.stream();
}
```

In C# code:

C#

```
var streamRef = new DotNetStreamReference(stream: {STREAM}, leaveOpen:
false);
await JS.InvokeVoidAsync("streamToJavaScript", streamRef);
```

In the preceding example:

- The `{STREAM}` placeholder represents the Stream sent to JS.
- `JS` is an injected IJSRuntime instance.

Disposing a DotNetStreamReference instance is usually unnecessary. When `leaveOpen` is set to its default value of `false`, the underlying Stream is automatically disposed after transmission to JS.

If `leaveOpen` is `true`, then disposing a DotNetStreamReference doesn't dispose its underlying Stream. The app's code determines when to dispose the underlying Stream. When deciding how to dispose the underlying Stream, consider the following:

- Disposing a Stream while it's being transmitted to JS is considered an application error and may cause an unhandled exception to occur.
- Stream transmission begins as soon as the DotNetStreamReference is passed as an argument to a JS interop call, regardless of whether the stream is actually used in JS logic.

Given these characteristics, we recommend disposing the underlying Stream only after it's fully consumed by JS (the promise returned by `arrayBuffer` or `stream` resolves). It follows that a DotNetStreamReference should only be passed to JS if it's unconditionally going to be consumed by JS logic.

Call .NET methods from JavaScript functions in ASP.NET Core Blazor covers the reverse operation, streaming from JavaScript to .NET.

ASP.NET Core Blazor file downloads covers how to download a file in Blazor.

Catch JavaScript exceptions

To catch JS exceptions, wrap the JS interop in a try-catch block and catch a JSException.

In the following example, the `nonFunction` JS function doesn't exist. When the function isn't found, the JSException is trapped with a Message that indicates the following error:

> `Could not find 'nonFunction' ('nonFunction' was undefined).`

`CallJs11.razor`:

razor

```razor
@page "/call-js-11"
@inject IJSRuntime JS

<PageTitle>Call JS 11</PageTitle>

<h1>Call JS Example 11</h1>

<p>
    <button @onclick="CatchUndefinedJSFunction">Catch Exception</button>
</p>

<p>
    @result
</p>

<p>
    @errorMessage
</p>

@code {
    private string? errorMessage;
    private string? result;

    private async Task CatchUndefinedJSFunction()
    {
        try
        {
            result = await JS.InvokeAsync<string>("nonFunction");
        }
        catch (JSException e)
        {
            errorMessage = $"Error Message: {e.Message}";
        }
    }
}
```

Abort a long-running JavaScript function

Use a JS AbortController with a CancellationTokenSource in the component to abort a long-running JavaScript function from C# code.

The following JS `Helpers` class contains a simulated long-running function, `longRunningFn`, to count continuously until the AbortController.signal indicates that AbortController.abort has been called. The `sleep` function is for demonstration purposes to simulate slow execution of the long-running function and wouldn't be present in production code. When a component calls `stopFn`, the `longRunningFn` is signalled to abort via the `while` loop conditional check on AbortSignal.aborted.

HTML

```
<script>
  class Helpers {
    static #controller = new AbortController();

    static async #sleep(ms) {
      return new Promise(resolve => setTimeout(resolve, ms));
    }

    static async longRunningFn() {
      var i = 0;
      while (!this.#controller.signal.aborted) {
        i++;
        console.log(`longRunningFn: ${i}`);
        await this.#sleep(1000);
      }
    }

    static stopFn() {
      this.#controller.abort();
      console.log('longRunningFn aborted!');
    }
  }

  window.Helpers = Helpers;
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

The following component:

- Invokes the JS function `longRunningFn` when the **`Start Task`** button is selected. A CancellationTokenSource is used to manage the execution of the long-running

function. CancellationToken.Register sets a JS interop call delegate to execute the JS function `stopFn` when the CancellationTokenSource.Token is cancelled.

- When the **`Cancel Task`** button is selected, the CancellationTokenSource.Token is cancelled with a call to Cancel.
- The CancellationTokenSource is disposed in the `Dispose` method.

`CallJs12.razor`:

razor

```
@page "/call-js-12"
@inject IJSRuntime JS

<h1>Cancel long-running JS interop</h1>

<p>
    <button @onclick="StartTask">Start Task</button>
    <button @onclick="CancelTask">Cancel Task</button>
</p>

@code {
    private CancellationTokenSource? cts;

    private async Task StartTask()
    {
        cts = new CancellationTokenSource();
        cts.Token.Register(() => JS.InvokeVoidAsync("Helpers.stopFn"));

        await JS.InvokeVoidAsync("Helpers.longRunningFn");
    }

    private void CancelTask()
    {
        cts?.Cancel();
    }

    public void Dispose()
    {
        cts?.Cancel();
        cts?.Dispose();
    }
}
```

A browser's developer tools console indicates the execution of the long-running JS function after the **`Start Task`** button is selected and when the function is aborted after the **`Cancel Task`** button is selected:

Console

```
longRunningFn: 1
longRunningFn: 2
longRunningFn: 3
longRunningFn aborted!
```

JavaScript `[JSImport]`/`[JSExport]` interop

This section applies to client-side components.

As an alternative to interacting with JavaScript (JS) in client-side components using Blazor's JS interop mechanism based on the IJSRuntime interface, a JS `[JSImport]`/`[JSExport]` interop API is available to apps targeting .NET 7 or later.

For more information, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

Unmarshalled JavaScript interop

This section applies to client-side components.

Unmarshalled interop using the IJSUnmarshalledRuntime interface is obsolete and should be replaced with JavaScript `[JSImport]`/`[JSExport]` interop.

For more information, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

Disposal of JavaScript interop object references

Examples throughout the JavaScript (JS) interop articles demonstrate typical object disposal patterns:

- When calling JS from .NET, as described in this article, dispose any created IJSObjectReference/IJSInProcessObjectReference/`JSObjectReference` either from .NET or from JS to avoid leaking JS memory.
- When calling .NET from JS, as described in Call .NET methods from JavaScript functions in ASP.NET Core Blazor, dispose of a created DotNetObjectReference either from .NET or from JS to avoid leaking .NET memory.

JS interop object references are implemented as a map keyed by an identifier on the side of the JS interop call that creates the reference. When object disposal is initiated

from either the .NET or JS side, Blazor removes the entry from the map, and the object can be garbage collected as long as no other strong reference to the object is present.

At a minimum, always dispose objects created on the .NET side to avoid leaking .NET managed memory.

DOM cleanup tasks during component disposal

For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

JavaScript interop calls without a circuit

For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

Additional resources

- Call .NET methods from JavaScript functions in ASP.NET Core Blazor
- InteropComponent.razor example (dotnet/AspNetCore GitHub repository main branch) : The `main` branch represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release (for example, `release/{VERSION}`, where the `{VERSION}` placeholder is the release version), use the **Switch branches or tags** dropdown list to select the branch. For a branch that no longer exists, use the **Tags** tab to find the API (for example, `v7.0.0`).
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)
- Handle errors in ASP.NET Core Blazor apps (*JavaScript interop* section)
- Threat mitigation: JavaScript functions invoked from .NET

Call .NET methods from JavaScript functions in ASP.NET Core Blazor

Article • 12/17/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to invoke .NET methods from JavaScript (JS).

For information on how to call JS functions from .NET, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Invoke a static .NET method

To invoke a static .NET method from JavaScript (JS), use the JS functions:

- `DotNet.invokeMethodAsync` (*recommended*): Asynchronous for both server-side and client-side components.
- `DotNet.invokeMethod`: Synchronous for client-side components only.

Pass in the name of the assembly containing the method, the identifier of the static .NET method, and any arguments.

In the following example:

- The `{ASSEMBLY NAME}` placeholder is the app's assembly name.
- The `{.NET METHOD ID}` placeholder is the .NET method identifier.
- The `{ARGUMENTS}` placeholder are optional, comma-separated arguments to pass to the method, each of which must be JSON-serializable.

JavaScript

```
DotNet.invokeMethodAsync('{ASSEMBLY NAME}', '{.NET METHOD ID}',
{ARGUMENTS});
```

`DotNet.invokeMethodAsync` returns a JS Promise representing the result of the operation. `DotNet.invokeMethod` (client-side components) returns the result of the operation.

> **Important**
>
> For server-side components, we recommend the asynchronous function (`invokeMethodAsync`) over the synchronous version (`invokeMethod`).

The .NET method must be public, static, and have the [JSInvokable] attribute.

In the following example:

- The `{<T>}` placeholder indicates the return type, which is only required for methods that return a value.
- The `{.NET METHOD ID}` placeholder is the method identifier.

razor

```
@code {
    [JSInvokable]
    public static Task{<T>} {.NET METHOD ID}()
    {
        ...
    }
}
```

> **Note**
>
> Calling open generic methods isn't supported with static .NET methods but is supported with instance methods. For more information, see the Call .NET generic class methods section.

In the following component, the `ReturnArrayAsync` C# method returns an `int` array. The [JSInvokable] attribute is applied to the method, which makes the method invokable by JS.

`CallDotnet1.razor`:

razor

```
@page "/call-dotnet-1"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET 1</PageTitle>

<h1>Call .NET Example 1</h1>
```

```razor
<p>
    <button id="btn">Trigger .NET static method</button>
</p>

<p>
    See the result in the developer tools console.
</p>

@code {
    private IJSObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/Pages/CallDotnet1.razor.js");

            await module.InvokeVoidAsync("addHandlers");
        }
    }

    [JSInvokable]
    public static Task<int[]> ReturnArrayAsync() =>
        Task.FromResult(new int[] { 11, 12, 13 });

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

`CallDotnet1.razor.js`:

JavaScript

```javascript
export function returnArrayAsync() {
  DotNet.invokeMethodAsync('BlazorSample', 'ReturnArrayAsync')
    .then(data => {
      console.log(data);
    });
}

export function addHandlers() {
```

```
  const btn = document.getElementById("btn");
  btn.addEventListener("click", returnArrayAsync);
}
```

The `addHandlers` JS function adds a click event to the button. The `returnArrayAsync` JS function is assigned as the handler.

The `returnArrayAsync` JS function calls the `ReturnArrayAsync` .NET method of the component, which logs the result to the browser's web developer tools console. `BlazorSample` is the app's assembly name.

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

When the **`Trigger .NET static method`** button is selected, the browser's developer tools console output displays the array data. The format of the output differs slightly among browsers. The following output shows the format used by Microsoft Edge:

Console

```
Array(3) [ 11, 12, 13 ]
```

Pass data to a .NET method when calling the `invokeMethodAsync` function by passing the data as arguments.

To demonstrate passing data to .NET, pass a starting position to the `ReturnArrayAsync` method where the method is invoked in JS:

JavaScript

```
export function returnArrayAsync() {
  DotNet.invokeMethodAsync('BlazorSample', 'ReturnArrayAsync', 14)
    .then(data => {
      console.log(data);
    });
}
```

The component's invokable `ReturnArrayAsync` method receives the starting position and constructs the array from it. The array is returned for logging to the console:

C#

```
[JSInvokable]
public static Task<int[]> ReturnArrayAsync(int startPosition) =>
    Task.FromResult(Enumerable.Range(startPosition, 3).ToArray());
```

After the app is recompiled and the browser is refreshed, the following output appears in the browser's console when the button is selected:

Console

```
Array(3) [ 14, 15, 16 ]
```

The .NET method identifier for the JS call is the .NET method name, but you can specify a different identifier using the [JSInvokable] attribute constructor. In the following example, `DifferentMethodName` is the assigned method identifier for the `ReturnArrayAsync` method:

C#

```
[JSInvokable("DifferentMethodName")]
```

In the call to `DotNet.invokeMethodAsync` (server-side or client-side components) or `DotNet.invokeMethod` (client-side components only), call `DifferentMethodName` to execute the `ReturnArrayAsync` .NET method:

- `DotNet.invokeMethodAsync('BlazorSample', 'DifferentMethodName');`
- `DotNet.invokeMethod('BlazorSample', 'DifferentMethodName');` (client-side components only)

> **Note**
>
> The `ReturnArrayAsync` method example in this section returns the result of a Task without the use of explicit C# async and await keywords. Coding methods with async and await is typical of methods that use the await keyword to return the value of asynchronous operations.
>
> `ReturnArrayAsync` method composed with async and await keywords:
>
> C#
>
> ```
> [JSInvokable]
> public static async Task<int[]> ReturnArrayAsync() =>
>     await Task.FromResult(new int[] { 11, 12, 13 });
> ```

For more information, see Asynchronous programming with async and await in the C# guide.

Create JavaScript object and data references to pass to .NET

Call `DotNet.createJSObjectReference(jsObject)` to construct a JS object reference so that it can be passed to .NET, where `jsObject` is the JS Object used to create the JS object reference. The following example passes a reference to the non-serializable `window` object to .NET, which receives it in the `ReceiveWindowObject` C# method as an IJSObjectReference:

JavaScript

```
DotNet.invokeMethodAsync('{ASSEMBLY NAME}', 'ReceiveWindowObject',
  DotNet.createJSObjectReference(window));
```

C#

```
[JSInvokable]
public static void ReceiveWindowObject(IJSObjectReference objRef)
{
    ...
}
```

In the preceding example, the `{ASSEMBLY NAME}` placeholder is the app's namespace.

> **Note**
>
> The preceding example doesn't require disposal of the `JSObjectReference`, as a reference to the `window` object isn't held in JS.

Maintaining a reference to a `JSObjectReference` requires disposing of it to avoid leaking JS memory on the client. The following example refactors the preceding code to capture a reference to the `JSObjectReference`, followed by a call to `DotNet.disposeJSObjectReference()` to dispose of the reference:

JavaScript

```
var jsObjectReference = DotNet.createJSObjectReference(window);
```

```
DotNet.invokeMethodAsync('{ASSEMBLY NAME}', 'ReceiveWindowObject',
jsObjectReference);

DotNet.disposeJSObjectReference(jsObjectReference);
```

In the preceding example, the `{ASSEMBLY NAME}` placeholder is the app's namespace.

Call `DotNet.createJSStreamReference(streamReference)` to construct a JS stream reference so that it can be passed to .NET, where `streamReference` is an ArrayBuffer, Blob, or any typed array, such as Uint8Array or Float32Array, used to create the JS stream reference.

Invoke an instance .NET method

To invoke an instance .NET method from JavaScript (JS):

- Pass the .NET instance by reference to JS by wrapping the instance in a DotNetObjectReference and calling Create on it.
- Invoke a .NET instance method from JS using `invokeMethodAsync` (*recommended*) or `invokeMethod` (client-side components only) from the passed DotNetObjectReference. Pass the identifier of the instance .NET method and any arguments. The .NET instance can also be passed as an argument when invoking other .NET methods from JS.

  In the following example:
  - `dotNetHelper` is a DotNetObjectReference.
  - The `{.NET METHOD ID}` placeholder is the .NET method identifier.
  - The `{ARGUMENTS}` placeholder are optional, comma-separated arguments to pass to the method, each of which must be JSON-serializable.

  JavaScript

  ```
  dotNetHelper.invokeMethodAsync('{.NET METHOD ID}', {ARGUMENTS});
  ```

  > **Note**
  >
  > `invokeMethodAsync` and `invokeMethod` don't accept an assembly name parameter when invoking an instance method.

`invokeMethodAsync` returns a JS Promise representing the result of the operation.
`invokeMethod` (client-side components only) returns the result of the operation.

> **Important**
>
> For server-side components, we recommend the asynchronous function (`invokeMethodAsync`) over the synchronous version (`invokeMethod`).

- Dispose of the DotNetObjectReference.

The following sections of this article demonstrate various approaches for invoking an instance .NET method:

- Pass a DotNetObjectReference to an individual JavaScript function
- Pass a DotNetObjectReference to a class with multiple JavaScript functions
- Call .NET generic class methods
- Class instance examples
- Component instance .NET method helper class

Avoid trimming JavaScript-invokable .NET methods

This section applies to client-side apps with ahead-of-time (AOT) compilation and runtime relinking enabled.

Several of the examples in the following sections are based on a class instance approach, where the JavaScript-invokable .NET method marked with the [JSInvokable] attribute is a member of a class that isn't a Razor component. When such .NET methods are located in a Razor component, they're protected from runtime relinking/trimming. In order to protect the .NET methods from trimming outside of Razor components, implement the methods with the DynamicDependency attribute on the class's constructor, as the following example demonstrates:

C#

```
using System.Diagnostics.CodeAnalysis;
using Microsoft.JSInterop;

public class ExampleClass {

    [DynamicDependency(nameof(ExampleJSInvokableMethod))]
    public ExampleClass()
    {
```

```
    }

    [JSInvokable]
    public string ExampleJSInvokableMethod()
    {
        ...
    }
}
```

For more information, see Prepare .NET libraries for trimming: DynamicDependency.

Pass a `DotNetObjectReference` to an individual JavaScript function

The example in this section demonstrates how to pass a DotNetObjectReference to an individual JavaScript (JS) function.

The following `sayHello1` JS function receives a DotNetObjectReference and calls `invokeMethodAsync` to call the `GetHelloMessage` .NET method of a component:

HTML

```
<script>
  window.sayHello1 = (dotNetHelper) => {
    return dotNetHelper.invokeMethodAsync('GetHelloMessage');
  };
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

For the following component:

- The component has a JS-invokable .NET method named `GetHelloMessage`.
- When the **`Trigger .NET instance method`** button is selected, the JS function `sayHello1` is called with the DotNetObjectReference.
- `sayHello1`:
  - Calls `GetHelloMessage` and receives the message result.

    - Returns the message result to the calling `TriggerDotNetInstanceMethod` method.
- The returned message from `sayHello1` in `result` is displayed to the user.
- To avoid a memory leak and allow garbage collection, the .NET object reference created by DotNetObjectReference is disposed in the `Dispose` method.

`CallDotnet2.razor`:

razor

```razor
@page "/call-dotnet-2"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET 2</PageTitle>

<h1>Call .NET Example 2</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>

<p>
    @result
</p>

@code {
    private string? name;
    private string? result;
    private DotNetObjectReference<CallDotnet2>? objRef;

    protected override void OnInitialized() =>
        objRef = DotNetObjectReference.Create(this);

    public async Task TriggerDotNetInstanceMethod() =>
        result = await JS.InvokeAsync<string>("sayHello1", objRef);

    [JSInvokable]
    public string GetHelloMessage() => $"Hello, {name}!";

    public void Dispose() => objRef?.Dispose();
}
```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

Use the following guidance to pass arguments to an instance method:

Add parameters to the .NET method invocation. In the following example, a name is passed to the method. Add additional parameters to the list as needed.

HTML

```
<script>
  window.sayHello2 = (dotNetHelper, name) => {
    return dotNetHelper.invokeMethodAsync('GetHelloMessage', name);
  };
</script>
```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

Provide the parameter list to the .NET method.

`CallDotnet3.razor`:

razor

```
@page "/call-dotnet-3"
@implements IDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET 3</PageTitle>

<h1>Call .NET Example 3</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>

<p>
    @result
</p>

@code {
```

```
    private string? name;
    private string? result;
    private DotNetObjectReference<CallDotnet3>? objRef;

    protected override void OnInitialized() =>
        objRef = DotNetObjectReference.Create(this);

    public async Task TriggerDotNetInstanceMethod() =>
        result = await JS.InvokeAsync<string>("sayHello2", objRef, name);

    [JSInvokable]
    public string GetHelloMessage(string passedName) => $"Hello,
{passedName}!";

    public void Dispose() => objRef?.Dispose();
}
```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

Pass a `DotNetObjectReference` to a class with multiple JavaScript functions

The example in this section demonstrates how to pass a DotNetObjectReference to a JavaScript (JS) class with multiple functions.

Create and pass a DotNetObjectReference from the OnAfterRenderAsync lifecycle method to a JS class for multiple functions to use. Make sure that the .NET code disposes of the DotNetObjectReference, as the following example shows.

In the following component, the `Trigger JS function` buttons call JS functions by setting the JS `onclick` property, not Blazor's `@onclick` directive attribute.

`CallDotNetExampleOneHelper.razor`:

razor

```
@page "/call-dotnet-example-one-helper"
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET Example</PageTitle>

<h1>Pass <code>DotNetObjectReference</code> to a JavaScript class</h1>

<p>
    <label>
        Message: <input @bind="name" />
```

```
    </label>
</p>

<p>
    <button id="sayHelloBtn">
        Trigger JS function <code>sayHello</code>
    </button>
</p>

<p>
    <button id="welcomeVisitorBtn">
        Trigger JS function <code>welcomeVisitor</code>
    </button>
</p>

@code {
    private IJSObjectReference? module;
    private string? name;
    private DotNetObjectReference<CallDotNetExampleOneHelper>? dotNetHelper;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/Pages/CallDotNetExampleOneHelper.razor.js");

            dotNetHelper = DotNetObjectReference.Create(this);
            await module.InvokeVoidAsync("GreetingHelpers.setDotNetHelper",
                dotNetHelper);

            await module.InvokeVoidAsync("addHandlers");
        }
    }

    [JSInvokable]
    public string GetHelloMessage() => $"Hello, {name}!";

    [JSInvokable]
    public string GetWelcomeMessage() => $"Welcome, {name}!";

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }

        dotNetHelper?.Dispose();
```

```
    }
}
```

In the preceding example:

- `JS` is an injected IJSRuntime instance. IJSRuntime is registered by the Blazor framework.
- The variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.
- The component must explicitly dispose of the DotNetObjectReference to permit garbage collection and prevent a memory leak.
- JSDisconnectedException is trapped during module disposal in case Blazor's SignalR circuit is lost. If the preceding code is used in a Blazor WebAssembly app, there's no SignalR connection to lose, so you can remove the `try`-`catch` block and leave the line that disposes the module (`await module.DisposeAsync();`). For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

`CallDotNetExampleOneHelper.razor.js`:

JavaScript

```
export class GreetingHelpers {
  static dotNetHelper;

  static setDotNetHelper(value) {
    GreetingHelpers.dotNetHelper = value;
  }

  static async sayHello() {
    const msg =
      await
GreetingHelpers.dotNetHelper.invokeMethodAsync('GetHelloMessage');
    alert(`Message from .NET: "${msg}"`);
  }

  static async welcomeVisitor() {
    const msg =
      await
GreetingHelpers.dotNetHelper.invokeMethodAsync('GetWelcomeMessage');
    alert(`Message from .NET: "${msg}"`);
  }
}

export function addHandlers() {
  const sayHelloBtn = document.getElementById("sayHelloBtn");
  sayHelloBtn.addEventListener("click", GreetingHelpers.sayHello);

  const welcomeVisitorBtn = document.getElementById("welcomeVisitorBtn");
  welcomeVisitorBtn.addEventListener("click",
```

```
GreetingHelpers.welcomeVisitor);
}
```

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

> ⓘ Note
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

Call .NET generic class methods

JavaScript (JS) functions can call .NET generic class methods, where a JS function calls a .NET method of a generic class.

In the following generic type class (`GenericType<TValue>`):

- The class has a single type parameter (`TValue`) with a single generic `Value` property.
- The class has two non-generic methods marked with the [JSInvokable] attribute, each with a generic type parameter named `newValue`:
  - `Update` synchronously updates the value of `Value` from `newValue`.
  - `UpdateAsync` asynchronously updates the value of `Value` from `newValue` after creating an awaitable task with Task.Yield that asynchronously yields back to the current context when awaited.
- Each of the class methods write the type of `TValue` and the value of `Value` to the console. Writing to the console is only for demonstration purposes. Production apps usually avoid writing to the console in favor of app *logging*. For more information, see ASP.NET Core Blazor logging and Logging in .NET Core and ASP.NET Core.

> ⓘ Note
>
> *Open generic types and methods* don't specify types for type placeholders. Conversely, *closed generics* supply types for all type placeholders. The examples in this section demonstrate closed generics, but invoking JS interop *instance methods* with open generics ***is supported***. Use of open generics isn't supported for static .NET method invocations, which were described earlier in this article.

For more information, see the following articles:

- Generic classes and methods (C# documentation)
- Generic Classes (C# Programming Guide)
- Generics in .NET (.NET documentation)

`GenericType.cs`:

C#

```
using Microsoft.JSInterop;

public class GenericType<TValue>
{
    public TValue? Value { get; set; }

    [JSInvokable]
    public void Update(TValue newValue)
    {
        Value = newValue;

        Console.WriteLine($"Update: GenericType<{typeof(TValue)}>:
{Value}");
    }

    [JSInvokable]
    public async Task UpdateAsync(TValue newValue)
    {
        await Task.Yield();
        Value = newValue;

        Console.WriteLine($"UpdateAsync: GenericType<{typeof(TValue)}>:
{Value}");
    }
}
```

In the following `invokeMethodsAsync` function:

- The generic type class's `Update` and `UpdateAsync` methods are called with arguments representing strings and numbers.
- Client-side components support calling .NET methods synchronously with `invokeMethod`. `syncInterop` receives a boolean value indicating if the JS interop is occurring on the client. When `syncInterop` is `true`, `invokeMethod` is safely called. If the value of `syncInterop` is `false`, only the asynchronous function `invokeMethodAsync` is called because the JS interop is executing in a server-side component.
- For demonstration purposes, the DotNetObjectReference function call (`invokeMethod` or `invokeMethodAsync`), the .NET method called (`Update` or

`UpdateAsync`), and the argument are written to the console. The arguments use a random number to permit matching the JS function call to the .NET method invocation (also written to the console on the .NET side). Production code usually doesn't write to the console, either on the client or the server. Production apps usually rely upon app *logging*. For more information, see ASP.NET Core Blazor logging and Logging in .NET Core and ASP.NET Core.

HTML

```
<script>
  const randomInt = () => Math.floor(Math.random() * 99999);

  window.invokeMethodsAsync = async (syncInterop, dotNetHelper1,
dotNetHelper2) => {
    var n = randomInt();
    console.log(`JS: invokeMethodAsync:Update('string ${n}')`);
    await dotNetHelper1.invokeMethodAsync('Update', `string ${n}`);

    n = randomInt();
    console.log(`JS: invokeMethodAsync:UpdateAsync('string ${n}')`);
    await dotNetHelper1.invokeMethodAsync('UpdateAsync', `string ${n}`);

    if (syncInterop) {
      n = randomInt();
      console.log(`JS: invokeMethod:Update('string ${n}')`);
      dotNetHelper1.invokeMethod('Update', `string ${n}`);
    }

    n = randomInt();
    console.log(`JS: invokeMethodAsync:Update(${n})`);
    await dotNetHelper2.invokeMethodAsync('Update', n);

    n = randomInt();
    console.log(`JS: invokeMethodAsync:UpdateAsync(${n})`);
    await dotNetHelper2.invokeMethodAsync('UpdateAsync', n);

    if (syncInterop) {
      n = randomInt();
      console.log(`JS: invokeMethod:Update(${n})`);
      dotNetHelper2.invokeMethod('Update', n);
    }
  };
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

In the following `GenericsExample` component:

- The JS function `invokeMethodsAsync` is called when the **`Invoke Interop`** button is selected.
- A pair of DotNetObjectReference types are created and passed to the JS function for instances of the `GenericType` as a `string` and an `int`.

`GenericsExample.razor`:

razor

```razor
@page "/generics-example"
@implements IDisposable
@inject IJSRuntime JS

<p>
    <button @onclick="InvokeInterop">Invoke Interop</button>
</p>

<ul>
    <li>genericType1: @genericType1?.Value</li>
    <li>genericType2: @genericType2?.Value</li>
</ul>

@code {
    private GenericType<string> genericType1 = new() { Value = "string 0" };
    private GenericType<int> genericType2 = new() { Value = 0 };
    private DotNetObjectReference<GenericType<string>>? objRef1;
    private DotNetObjectReference<GenericType<int>>? objRef2;

    protected override void OnInitialized()
    {
        objRef1 = DotNetObjectReference.Create(genericType1);
        objRef2 = DotNetObjectReference.Create(genericType2);
    }

    public async Task InvokeInterop()
    {
        var syncInterop = OperatingSystem.IsBrowser();

        await JS.InvokeVoidAsync(
            "invokeMethodsAsync", syncInterop, objRef1, objRef2);
    }

    public void Dispose()
    {
        objRef1?.Dispose();
        objRef2?.Dispose();
    }
}
```

In the preceding example, `JS` is an injected IJSRuntime instance. IJSRuntime is registered by the Blazor framework.

The following demonstrates typical output of the preceding example when the **`Invoke Interop`** button is selected in a client-side component:

```
JS: invokeMethodAsync:Update('string 37802')
.NET: Update: GenericType<System.String>: string 37802
JS: invokeMethodAsync:UpdateAsync('string 53051')
JS: invokeMethod:Update('string 26784')
.NET: Update: GenericType<System.String>: string 26784
JS: invokeMethodAsync:Update(14107)
.NET: Update: GenericType<System.Int32>: 14107
JS: invokeMethodAsync:UpdateAsync(48995)
JS: invokeMethod:Update(12872)
.NET: Update: GenericType<System.Int32>: 12872
.NET: UpdateAsync: GenericType<System.String>: string 53051
.NET: UpdateAsync: GenericType<System.Int32>: 48995
```

If the preceding example is implemented in a server-side component, the synchronous calls with `invokeMethod` are avoided. For server-side components, we recommend the asynchronous function (`invokeMethodAsync`) over the synchronous version (`invokeMethod`).

Typical output of a server-side component:

```
JS: invokeMethodAsync:Update('string 34809')
.NET: Update: GenericType<System.String>: string 34809
JS: invokeMethodAsync:UpdateAsync('string 93059')
JS: invokeMethodAsync:Update(41997)
.NET: Update: GenericType<System.Int32>: 41997
JS: invokeMethodAsync:UpdateAsync(24652)
.NET: UpdateAsync: GenericType<System.String>: string 93059
.NET: UpdateAsync: GenericType<System.Int32>: 24652
```

The preceding output examples demonstrate that asynchronous methods execute and complete in an *arbitrary order* depending on several factors, including thread scheduling and the speed of method execution. It isn't possible to reliably predict the order of completion for asynchronous method calls.

Class instance examples

The following `sayHello1` JS function:

- Calls the `GetHelloMessage` .NET method on the passed DotNetObjectReference.
- Returns the message from `GetHelloMessage` to the `sayHello1` caller.

HTML

```
<script>
  window.sayHello1 = (dotNetHelper) => {
    return dotNetHelper.invokeMethodAsync('GetHelloMessage');
  };
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

The following `HelloHelper` class has a JS-invokable .NET method named `GetHelloMessage`. When `HelloHelper` is created, the name in the `Name` property is used to return a message from `GetHelloMessage`.

`HelloHelper.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;

public class HelloHelper(string? name)
{
    public string? Name { get; set; } = name ?? "No Name";

    [JSInvokable]
    public string GetHelloMessage() => $"Hello, {Name}!";
}
```

The `CallHelloHelperGetHelloMessage` method in the following `JsInteropClasses3` class invokes the JS function `sayHello1` with a new instance of `HelloHelper`.

`JsInteropClasses3.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;

public class JsInteropClasses3(IJSRuntime js)
{
    private readonly IJSRuntime js = js;

    public async ValueTask<string> CallHelloHelperGetHelloMessage(string?
name)
    {
        using var objRef = DotNetObjectReference.Create(new
HelloHelper(name));
        return await js.InvokeAsync<string>("sayHello1", objRef);
    }
}
```

To avoid a memory leak and allow garbage collection, the .NET object reference created by DotNetObjectReference is disposed when the object reference goes out of scope with using var syntax.

When the **`Trigger .NET instance method`** button is selected in the following component, `JsInteropClasses3.CallHelloHelperGetHelloMessage` is called with the value of `name`.

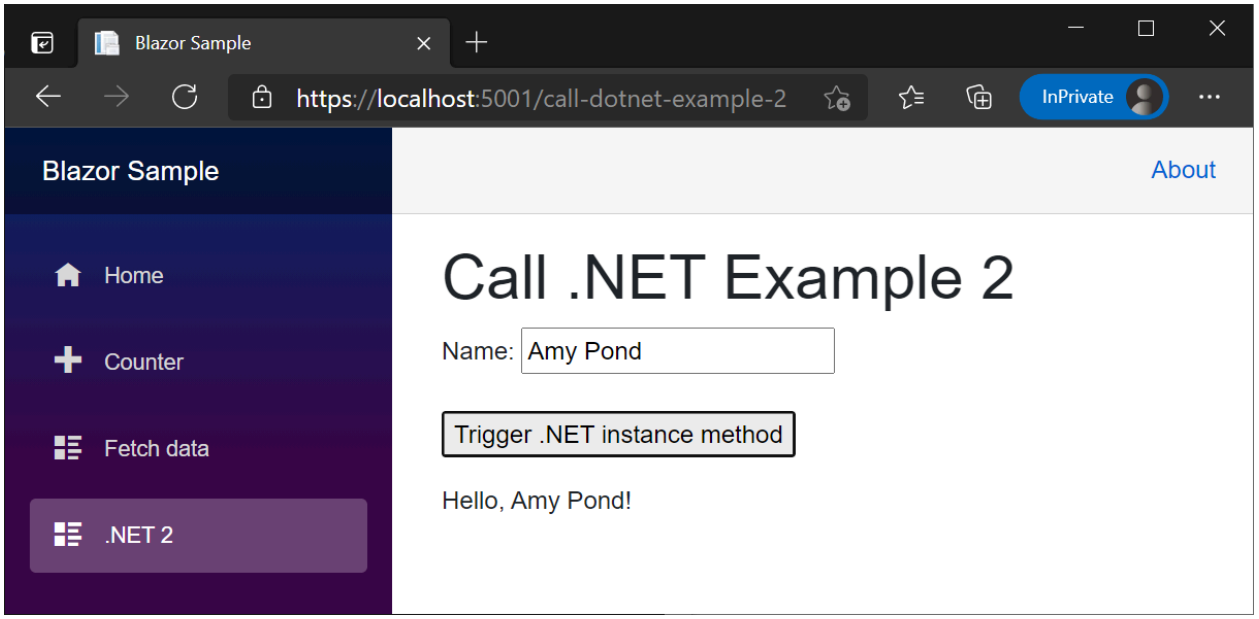
`CallDotnet4.razor`:

razor

```
@page "/call-dotnet-4"
@inject IJSRuntime JS

<PageTitle>Call .NET 4</PageTitle>

<h1>Call .NET Example 4</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>
```

```
<p>
    @result
</p>

@code {
    private string? name;
    private string? result;
    private JsInteropClasses3? jsInteropClasses;

    protected override void OnInitialized() =>
        jsInteropClasses = new JsInteropClasses3(JS);

    private async Task TriggerDotNetInstanceMethod()
    {
        if (jsInteropClasses is not null)
        {
            result = await
jsInteropClasses.CallHelloHelperGetHelloMessage(name);
        }
    }
}
```

The following image shows the rendered component with the name `Amy Pond` in the `Name` field. After the button is selected, `Hello, Amy Pond!` is displayed in the UI:



The preceding pattern shown in the `JsInteropClasses3` class can also be implemented entirely in a component.

`CallDotnet5.razor`:

razor

```
@page "/call-dotnet-5"
@inject IJSRuntime JS
```

```
<PageTitle>Call .NET 5</PageTitle>

<h1>Call .NET Example 5</h1>

<p>
    <label>
        Name: <input @bind="name" />
    </label>
</p>

<p>
    <button @onclick="TriggerDotNetInstanceMethod">
        Trigger .NET instance method
    </button>
</p>

<p>
    @result
</p>

@code {
    private string? name;
    private string? result;

    public async Task TriggerDotNetInstanceMethod()
    {
        using var objRef = DotNetObjectReference.Create(new
HelloHelper(name));
        result = await JS.InvokeAsync<string>("sayHello1", objRef);
    }
}
```

To avoid a memory leak and allow garbage collection, the .NET object reference created by DotNetObjectReference is disposed when the object reference goes out of scope with using var syntax.

The output displayed by the component is `Hello, Amy Pond!` when the name `Amy Pond` is provided in the `name` field.

In the preceding component, the .NET object reference is disposed. If a class or component doesn't dispose the DotNetObjectReference, dispose it from the client by calling `dispose` on the passed DotNetObjectReference:

JavaScript

```
window.{JS FUNCTION NAME} = (dotNetHelper) => {
  dotNetHelper.invokeMethodAsync('{.NET METHOD ID}');
  dotNetHelper.dispose();
}
```

In the preceding example:

- The `{JS FUNCTION NAME}` placeholder is the JS function's name.
- The variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.
- The `{.NET METHOD ID}` placeholder is the .NET method identifier.

Component instance .NET method helper class

A helper class can invoke a .NET instance method as an Action. Helper classes are useful in scenarios where using static .NET methods aren't applicable:

- When several components of the same type are rendered on the same page.
- In server-side apps with multiple users concurrently using the same component.

In the following example:

- The component contains several `ListItem1` components.
- Each `ListItem1` component is composed of a message and a button.
- When a `ListItem1` component button is selected, that `ListItem1`'s `UpdateMessage` method changes the list item text and hides the button.

The following `MessageUpdateInvokeHelper` class maintains a JS-invokable .NET method, `UpdateMessageCaller`, to invoke the Action specified when the class is instantiated.

`MessageUpdateInvokeHelper.cs`:

C#

```
using Microsoft.JSInterop;

namespace BlazorSample;

public class MessageUpdateInvokeHelper(Action action)
{
    private readonly Action action = action;

    [JSInvokable]
    public void UpdateMessageCaller() => action.Invoke();
}
```

The following `updateMessageCaller` JS function invokes the `UpdateMessageCaller` .NET method.

HTML

```
<script>
  window.updateMessageCaller = (dotNetHelper) => {
    dotNetHelper.invokeMethodAsync('UpdateMessageCaller');
    dotNetHelper.dispose();
  }
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

The following `ListItem1` component is a shared component that can be used any number of times in a parent component and creates list items (`<li>...</li>`) for an HTML list (`<ul>...</ul>` or `<ol>...</ol>`). Each `ListItem1` component instance establishes an instance of `MessageUpdateInvokeHelper` with an Action set to its `UpdateMessage` method.

When a `ListItem1` component's **`InteropCall`** button is selected, `updateMessageCaller` is invoked with a created DotNetObjectReference for the `MessageUpdateInvokeHelper` instance. This permits the framework to call `UpdateMessageCaller` on that `ListItem1`'s `MessageUpdateInvokeHelper` instance. The passed DotNetObjectReference is disposed in JS (`dotNetHelper.dispose()`).

`ListItem1.razor`:

razor

```
@inject IJSRuntime JS

<li>
    @message
    <button @onclick="InteropCall"
style="display:@display">InteropCall</button>
</li>

@code {
    private string message = "Select one of these list item buttons.";
    private string display = "inline-block";
    private MessageUpdateInvokeHelper? messageUpdateInvokeHelper;

    protected override void OnInitialized()
```

```
    {
        messageUpdateInvokeHelper = new
MessageUpdateInvokeHelper(UpdateMessage);
    }

    protected async Task InteropCall()
    {
        if (messageUpdateInvokeHelper is not null)
        {
            await JS.InvokeVoidAsync("updateMessageCaller",
                DotNetObjectReference.Create(messageUpdateInvokeHelper));
        }
    }

    private void UpdateMessage()
    {
        message = "UpdateMessage Called!";
        display = "none";
        StateHasChanged();
    }
}
```

StateHasChanged is called to update the UI when `message` is set in `UpdateMessage`. If `StateHasChanged` isn't called, Blazor has no way of knowing that the UI should be updated when the Action is invoked.

The following parent component includes four list items, each an instance of the `ListItem1` component.

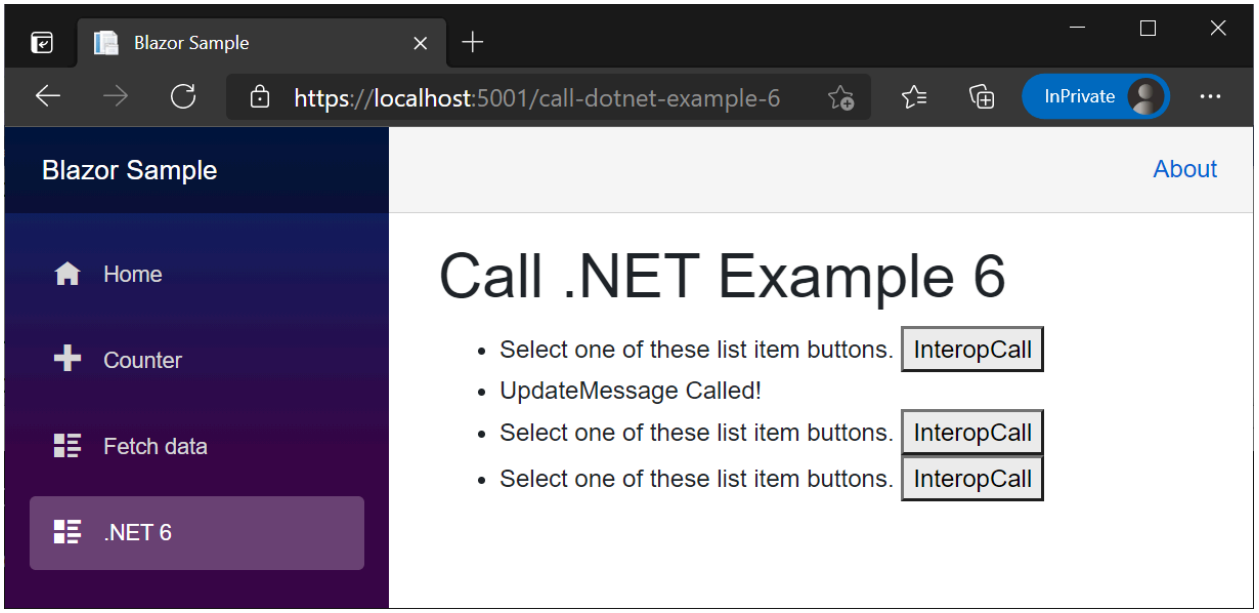
`CallDotnet6.razor`:

razor

```
@page "/call-dotnet-6"

<PageTitle>Call .NET 6</PageTitle>

<h1>Call .NET Example 6</h1>

<ul>
    <ListItem1 />
    <ListItem1 />
    <ListItem1 />
    <ListItem1 />
</ul>
```

The following image shows the rendered parent component after the second **`InteropCall`** button is selected:

- The second `ListItem1` component has displayed the `UpdateMessage Called!` message.
- The `InteropCall` button for the second `ListItem1` component isn't visible because the button's CSS `display` property is set to `none`.



Component instance .NET method called from `DotNetObjectReference` assigned to an element property

The assignment of a DotNetObjectReference to a property of an HTML element permits calling .NET methods on a component instance:

- An element reference is captured (ElementReference).
- In the component's OnAfterRender{Async} method, a JavaScript (JS) function is invoked with the element reference and the component instance as a DotNetObjectReference. The JS function attaches the DotNetObjectReference to the element in a property.
- When an element event is invoked in JS (for example, `onclick`), the element's attached DotNetObjectReference is used to call a .NET method.

Similar to the approach described in the Component instance .NET method helper class section, this approach is useful in scenarios where using static .NET methods aren't applicable:

- When several components of the same type are rendered on the same page.
- In server-side apps with multiple users concurrently using the same component.

- The .NET method is invoked from a JS event (for example, `onclick`), not from a Blazor event (for example, `@onclick`).

In the following example:

- The component contains several `ListItem2` components, which is a shared component.
- Each `ListItem2` component is composed of a list item message `<span>` and a second `<span>` with a `display` CSS property set to `inline-block` for display.
- When a `ListItem2` component list item is selected, that `ListItem2`'s `UpdateMessage` method changes the list item text in the first `<span>` and hides the second `<span>` by setting its `display` property to `none`.

The following `assignDotNetHelper` JS function assigns the DotNetObjectReference to an element in a property named `dotNetHelper`. The following `interopCall` JS function uses the DotNetObjectReference for the passed element to invoke a .NET method named `UpdateMessage`.

`ListItem2.razor.js`:

JavaScript

```
export function assignDotNetHelper(element, dotNetHelper) {
  element.dotNetHelper = dotNetHelper;
}

export async function interopCall(element) {
  await element.dotNetHelper.invokeMethodAsync('UpdateMessage');
}
```

> ⓘ **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

In the preceding example, the variable name `dotNetHelper` is arbitrary and can be changed to any preferred name.

The following `ListItem2` component is a shared component that can be used any number of times in a parent component and creates list items (`<li>...</li>`) for an HTML list (`<ul>...</ul>` or `<ol>...</ol>`).

Each `ListItem2` component instance invokes the `assignDotNetHelper` JS function in OnAfterRenderAsync with an element reference (the first `<span>` element of the list item) and the component instance as a DotNetObjectReference.

When a `ListItem2` component's message `<span>` is selected, `interopCall` is invoked passing the `<span>` element as a parameter (`this`), which invokes the `UpdateMessage` .NET method. In `UpdateMessage`, StateHasChanged is called to update the UI when `message` is set and the `display` property of the second `<span>` is updated. If `StateHasChanged` isn't called, Blazor has no way of knowing that the UI should be updated when the method is invoked.

The DotNetObjectReference is disposed when the component is disposed.

`ListItem2.razor`:

razor

```
@inject IJSRuntime JS
@implements IAsyncDisposable

<li>
    <span style="font-weight:bold;color:@color" @ref="elementRef"
        @onclick="CallJSToInvokeDotnet">
        @message
    </span>
    <span style="display:@display">
        Not Updated Yet!
    </span>
</li>

@code {
    private IJSObjectReference? module;
    private DotNetObjectReference<ListItem2>? objRef;
    private ElementReference elementRef;
    private string display = "inline-block";
    private string message = "Select one of these list items.";
    private string color = "initial";

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/ListItem2.razor.js");

            objRef = DotNetObjectReference.Create(this);
            await module.InvokeVoidAsync("assignDotNetHelper", elementRef,
objRef);
        }
    }
```

```
    public async Task CallJSToInvokeDotnet()
    {
        if (module is not null)
        {
            await module.InvokeVoidAsync("interopCall", elementRef);
        }
    }

    [JSInvokable]
    public void UpdateMessage()
    {
        message = "UpdateMessage Called!";
        display = "none";
        color = "MediumSeaGreen";
        StateHasChanged();
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }

        objRef?.Dispose();
    }
}
```

The following parent component includes four list items, each an instance of the `ListItem2` component.

`CallDotnet7.razor`:

razor

```
@page "/call-dotnet-7"

<PageTitle>Call .NET 7</PageTitle>

<h1>Call .NET Example 7</h1>

<ul>
    <ListItem2 />
    <ListItem2 />
    <ListItem2 />
```

```
    <ListItem2 />
</ul>
```

Synchronous JS interop in client-side components

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from JavaScript to .NET in a client-side component, use `DotNet.invokeMethod` instead of `DotNet.invokeMethodAsync`.

Synchronous calls work if:

- The component is only rendered for execution on WebAssembly.
- The called function returns a value synchronously. The function isn't an `async` method and doesn't return a .NET Task or JavaScript Promise.

JavaScript location

Load JavaScript (JS) code using any of approaches described by the article on JavaScript location:

- Load a script in <head> markup (*Not generally recommended*)
- Load a script in <body> markup
- Load a script from an external JavaScript file (.js) collocated with a component
- Load a script from an external JavaScript file (.js)
- Inject a script before or after Blazor starts

Using JS modules to load JS is described in this article in the JavaScript isolation in JavaScript modules section.

> ⚠ **Warning**
>
> Only place a `<script>` tag in a component file (`.razor`) if the component is guaranteed to adopt **static server-side rendering (static SSR)** because the `<script>` tag can't be updated dynamically.

JavaScript isolation in JavaScript modules

Blazor enables JavaScript (JS) isolation in standard JavaScript modules (ECMAScript specification). JavaScript module loading works the same way in Blazor as it does for other types of web apps, and you're free to customize how modules are defined in your app. For a guide on how to use JavaScript modules, see MDN Web Docs: JavaScript modules.

JS isolation provides the following benefits:

- Imported JS no longer pollutes the global namespace.
- Consumers of a library and components aren't required to import the related JS.

For more information, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Dynamic import with the import() operator is supported with ASP.NET Core and Blazor:

JavaScript

```
if ({CONDITION}) import("/additionalModule.js");
```

In the preceding example, the `{CONDITION}` placeholder represents a conditional check to determine if the module should be loaded.

For browser compatibility, see Can I use: JavaScript modules: dynamic import.

Avoid circular object references

Objects that contain circular references can't be serialized on the client for either:

- .NET method calls.
- JavaScript method calls from C# when the return type has circular references.

Byte array support

Blazor supports optimized byte array JavaScript (JS) interop that avoids encoding/decoding byte arrays into Base64. The following example uses JS interop to pass a byte array to .NET.

Provide a `sendByteArray` JS function. The function is called statically, which includes the assembly name parameter in the `invokeMethodAsync` call, by a button in the component and doesn't return a value:

`CallDotnet8.razor.js`:

JavaScript

```
export function sendByteArray() {
  const data = new Uint8Array([0x45, 0x76, 0x65, 0x72, 0x79, 0x74, 0x68,
0x69,
    0x6e, 0x67, 0x27, 0x73, 0x20, 0x73, 0x68, 0x69, 0x6e, 0x79, 0x2c,
    0x20, 0x43, 0x61, 0x70, 0x74, 0x61, 0x69, 0x6e, 0x2e, 0x20, 0x4e,
    0x6f, 0x74, 0x20, 0x74, 0x6f, 0x20, 0x66, 0x72, 0x65, 0x74, 0x2e]);
  DotNet.invokeMethodAsync('BlazorSample', 'ReceiveByteArray', data)
    .then(str => {
      alert(str);
    });
}

export function addHandlers() {
  const btn = document.getElementById("btn");
  btn.addEventListener("click", sendByteArray);
}
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

`CallDotnet8.razor`:

razor

```
@page "/call-dotnet-8"
@using System.Text
@implements IAsyncDisposable
@inject IJSRuntime JS

<PageTitle>Call .NET 8</PageTitle>
```

```razor
<h1>Call .NET Example 8</h1>

<p>
    <button id="btn">Send Bytes</button>
</p>

<p>
    Quote ©2005 <a href="https://www.uphe.com">Universal Pictures</a>:
    <a href="https://www.uphe.com/movies/serenity-2005">Serenity</a><br>
    <a href="https://www.imdb.com/name/nm0821612/">Jewel Staite on IMDB</a>
</p>

@code {
    private IJSObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/Pages/CallDotnet8.razor.js");

            await module.InvokeVoidAsync("addHandlers");
        }
    }

    [JSInvokable]
    public static Task<string> ReceiveByteArray(byte[] receivedBytes) =>
        Task.FromResult(Encoding.UTF8.GetString(receivedBytes, 0,
            receivedBytes.Length));

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

For information on using a byte array when calling JavaScript from .NET, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.

Stream from JavaScript to .NET

Blazor supports streaming data directly from JavaScript to .NET. Streams are requested using the `Microsoft.JSInterop.IJSStreamReference` interface.

`Microsoft.JSInterop.IJSStreamReference.OpenReadStreamAsync` returns a Stream and uses the following parameters:

- `maxAllowedSize`: Maximum number of bytes permitted for the read operation from JavaScript, which defaults to 512,000 bytes if not specified.
- `cancellationToken`: A CancellationToken for cancelling the read.

In JavaScript:

JavaScript

```
function streamToDotNet() {
  return new Uint8Array(10000000);
}
```

In C# code:

C#

```
var dataReference =
    await JS.InvokeAsync<IJSStreamReference>("streamToDotNet");
using var dataReferenceStream =
    await dataReference.OpenReadStreamAsync(maxAllowedSize: 10_000_000);

var outputPath = Path.Combine(Path.GetTempPath(), "file.txt");
using var outputFileStream = File.OpenWrite(outputPath);
await dataReferenceStream.CopyToAsync(outputFileStream);
```

In the preceding example:

- `JS` is an injected IJSRuntime instance. IJSRuntime is registered by the Blazor framework.
- The `dataReferenceStream` is written to disk (`file.txt`) at the current user's temporary folder path (GetTempPath).

Call JavaScript functions from .NET methods in ASP.NET Core Blazor covers the reverse operation, streaming from .NET to JavaScript using a DotNetStreamReference.

ASP.NET Core Blazor file uploads covers how to upload a file in Blazor. For a forms example that streams `<textarea>` data in a server-side component, see Troubleshoot ASP.NET Core Blazor forms.

JavaScript `[JSImport]`/`[JSExport]` interop

This section applies to client-side components.

As an alternative to interacting with JavaScript (JS) in client-side components using Blazor's JS interop mechanism based on the IJSRuntime interface, a JS `[JSImport]`/`[JSExport]` interop API is available to apps targeting .NET 7 or later.

For more information, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

Disposal of JavaScript interop object references

Examples throughout the JavaScript (JS) interop articles demonstrate typical object disposal patterns:

- When calling .NET from JS, as described in this article, dispose of a created DotNetObjectReference either from .NET or from JS to avoid leaking .NET memory.
- When calling JS from .NET, as described in Call JavaScript functions from .NET methods in ASP.NET Core Blazor, dispose any created IJSObjectReference/IJSInProcessObjectReference/`JSObjectReference` either from .NET or from JS to avoid leaking JS memory.

JS interop object references are implemented as a map keyed by an identifier on the side of the JS interop call that creates the reference. When object disposal is initiated from either the .NET or JS side, Blazor removes the entry from the map, and the object can be garbage collected as long as no other strong reference to the object is present.

At a minimum, always dispose objects created on the .NET side to avoid leaking .NET managed memory.

DOM cleanup tasks during component disposal

For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

JavaScript interop calls without a circuit

For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

Additional resources

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- InteropComponent.razor example (dotnet/AspNetCore GitHub repository main branch): The `main` branch represents the product unit's current development for the next release of ASP.NET Core. To select the branch for a different release (for example, `release/{VERSION}`, where the `{VERSION}` placeholder is the release version), use the **Switch branches or tags** dropdown list to select the branch. For a branch that no longer exists, use the **Tags** tab to find the API (for example, `v7.0.0`).
- Interaction with the DOM
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)
- Handle errors in ASP.NET Core Blazor apps (*JavaScript interop* section)
- Threat mitigation: .NET methods invoked from the browser

JavaScript `[JSImport]`/`[JSExport]` interop with ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to interact with JavaScript (JS) in client-side components using JavaScript (JS) `[JSImport]`/`[JSExport]` interop API. For additional information and examples, see JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly.

For additional guidance, see the Configuring and hosting .NET WebAssembly applications guidance in the .NET Runtime (`dotnet/runtime`) GitHub repository.

Blazor provides its own JS interop mechanism based on the IJSRuntime interface. Blazor's JS interop is uniformly supported across Blazor render modes and for Blazor Hybrid apps. IJSRuntime also enables library authors to build JS interop libraries for sharing across the Blazor ecosystem and remains the recommended approach for JS interop in Blazor. See the following articles:

- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor

This article describes an alternative JS interop approach specific to client-side components executed on WebAssembly. These approaches are appropriate when you only expect to run on client-side WebAssembly. Library authors can use these approaches to optimize JS interop by checking during code execution if the app is running on WebAssembly in a browser (OperatingSystem.IsBrowser). The approaches described in this article should be used to replace the obsolete unmarshalled JS interop API when migrating to .NET 7 or later.

> **Note**
>
> This article focuses on JS interop in client-side components. For guidance on calling .NET in JavaScript apps, see JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project.

Obsolete JavaScript interop API

Unmarshalled JS interop using IJSUnmarshalledRuntime API is obsolete in ASP.NET Core in .NET 7 or later. Follow the guidance in this article to replace the obsolete API.

Prerequisites

Visual Studio with the **ASP.NET and web development** workload.

No further tooling is required if you plan on implementing `[JSImport]`/`[JSExport]` interop in a Blazor WebAssembly app generated from the Blazor WebAssembly project template.

If you plan to use the WebAssembly Browser or WebAssembly Console app project templates, install the Microsoft.NET.Runtime.WebAssembly.Templates NuGet package with the following command:

.NET CLI

```
dotnet new install Microsoft.NET.Runtime.WebAssembly.Templates
```

For more information, see JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project.

Namespace

The JS interop API (JSHost.ImportAsync) described in this article is controlled by attributes in the System.Runtime.InteropServices.JavaScript namespace.

Enable unsafe blocks

Enable the AllowUnsafeBlocks property in app's project file, which permits the code generator in the Roslyn compiler to use pointers for JS interop:

XML

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

⚠ **Warning**

The JS interop API requires enabling **AllowUnsafeBlocks**. Be careful when implementing your own unsafe code in .NET apps, which can introduce security and stability risks. For more information, see **Unsafe code, pointer types, and function pointers**.

Razor class library (RCL) collocated JS is unsupported

Generally, the JS location support for IJSRuntime-based JS interop (JavaScript location in ASP.NET Core Blazor apps) is also present for the `[JSImport]`/`[JSExport]` interop described by this article. The only unsupported JS location feature is for *collocated JS in a Razor class library (RCL)*.

Instead of using collocated JS in an RCL, place the JS file in the RCL's `wwwroot` folder and reference it using the usual path for RCL static assets:

`_content/{PACKAGE ID}/{PATH}/{FILE NAME}.js`

- The `{PACKAGE ID}` placeholder is the RCL's package identifier (or library name for a class library).
- The `{PATH}` placeholder is the path to the file.
- The `{FILE NAME}` placeholder is the file name.

Although collocated JS in an RCL isn't supported by `[JSImport]`/`[JSExport]` interop, you can keep your JS files organized by taking either or both of the following approaches:

- Name the JS file the same as the component where the JS is used. For a component in the RCL named `CallJavaScriptFromLib` (`CallJavaScriptFromLib.razor`), name the file `CallJavaScriptFromLib.js` in the `wwwroot` folder.
- Place component-specific JS files in a `Components` folder inside the RCL's `wwwroot` folder and use "`Components`" in the path to the file: `_content/{PACKAGE ID}/Components/CallJavaScriptFromLib.js`.

Call JavaScript from .NET

This section explains how to call JS functions from .NET.

In the following `CallJavaScript1` component:

- The `CallJavaScript1` module is imported asynchronously from the collocated JS file with JSHost.ImportAsync.
- The imported `getMessage` JS function is called by `GetWelcomeMessage`.
- The returned welcome message string is displayed in the UI via the `message` field.

`CallJavaScript1.razor`:

razor

```razor
@page "/call-javascript-1"
@rendermode InteractiveWebAssembly
@using System.Runtime.InteropServices.JavaScript

<h1>
    JS <code>[JSImport]</code>/<code>[JSExport]</code> Interop
    (Call JS Example 1)
</h1>

@(message is not null ? message : string.Empty)

@code {
    private string? message;

    protected override async Task OnInitializedAsync()
    {
        await JSHost.ImportAsync("CallJavaScript1",
            "../Components/Pages/CallJavaScript1.razor.js");

        message = GetWelcomeMessage();
    }
}
```

> **Note**
>
> Include a conditional check in code with OperatingSystem.IsBrowser to ensure that the JS interop is only called by a component rendered on the client. This is important for libraries/NuGet packages that target server-side components, which can't execute the code provided by this JS interop API.

To import a JS function to call it from C#, use the [JSImport] attribute on a C# method signature that matches the JS function's signature. The first parameter to the `[JSImport]` attribute is the name of the JS function to import, and the second parameter is the name of the JS module.

In the following example, `getMessage` is a JS function that returns a `string` for a module named `CallJavaScript1`. The C# method signature matches: No parameters are passed

to the JS function, and the JS function returns a `string`. The JS function is called by `GetWelcomeMessage` in C# code.

`CallJavaScript1.razor.cs`:

C#

```
using System.Runtime.InteropServices.JavaScript;
using System.Runtime.Versioning;

namespace BlazorSample.Components.Pages;

[SupportedOSPlatform("browser")]
public partial class CallJavaScript1
{
    [JSImport("getMessage", "CallJavaScript1")]
    internal static partial string GetWelcomeMessage();
}
```

The app's namespace for the preceding `CallJavaScript1` partial class is `BlazorSample`. The component's namespace is `BlazorSample.Components.Pages`. If using the preceding component in a local test app, update the namespace to match the app. For example, the namespace is `ContosoApp.Components.Pages` if the app's namespace is `ContosoApp`. For more information, see ASP.NET Core Razor components.

In the imported method signature, you can use .NET types for parameters and return values, which are marshalled automatically by the runtime. Use JSMarshalAsAttribute<T> to control how the imported method parameters are marshalled. For example, you might choose to marshal a `long` as System.Runtime.InteropServices.JavaScript.JSType.Number or System.Runtime.InteropServices.JavaScript.JSType.BigInt. You can pass Action/Func<TResult> callbacks as parameters, which are marshalled as callable JS functions. You can pass both JS and managed object references, and they are marshaled as proxy objects, keeping the object alive across the boundary until the proxy is garbage collected. You can also import and export asynchronous methods with a Task result, which are marshaled as JS promises. Most of the marshalled types work in both directions, as parameters and as return values, on both imported and exported methods, which are covered in the Call .NET from JavaScript section later in this article.

For additional type mapping information and examples, see JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly.

The module name in the `[JSImport]` attribute and the call to load the module in the component with JSHost.ImportAsync must match and be unique in the app. When

authoring a library for deployment in a NuGet package, we recommend using the NuGet package namespace as a prefix in module names. In the following example, the module name reflects the `Contoso.InteropServices.JavaScript` package and a folder of user message interop classes (`UserMessages`):

C#

```
[JSImport("getMessage",
    "Contoso.InteropServices.JavaScript.UserMessages.CallJavaScript1")]
```

Functions accessible on the global namespace can be imported by using the globalThis prefix in the function name and by using the `[JSImport]` attribute without providing a module name. In the following example, console.log is prefixed with `globalThis`. The imported function is called by the C# `Log` method, which accepts a C# string message (`message`) and marshalls the C# string to a JS String for `console.log`:

C#

```
[JSImport("globalThis.console.log")]
internal static partial void Log([JSMarshalAs<JSType.String>] string
message);
```

Export scripts from a standard JavaScript module either collocated with a component or placed with other JavaScript static assets in a JS file (for example, `wwwroot/js/{FILE NAME}.js`, where JS static assets are maintained in a folder named `js` in the app's `wwwroot` folder and the `{FILE NAME}` placeholder is the file name).

In the following example, a JS function named `getMessage` is exported from a collocated JS file that returns a welcome message, "Hello from Blazor!" in Portuguese:

`CallJavaScript1.razor.js`:

JavaScript

```
export function getMessage() {
  return 'Olá do Blazor!';
}
```

Call .NET from JavaScript

This section explains how to call .NET methods from JS.

The following `CallDotNet1` component calls JS that directly interacts with the DOM to render the welcome message string:

- The `CallDotNet` JS module is imported asynchronously from the collocated JS file for this component.
- The imported `setMessage` JS function is called by `SetWelcomeMessage`.
- The returned welcome message is displayed by `setMessage` in the UI via the `message` field.

> **ⓘ Important**
>
> In this section's example, JS interop is used to mutate a DOM element *purely for demonstration purposes* after the component is rendered in OnAfterRender. Typically, you should only mutate the DOM with JS when the object doesn't interact with Blazor. The approach shown in this section is similar to cases where a third-party JS library is used in a Razor component, where the component interacts with the JS library via JS interop, the third-party JS library interacts with part of the DOM, and Blazor isn't involved directly with the DOM updates to that part of the DOM. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

`CallDotNet1.razor`:

razor

```
@page "/call-dotnet-1"
@rendermode InteractiveWebAssembly
@using System.Runtime.InteropServices.JavaScript

<h1>
    JS <code>[JSImport]</code>/<code>[JSExport]</code> Interop
    (Call .NET Example 1)
</h1>

<p>
    <span id="result">.NET method not executed yet</span>
</p>

@code {
    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            await JSHost.ImportAsync("CallDotNet1",
                "../Components/Pages/CallDotNet1.razor.js");
```

```
            SetWelcomeMessage();
        }
    }
}
```

To export a .NET method so that it can be called from JS, use the [JSExport] attribute.

In the following example:

- `SetWelcomeMessage` calls a JS function named `setMessage`. The JS function calls into .NET to receive the welcome message from `GetMessageFromDotnet` and displays the message in the UI.
- `GetMessageFromDotnet` is a .NET method with the `[JSExport]` attribute that returns a welcome message, "Hello from Blazor!" in Portuguese.

`CallDotNet1.razor.cs`:

C#

```
using System.Runtime.InteropServices.JavaScript;
using System.Runtime.Versioning;

namespace BlazorSample.Components.Pages;

[SupportedOSPlatform("browser")]
public partial class CallDotNet1
{
    [JSImport("setMessage", "CallDotNet1")]
    internal static partial void SetWelcomeMessage();

    [JSExport]
    internal static string GetMessageFromDotnet() => "Olá do Blazor!";
}
```

The app's namespace for the preceding `CallDotNet1` partial class is `BlazorSample`. The component's namespace is `BlazorSample.Components.Pages`. If using the preceding component in a local test app, update the app's namespace to match the app. For example, the component namespace is `ContosoApp.Components.Pages` if the app's namespace is `ContosoApp`. For more information, see ASP.NET Core Razor components.

In the following example, a JS function named `setMessage` is imported from a collocated JS file.

The `setMessage` method:

- Calls `globalThis.getDotnetRuntime(0)` to expose the WebAssembly .NET runtime instance for calling exported .NET methods.
- Obtains the app assembly's JS exports. The name of the app's assembly in the following example is `BlazorSample`.
- Calls the `BlazorSample.Components.Pages.CallDotNet1.GetMessageFromDotnet` method from the exports (`exports`). The returned value, which is the welcome message, is assigned to the `CallDotNet1` component's `<span>` text. The app's namespace is `BlazorSample`, and the `CallDotNet1` component's namespace is `BlazorSample.Components.Pages`.

`CallDotNet1.razor.js`:

JavaScript

```
export async function setMessage() {
  const { getAssemblyExports } = await globalThis.getDotnetRuntime(0);
  var exports = await getAssemblyExports("BlazorSample.dll");

  document.getElementById("result").innerText =

exports.BlazorSample.Components.Pages.CallDotNet1.GetMessageFromDotnet();
}
```

> **Note**
>
> Calling `getAssemblyExports` to obtain the exports can occur in a JavaScript initializer for availability across the app.

Multiple module import calls

After a JS module is loaded, the module's JS functions are available to the app's components and classes as long as the app is running in the browser window or tab without the user manually reloading the app. JSHost.ImportAsync can be called multiple times on the same module without a significant performance penalty when:

- The user visits a component that calls JSHost.ImportAsync to import a module, navigates away from the component, and then returns to the component where JSHost.ImportAsync is called again for the same module import.
- The same module is used by different components and loaded by JSHost.ImportAsync in each of the components.

Use of a single JavaScript module across components

Before following the guidance in this section, read the Call JavaScript from .NET and Call .NET from JavaScript sections of this article, which provide general guidance on `[JSImport]`/`[JSExport]` interop.

The example in this section shows how to use JS interop from a shared JS module in a client-side app. The guidance in this section isn't applicable to Razor class libraries (RCLs).

The following components, classes, C# methods, and JS functions are used:

- `Interop` class (`Interop.cs`): Sets up import and export JS interop with the `[JSImport]` and `[JSExport]` attributes for a module named `Interop`.
  - `GetWelcomeMessage`: .NET method that calls the imported `getMessage` JS function.
  - `SetWelcomeMessage`: .NET method that calls the imported `setMessage` JS function.
  - `GetMessageFromDotnet`: An exported C# method that returns a welcome message string when called from JS.
- `wwwroot/js/interop.js` file: Contains the JS functions.
  - `getMessage`: Returns a welcome message when called by C# code in a component.
  - `setMessage`: Calls the `GetMessageFromDotnet` C# method and assigns the returned welcome message to a DOM `<span>` element.
- `Program.cs` calls JSHost.ImportAsync to load the module from `wwwroot/js/interop.js`.
- `CallJavaScript2` component (`CallJavaScript2.razor`): Calls `GetWelcomeMessage` and displays the returned welcome message in the component's UI.
- `CallDotNet2` component (`CallDotNet2.razor`): Calls `SetWelcomeMessage`.

`Interop.cs`:

C#

```
using System.Runtime.InteropServices.JavaScript;
using System.Runtime.Versioning;

namespace BlazorSample.JavaScriptInterop;

[SupportedOSPlatform("browser")]
public partial class Interop
{
    [JSImport("getMessage", "Interop")]
```

```
    internal static partial string GetWelcomeMessage();

    [JSImport("setMessage", "Interop")]
    internal static partial void SetWelcomeMessage();

    [JSExport]
    internal static string GetMessageFromDotnet() => "Olá do Blazor!";
}
```

In the preceding example, the app's namespace is `BlazorSample`, and the full namespace for C# interop classes is `BlazorSample.JavaScriptInterop`.

`wwwroot/js/interop.js`:

JavaScript

```
export function getMessage() {
  return 'Olá do Blazor!';
}

export async function setMessage() {
  const { getAssemblyExports } = await globalThis.getDotnetRuntime(0);
  var exports = await getAssemblyExports("BlazorSample.dll");

  document.getElementById("result").innerText =
    exports.BlazorSample.JavaScriptInterop.Interop.GetMessageFromDotnet();
}
```

Make the System.Runtime.InteropServices.JavaScript namespace available at the top of the `Program.cs` file:

C#

```
using System.Runtime.InteropServices.JavaScript;
```

Load the module in `Program.cs` before WebAssemblyHost.RunAsync is called:

C#

```
if (OperatingSystem.IsBrowser())
{
    await JSHost.ImportAsync("Interop", "../js/interop.js");
}
```

`CallJavaScript2.razor`:

razor

```razor
@page "/call-javascript-2"
@rendermode InteractiveWebAssembly
@using BlazorSample.JavaScriptInterop

<h1>
    JS <code>[JSImport]</code>/<code>[JSExport]</code> Interop
    (Call JS Example 2)
</h1>

@(message is not null ? message : string.Empty)

@code {
    private string? message;

    protected override void OnInitialized()
    {
        message = Interop.GetWelcomeMessage();
    }
}
```

`CallDotNet2.razor`:

razor

```razor
@page "/call-dotnet-2"
@rendermode InteractiveWebAssembly
@using BlazorSample.JavaScriptInterop

<h1>
    JS <code>[JSImport]</code>/<code>[JSExport]</code> Interop
    (Call .NET Example 2)
</h1>

<p>
    <span id="result">.NET method not executed</span>
</p>

@code {
    protected override void OnAfterRender(bool firstRender)
    {
        if (firstRender)
        {
            Interop.SetWelcomeMessage();
        }
    }
}
```

ⓘ Important

In this section's example, JS interop is used to mutate a DOM element *purely for demonstration purposes* after the component is rendered in **OnAfterRender**. Typically, you should only mutate the DOM with JS when the object doesn't interact with Blazor. The approach shown in this section is similar to cases where a third-party JS library is used in a Razor component, where the component interacts with the JS library via JS interop, the third-party JS library interacts with part of the DOM, and Blazor isn't involved directly with the DOM updates to that part of the DOM. For more information, see **ASP.NET Core Blazor JavaScript interoperability (JS interop)**.

Additional resources

- JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly
- JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project
- API documentation
  - [JSImport] attribute
  - [JSExport] attribute
- In the `dotnet/runtime` GitHub repository:
  - Configuring and hosting .NET WebAssembly applications
  - .NET WebAssembly runtime
  - dotnet.d.ts file (.NET WebAssembly runtime configuration)

ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)

Article • 01/13/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to load JavaScript (JS) in a Blazor Web App with static server-side rendering (static SSR) and enhanced navigation.

Some apps depend on JS to perform initialization tasks that are specific to each page. When using Blazor's enhanced navigation feature, which allows the user to avoid reloading the entire page, page-specific JS may not be executed again as expected each time an enhanced page navigation occurs.

To avoid this problem, we don't recommended relying on page-specific `<script>` elements placed outside of the layout file applied to the component. Instead, scripts should register an afterWebStarted JS initializer to perform initialization logic and use an event listener to listen for page updates caused by enhanced navigation:

JavaScript

```
blazor.addEventListener("enhancedload", {CALLBACK});
```

In the preceding example, the `{CALLBACK}` placeholder is the callback function.

Enhanced page load script example

The following example demonstrates one way to configure JS code to run when a statically-rendered page with enhanced navigation is initially loaded or updated.

The following `PageWithScript` component example is a component in the app that requires scripts to run with static SSR and enhanced navigation. The following component example includes a `PageScript` component from a Razor class library (RCL) that's added to the solution later in this article.

`Components/Pages/PageWithScript.razor`:

razor

```razor
@page "/page-with-script"
@using BlazorPageScript

<PageTitle>Enhanced Load Script Example</PageTitle>

<PageScript Src="./Components/Pages/PageWithScript.razor.js" />

Welcome to my page.
```

In the Blazor Web App, add the following collocated JS file:

- `onLoad` is called when the script is added to the page.
- `onUpdate` is called when the script still exists on the page after an enhanced update.
- `onDispose` is called when the script is removed from the page after an enhanced update.

`Components/Pages/PageWithScript.razor.js`:

JavaScript

```javascript
export function onLoad() {
  console.log('Loaded');
}

export function onUpdate() {
  console.log('Updated');
}

export function onDispose() {
  console.log('Disposed');
}
```

In a Razor Class Library (RCL) (the example RCL is named `BlazorPageScript`), add the following module, which is a JS initializer.

`wwwroot/BlazorPageScript.lib.module.js`:

JavaScript

```javascript
const pageScriptInfoBySrc = new Map();

function registerPageScriptElement(src) {
  if (!src) {
    throw new Error('Must provide a non-empty value for the "src"
attribute.');
```

```
  }

  let pageScriptInfo = pageScriptInfoBySrc.get(src);

  if (pageScriptInfo) {
    pageScriptInfo.referenceCount++;
  } else {
    pageScriptInfo = { referenceCount: 1, module: null };
    pageScriptInfoBySrc.set(src, pageScriptInfo);
    initializePageScriptModule(src, pageScriptInfo);
  }
}

function unregisterPageScriptElement(src) {
  if (!src) {
    return;
  }

  const pageScriptInfo = pageScriptInfoBySrc.get(src);

  if (!pageScriptInfo) {
    return;
  }

  pageScriptInfo.referenceCount--;
}

async function initializePageScriptModule(src, pageScriptInfo) {
  if (src.startsWith("./")) {
    src = new URL(src.substr(2), document.baseURI).toString();
  }

  const module = await import(src);

  if (pageScriptInfo.referenceCount <= 0) {
    return;
  }

  pageScriptInfo.module = module;
  module.onLoad?.();
  module.onUpdate?.();
}

function onEnhancedLoad() {
  for (const [src, { module, referenceCount }] of pageScriptInfoBySrc) {
    if (referenceCount <= 0) {
      module?.onDispose?.();
      pageScriptInfoBySrc.delete(src);
    }
  }

  for (const { module } of pageScriptInfoBySrc.values()) {
    module?.onUpdate?.();
  }
}
```

```
export function afterWebStarted(blazor) {
  customElements.define('page-script', class extends HTMLElement {
    static observedAttributes = ['src'];

    attributeChangedCallback(name, oldValue, newValue) {
      if (name !== 'src') {
        return;
      }

      this.src = newValue;
      unregisterPageScriptElement(oldValue);
      registerPageScriptElement(newValue);
    }

    disconnectedCallback() {
      unregisterPageScriptElement(this.src);
    }
  });

  blazor.addEventListener('enhancedload', onEnhancedLoad);
}
```

Do ***not*** add a `<script>` tag to the app's root component, typically the `App` componennt, for `BlazorPageScript.lib.module.js` because the module in this case is a JS initializer (afterWebStarted). JS initializers are detected and loaded automatically by the Blazor framework.

In the RCL, add the following `PageScript` component.

`PageScript.razor`:

razor

```
<page-script src="@Src"></page-script>

@code {
    [Parameter]
    [EditorRequired]
    public string Src { get; set; } = default!;
}
```

The `PageScript` component functions normally on the top-level of a page.

If you place the `PageScript` component in an app's layout (for example, `MainLayout.razor`), which results in a shared `PageScript` among pages that use the layout, then the component only runs `onLoad` after a full page reload and `onUpdate` when any enhanced page update occurs, including enhanced navigation.

To reuse the same module among pages, but have the `onLoad` and `onDispose` callbacks invoked on each page change, append a query string to the end of the script so that it's recognized as a different module. An app could adopt the convention of using the component's name as the query string value. In the following example, the query string is "`counter`" because this `PageScript` component reference is placed in a `Counter` component. This is merely a suggestion, and you can use whatever query string scheme that you prefer.

razor

```
<PageScript Src="./Components/Pages/PageWithScript.razor.js?counter" />
```

To monitor changes in specific DOM elements, use the MutationObserver pattern in JS on the client. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

Implementation without using an RCL

The approach described in this article can be implemented directly in a Blazor Web App without using a Razor class library (RCL) by moving the assets of the RCL into the app. However, the use of an RCL is convenient because the RCL can be packaged into a NuGet package for consumption by Blazor apps across an organization.

If you move the assets into a Blazor Web App, be sure to rename the module (`BlazorPageScript.lib.module.js`) to match the app per the file naming rules for JS initializers. If the file isn't named correctly, Blazor can't detect and load the module, and the `afterWebStarted` event doesn't execute automatically when the app starts.

Call a web API from ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes how to call a web API from a Blazor app.

Package

The System.Net.Http.Json package provides extension methods for System.Net.Http.HttpClient and System.Net.Http.HttpContent that perform automatic serialization and deserialization using System.Text.Json. The `System.Net.Http.Json` package is provided by the .NET shared framework and doesn't require adding a package reference to the app.

Sample apps

See the sample apps in the dotnet/blazor-samples GitHub repository.

`BlazorWebAppCallWebApi`

Call an external (not in the Blazor Web App) todo list web API from a Blazor Web App:

- `Backend`: A web API app for maintaining a todo list, based on Minimal APIs. The web API app is a separate app from the Blazor Web App, possibly hosted on a different server.
- `BlazorApp`/`BlazorApp.Client`: A Blazor Web App that calls the web API app with an HttpClient for todo list operations, such as creating, reading, updating, and deleting (CRUD) items from the todo list.

For client-side rendering (CSR), which includes Interactive WebAssembly components and Auto components that have adopted CSR, calls are made with a preconfigured HttpClient registered in the `Program` file of the client project (`BlazorApp.Client`):

C#

```
builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri(builder.Configuration["FrontendUrl"] ??
"https://localhost:5002")
    });
```

For server-side rendering (SSR), which includes prerendered and interactive Server components, prerendered WebAssembly components, and Auto components that are prerendered or have adopted SSR, calls are made with an HttpClient registered in the `Program` file of the server project (`BlazorApp`):

C#

```
builder.Services.AddHttpClient();
```

Call an internal (inside the Blazor Web App) movie list API, where the API resides in the server project of the Blazor Web App:

- `BlazorApp`: A Blazor Web App that maintains a movie list:
  - When operations are performed on the movie list within the app on the server, ordinary API calls are used.
  - When API calls are made by a web-based client, a web API is used for movie list operations, based on Minimal APIs.
- `BlazorApp.Client`: The client project of the Blazor Web App, which contains Interactive WebAssembly and Auto components for user management of the movie list.

For CSR, which includes Interactive WebAssembly components and Auto components that have adopted CSR, calls to the API are made via a client-based service (`ClientMovieService`) that uses a preconfigured HttpClient registered in the `Program` file of the client project (`BlazorApp.Client`). Because these calls are made over a public or private web, the movie list API is a *web API*.

The following example obtains a list of movies from the `/movies` endpoint:

C#

```
public class ClientMovieService(HttpClient http) : IMovieService
{
    public async Task<Movie[]> GetMoviesAsync(bool watchedMovies) =>
```

```
        await http.GetFromJsonAsync<Movie[]>("movies") ?? [];
}
```

For SSR, which includes prerendered and interactive Server components, prerendered WebAssembly components, and Auto components that are prerendered or have adopted SSR, calls are made directly via a server-based service (`ServerMovieService`). The API doesn't rely on a network, so it's a standard API for movie list CRUD operations.

The following example obtains a list of movies:

C#

```
public class ServerMovieService(MovieContext db) : IMovieService
{
    public async Task<Movie[]> GetMoviesAsync(bool watchedMovies) =>
        watchedMovies ?
        await db.Movies.Where(t => t.IsWatched).ToArrayAsync() :
        await db.Movies.ToArrayAsync();
}
```

`BlazorWebAppCallWebApi_Weather`

A weather data sample app that uses streaming rendering for weather data.

`BlazorWebAssemblyCallWebApi`

Calls a todo list web API from a Blazor WebAssembly app:

- `Backend`: A web API app for maintaining a todo list, based on Minimal APIs.
- `BlazorTodo`: A Blazor WebAssembly app that calls the web API with a preconfigured HttpClient for todo list CRUD operations.

Server-side scenarios for calling external web APIs

Server-based components call external web APIs using HttpClient instances, typically created using IHttpClientFactory. For guidance that applies to server-side apps, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

A server-side app doesn't include an HttpClient service. Provide an HttpClient to the app using the HttpClient factory infrastructure.

In the `Program` file:

C#

```
builder.Services.AddHttpClient();
```

The following Razor component makes a request to a web API for GitHub branches similar to the *Basic Usage* example in the Make HTTP requests using IHttpClientFactory in ASP.NET Core article.

`CallWebAPI.razor`:

razor

```
@page "/call-web-api"
@using System.Text.Json
@using System.Text.Json.Serialization
@inject IHttpClientFactory ClientFactory

<h1>Call web API from a Blazor Server Razor component</h1>

@if (getBranchesError || branches is null)
{
    <p>Unable to get branches from GitHub. Please try again later.</p>
}
else
{
    <ul>
        @foreach (var branch in branches)
        {
            <li>@branch.Name</li>
        }
    </ul>
}

@code {
    private IEnumerable<GitHubBranch>? branches = [];
    private bool getBranchesError;
    private bool shouldRender;

    protected override bool ShouldRender() => shouldRender;

    protected override async Task OnInitializedAsync()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches");
        request.Headers.Add("Accept", "application/vnd.github.v3+json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");

        var client = ClientFactory.CreateClient();
```

```
        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            using var responseStream = await
response.Content.ReadAsStreamAsync();
            branches = await JsonSerializer.DeserializeAsync
                <IEnumerable<GitHubBranch>>(responseStream);
        }
        else
        {
            getBranchesError = true;
        }

        shouldRender = true;
    }

    public class GitHubBranch
    {
        [JsonPropertyName("name")]
        public string? Name { get; set; }
    }
}
```

In the preceding example for C# 12 or later, an empty array (`[]`) is created for the `branches` variable. For earlier versions of C#, create an empty array (`Array.Empty<GitHubBranch>()`).

For an additional working example, see the server-side file upload example that uploads files to a web API controller in the ASP.NET Core Blazor file uploads article.

Service abstractions for web API calls

This section applies to Blazor Web Apps that maintain a web API in the server project or transform web API calls to an external web API.

When using the interactive WebAssembly and Auto render modes, components are prerendered by default. Auto components are also initially rendered interactively from the server before the Blazor bundle downloads to the client and the client-side runtime activates. This means that components using these render modes should be designed so that they run successfully from both the client and the server. If the component must call a server project-based API or transform a request to an external web API (one that's outside of the Blazor Web App) when running on the client, the recommended approach is to abstract that API call behind a service interface and implement client and server versions of the service:

- The client version calls the web API with a preconfigured HttpClient.

- The server version can typically access the server-side resources directly. Injecting an HttpClient on the server that makes calls back to the server isn't recommended, as the network request is typically unnecessary. Alternatively, the API might be external to the server project, but a service abstraction for the server is required to transform the request in some way, for example to add an access token to a proxied request.

When using the WebAssembly render mode, you also have the option of disabling prerendering, so the components only render from the client. For more information, see ASP.NET Core Blazor render modes.

Examples (sample apps):

- Movie list web API in the `BlazorWebAppCallWebApi` sample app.
- Streaming rendering weather data web API in the `BlazorWebAppCallWebApi_Weather` sample app.
- Weather data returned to the client in either the `BlazorWebAppOidc` (non-BFF pattern) or `BlazorWebAppOidcBff` (BFF pattern) sample apps. These apps demonstrate secure (web) API calls. For more information, see Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC).

Blazor Web App external web APIs

This section applies to Blazor Web Apps that call a web API maintained by a separate (external) project, possibly hosted on a different server.

Blazor Web Apps normally prerender client-side WebAssembly components, and Auto components render on the server during static or interactive server-side rendering (SSR). HttpClient services aren't registered by default in a Blazor Web App's main project. If the app is run with only the HttpClient services registered in the `.Client` project, as described in the Add the HttpClient service section, executing the app results in a runtime error:

> InvalidOperationException: Cannot provide a value for property 'Http' on type '... {COMPONENT}'. There is no registered service of type 'System.Net.Http.HttpClient'.

Use ***either*** of the following approaches:

- Add the HttpClient services to the server project to make the HttpClient available during SSR. Use the following service registration in the server project's `Program` file:

```
C#

builder.Services.AddHttpClient();
```

HttpClient services are provided by the shared framework, so a package reference in the app's project file isn't required.

Example: Todo list web API in the `BlazorWebAppCallWebApi` sample app

- If prerendering isn't required for a WebAssembly component that calls the web API, disable prerendering by following the guidance in ASP.NET Core Blazor render modes. If you adopt this approach, you don't need to add HttpClient services to the main project of the Blazor Web App because the component isn't prerendered on the server.

For more information, see Client-side services fail to resolve during prerendering.

Prerendered data

When prerendering, components render twice: first statically, then interactively. State doesn't automatically flow from the prerendered component to the interactive one. If a component performs asynchronous initialization operations and renders different content for different states during initialization, such as a "Loading..." progress indicator, you may see a flicker when the component renders twice.

You can address this by flowing prerendered state using the Persistent Component State API, which the `BlazorWebAppCallWebApi` and `BlazorWebAppCallWebApi_Weather` sample apps demonstrate. When the component renders interactively, it can render the same way using the same state. However, the API doesn't currently work with enhanced navigation, which you can work around by disabling enhanced navigation on links to the page (`data-enhanced-nav=false`). For more information, see the following resources:

- Prerender ASP.NET Core Razor components
- ASP.NET Core Blazor routing and navigation
- Support persistent component state across enhanced page navigations (dotnet/aspnetcore #51584)

Add the `HttpClient` service

The guidance in this section applies to client-side scenarios.

Client-side components call web APIs using a preconfigured HttpClient service, which is focused on making requests back to the server of origin. Additional HttpClient service configurations for other web APIs can be created in developer code. Requests are composed using Blazor JSON helpers or with HttpRequestMessage. Requests can include Fetch API option configuration.

The configuration examples in this section are only useful when a single web API is called for a single HttpClient instance in the app. When the app must call multiple web APIs, each with its own base address and configuration, you can adopt the following approaches, which are covered later in this article:

- Named HttpClient with IHttpClientFactory: Each web API is provided a unique name. When app code or a Razor component calls a web API, it uses a named HttpClient instance to make the call.
- Typed HttpClient: Each web API is typed. When app code or a Razor component calls a web API, it uses a typed HttpClient instance to make the call.

In the `Program` file, add an HttpClient service if it isn't already present from a Blazor project template used to create the app:

C#

```
builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
    });
```

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the app and is typically derived from the `<base>` tag's `href` value in the host page.

The most common use cases for using the client's own base address are:

- The client project (`.Client`) of a Blazor Web App (.NET 8 or later) makes web API calls from WebAssembly components or code that runs on the client in WebAssembly to APIs in the server app.
- The client project (**Client**) of a hosted Blazor WebAssembly app makes web API calls to the server project (**Server**). Note that the hosted Blazor WebAssembly project template is no longer available in .NET 8 or later. However, hosted Blazor WebAssembly apps remain supported for .NET 8.

If you're calling an external web API (not in the same URL space as the client app), set the URI to the web API's base address. The following example sets the base address of the web API to `https://localhost:5001`, where a separate web API app is running and ready to respond to requests from the client app:

C#

```
builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri("https://localhost:5001")
    });
```

JSON helpers

HttpClient is available as a preconfigured service for making requests back to the origin server.

HttpClient and JSON helpers (System.Net.Http.Json.HttpClientJsonExtensions) are also used to call third-party web API endpoints. HttpClient is implemented using the browser's Fetch API and is subject to its limitations, including enforcement of the same-origin policy, which is discussed later in this article in the *Cross-Origin Resource Sharing (CORS)* section.

The client's base address is set to the originating server's address. Inject an HttpClient instance into a component using the @inject directive:

razor

```
@using System.Net.Http
@inject HttpClient Http
```

Use the System.Net.Http.Json namespace for access to HttpClientJsonExtensions, including GetFromJsonAsync, PutAsJsonAsync, and PostAsJsonAsync:

razor

```
@using System.Net.Http.Json
```

The following sections cover JSON helpers:

- GET
- POST

- PUT
- PATCH

System.Net.Http includes additional methods for sending HTTP requests and receiving HTTP responses, for example to send a DELETE request. For more information, see the DELETE and additional extension methods section.

GET from JSON (`GetFromJsonAsync`)

GetFromJsonAsync sends an HTTP GET request and parses the JSON response body to create an object.

In the following component code, the `todoItems` are displayed by the component. GetFromJsonAsync is called when the component is finished initializing (OnInitializedAsync).

```
C#

todoItems = await Http.GetFromJsonAsync<TodoItem[]>("todoitems");
```

POST as JSON (`PostAsJsonAsync`)

PostAsJsonAsync sends a POST request to the specified URI containing the value serialized as JSON in the request body.

In the following component code, `newItemName` is provided by a bound element of the component. The `AddItem` method is triggered by selecting a `<button>` element.

```
C#

await Http.PostAsJsonAsync("todoitems", addItem);
```

PostAsJsonAsync returns an HttpResponseMessage. To deserialize the JSON content from the response message, use the ReadFromJsonAsync extension method. The following example reads JSON weather data as an array:

```
C#

var content = await response.Content.ReadFromJsonAsync<WeatherForecast[]>()
??
    Array.Empty<WeatherForecast>();
```

PUT as JSON (`PutAsJsonAsync`)

PutAsJsonAsync sends an HTTP PUT request with JSON-encoded content.

In the following component code, `editItem` values for `Name` and `IsCompleted` are provided by bound elements of the component. The item's `Id` is set when the item is selected in another part of the UI (not shown) and `EditItem` is called. The `SaveItem` method is triggered by selecting the `<button>` element. The following example doesn't show loading `todoItems` for brevity. See the GET from JSON (GetFromJsonAsync) section for an example of loading items.

C#

```
await Http.PutAsJsonAsync($"todoitems/{editItem.Id}", editItem);
```

PutAsJsonAsync returns an HttpResponseMessage. To deserialize the JSON content from the response message, use the ReadFromJsonAsync extension method. The following example reads JSON weather data as an array:

C#

```
var content = await response.Content.ReadFromJsonAsync<WeatherForecast[]>()
??
    Array.Empty<WeatherForecast>();
```

PATCH as JSON (`PatchAsJsonAsync`)

PatchAsJsonAsync sends an HTTP PATCH request with JSON-encoded content.

> ⓘ Note
>
> For more information, see JsonPatch in ASP.NET Core web API.

In the following example, PatchAsJsonAsync receives a JSON PATCH document as a plain text string with escaped quotes:

C#

```
await Http.PatchAsJsonAsync(
    $"todoitems/{id}",
    "
```

```
[{\"operationType\":2,\"path\":\"/IsComplete\",\"op\":\"replace\",\"value\":
true}]");
```

PatchAsJsonAsync returns an HttpResponseMessage. To deserialize the JSON content from the response message, use the ReadFromJsonAsync extension method. The following example reads JSON todo item data as an array. An empty array is created if no item data is returned by the method, so `content` isn't null after the statement executes:

C#

```
var response = await Http.PatchAsJsonAsync(...);
var content = await response.Content.ReadFromJsonAsync<TodoItem[]>() ??
    Array.Empty<TodoItem>();
```

Laid out with indentation, spacing, and unescaped quotes, the unencoded PATCH document appears as the following JSON:

JSON

```
[
  {
    "operationType": 2,
    "path": "/IsComplete",
    "op": "replace",
    "value": true
  }
]
```

To simplify the creation of PATCH documents in the app issuing PATCH requests, an app can use .NET JSON PATCH support, as the following guidance demonstrates.

Install the Microsoft.AspNetCore.JsonPatch NuGet package and use the API features of the package to compose a JsonPatchDocument for a PATCH request.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Add `@using` directives for the System.Text.Json, System.Text.Json.Serialization, and Microsoft.AspNetCore.JsonPatch namespaces to the top of the Razor component:

razor

```
@using System.Text.Json
@using System.Text.Json.Serialization
@using Microsoft.AspNetCore.JsonPatch
```

Compose the JsonPatchDocument for a `TodoItem` with `IsComplete` set to `true` using the Replace method:

C#

```
var patchDocument = new JsonPatchDocument<TodoItem>()
    .Replace(p => p.IsComplete, true);
```

Pass the document's operations (`patchDocument.Operations`) to the PatchAsJsonAsync call:

C#

```
private async Task UpdateItem(long id)
{
    await Http.PatchAsJsonAsync(
        $"todoitems/{id}",
        patchDocument.Operations,
        new JsonSerializerOptions()
        {
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingDefault
        });
}
```

JsonSerializerOptions.DefaultIgnoreCondition is set to JsonIgnoreCondition.WhenWritingDefault to ignore a property only if it equals the default value for its type.

Add JsonSerializerOptions.WriteIndented set to `true` if you want to present the JSON payload in a pleasant format for display. Writing indented JSON has no bearing on processing PATCH requests and isn't typically performed in production apps for web API requests.

Follow the guidance in the JsonPatch in ASP.NET Core web API article to add a PATCH controller action to the web API. Alternatively, PATCH request processing can be implemented as a Minimal API with the following steps.

Add a package reference for the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package to the web API app.

> **Note**
>
> There's no need to add a package reference for the Microsoft.AspNetCore.JsonPatch package to the app because the reference to the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` package automatically transitively adds a package reference for `Microsoft.AspNetCore.JsonPatch`.

In the `Program` file add an `@using` directive for the Microsoft.AspNetCore.JsonPatch namespace:

C#

```
using Microsoft.AspNetCore.JsonPatch;
```

Provide the endpoint to the request processing pipeline of the web API:

C#

```
app.MapPatch("/todoitems/{id}", async (long id, TodoContext db) =>
{
    if (await db.TodoItems.FindAsync(id) is TodoItem todo)
    {
        var patchDocument =
            new JsonPatchDocument<TodoItem>().Replace(p => p.IsComplete,
true);
        patchDocument.ApplyTo(todo);
        await db.SaveChangesAsync();

        return TypedResults.Ok(todo);
    }

    return TypedResults.NoContent();
});
```

> **⚠ Warning**
>
> As with the other examples in the JsonPatch in ASP.NET Core web API article, the preceding PATCH API doesn't protect the web API from over-posting attacks. For more information, see Tutorial: Create a web API with ASP.NET Core.

For a fully working PATCH experience, see the `BlazorWebAppCallWebApi` sample app.

DELETE (`DeleteAsync`) and additional extension methods

System.Net.Http includes additional extension methods for sending HTTP requests and receiving HTTP responses. HttpClient.DeleteAsync is used to send an HTTP DELETE request to a web API.

In the following component code, the `<button>` element calls the `DeleteItem` method. The bound `<input>` element supplies the `id` of the item to delete.

C#

```
await Http.DeleteAsync($"todoitems/{id}");
```

Named `HttpClient` with `IHttpClientFactory`

IHttpClientFactory services and the configuration of a named HttpClient are supported.

> **Note**
>
> An alternative to using a named HttpClient from an IHttpClientFactory is to use a typed HttpClient. For more information, see the Typed HttpClient section.

Add the Microsoft.Extensions.Http NuGet package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the `Program` file of a client project:

C#

```
builder.Services.AddHttpClient("WebAPI", client =>
    client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));
```

If the named client is to be used by prerendered client-side components of a Blazor Web App, the preceding service registration should appear in both the server project

and the `.Client` project. On the server, `builder.HostEnvironment.BaseAddress` is replaced by the web API's base address, which is described further below.

The preceding client-side example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the client-side app and is typically derived from the `<base>` tag's `href` value in the host page.

The most common use cases for using the client's own base address are:

- The client project (`.Client`) of a Blazor Web App that makes web API calls from WebAssembly/Auto components or code that runs on the client in WebAssembly to APIs in the server app at the same host address.
- The client project (**Client**) of a hosted Blazor WebAssembly app that makes web API calls to the server project (**Server**).

If you're calling an external web API (not in the same URL space as the client app) or you're configuring the services in a server-side app (for example to deal with prerendering of client-side components on the server), set the URI to the web API's base address. The following example sets the base address of the web API to `https://localhost:5001`, where a separate web API app is running and ready to respond to requests from the client app:

C#

```
builder.Services.AddHttpClient("WebAPI", client =>
    client.BaseAddress = new Uri("https://localhost:5001"));
```

In the following component code:

- An instance of IHttpClientFactory creates a named HttpClient.
- The named HttpClient is used to issue a GET request for JSON weather forecast data from the web API at `/forecast`.

razor

```
@inject IHttpClientFactory ClientFactory

...

@code {
    private Forecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        var client = ClientFactory.CreateClient("WebAPI");
```

```
        forecasts = await client.GetFromJsonAsync<Forecast[]>("forecast") ??
[];
    }
}
```

The `BlazorWebAppCallWebApi` sample app demonstrates calling a web API with a named HttpClient in its `CallTodoWebApiCsrNamedClient` component. For an additional working demonstration in a client app based on calling Microsoft Graph with a named HttpClient, see Use Graph API with ASP.NET Core Blazor WebAssembly.

Typed `HttpClient`

Typed HttpClient uses one or more of the app's HttpClient instances, default or named, to return data from one or more web API endpoints.

> **Note**
>
> An alternative to using a typed HttpClient is to use a named HttpClient from an IHttpClientFactory. For more information, see the Named HttpClient with IHttpClientFactory section.

Add the Microsoft.Extensions.Http NuGet package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

The following example issues a GET request for JSON weather forecast data from the web API at `/forecast`.

`ForecastHttpClient.cs`:

C#

```
using System.Net.Http.Json;

namespace BlazorSample.Client;

public class ForecastHttpClient(HttpClient http)
{
```

```
    public async Task<Forecast[]> GetForecastAsync() =>
        await http.GetFromJsonAsync<Forecast[]>("forecast") ?? [];
}
```

In the `Program` file of a client project:

C#

```
builder.Services.AddHttpClient<ForecastHttpClient>(client =>
    client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));
```

If the typed client is to be used by prerendered client-side components of a Blazor Web App, the preceding service registration should appear in both the server project and the `.Client` project. On the server, `builder.HostEnvironment.BaseAddress` is replaced by the web API's base address, which is described further below.

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the client-side app and is typically derived from the `<base>` tag's `href` value in the host page.

The most common use cases for using the client's own base address are:

- The client project (`.Client`) of a Blazor Web App that makes web API calls from WebAssembly/Auto components or code that runs on the client in WebAssembly to APIs in the server app at the same host address.
- The client project (**Client**) of a hosted Blazor WebAssembly app that makes web API calls to the server project (**Server**).

If you're calling an external web API (not in the same URL space as the client app) or you're configuring the services in a server-side app (for example to deal with prerendering of client-side components on the server), set the URI to the web API's base address. The following example sets the base address of the web API to `https://localhost:5001`, where a separate web API app is running and ready to respond to requests from the client app:

C#

```
builder.Services.AddHttpClient<ForecastHttpClient>(client =>
    client.BaseAddress = new Uri("https://localhost:5001"));
```

Components inject the typed HttpClient to call the web API.

In the following component code:

- An instance of the preceding `ForecastHttpClient` is injected, which creates a typed HttpClient.
- The typed HttpClient is used to issue a GET request for JSON weather forecast data from the web API.

razor

```
@inject ForecastHttpClient Http

...

@code {
    private Forecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await Http.GetForecastAsync();
    }
}
```

The `BlazorWebAppCallWebApi` sample app demonstrates calling a web API with a typed HttpClient in its `CallTodoWebApiCsrTypedClient` component. Note that the component adopts and client-side rendering (CSR) (`InteractiveWebAssembly` render mode) ***with prerendering***, so the typed client service registration appears in the `Program` file of both the server project and the `.Client` project.

Cookie-based request credentials

The guidance in this section applies to client-side scenarios that rely upon an authentication cookie.

For cookie-based authentication, which is considered more secure than bearer token authentication, cookie credentials can be sent with each web API request by calling AddHttpMessageHandler with a DelegatingHandler on a preconfigured HttpClient. The handler configures SetBrowserRequestCredentials with BrowserRequestCredentials.Include, which advises the browser to send credentials with each request, such as cookies or HTTP authentication headers, including for cross-origin requests.

`CookieHandler.cs`:

C#

```
public class CookieHandler : DelegatingHandler
{
```

```
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {

request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        request.Headers.Add("X-Requested-With", ["XMLHttpRequest"]);

        return base.SendAsync(request, cancellationToken);
    }
}
```

The `CookieHandler` is registered in the `Program` file:

C#

```
builder.Services.AddTransient<CookieHandler>();
```

The message handler is added to any preconfigured HttpClient that requires cookie authentication:

C#

```
builder.Services.AddHttpClient(...)
    .AddHttpMessageHandler<CookieHandler>();
```

For a demonstration, see Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

When composing an HttpRequestMessage, set the browser request credentials and header directly:

C#

```
var requestMessage = new HttpRequestMessage() { ... };

requestMessage.SetBrowserRequestCredentials(BrowserRequestCredentials.Includ
e);
requestMessage.Headers.Add("X-Requested-With", ["XMLHttpRequest"]);
```

`HttpClient` and `HttpRequestMessage` with Fetch API request options

The guidance in this section applies to client-side scenarios that rely upon bearer token authentication.

HttpClient (API documentation) and HttpRequestMessage can be used to customize requests. For example, you can specify the HTTP method and request headers. The following component makes a `POST` request to a web API endpoint and shows the response body.

`TodoRequest.razor`:

razor

```razor
@page "/todo-request"
@using System.Net.Http.Headers
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject HttpClient Http
@inject IAccessTokenProvider TokenProvider

<h1>ToDo Request</h1>

<h1>ToDo Request Example</h1>

<button @onclick="PostRequest">Submit POST request</button>

<p>Response body returned by the server:</p>

<p>@responseBody</p>

@code {
    private string? responseBody;

    private async Task PostRequest()
    {
        var requestMessage = new HttpRequestMessage()
        {
            Method = new HttpMethod("POST"),
            RequestUri = new Uri("https://localhost:10000/todoitems"),
            Content =
                JsonContent.Create(new TodoItem
                {
                    Name = "My New Todo Item",
                    IsComplete = false
                })
        };

        var tokenResult = await TokenProvider.RequestAccessToken();

        if (tokenResult.TryGetToken(out var token))
        {
            requestMessage.Headers.Authorization =
                new AuthenticationHeaderValue("Bearer", token.Value);

            requestMessage.Content.Headers.TryAddWithoutValidation(
                "x-custom-header", "value");

            var response = await Http.SendAsync(requestMessage);
```

```
            var responseStatusCode = response.StatusCode;

            responseBody = await response.Content.ReadAsStringAsync();
        }
    }

    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

Blazor's client-side implementation of HttpClient uses Fetch API and configures the underlying request-specific Fetch API options via HttpRequestMessage extension methods and WebAssemblyHttpRequestMessageExtensions. Set additional options using the generic SetBrowserRequestOption extension method. Blazor and the underlying Fetch API don't directly add or modify request headers. For more information on how user agents, such as browsers, interact with headers, consult external user agent documentation sets and other web resources.

The HTTP response is typically buffered to enable support for synchronous reads on the response content. To enable support for response streaming, use the SetBrowserResponseStreamingEnabled extension method on the request.

To include credentials in a cross-origin request, use the SetBrowserRequestCredentials extension method:

C#

```
requestMessage.SetBrowserRequestCredentials(BrowserRequestCredentials.Includ
e);
```

For more information on Fetch API options, see MDN web docs: WindowOrWorkerGlobalScope.fetch(): Parameters.

Handle errors

Handle web API response errors in developer code when they occur. For example, GetFromJsonAsync expects a JSON response from the web API with a `Content-Type` of `application/json`. If the response isn't in JSON format, content validation throws a NotSupportedException.

In the following example, the URI endpoint for the weather forecast data request is misspelled. The URI should be to `WeatherForecast` but appears in the call as `WeatherForcast`, which is missing the letter `e` in `Forecast`.

The GetFromJsonAsync call expects JSON to be returned, but the web API returns HTML for an unhandled exception with a `Content-Type` of `text/html`. The unhandled exception occurs because the path to `/WeatherForcast` isn't found and middleware can't serve a page or view for the request.

In OnInitializedAsync on the client, NotSupportedException is thrown when the response content is validated as non-JSON. The exception is caught in the `catch` block, where custom logic could log the error or present a friendly error message to the user.

`ReturnHTMLOnException.razor`:

razor

```
@page "/return-html-on-exception"
@using {PROJECT NAME}.Shared
@inject HttpClient Http

<h1>Fetch data but receive HTML on unhandled exception</h1>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <h2>Temperatures by Date</h2>

    <ul>
        @foreach (var forecast in forecasts)
        {
            <li>
                @forecast.Date.ToShortDateString():
                @forecast.TemperatureC &#8451;
                @forecast.TemperatureF &#8457;
            </li>
        }
    </ul>
}

<p>
    @exceptionMessage
</p>

@code {
    private WeatherForecast[]? forecasts;
    private string? exceptionMessage;
```

```
    protected override async Task OnInitializedAsync()
    {
        try
        {
            // The URI endpoint "WeatherForecast" is misspelled on purpose
on the
            // next line. See the preceding text for more information.
            forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>
("WeatherForcast");
        }
        catch (NotSupportedException exception)
        {
            exceptionMessage = exception.Message;
        }
    }
}
```

> **Note**
>
> The preceding example is for demonstration purposes. A web API can be configured to return JSON even when an endpoint doesn't exist or an unhandled exception occurs on the server.

For more information, see Handle errors in ASP.NET Core Blazor apps.

Cross-Origin Resource Sharing (CORS)

Browser security restricts a webpage from making requests to a different domain than the one that served the webpage. This restriction is called the *same-origin policy*. The same-origin policy restricts (but doesn't prevent) a malicious site from reading sensitive data from another site. To make requests from the browser to an endpoint with a different origin, the *endpoint* must enable Cross-Origin Resource Sharing (CORS).

For more information on server-side CORS, see Enable Cross-Origin Requests (CORS) in ASP.NET Core. The article's examples don't pertain directly to Razor component scenarios, but the article is useful for learning general CORS concepts.

For information on client-side CORS requests, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Antiforgery support

To add antiforgery support to an HTTP request, inject the `AntiforgeryStateProvider` and add a `RequestToken` to the headers collection as a `RequestVerificationToken`:

razor

```
@inject AntiforgeryStateProvider Antiforgery
```

C#

```
private async Task OnSubmit()
{
    var antiforgery = Antiforgery.GetAntiforgeryToken();
    var request = new HttpRequestMessage(HttpMethod.Post, "action");
    request.Headers.Add("RequestVerificationToken",
antiforgery.RequestToken);
    var response = await client.SendAsync(request);
    ...
}
```

For more information, see ASP.NET Core Blazor authentication and authorization.

Blazor framework component examples for testing web API access

Various network tools are publicly available for testing web API backend apps directly, such as Firefox Browser Developer. Blazor framework's reference source includes HttpClient test assets that are useful for testing:

HttpClientTest assets in the dotnet/aspnetcore GitHub repository

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Additional resources

General

- Cross-Origin Resource Sharing (CORS) at W3C
- Enable Cross-Origin Requests (CORS) in ASP.NET Core: Although the content applies to ASP.NET Core apps, not Razor components, the article covers general CORS concepts.

Mitigation of overposting attacks

Web APIs can be vulnerable to an *overposting* attack, also known as a *mass assignment* attack. An overposting attack occurs when a malicious user issues an HTML form POST to the server that processes data for properties that aren't part of the rendered form and that the developer doesn't wish to allow users to modify. The term "overposting" literally means that the malicious user has *over*-POSTed with the form.

For guidance on mitigating overposting attacks, see Tutorial: Create a web API with ASP.NET Core.

Server-side

- ASP.NET Core server-side and Blazor Web App additional security scenarios: Includes coverage on using HttpClient to make secure web API requests.
- Make HTTP requests using IHttpClientFactory in ASP.NET Core
- Enforce HTTPS in ASP.NET Core
- Kestrel HTTPS endpoint configuration

Client-side

- ASP.NET Core Blazor WebAssembly additional security scenarios: Includes coverage on using HttpClient to make secure web API requests.
- Use Graph API with ASP.NET Core Blazor WebAssembly
- Fetch API

Display images and documents in ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes approaches for displaying images and documents in Blazor apps.

The examples in this article are available for inspection and use in the Blazor sample apps:

dotnet/blazor-samples GitHub repository: Navigate to the app named `BlazorSample_BlazorWebApp` (8.0 or later), `BlazorSample_Server` (7.0 or earlier), or `BlazorSample_WebAssembly`.

Dynamically set an image source

The following example demonstrates how to dynamically set an image's source with a C# field.

The example in this section uses three image files, named `image1.png`, `image2.png`, and `image3.png`. The images are placed in a folder named `images` in the app's web root (`wwwroot`). The use of the `images` folder is only for demonstration purposes. You can organize static assets in any folder layout that you prefer, including serving assets directly from the `wwwroot` folder.

In the following `ShowImage1` component:

- The image's source (`src`) is dynamically set to the value of `imageSource` in C#.
- The `ShowImage` method updates the `imageSource` field based on an image `id` argument passed to the method.
- Rendered buttons call the `ShowImage` method with an image argument for each of the three available images in the `images` folder. The file name is composed using the argument passed to the method and matches one of the three images in the `images` folder.

`ShowImage1.razor`:

```razor
@page "/show-image-1"

<PageTitle>Show Image 1</PageTitle>

<h1>Show Image Example 1</h1>

@if (imageSource is not null)
{
    <p>
        <img src="@imageSource" />
    </p>
}

@for (var i = 1; i <= 3; i++)
{
    var imageId = i;
    <button @onclick="() => ShowImage(imageId)">
        Image @imageId
    </button>
}

@code {
    private string? imageSource;

    private void ShowImage(int id) => imageSource = $"images/image{id}.png";
}
```

The preceding example uses a C# field to hold the image's source data, but you can also use a C# property to hold the data.

Avoid using a loop variable directly in a lambda expression, such as `i` in the preceding `for` loop example. Otherwise, the same variable is used by all lambda expressions, which results in use of the same value in all lambdas. Capture the variable's value in a local variable. In the preceding example:

- The loop variable `i` is assigned to `imageId`.
- `imageId` is used in the lambda expression.

Alternatively, use a `foreach` loop with Enumerable.Range, which doesn't suffer from the preceding problem:

```razor
@foreach (var imageId in Enumerable.Range(1, 3))
{
```

```
    <button @onclick="() => ShowImage(imageId)">
        Image @imageId
    </button>
}
```

For more information on lambda expressions with event handling, see ASP.NET Core Blazor event handling.

Stream image or document data

An image or other document type, such as a PDF, can be directly transmitted to the client using Blazor's streaming interop features instead of hosting the file at a public URL.

The example in this section streams source data using JavaScript (JS) interop. The following `setSource` JS function:

- Can be used to stream content for the following elements: `<body>`, `<embed>`, `<iframe>`, `<img>`, `<link>`, `<object>`, `<script>`, `<style>`, and `<track>`.
- Accepts an element `id` to display the file's contents, a data stream for the document, the content type, and a title for the display element.

The function:

- Reads the provided stream into an ArrayBuffer.
- Creates a Blob to wrap the `ArrayBuffer`, setting the blob's content type.
- Creates an object URL to serve as the address for the document to be shown.
- Set's the element's title (`title`) from the `title` parameter and sets the element's source (`src`) from the created object URL.
- To prevent memory leaks, the function calls revokeObjectURL to dispose of the object URL after the element loads the resource (load event).

HTML

```
<script>
  window.setSource = async (elementId, stream, contentType, title) => {
    const arrayBuffer = await stream.arrayBuffer();
    let blobOptions = {};
    if (contentType) {
      blobOptions['type'] = contentType;
    }
    const blob = new Blob([arrayBuffer], blobOptions);
    const url = URL.createObjectURL(blob);
    const element = document.getElementById(elementId);
    element.title = title;
```

```
    element.onload = () => {
      URL.revokeObjectURL(url);
    }
    element.src = url;
  }
</script>
```

> **Note**
>
> For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

The following `ShowImage2` component:

- Injects services for an System.Net.Http.HttpClient and Microsoft.JSInterop.IJSRuntime.
- Includes an `<img>` tag to display an image.
- Has a `GetImageStreamAsync` C# method to retrieve a Stream for an image. A production app may dynamically generate an image based on the specific user or retrieve an image from storage. The following example retrieves the .NET avatar for the `dotnet` GitHub repository.
- Has a `SetImageAsync` method that's triggered on the button's selection by the user. `SetImageAsync` performs the following steps:
  - Retrieves the Stream from `GetImageStreamAsync`.
  - Wraps the Stream in a DotNetStreamReference, which allows streaming the image data to the client.
  - Invokes the `setSource` JavaScript function, which accepts the data on the client.

> **Note**
>
> Server-side apps use a dedicated HttpClient service to make requests, so no action is required by the developer of a server-side Blazor app to register an HttpClient service. Client-side apps have a default HttpClient service registration when the app is created from a Blazor project template. If an HttpClient service registration isn't present in the `Program` file of a client-side app, provide one by adding `builder.Services.AddHttpClient();`. For more information, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

`ShowImage2.razor`:

```
razor
```

```razor
@page "/show-image-2"
@inject HttpClient Http
@inject IJSRuntime JS

<PageTitle>Show Image 2</PageTitle>

<h1>Show Image Example 2</h1>

<button @onclick="SetImageAsync">
    Set Image
</button>

<div class="p-3">
    <img id="avatar" />
</div>

@code {
    private async Task<Stream> GetImageStreamAsync() =>
        await
Http.GetStreamAsync("https://avatars.githubusercontent.com/u/9141961");

    private async Task SetImageAsync()
    {
        var imageStream = await GetImageStreamAsync();
        var strRef = new DotNetStreamReference(imageStream);
        await JS.InvokeVoidAsync("setSource", "avatar", strRef, "image/png",
            ".NET GitHub avatar");
    }
}
```

The following `ShowFile` component loads either a text file (`files/quote.txt`) or a PDF file (`files/quote.pdf`) into an <iframe> element (MDN documentation).

> **⊗ Caution**
>
> Use of the `<iframe>` element in the following example is safe and doesn't require **sandboxing** because content is loaded from the app, which is a trusted source.
>
> When loading content from an untrusted source or user input, an improperly implemented `<iframe>` element risks creating security vulnerabilities.

`ShowFile.razor`:

razor

```razor
@page "/show-file"
@inject NavigationManager Navigation
@inject HttpClient Http
```

```
@inject IJSRuntime JS

<PageTitle>Show File</PageTitle>

<div class="d-flex flex-column">
    <h1>Show File Example</h1>
    <div class="mb-4">
        <button @onclick="@(() => ShowFileAsync("files/quote.txt",
                "General Ravon quote (text file)"))">
            Show text ('quote.txt')
        </button>
        <button @onclick="@(() => ShowFileAsync("files/quote.pdf",
                "General Ravon quote (PDF file)"))">
            Show PDF ('quote.pdf')
        </button>
    </div>
    <iframe id="iframe" style="height: calc(100vh - 200px)" />
</div>

@code
{
    private async Task<(Stream, string?)> DownloadFileAsync(string url)
    {
        var absoluteUrl = Navigation.ToAbsoluteUri(url);
        Console.WriteLine($"Downloading file from {absoluteUrl}");

        var response = await Http.GetAsync(absoluteUrl);
        string? contentType = null;

        if (response.Content.Headers.TryGetValues("Content-Type", out var
values))
        {
            contentType = values.FirstOrDefault();
        }

        return (await response.Content.ReadAsStreamAsync(), contentType);
    }

    private async Task ShowFileAsync(string url, string title)
    {
        var (fileStream, contentType) = await DownloadFileAsync(url);
        var strRef = new DotNetStreamReference(fileStream);
        await JS.InvokeVoidAsync("setSource", "iframe", strRef, contentType,
title);
    }
}
```

Additional resources

- ASP.NET Core Blazor file uploads
- File uploads: Upload image preview

- ASP.NET Core Blazor file downloads
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor
- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor authentication and authorization

Article • 11/19/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes ASP.NET Core's support for the configuration and management of security in Blazor apps.

Blazor uses the existing ASP.NET Core authentication mechanisms to establish the user's identity. The exact mechanism depends on how the Blazor app is hosted, server-side or client-side.

Security scenarios differ between authorization code running server-side and client-side in Blazor apps. For authorization code that runs on the server, authorization checks are able to enforce access rules for areas of the app and components. Because client-side code execution can be tampered with, authorization code executing on the client can't be trusted to absolutely enforce access rules or control the display of client-side content.

If authorization rule enforcement must be guaranteed, don't implement authorization checks in client-side code. Build a Blazor Web App that only relies on server-side rendering (SSR) for authorization checks and rule enforcement.

Razor Pages authorization conventions don't apply to routable Razor components. If a non-routable Razor component is embedded in a page of a Razor Pages app, the page's authorization conventions indirectly affect the Razor component along with the rest of the page's content.

> **Note**
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from examples in this article.

Securely maintain sensitive data and credentials

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private .NET/C# code, or private keys/tokens in client-side code, which is ***always insecure***. Client-side Blazor code should access secure services and databases through a secure web API that you control.

In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see the following resources:

- Secure authentication flows (ASP.NET Core documentation)
- Managed identities for Microsoft Azure services (this article)

For client-side and server-side local development and testing, use the Secret Manager tool to secure sensitive credentials.

Managed identities for Microsoft Azure services

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Antiforgery support

The Blazor template:

- Adds antiforgery services automatically when AddRazorComponents is called in the `Program` file.

- Adds Antiforgery Middleware by calling UseAntiforgery in its request processing pipeline in the `Program` file and requires endpoint antiforgery protection to mitigate the threats of Cross-Site Request Forgery (CSRF/XSRF). UseAntiforgery is called after UseHttpsRedirection. A call to UseAntiforgery must be placed after calls, if present, to UseAuthentication and UseAuthorization.

The AntiforgeryToken component renders an antiforgery token as a hidden field, and this component is automatically added to form (EditForm) instances. For more information, see ASP.NET Core Blazor forms overview.

The AntiforgeryStateProvider service provides access to an antiforgery token associated with the current session. Inject the service and call its GetAntiforgeryToken() method to obtain the current AntiforgeryRequestToken. For more information, see Call a web API from an ASP.NET Core Blazor app.

Blazor stores request tokens in component state, which guarantees that antiforgery tokens are available to interactive components, even when they don't have access to the request.

> **Note**
>
> Antiforgery mitigation is only required when submitting form data to the server encoded as `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain` since these are the only valid form enctypes.

For more information, see the following resources:

- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core: This article is the primary ASP.NET Core article on the subject, which applies to server-side Blazor Server, the server project of Blazor Web Apps, and Blazor integration with MVC/Razor Pages.
- ASP.NET Core Blazor forms overview: The *Antiforgery support* section of the article pertains to Blazor forms antiforgery support.

Server-side Blazor authentication

Server-side Blazor apps are configured for security in the same manner as ASP.NET Core apps. For more information, see the articles under ASP.NET Core security topics.

The authentication context is only established when the app starts, which is when the app first connects to the WebSocket over a SignalR connection with the client. Authentication can be based on a cookie or some other bearer token, but authentication

is managed via the SignalR hub and entirely within the circuit. The authentication context is maintained for the lifetime of the circuit. Apps periodically revalidate the user's authentication state every 30 minutes.

If the app must capture users for custom services or react to updates to the user, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Blazor differs from a traditional server-rendered web apps that make new HTTP requests with cookies on every page navigation. Authentication is checked during navigation events. However, cookies aren't involved. Cookies are only sent when making an HTTP request to a server, which isn't what happens when the user navigates in a Blazor app. During navigation, the user's authentication state is checked within the Blazor circuit, which you can update at any time on the server using the RevalidatingAuthenticationStateProvider abstraction.

> **Important**
>
> Implementing a custom `NavigationManager` to achieve authentication validation during navigation isn't recommended. If the app must execute custom authentication state logic during navigation, use a **custom AuthenticationStateProvider**.

> **Note**
>
> The code examples in this article adopt **nullable reference types (NRTs) and .NET compiler null-state static analysis**, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from the examples in this article.

The built-in or custom AuthenticationStateProvider service obtains authentication state data from ASP.NET Core's HttpContext.User. This is how authentication state integrates with existing ASP.NET Core authentication mechanisms.

For more information on server-side authentication, see ASP.NET Core Blazor authentication and authorization.

`IHttpContextAccessor`/`HttpContext` in Razor components

IHttpContextAccessor must be avoided with interactive rendering because there isn't a valid `HttpContext` available.

IHttpContextAccessor can be used for components that are statically rendered on the server. **However, we recommend avoiding it if possible.**

HttpContext can be used as a cascading parameter only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

C#

```
[CascadingParameter]
public HttpContext? HttpContext { get; set; }
```

For scenarios where the HttpContext is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Shared state

Server-side Blazor apps live in server memory, and multiple app sessions are hosted within the same process. For each app session, Blazor starts a circuit with its own dependency injection container scope, thus scoped services are unique per Blazor session.

> ⚠ **Warning**
>
> We don't recommend apps on the same server share state using singleton services unless extreme care is taken, as this can introduce security vulnerabilities, such as leaking user state across circuits.

You can use stateful singleton services in Blazor apps if they're specifically designed for it. For example, use of a singleton memory cache is acceptable because a memory cache requires a key to access a given entry. Assuming users don't have control over the cache keys that are used with the cache, state stored in the cache doesn't leak across circuits.

For general guidance on state management, see ASP.NET Core Blazor state management.

Server-side security of sensitive data and credentials

In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see the following resources:

- Secure authentication flows (ASP.NET Core documentation)
- Managed identities for Microsoft Azure services (Blazor documentation)

For client-side and server-side local development and testing, use the Secret Manager tool to secure sensitive credentials.

Project template

Create a new server-side Blazor app by following the guidance in Tooling for ASP.NET Core Blazor.

Visual Studio

After choosing the server-side app template and configuring the project, select the app's authentication under **Authentication type**:

- **None** (default): No authentication.
- **Individual Accounts**: User accounts are stored within the app using ASP.NET Core Identity.

Blazor Identity UI (Individual Accounts)

Blazor supports generating a full Blazor-based Identity UI when you choose the authentication option for *Individual Accounts*.

The Blazor Web App template scaffolds Identity code for a SQL Server database. The command line version uses SQLite and includes a SQLite database for Identity.

The template:

- Supports interactive server-side rendering (interactive SSR) and client-side rendering (CSR) scenarios with authenticated users.

- Adds Identity Razor components and related logic for routine authentication tasks, such as signing users in and out. The Identity components also support advanced Identity features, such as account confirmation and password recovery and multifactor authentication using a third-party app. Note that the Identity components themselves don't support interactivity.
- Adds the Identity-related packages and dependencies.
- References the Identity packages in `_Imports.razor`.
- Creates a custom user Identity class (`ApplicationUser`).
- Creates and registers an EF Core database context (`ApplicationDbContext`).
- Configures routing for the built-in Identity endpoints.
- Includes Identity validation and business logic.

To inspect the Blazor framework's Identity components, access them in the `Pages` and `Shared` folders of the Account folder in the Blazor Web App project template (reference source).

When you choose the Interactive WebAssembly or Interactive Auto render modes, the server handles all authentication and authorization requests, and the Identity components render statically on the server in the Blazor Web App's main project. The project template includes a PersistentAuthenticationStateProvider class (reference source) in the `.Client` project to synchronize the user's authentication state between the server and the browser. The class is a custom implementation of AuthenticationStateProvider. The provider uses the Persistent Component State service (PersistentComponentState) to prerender the authentication state and persist it to the page.

In the main project of a Blazor Web App, the authentication state provider is named either IdentityRevalidatingAuthenticationStateProvider (reference source) (Server interactivity solutions only) or PersistingRevalidatingAuthenticationStateProvider (reference source) (WebAssembly or Auto interactivity solutions).

Blazor Identity depends on DbContext instances not created by a factory, which is intentional because DbContext is sufficient for the project template's Identity components to render statically without supporting interactivity.

For a description on how global interactive render modes are applied to non-Identity components while at the same time enforcing static SSR for the Identity components, see ASP.NET Core Blazor render modes.

For more information on persisting prerendered state, see Prerender ASP.NET Core Razor components.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Manage authentication state in Blazor Web Apps

This section applies to Blazor Web Apps that adopt:

- Individual Accounts
- *Client-side rendering (CSR, WebAssembly-based interactivity).*

A client-side authentication state provider is only used within Blazor and isn't integrated with the ASP.NET Core authentication system. During prerendering, Blazor respects the metadata defined on the page and uses the ASP.NET Core authentication system to determine if the user is authenticated. When a user navigates from one page to another, a client-side authentication provider is used. When the user refreshes the page (full-page reload), the client-side authentication state provider isn't involved in the authentication decision on the server. Since the user's state isn't persisted by the server, any authentication state maintained client-side is lost.

To address this, the best approach is to perform authentication within the ASP.NET Core authentication system. The client-side authentication state provider only takes care of reflecting the user's authentication state. Examples for how to accomplish this with authentication state providers are demonstrated by the Blazor Web App project template and described below.

- PersistingRevalidatingAuthenticationStateProvider (reference source): For Blazor Web Apps that adopt interactive server-side rendering (interactive SSR) and client-side rendering (CSR). This is a server-side AuthenticationStateProvider that revalidates the security stamp for the connected user every 30 minutes an interactive circuit is connected. It also uses the Persistent Component State service to flow the authentication state to the client, which is then fixed for the lifetime of CSR.

- PersistingServerAuthenticationStateProvider (reference source): For Blazor Web Apps that only adopt CSR. This is a server-side AuthenticationStateProvider that

uses the Persistent Component State service to flow the authentication state to the client, which is then fixed for the lifetime of CSR.

- PersistentAuthenticationStateProvider (reference source): For Blazor Web Apps that adopt CSR. This is a client-side AuthenticationStateProvider that determines the user's authentication state by looking for data persisted in the page when it was rendered on the server. This authentication state is fixed for the lifetime of CSR. If the user needs to log in or out, a full-page reload is required. This only provides a user name and email for display purposes. It doesn't include tokens that authenticate to the server when making subsequent requests, which is handled separately using a cookie that's included on `HttpClient` requests to the server.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Additional claims and tokens from external providers

To store additional claims from external providers, see Persist additional claims and tokens from external providers in ASP.NET Core.

Azure App Service on Linux with Identity Server

Specify the issuer explicitly when deploying to Azure App Service on Linux with Identity Server. For more information, see Use Identity to secure a Web API backend for SPAs.

Inject `AuthenticationStateProvider` for services scoped to a component

Don't attempt to resolve AuthenticationStateProvider within a custom scope because it results in the creation of a new instance of the AuthenticationStateProvider that isn't correctly initialized.

To access the AuthenticationStateProvider within a service scoped to a component, inject the AuthenticationStateProvider with the @inject directive or the [Inject] attribute

and pass it to the service as a parameter. This approach ensures that the correct, initialized instance of the AuthenticationStateProvider is used for each user app instance.

`ExampleService.cs`:

```
C#

public class ExampleService
{
    public async Task<string> ExampleMethod(AuthenticationStateProvider
authStateProvider)
    {
        var authState = await
authStateProvider.GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity is not null && user.Identity.IsAuthenticated)
        {
            return $"{user.Identity.Name} is authenticated.";
        }
        else
        {
            return "The user is NOT authenticated.";
        }
    }
}
```

Register the service as scoped. In a server-side Blazor app, scoped services have a lifetime equal to the duration of the client connection circuit.

In the `Program` file:

```
C#

builder.Services.AddScoped<ExampleService>();
```

In the following `InjectAuthStateProvider` component:

- The component inherits OwningComponentBase.
- The AuthenticationStateProvider is injected and passed to `ExampleService.ExampleMethod`.
- `ExampleService` is resolved with OwningComponentBase.ScopedServices and GetRequiredService, which returns the correct, initialized instance of `ExampleService` that exists for the lifetime of the user's circuit.

`InjectAuthStateProvider.razor`:

razor

```razor
@page "/inject-auth-state-provider"
@inherits OwningComponentBase
@inject AuthenticationStateProvider AuthenticationStateProvider

<h1>Inject <code>AuthenticationStateProvider</code> Example</h1>

<p>@message</p>

@code {
    private string? message;
    private ExampleService? ExampleService { get; set; }

    protected override async Task OnInitializedAsync()
    {
        ExampleService = ScopedServices.GetRequiredService<ExampleService>
();

        message = await
ExampleService.ExampleMethod(AuthenticationStateProvider);
    }
}
```

For more information, see the guidance on OwningComponentBase in ASP.NET Core Blazor dependency injection.

Unauthorized content display while prerendering with a custom `AuthenticationStateProvider`

To avoid showing unauthorized content, for example content in an AuthorizeView component, while prerendering with a custom AuthenticationStateProvider, adopt ***one*** of the following approaches:

- Implement IHostEnvironmentAuthenticationStateProvider for the custom AuthenticationStateProvider to support prerendering: For an example implementation of IHostEnvironmentAuthenticationStateProvider, see the Blazor framework's ServerAuthenticationStateProvider implementation in ServerAuthenticationStateProvider.cs (reference source).

  > **Note**
  >
  > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags**

dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

- Disable prerendering: Indicate the render mode with the `prerender` parameter set to `false` at the highest-level component in the app's component hierarchy that isn't a root component.

  > **Note**
  >
  > Making a root component interactive, such as the `App` component, isn't supported. Therefore, prerendering can't be disabled directly by the `App` component.

  For apps based on the Blazor Web App project template, prerendering is typically disabled where the `Routes` component is used in the `App` component (`Components/App.razor`):

  ```razor
  <Routes @rendermode="new InteractiveServerRenderMode(prerender: false)" />
  ```

  Also, disable prerendering for the `HeadOutlet` component:

  ```razor
  <HeadOutlet @rendermode="new InteractiveServerRenderMode(prerender: false)" />
  ```

  You can also selectively control the render mode applied to the `Routes` component instance. For example, see ASP.NET Core Blazor render modes.

- Authenticate the user on the server before the app starts: To adopt this approach, the app must respond to a user's initial request with the Identity-based sign-in page or view and prevent any requests to Blazor endpoints until they're authenticated. For more information, see Create an ASP.NET Core app with user data protected by authorization. After authentication, unauthorized content in prerendered Razor components is only shown when the user is truly unauthorized to view the content.

User state management

In spite of the word "state" in the name, AuthenticationStateProvider isn't for storing *general user state*. AuthenticationStateProvider only indicates the user's authentication state to the app, whether they are signed into the app and who they are signed in as.

Authentication uses the same ASP.NET Core Identity authentication as Razor Pages and MVC apps. The user state stored for ASP.NET Core Identity flows to Blazor without adding additional code to the app. Follow the guidance in the ASP.NET Core Identity articles and tutorials for the Identity features to take effect in the Blazor parts of the app.

For guidance on general state management outside of ASP.NET Core Identity, see ASP.NET Core Blazor state management.

Additional security abstractions

Two additional abstractions participate in managing authentication state:

- ServerAuthenticationStateProvider (reference source): An AuthenticationStateProvider used by the Blazor framework to obtain authentication state from the server.
- RevalidatingServerAuthenticationStateProvider (reference source): A base class for AuthenticationStateProvider services used by the Blazor framework to receive an authentication state from the host environment and revalidate it at regular intervals.

  The default 30 minute revalidation interval can be adjusted in RevalidatingIdentityAuthenticationStateProvider (Areas/Identity/RevalidatingIdentityAuthenticationStateProvider.cs). The following example shortens the interval to 20 minutes:

  C#

  ```
  protected override TimeSpan RevalidationInterval =>
  TimeSpan.FromMinutes(20);
  ```

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Authentication state management at sign out

Server-side Blazor persists user authentication state for the lifetime of the circuit, including across browser tabs. To proactively sign off a user across browser tabs when the user signs out on one tab, you must implement a RevalidatingServerAuthenticationStateProvider (reference source) with a short RevalidationInterval.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Temporary redirection URL validity duration

This section applies to Blazor Web Apps.

Use the RazorComponentsServiceOptions.TemporaryRedirectionUrlValidityDuration option to get or set the lifetime of ASP.NET Core Data Protection validity for temporary redirection URLs emitted by Blazor server-side rendering. These are only used transiently, so the lifetime only needs to be long enough for a client to receive the URL and begin navigation to it. However, it should also be long enough to allow for clock skew across servers. The default value is five minutes.

In the following example the value is extended to seven minutes:

C#

```
builder.Services.AddRazorComponents(options =>
    options.TemporaryRedirectionUrlValidityDuration =
        TimeSpan.FromMinutes(7));
```

Client-side Blazor authentication

In client-side Blazor apps, client-side authentication checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks and native apps for any operating system.

Add the following:

- A package reference for the Microsoft.AspNetCore.Components.Authorization NuGet package.

  > **Note**
  >
  > For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

- The Microsoft.AspNetCore.Components.Authorization namespace to the app's `_Imports.razor` file.

To handle authentication, use the built-in or custom AuthenticationStateProvider service.

For more information on client-side authentication, see Secure ASP.NET Core Blazor WebAssembly.

`AuthenticationStateProvider` service

AuthenticationStateProvider is the underlying service used by the AuthorizeView component and cascading authentication services to obtain the authentication state for a user.

You don't typically use AuthenticationStateProvider directly. Use the AuthorizeView component or Task<AuthenticationState> approaches described later in this article. The main drawback to using AuthenticationStateProvider directly is that the component isn't notified automatically if the underlying authentication state data changes.

To implement a custom AuthenticationStateProvider, see ASP.NET Core Blazor authentication state, which includes guidance on implementing user authentication state change notifications.

Obtain a user's claims principal data

The AuthenticationStateProvider service can provide the current user's ClaimsPrincipal data, as shown in the following example.

`ClaimsPrincipalData.razor`:

```
razor
```

```razor
@page "/claims-principal-data"
@using System.Security.Claims
@inject AuthenticationStateProvider AuthenticationStateProvider

<h1>ClaimsPrincipal Data</h1>

<button @onclick="GetClaimsPrincipalData">Get ClaimsPrincipal Data</button>

<p>@authMessage</p>

@if (claims.Any())
{
    <ul>
        @foreach (var claim in claims)
        {
            <li>@claim.Type: @claim.Value</li>
        }
    </ul>
}

<p>@surname</p>

@code {
    private string? authMessage;
    private string? surname;
    private IEnumerable<Claim> claims = Enumerable.Empty<Claim>();

    private async Task GetClaimsPrincipalData()
    {
        var authState = await AuthenticationStateProvider
            .GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity is not null && user.Identity.IsAuthenticated)
        {
            authMessage = $"{user.Identity.Name} is authenticated.";
            claims = user.Claims;
            surname = user.FindFirst(c => c.Type ==
ClaimTypes.Surname)?.Value;
        }
        else
        {
            authMessage = "The user is NOT authenticated.";
        }
    }
}
```

In the preceding example:

- ClaimsPrincipal.Claims returns the user's claims (`claims`) for display in the UI.
- The line that obtains the user's surname (`surname`) calls ClaimsPrincipal.FindAll with a predicate to filter the user's claims.

If `user.Identity.IsAuthenticated` is `true` and because the user is a ClaimsPrincipal, claims can be enumerated and membership in roles evaluated.

For more information on dependency injection (DI) and services, see ASP.NET Core Blazor dependency injection and Dependency injection in ASP.NET Core. For information on how to implement a custom AuthenticationStateProvider, see ASP.NET Core Blazor authentication state.

Expose the authentication state as a cascading parameter

If authentication state data is required for procedural logic, such as when performing an action triggered by the user, obtain the authentication state data by defining a cascading parameter of type `Task<AuthenticationState>`, as the following example demonstrates.

`CascadeAuthState.razor`:

razor

```razor
@page "/cascade-auth-state"

<h1>Cascade Auth State</h1>

<p>@authMessage</p>

@code {
    private string authMessage = "The user is NOT authenticated.";

    [CascadingParameter]
    private Task<AuthenticationState>? authenticationState { get; set; }

    protected override async Task OnInitializedAsync()
    {
        if (authenticationState is not null)
        {
            var authState = await authenticationState;
            var user = authState?.User;

            if (user?.Identity is not null && user.Identity.IsAuthenticated)
            {
                authMessage = $"{user.Identity.Name} is authenticated.";
            }
        }
    }
}
```

If `user.Identity.IsAuthenticated` is `true`, claims can be enumerated and membership in roles evaluated.

Set up the `Task<`AuthenticationState`>` cascading parameter using the AuthorizeRouteView and cascading authentication state services.

When you create a Blazor app from one of the Blazor project templates with authentication enabled, the app includes the AuthorizeRouteView and the call to AddCascadingAuthenticationState shown in the following example. A client-side Blazor app includes the required service registrations as well. Additional information is presented in the Customize unauthorized content with the Router component section.

razor

```
<Router ...>
    <Found ...>
        <AuthorizeRouteView RouteData="routeData"
            DefaultLayout="typeof(Layout.MainLayout)" />
        ...
    </Found>
</Router>
```

In the `Program` file, register cascading authentication state services:

C#

```
builder.Services.AddCascadingAuthenticationState();
```

In a client-side Blazor app, add authorization services to the `Program` file:

C#

```
builder.Services.AddAuthorizationCore();
```

In a server-side Blazor app, services for options and authorization are already present, so no further steps are required.

Authorization

After a user is authenticated, *authorization* rules are applied to control what the user can do.

Access is typically granted or denied based on whether:

- A user is authenticated (signed in).
- A user is in a *role*.
- A user has a *claim*.
- A *policy* is satisfied.

Each of these concepts is the same as in an ASP.NET Core MVC or Razor Pages app. For more information on ASP.NET Core security, see the articles under ASP.NET Core Security and Identity.

`AuthorizeView` component

The AuthorizeView component selectively displays UI content depending on whether the user is authorized. This approach is useful when you only need to *display* data for the user and don't need to use the user's identity in procedural logic.

The component exposes a `context` variable of type AuthenticationState (`@context` in Razor syntax), which you can use to access information about the signed-in user:

razor

```
<AuthorizeView>
    <p>Hello, @context.User.Identity?.Name!</p>
</AuthorizeView>
```

You can also supply different content for display if the user isn't authorized with a combination of the Authorized and NotAuthorized parameters:

razor

```
<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
        <p><button @onclick="HandleClick">Authorized Only Button</button>
</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
</AuthorizeView>

@code {
    private void HandleClick() { ... }
}
```

Although the AuthorizeView component controls the visibility of elements based on the user's authorization status, it doesn't enforce security on the event handler itself. In the preceding example, the `HandleClick` method is only associated with a button visible to authorized users, but nothing prevents invoking this method from other places. To ensure method-level security, implement additional authorization logic within the handler itself or in the relevant API.

Razor components of Blazor Web Apps never display `<NotAuthorized>` content when authorization fails server-side during static server-side rendering (static SSR). The server-side ASP.NET Core pipeline processes authorization on the server. Use server-side techniques to handle unauthorized requests. For more information, see ASP.NET Core Blazor render modes.

> ⚠ **Warning**
>
> Client-side markup and methods associated with an **AuthorizeView** are only protected from view and execution in the ***rendered UI*** in client-side Blazor apps. In order to protect authorized content and secure methods in client-side Blazor, the content is usually supplied by a secure, authorized web API call to a server API and never stored in the app. For more information, see **Call a web API from an ASP.NET Core Blazor app** and **ASP.NET Core Blazor WebAssembly additional security scenarios**.

The content of Authorized and NotAuthorized can include arbitrary items, such as other interactive components.

Authorization conditions, such as roles or policies that control UI options or access, are covered in the Authorization section.

If authorization conditions aren't specified, AuthorizeView uses a default policy:

- Authenticated (signed-in) users are authorized.
- Unauthenticated (signed-out) users are unauthorized.

The AuthorizeView component can be used in the `NavMenu` component (`Shared/NavMenu.razor`) to display a NavLink component (NavLink), but note that this approach only removes the list item from the rendered output. It doesn't prevent the user from navigating to the component. Implement authorization separately in the destination component.

Role-based and policy-based authorization

The AuthorizeView component supports *role-based* or *policy-based* authorization.

For role-based authorization, use the Roles parameter. In the following example, the user must have a role claim for either the `Admin` or `Superuser` roles:

razor

```
<AuthorizeView Roles="Admin, Superuser">
    <p>You have an 'Admin' or 'Superuser' role claim.</p>
</AuthorizeView>
```

To require a user have both `Admin` and `Superuser` role claims, nest AuthorizeView components:

razor

```
<AuthorizeView Roles="Admin">
    <p>User: @context.User</p>
    <p>You have the 'Admin' role claim.</p>
    <AuthorizeView Roles="Superuser" Context="innerContext">
        <p>User: @innerContext.User</p>
        <p>You have both 'Admin' and 'Superuser' role claims.</p>
    </AuthorizeView>
</AuthorizeView>
```

The preceding code establishes a `Context` for the inner AuthorizeView component to prevent an AuthenticationState context collision. The AuthenticationState context is accessed in the outer AuthorizeView with the standard approach for accessing the context (`@context.User`). The context is accessed in the inner AuthorizeView with the named `innerContext` context (`@innerContext.User`).

For more information, including configuration guidance, see Role-based authorization in ASP.NET Core.

For policy-based authorization, use the Policy parameter with a single policy name:

razor

```
<AuthorizeView Policy="Over21">
    <p>You satisfy the 'Over21' policy.</p>
</AuthorizeView>
```

To handle the case where the user should satisfy one of several policies, create a policy that confirms that the user satisfies other policies.

To handle the case where the user must satisfy several policies simultaneously, take *either* of the following approaches:

- Create a policy for AuthorizeView that confirms that the user satisfies several other policies.
- Nest the policies in multiple AuthorizeView components:

```razor
<AuthorizeView Policy="Over21">
    <AuthorizeView Policy="LivesInCalifornia">
        <p>You satisfy the 'Over21' and 'LivesInCalifornia' policies.
</p>
    </AuthorizeView>
</AuthorizeView>
```

Claims-based authorization is a special case of policy-based authorization. For example, you can define a policy that requires users to have a certain claim. For more information, see Policy-based authorization in ASP.NET Core.

If neither Roles nor Policy is specified, AuthorizeView uses the default policy:

- Authenticated (signed-in) users are authorized.
- Unauthenticated (signed-out) users are unauthorized.

Because .NET string comparisons are case-sensitive, matching role and policy names is also case-sensitive. For example, `Admin` (uppercase `A`) is not treated as the same role as `admin` (lowercase `a`).

Pascal case is typically used for role and policy names (for example, `BillingAdministrator`), but the use of Pascal case isn't a strict requirement. Different casing schemes, such as camel case, kebab case, and snake case, are permitted. Using spaces in role and policy names is unusual but permitted by the framework. For example, `billing administrator` is an unusual role or policy name format in .NET apps, but it's a valid role or policy name.

Content displayed during asynchronous authentication

Blazor allows for authentication state to be determined *asynchronously*. The primary scenario for this approach is in client-side Blazor apps that make a request to an external endpoint for authentication.

While authentication is in progress, AuthorizeView displays no content. To display content while authentication occurs, assign content to the Authorizing parameter:

razor

```
<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <Authorizing>
        <p>You can only see this content while authentication is in
progress.</p>
    </Authorizing>
</AuthorizeView>
```

This approach isn't normally applicable to server-side Blazor apps. Server-side Blazor apps know the authentication state as soon as the state is established. Authorizing content can be provided in an app's AuthorizeView component, but the content is never displayed.

`[Authorize]` attribute

The [Authorize] attribute is available in Razor components:

razor

```
@page "/"
@attribute [Authorize]

You can only see this if you're signed in.
```

> **ⓘ Important**
>
> Only use **[Authorize]** on `@page` components reached via the Blazor router. Authorization is only performed as an aspect of routing and *not* for child components rendered within a page. To authorize the display of specific parts within a page, use **AuthorizeView** instead.

The [Authorize] attribute also supports role-based or policy-based authorization. For role-based authorization, use the Roles parameter:

razor

```
@page "/"
@attribute [Authorize(Roles = "Admin, Superuser")]

<p>You can only see this if you're in the 'Admin' or 'Superuser' role.</p>
```

For policy-based authorization, use the Policy parameter:

razor

```
@page "/"
@attribute [Authorize(Policy = "Over21")]

<p>You can only see this if you satisfy the 'Over21' policy.</p>
```

If neither Roles nor Policy is specified, [Authorize] uses the default policy:

- Authenticated (signed-in) users are authorized.
- Unauthenticated (signed-out) users are unauthorized.

When the user isn't authorized and if the app doesn't customize unauthorized content with the Router component, the framework automatically displays the following fallback message:

HTML

```
Not authorized.
```

Resource authorization

To authorize users for resources, pass the request's route data to the Resource parameter of AuthorizeRouteView.

In the Router.Found content for a requested route:

razor

```
<AuthorizeRouteView Resource="routeData" RouteData="routeData"
    DefaultLayout="typeof(MainLayout)" />
```

For more information on how authorization state data is passed and used in procedural logic, see the Expose the authentication state as a cascading parameter section.

When the AuthorizeRouteView receives the route data for the resource, authorization policies have access to RouteData.PageType and RouteData.RouteValues that permit

custom logic to make authorization decisions.

In the following example, an `EditUser` policy is created in AuthorizationOptions for the app's authorization service configuration (AddAuthorizationCore) with the following logic:

- Determine if a route value exists with a key of `id`. If the key exists, the route value is stored in `value`.
- In a variable named `id`, store `value` as a string or set an empty string value (`string.Empty`).
- If `id` isn't an empty string, assert that the policy is satisfied (return `true`) if the string's value starts with `EMP`. Otherwise, assert that the policy fails (return `false`).

In the `Program` file:

- Add namespaces for Microsoft.AspNetCore.Components and System.Linq:

  C#

  ```
  using Microsoft.AspNetCore.Components;
  using System.Linq;
  ```

- Add the policy:

  C#

  ```
  options.AddPolicy("EditUser", policy =>
      policy.RequireAssertion(context =>
      {
          if (context.Resource is RouteData rd)
          {
              var routeValue = rd.RouteValues.TryGetValue("id", out var
  value);
              var id = Convert.ToString(value,
                  System.Globalization.CultureInfo.InvariantCulture) ??
  string.Empty;

              if (!string.IsNullOrEmpty(id))
              {
                  return id.StartsWith("EMP",
  StringComparison.InvariantCulture);
              }
          }

          return false;
      })
  );
  ```

The preceding example is an oversimplified authorization policy, merely used to demonstrate the concept with a working example. For more information on creating and configuring authorization policies, see Policy-based authorization in ASP.NET Core.

In the following `EditUser` component, the resource at `/users/{id}/edit` has a route parameter for the user's identifier (`{id}`). The component uses the preceding `EditUser` authorization policy to determine if the route value for `id` starts with `EMP`. If `id` starts with `EMP`, the policy succeeds and access to the component is authorized. If `id` starts with a value other than `EMP` or if `id` is an empty string, the policy fails, and the component doesn't load.

`EditUser.razor`:

razor

```razor
@page "/users/{id}/edit"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize(Policy = "EditUser")]

<h1>Edit User</h1>

<p>The "EditUser" policy is satisfied! <code>Id</code> starts with 'EMP'.
</p>

@code {
    [Parameter]
    public string? Id { get; set; }
}
```

Customize unauthorized content with the `Router` component

The Router component, in conjunction with the AuthorizeRouteView component, allows the app to specify custom content if:

- The user fails an [Authorize] condition applied to the component. The markup of the <NotAuthorized> element is displayed. The [Authorize] attribute is covered in the [Authorize] attribute section.
- Asynchronous authorization is in progress, which usually means that the process of authenticating the user is in progress. The markup of the <Authorizing> element is displayed.

ⓘ Important

Blazor router features that display `<NotAuthorized>` and `<NotFound>` content aren't operational during static server-side rendering (static SSR) because request processing is entirely handled by ASP.NET Core middleware pipeline request processing and Razor components aren't rendered at all for unauthorized or bad requests. Use server-side techniques to handle unauthorized and bad requests during static SSR. For more information, see **ASP.NET Core Blazor render modes**.

razor

```
<Router ...>
    <Found ...>
        <AuthorizeRouteView ...>
            <NotAuthorized>
                ...
            </NotAuthorized>
            <Authorizing>
                ...
            </Authorizing>
        </AuthorizeRouteView>
    </Found>
</Router>
```

The content of Authorized and NotAuthorized can include arbitrary items, such as other interactive components.

Note

The preceding requires cascading authentication state services registration in the app's `Program` file:

C#

```
builder.Services.AddCascadingAuthenticationState();
```

If NotAuthorized content isn't specified, the AuthorizeRouteView uses the following fallback message:

HTML

```
Not authorized.
```

An app created from the Blazor WebAssembly project template with authentication enabled includes a `RedirectToLogin` component, which is positioned in the

`<NotAuthorized>` content of the Router component. When a user isn't authenticated (`context.User.Identity?.IsAuthenticated != true`), the `RedirectToLogin` component redirects the browser to the `authentication/login` endpoint for authentication. The user is returned to the requested URL after authenticating with the identity provider.

Procedural logic

If the app is required to check authorization rules as part of procedural logic, use a cascaded parameter of type `Task<AuthenticationState>` to obtain the user's ClaimsPrincipal. `Task<AuthenticationState>` can be combined with other services, such as `IAuthorizationService`, to evaluate policies.

In the following example:

- The `user.Identity.IsAuthenticated` executes code for authenticated (signed-in) users.
- The `user.IsInRole("admin")` executes code for users in the 'Admin' role.
- The `(await AuthorizationService.AuthorizeAsync(user, "content-editor")).Succeeded` executes code for users satisfying the 'content-editor' policy.

A server-side Blazor app includes the appropriate namespaces when created from the project template. In a client-side Blazor app, confirm the presence of the Microsoft.AspNetCore.Authorization and Microsoft.AspNetCore.Components.Authorization namespaces either in the component or in the app's `_Imports.razor` file:

razor

```
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
```

`ProceduralLogic.razor`:

razor

```
@page "/procedural-logic"
@inject IAuthorizationService AuthorizationService

<h1>Procedural Logic Example</h1>

<button @onclick="@DoSomething">Do something important</button>

@code {
    [CascadingParameter]
```

```
    private Task<AuthenticationState>? authenticationState { get; set; }

    private async Task DoSomething()
    {
        if (authenticationState is not null)
        {
            var authState = await authenticationState;
            var user = authState?.User;

            if (user is not null)
            {
                if (user.Identity is not null &&
user.Identity.IsAuthenticated)
                {
                    // ...
                }

                if (user.IsInRole("Admin"))
                {
                    // ...
                }

                if ((await AuthorizationService.AuthorizeAsync(user,
"content-editor"))
                    .Succeeded)
                {
                    // ...
                }
            }
        }
    }
}
```

Troubleshoot errors

Common errors:

- **Authorization requires a cascading parameter of type `Task<AuthenticationState>`. Consider using `CascadingAuthenticationState` to supply this.**
- **`null` value is received for `authenticationStateTask`**

It's likely that the project wasn't created using a server-side Blazor template with authentication enabled.

In .NET 7 or earlier, wrap a `<CascadingAuthenticationState>` around some part of the UI tree, for example around the Blazor router:

```
```

razor

```
<CascadingAuthenticationState>
    <Router ...>
        ...
    </Router>
</CascadingAuthenticationState>
```

In .NET 8 or later, don't use the CascadingAuthenticationState component:

diff

```
- <CascadingAuthenticationState>
      <Router ...>
          ...
      </Router>
- </CascadingAuthenticationState>
```

Instead, add cascading authentication state services to the service collection in the `Program` file:

C#

```
builder.Services.AddCascadingAuthenticationState();
```

The CascadingAuthenticationState component (.NET 7 or earlier) or services provided by AddCascadingAuthenticationState (.NET 8 or later) supplies the `Task<`AuthenticationState`>` cascading parameter, which in turn it receives from the underlying AuthenticationStateProvider dependency injection service.

Personally Identifiable Information (PII)

Microsoft uses the GDPR definition for 'personal data' (GDPR 4.1) when documentation discusses Personally Identifiable Information (PII).

PII refers any information relating to an identified or identifiable natural person. An identifiable natural person is one who can be identified, directly or indirectly, with any of the following:

- Name
- Identification number
- Location coordinates
- Online identifier
- Other specific factors

    - Physical
    - Physiological
    - Genetic
    - Mental (psychological)
    - Economic
    - Cultural
    - Social identity

Additional resources

- Server-side and Blazor Web App resources
    - Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app
    - Quickstart: Protect an ASP.NET Core web API with Microsoft identity platform
    - Configure ASP.NET Core to work with proxy servers and load balancers: Includes guidance on:
        - Using Forwarded Headers Middleware to preserve HTTPS scheme information across proxy servers and internal networks.
        - Additional scenarios and use cases, including manual scheme configuration, request path changes for correct request routing, and forwarding the request scheme for Linux and non-IIS reverse proxies.
- Microsoft identity platform documentation
    - Overview
    - OAuth 2.0 and OpenID Connect protocols on the Microsoft identity platform
    - Microsoft identity platform and OAuth 2.0 authorization code flow
    - Microsoft identity platform ID tokens
    - Microsoft identity platform access tokens
- ASP.NET Core security topics
- Configure Windows Authentication in ASP.NET Core
- Build a custom version of the Authentication.MSAL JavaScript library
- Awesome Blazor: Authentication community sample links
- ASP.NET Core Blazor Hybrid authentication and authorization

ASP.NET Core Blazor authentication state

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to create a custom authentication state provider and receive user authentication state change notifications in code.

The general approaches taken for server-side and client-side Blazor apps are similar but differ in their exact implementations, so this article pivots between server-side Blazor apps and client-side Blazor apps. Use the pivot selector at the top of the article to change the article's pivot to match the type of Blazor project that you're working with:

- Server-side Blazor apps (**Server** pivot): Blazor Server for .NET 7 or earlier and the server project of a Blazor Web App for .NET 8 or later.
- Client-side Blazor apps (**Blazor WebAssembly** pivot): Blazor WebAssembly for all versions of .NET or the `.Client` project of a Blazor Web App for .NET 8 or later.

Abstract `AuthenticationStateProvider` class

The Blazor framework includes an abstract AuthenticationStateProvider class to provide information about the authentication state of the current user with the following members:

- GetAuthenticationStateAsync: Asynchronously gets the authentication state of the current user.
- AuthenticationStateChanged: An event that provides notification when the authentication state has changed. For example, this event may be raised if a user signs in or out of the app.
- NotifyAuthenticationStateChanged: Raises an authentication state changed event.

Implement a custom `AuthenticationStateProvider`

The app must reference the Microsoft.AspNetCore.Components.Authorization NuGet package, which provides authentication and authorization support for Blazor apps.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Configure the following authentication, authorization, and cascading authentication state services in the `Program` file.

When you create a Blazor app from one of the Blazor project templates with authentication enabled, the app is preconfigured with the following service registrations, which includes exposing the authentication state as a cascading parameter. For more information, see ASP.NET Core Blazor authentication and authorization with additional information presented in the article's Customize unauthorized content with the Router component section.

C#

```
using Microsoft.AspNetCore.Components.Authorization;

...

builder.Services.AddAuthorization();
builder.Services.AddCascadingAuthenticationState();
```

Subclass AuthenticationStateProvider and override GetAuthenticationStateAsync to create the user's authentication state. In the following example, all users are authenticated with the username `mrfibuli`.

`CustomAuthStateProvider.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
    {
        var identity = new ClaimsIdentity(
        [
            new Claim(ClaimTypes.Name, "mrfibuli"),
```

```
        ], "Custom Authentication");

        var user = new ClaimsPrincipal(identity);

        return Task.FromResult(new AuthenticationState(user));
    }
}
```

> **Note**
>
> The preceding code that creates a new ClaimsIdentity uses simplified collection initialization introduced with C# 12 (.NET 8). For more information, see Collection expressions - C# language reference.

The `CustomAuthStateProvider` service is registered in the `Program` file. Register the service *scoped* with AddScoped:

C#

```
builder.Services.AddScoped<AuthenticationStateProvider,
CustomAuthStateProvider>();
```

If it isn't present, add an @using statement to the `_Imports.razor` file to make the Microsoft.AspNetCore.Components.Authorization namespace available across components:

razor

```
@using Microsoft.AspNetCore.Components.Authorization;
```

Confirm or change the route view component to an AuthorizeRouteView in the Router component definition. The location of the `Router` component differs depending on the type of app. Use search to locate the component if you're unaware of its location in the project.

razor

```
<Router ...>
    <Found ...>
        <AuthorizeRouteView RouteData="routeData"
            DefaultLayout="typeof(Layout.MainLayout)" />
        ...
    </Found>
</Router>
```

> **Note**
>
> When you create a Blazor app from one of the Blazor project templates with authentication enabled, the app includes the AuthorizeRouteView component. For more information, see ASP.NET Core Blazor authentication and authorization with additional information presented in the article's Customize unauthorized content with the Router component section.

The following example AuthorizeView component demonstrates the authenticated user's name:

razor

```
<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
</AuthorizeView>
```

For guidance on the use of AuthorizeView, see ASP.NET Core Blazor authentication and authorization.

Authentication state change notifications

A custom AuthenticationStateProvider can invoke NotifyAuthenticationStateChanged on the AuthenticationStateProvider base class to notify consumers of the authentication state change to rerender.

The following example is based on implementing a custom AuthenticationStateProvider by following the guidance in the Implement a custom AuthenticationStateProvider section earlier in this article. If you already followed the guidance in that section, the following `CustomAuthStateProvider` replaces the one shown in the section.

The following `CustomAuthStateProvider` implementation exposes a custom method, `AuthenticateUser`, to sign in a user and notify consumers of the authentication state change.

`CustomAuthStateProvider.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
    {
        var identity = new ClaimsIdentity();
        var user = new ClaimsPrincipal(identity);

        return Task.FromResult(new AuthenticationState(user));
    }

    public void AuthenticateUser(string userIdentifier)
    {
        var identity = new ClaimsIdentity(
        [
            new Claim(ClaimTypes.Name, userIdentifier),
        ], "Custom Authentication");

        var user = new ClaimsPrincipal(identity);

        NotifyAuthenticationStateChanged(
            Task.FromResult(new AuthenticationState(user)));
    }
}
```

> **Note**
>
> The preceding code that creates a new ClaimsIdentity uses simplified collection initialization introduced with C# 12 (.NET 8). For more information, see Collection expressions - C# language reference.

In a component:

- Inject AuthenticationStateProvider.
- Add a field to hold the user's identifier.
- Add a button and a method to cast the AuthenticationStateProvider to `CustomAuthStateProvider` and call `AuthenticateUser` with the user's identifier.

razor

```
@inject AuthenticationStateProvider AuthenticationStateProvider

<input @bind="userIdentifier" />
<button @onclick="SignIn">Sign in</button>

<AuthorizeView>
```

```
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
</AuthorizeView>

@code {
    public string userIdentifier = string.Empty;

    private void SignIn()
    {
        ((CustomAuthStateProvider)AuthenticationStateProvider)
            .AuthenticateUser(userIdentifier);
    }
}
```

The preceding approach can be enhanced to trigger notifications of authentication state changes via a custom service. The following `CustomAuthenticationService` class maintains the current user's claims principal in a backing field (`currentUser`) with an event (`UserChanged`) that the authentication state provider can subscribe to, where the event invokes NotifyAuthenticationStateChanged. With the additional configuration later in this section, the `CustomAuthenticationService` can be injected into a component with logic that sets the `CurrentUser` to trigger the `UserChanged` event.

`CustomAuthenticationService.cs`:

C#

```
using System.Security.Claims;

public class CustomAuthenticationService
{
    public event Action<ClaimsPrincipal>? UserChanged;
    private ClaimsPrincipal? currentUser;

    public ClaimsPrincipal CurrentUser
    {
        get { return currentUser ?? new(); }
        set
        {
            currentUser = value;

            if (UserChanged is not null)
            {
                UserChanged(currentUser);
            }
        }
```

```
    }
}
```

In the `Program` file, register the `CustomAuthenticationService` in the dependency injection container:

C#

```
builder.Services.AddScoped<CustomAuthenticationService>();
```

The following `CustomAuthStateProvider` subscribes to the `CustomAuthenticationService.UserChanged` event. The `GetAuthenticationStateAsync` method returns the user's authentication state. Initially, the authentication state is based on the value of the `CustomAuthenticationService.CurrentUser`. When the user changes, a new authentication state is created for the new user (`new AuthenticationState(newUser)`) for calls to `GetAuthenticationStateAsync`:

C#

```
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    private AuthenticationState authenticationState;

    public CustomAuthStateProvider(CustomAuthenticationService service)
    {
        authenticationState = new AuthenticationState(service.CurrentUser);

        service.UserChanged += (newUser) =>
        {
            authenticationState = new AuthenticationState(newUser);

NotifyAuthenticationStateChanged(Task.FromResult(authenticationState));
        };
    }

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        Task.FromResult(authenticationState);
}
```

The following component's `SignIn` method creates a claims principal for the user's identifier to set on `CustomAuthenticationService.CurrentUser`:

razor

```razor
@using System.Security.Claims
@inject CustomAuthenticationService AuthService

<input @bind="userIdentifier" />
<button @onclick="SignIn">Sign in</button>

<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
</AuthorizeView>

@code {
    public string userIdentifier = string.Empty;

    private void SignIn()
    {
        var currentUser = AuthService.CurrentUser;

        var identity = new ClaimsIdentity(
            [
                new Claim(ClaimTypes.Name, userIdentifier),
            ],
            "Custom Authentication");

        var newUser = new ClaimsPrincipal(identity);

        AuthService.CurrentUser = newUser;
    }
}
```

> [!NOTE]
> The preceding code that creates a new ClaimsIdentity uses simplified collection initialization introduced with C# 12 (.NET 8). For more information, see Collection expressions - C# language reference.

Additional resources

- Server-side unauthorized content display while prerendering with a custom AuthenticationStateProvider
- How to access an AuthenticationStateProvider from a DelegatingHandler set up using an IHttpClientFactory
- Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)

- Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

Secure ASP.NET Core Blazor WebAssembly

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Blazor WebAssembly apps are secured in the same manner as single-page applications (SPAs). There are several approaches for authenticating users to SPAs, but the most common and comprehensive approach is to use an implementation based on the OAuth 2.0 protocol, such as OpenID Connect (OIDC).

The Blazor WebAssembly security documentation primarily focuses on how to accomplish user authentication and authorization tasks. For OAuth 2.0/OIDC general concept coverage, see the resources in the main overview article's *Additional resources* section.

Client-side/SPA security of sensitive data and credentials

A Blazor WebAssembly app's .NET/C# codebase is served to clients, and the app's code can't be protected from inspection and tampering by users. Never place credentials or secrets into a Blazor WebAssembly app, such as app secrets, connection strings, passwords, private .NET/C# code, or other sensitive data.

To protect .NET/C# code and use ASP.NET Core Data Protection features to secure data, use a server-side ASP.NET Core web API. Have the client-side Blazor WebAssembly app call the server-side web API for secure app features and data processing. For more information, see Call a web API from an ASP.NET Core Blazor app and the articles in this node.

For local development testing, the Secret Manager tool is recommended for securing sensitive data.

Authentication library

Blazor WebAssembly supports authenticating and authorizing apps using OIDC via the Microsoft.AspNetCore.Components.WebAssembly.Authentication library using the Microsoft identity platform. The library provides a set of primitives for seamlessly authenticating against ASP.NET Core backends. The library can authenticate against any third-party Identity Provider (IP) that supports OIDC, which are called OpenID Providers (OP).

The authentication support in the Blazor WebAssembly Library (`Authentication.js`) is built on top of the Microsoft Authentication Library (MSAL, msal.js), which is used to handle the underlying authentication protocol details. The Blazor WebAssembly Library only supports the Proof Key for Code Exchange (PKCE) authorization code flow. Implicit grant isn't supported.

Other options for authenticating SPAs exist, such as the use of SameSite cookies. However, the engineering design of Blazor WebAssembly uses OAuth and OIDC as the best option for authentication in Blazor WebAssembly apps. Token-based authentication based on JSON Web Tokens (JWTs) was chosen over cookie-based authentication for functional and security reasons:

- Using a token-based protocol offers fewer vulnerabilities, as the tokens aren't sent in all requests.
- The tokens are explicitly sent to the server, so server endpoints don't require protection against Cross-Site Request Forgery (CSRF). This allows you to host Blazor WebAssembly apps alongside MVC or Razor pages apps.
- Tokens have narrower permissions than cookies. For example, tokens can't be used to manage the user account or change a user's password unless such functionality is explicitly implemented.
- Tokens have a short lifetime, one hour, which limits the attack window. Tokens can also be revoked at any time.
- Self-contained JWTs offer guarantees to the client and server about the authentication process. For example, a client has the means to detect and validate that the tokens it receives are legitimate and were emitted as part of a given authentication process. If a third party attempts to switch a token in the middle of the authentication process, the client can detect the switched token and avoid using it.
- Tokens with OAuth and OIDC don't rely on the user agent behaving correctly to ensure that the app is secure.
- Token-based protocols, such as OAuth and OIDC, allow for authenticating and authorizing users in standalone Blazor Webassembly apps with the same set of security characteristics.

> **ⓘ Important**
>
> For versions of ASP.NET Core that adopt Duende Identity Server in Blazor project templates, Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

Authentication process with OIDC

The Microsoft.AspNetCore.Components.WebAssembly.Authentication library offers several primitives to implement authentication and authorization using OIDC. In broad terms, authentication works as follows:

- When an anonymous user selects the login button or requests a Razor component or page with the [Authorize] attribute applied, the user is redirected to the app's login page (`/authentication/login`).
- In the login page, the authentication library prepares for a redirect to the authorization endpoint. The authorization endpoint is outside of the Blazor WebAssembly app and can be hosted at a separate origin. The endpoint is responsible for determining whether the user is authenticated and for issuing one or more tokens in response. The authentication library provides a login callback to receive the authentication response.
  - If the user isn't authenticated, the user is redirected to the underlying authentication system, which is usually ASP.NET Core Identity.
  - If the user was already authenticated, the authorization endpoint generates the appropriate tokens and redirects the browser back to the login callback endpoint (`/authentication/login-callback`).
- When the Blazor WebAssembly app loads the login callback endpoint (`/authentication/login-callback`), the authentication response is processed.
  - If the authentication process completes successfully, the user is authenticated and optionally sent back to the original protected URL that the user requested.
  - If the authentication process fails for any reason, the user is sent to the login failed page (`/authentication/login-failed`), where an error is displayed.

`Authentication` component

The `Authentication` component (`Authentication.razor`) handles remote authentication operations and permits the app to:

- Configure app routes for authentication states.
- Set UI content for authentication states.
- Manage authentication state.

Authentication actions, such as registering or signing in a user, are passed to the Blazor framework's RemoteAuthenticatorViewCore<TAuthenticationState> component, which persists and controls state across authentication operations.

For more information and examples, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Authorization

In Blazor WebAssembly apps, authorization checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks or native apps for any operating system.

Always perform authorization checks on the server within any API endpoints accessed by your client-side app.

Customize authentication

Blazor WebAssembly provides methods to add and retrieve additional parameters for the underlying Authentication library to conduct remote authentication operations with external identity providers.

To pass additional parameters, NavigationManager supports passing and retrieving history entry state when performing external location changes. For more information, see the following resources:

- Blazor *Fundamentals* > *Routing and navigation* article
  - Navigation history state
  - Navigation options
- MDN documentation: History API

The state stored by the History API provides the following benefits for remote authentication:

- The state passed to the secured app endpoint is tied to the navigation performed to authenticate the user at the `authentication/login` endpoint.
- Extra work encoding and decoding data is avoided.

- Vulnerabilities are reduced. Unlike using the query string to store navigation state, a top-level navigation or influence from a different origin can't set the state stored by the History API.
- The history entry is replaced upon successful authentication, so the state attached to the history entry is removed and doesn't require clean up.

InteractiveRequestOptions represents the request to the identity provider for logging in or provisioning an access token.

NavigationManagerExtensions provides the NavigateToLogin method for a login operation and NavigateToLogout for a logout operation. The methods call NavigationManager.NavigateTo, setting the history entry state with a passed InteractiveRequestOptions or a new InteractiveRequestOptions instance created by the method for:

- A user signing in (InteractionType.SignIn) with the current URI for the return URL.
- A user signing out (InteractionType.SignOut) with the return URL.

The following authentication scenarios are covered in the ASP.NET Core Blazor WebAssembly additional security scenarios article:

- Customize the login process
- Logout with a custom return URL
- Customize options before obtaining a token interactively
- Customize options when using an IAccessTokenProvider
- Obtain the login path from authentication options

Require authorization for the entire app

Apply the [Authorize] attribute (API documentation) to each Razor component of the app using ***one*** of the following approaches:

- In the app's Imports file, add an @using directive for the Microsoft.AspNetCore.Authorization namespace with an @attribute directive for the [Authorize] attribute.

  `_Imports.razor`:

  razor

  ```
  @using Microsoft.AspNetCore.Authorization
  @attribute [Authorize]
  ```

Allow anonymous access to the `Authentication` component to permit redirection to the identity provider. Add the following Razor code to the `Authentication` component under its @page directive.

`Authentication.razor`:

```razor
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@attribute [AllowAnonymous]
```

- Add the attribute to each Razor component under the @page directive:

```razor
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
```

> ⓘ Note
>
> Setting an AuthorizationOptions.FallbackPolicy to a policy with RequireAuthenticatedUser is **not** supported.

Use one identity provider app registration per app

Some of the articles under this *Overview* pertain to Blazor hosting scenarios that involve two or more apps. A standalone Blazor WebAssembly app uses web API with authenticated users to access server resources and data provided by a server app.

When this scenario is implemented in documentation examples, ***two*** identity provider registrations are used, one for the client app and one for the server app. Using separate registrations, for example in Microsoft Entra ID, isn't strictly required. However, using two registrations is a security best practice because it isolates the registrations by app. Using separate registrations also allows independent configuration of the client and server registrations.

Refresh tokens

Although refresh tokens can't be secured in Blazor WebAssembly apps, they can be used if you implement them with appropriate security strategies.

For standalone Blazor WebAssembly apps in ASP.NET Core in .NET 6 or later, we recommend using:

- The OAuth 2.0 Authorization Code flow (Code) with Proof Key for Code Exchange (PKCE).
- A refresh token that has a short expiration.
- A rotated refresh token.
- A refresh token with an expiration after which a new interactive authorization flow is required to refresh the user's credentials.

For more information, see the following resources:

- Microsoft identity platform refresh tokens: Refresh token lifetime
- OAuth 2.0 for Browser-Based Apps (IETF specification)

Establish claims for users

Apps often require claims for users based on a web API call to a server. For example, claims are frequently used to establish authorization in an app. In these scenarios, the app requests an access token to access the service and uses the token to obtain user data for creating claims.

For examples, see the following resources:

- Additional scenarios: Customize the user
- ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles

Prerendering support

Prerendering isn't supported for authentication endpoints (`/authentication/` path segment).

For more information, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Azure App Service on Linux with Identity Server

Specify the issuer explicitly when deploying to Azure App Service on Linux with Identity Server.

For more information, see Use Identity to secure a Web API backend for SPAs.

Windows Authentication

We don't recommend using Windows Authentication with Blazor Webassembly or with any other SPA framework. We recommend using token-based protocols instead of Windows Authentication, such as OIDC with Active Directory Federation Services (ADFS).

If Windows Authentication is used with Blazor Webassembly or with any other SPA framework, additional measures are required to protect the app from cross-site request forgery (CSRF) tokens. The same concerns that apply to cookies apply to Windows Authentication with the addition that Windows Authentication doesn't offer a mechanism to prevent sharing of the authentication context across origins. Apps using Windows Authentication without additional protection from CSRF should at least be restricted to an organization's intranet and not be used on the open Internet.

For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Secure a SignalR hub

To secure a SignalR hub in the server API project, apply the [Authorize] attribute to the hub class or to methods of the hub class.

In a client project with prerendering, such as hosted Blazor WebAssembly (ASP.NET Core in .NET 7 or earlier) or a Blazor Web App (ASP.NET Core in .NET 8 or later), see the guidance in ASP.NET Core Blazor SignalR guidance.

In a client project component without prerendering, such as standalone Blazor WebAssembly, or non-browser apps, supply an access token to the hub connection, as the following example demonstrates. For more information, see Authentication and authorization in ASP.NET Core SignalR.

razor

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IAccessTokenProvider TokenProvider
@inject NavigationManager Navigation

...

var tokenResult = await TokenProvider.RequestAccessToken();

if (tokenResult.TryGetToken(out var token))
```

```
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"),
            options => { options.AccessTokenProvider = () =>
Task.FromResult(token?.Value); })
        .Build();

  ...
}
```

Logging

This section applies to Blazor WebAssembly apps in ASP.NET Core in .NET 7 or later.

To enable debug or trace logging, see the *Authentication logging (Blazor WebAssembly)* section in a 7.0 or later version of the ASP.NET Core Blazor logging article.

The WebAssembly sandbox

The WebAssembly *sandbox* restricts access to the environment of the system executing WebAssembly code, including access to I/O subsystems, system storage and resources, and the operating system. The isolation between WebAssembly code and the system that executes the code makes WebAssembly a safe coding framework for systems. However, WebAssembly is vulnerable to side-channel attacks at the hardware level. Normal precautions and due diligence in sourcing hardware and placing limitations on accessing hardware apply.

WebAssembly isn't owned or maintained by Microsoft.

For more information, see the following W3C resources:

- WebAssembly: Security
- WebAssembly Specification: Security Considerations
- W3C WebAssembly Community Group: Feedback and issues : The W3C WebAssembly Community Group link is only provided for reference, making it clear that WebAssembly security vulnerabilities and bugs are patched on an ongoing basis, often reported and addressed by browser. ***Don't send feedback or bug reports on Blazor to the W3C WebAssembly Community Group.*** Blazor feedback should be reported to the Microsoft ASP.NET Core product unit . If the Microsoft product unit determines that an underlying problem with WebAssembly exists, they take the appropriate steps to report the problem to the W3C WebAssembly Community Group.

Implementation guidance

Articles under this *Overview* provide information on authenticating users in Blazor WebAssembly apps against specific providers.

Standalone Blazor WebAssembly apps:

- General guidance for OIDC providers and the WebAssembly Authentication Library
- Microsoft Accounts
- Microsoft Entra ID (ME-ID)
- Azure Active Directory (AAD) B2C

Further configuration guidance is found in the following articles:

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Use Graph API with ASP.NET Core Blazor WebAssembly

Use the Authorization Code flow with PKCE

Microsoft identity platform's Microsoft Authentication Library for JavaScript (MSAL) v2.0 or later provides support for the Authorization Code flow with Proof Key for Code Exchange (PKCE) and Cross-Origin Resource Sharing (CORS) for single-page applications, including Blazor.

Microsoft doesn't recommend using Implicit grant.

For more information, see the following resources:

- Authentication flow support in MSAL: Implicit grant
- Microsoft identity platform and implicit grant flow: Prefer the auth code flow
- Microsoft identity platform and OAuth 2.0 authorization code flow

Additional resources

- Microsoft identity platform documentation
  - General documentation
  - Access tokens
- Configure ASP.NET Core to work with proxy servers and load balancers
  - Using Forwarded Headers Middleware to preserve HTTPS scheme information across proxy servers and internal networks.
  - Additional scenarios and use cases, including manual scheme configuration, request path changes for correct request routing, and forwarding the request

scheme for Linux and non-IIS reverse proxies.

- Prerendering with authentication
- WebAssembly: Security
- WebAssembly Specification: Security Considerations

Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

Article • 11/19/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Standalone Blazor WebAssembly apps can be secured with ASP.NET Core Identity by following the guidance in this article.

Endpoints for registering, logging in, and logging out

Instead of using the default UI provided by ASP.NET Core Identity for SPA and Blazor apps, which is based on Razor Pages, call MapIdentityApi in a backend API to add JSON API endpoints for registering and logging in users with ASP.NET Core Identity. Identity API endpoints also support advanced features, such as two-factor authentication and email verification.

On the client, call the `/register` endpoint to register a user with their email address and password:

C#

```
var result = await _httpClient.PostAsJsonAsync(
    "register", new
    {
        email,
        password
    });
```

On the client, log in a user with cookie authentication using the `/login` endpoint with `useCookies` query string set to `true`:

C#

```
var result = await _httpClient.PostAsJsonAsync(
    "login?useCookies=true", new
    {
        email,
        password
    });
```

The backend server API establishes cookie authentication with a call to AddIdentityCookies on the authentication builder:

C#

```
builder.Services
    .AddAuthentication(IdentityConstants.ApplicationScheme)
    .AddIdentityCookies();
```

Token authentication

For native and mobile scenarios where some clients don't support cookies, the login API provides a parameter to request tokens.

> ⚠ **Warning**
>
> We recommend using cookies for browser-based apps instead of tokens because the browser handles cookies without exposing them to JavaScript. If you opt to use token-based security in web apps, you're responsible for ensuring the tokens are kept secure.

A custom token (one that is proprietary to the ASP.NET Core Identity platform) is issued that can be used to authenticate subsequent requests. The token should be passed in the `Authorization` header as a bearer token. A refresh token is also provided. This token allows the app to request a new token when the old one expires without forcing the user to log in again.

The tokens are not standard JSON Web Tokens (JWTs). The use of custom tokens is intentional, as the built-in Identity API is meant primarily for simple scenarios. The token option is not intended to be a fully-featured identity service provider or token server, but instead an alternative to the cookie option for clients that can't use cookies.

The following guidance begins the process of implementing token-based authentication with the login API. Custom code is required to complete the implementation. For more information, see Use Identity to secure a Web API backend for SPAs.

Instead of the backend server API establishing cookie authentication with a call to AddIdentityCookies on the authentication builder, the server API sets up bearer token auth with the AddBearerToken extension method. Specify the scheme for bearer authentication tokens with IdentityConstants.BearerScheme.

In `Backend/Program.cs`, change the authentication services and configuration to the following:

C#

```
builder.Services
    .AddAuthentication()
    .AddBearerToken(IdentityConstants.BearerScheme);
```

In `BlazorWasmAuth/Identity/CookieAuthenticationStateProvider.cs`, remove the `useCookies` query string parameter in the `LoginAsync` method of the `CookieAuthenticationStateProvider`:

diff

```
- login?useCookies=true
+ login
```

At this point, you must provide custom code to parse the AccessTokenResponse on the client and manage the access and refresh tokens. For more information, see Use Identity to secure a Web API backend for SPAs.

Additional Identity scenarios

Scenarios covered by the Blazor documentation set:

- Account confirmation, password management, and recovery codes
- Two-factor authentication (2FA) with a TOTP authenticator app

For information on additional Identity scenarios provided by the API, see Use Identity to secure a Web API backend for SPAs:

- Secure selected endpoints
- User info management

Use secure authentication flows to maintain sensitive data and credentials

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

Sample apps

In this article, sample apps serve as a reference for standalone Blazor WebAssembly apps that access ASP.NET Core Identity through a backend web API. The demonstration includes two apps:

- `Backend`: A backend web API app that maintains a user identity store for ASP.NET Core Identity.
- `BlazorWasmAuth`: A standalone Blazor WebAssembly frontend app with user authentication.

Access the sample apps through the latest version folder from the repository's root with the following link. The samples are provided for .NET 8 or later. See the `README` file in the `BlazorWebAssemblyStandaloneWithIdentity` folder for steps on how to run the sample apps.

View or download sample code (how to download)

Backend web API app packages and code

The backend web API app maintains a user identity store for ASP.NET Core Identity.

Packages

The app uses the following NuGet packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore

- Microsoft.EntityFrameworkCore.InMemory
- NSwag.AspNetCore

If your app is to use a different EF Core database provider than the in-memory provider, don't create a package reference in your app for `Microsoft.EntityFrameworkCore.InMemory`.

In the app's project file (`.csproj`), invariant globalization is configured.

Sample app code

App settings configure backend and frontend URLs:

- `Backend` app (`BackendUrl`): `https://localhost:7211`
- `BlazorWasmAuth` app (`FrontendUrl`): `https://localhost:7171`

The Backend.http file can be used for testing the weather data request. Note that the `BlazorWasmAuth` app must be running to test the endpoint, and the endpoint is hardcoded into the file. For more information, see Use .http files in Visual Studio 2022.

The following setup and configuration is found in the app's Program file.

User identity with cookie authentication is added by calling AddAuthentication and AddIdentityCookies. Services for authorization checks are added by a call to AddAuthorizationBuilder.

Only recommended for demonstrations, the app uses the EF Core in-memory database provider for the database context registration (AddDbContext). The in-memory database provider makes it easy to restart the app and test the registration and login user flows. Each run starts with a fresh database, but the app includes test user seeding demonstration code, which is described later in this article. If the database is changed to SQLite, users are saved between sessions, but the database must be created through migrations, as shown in the EF Core getting started tutorial†. You can use other relational providers such as SQL Server for your production code.

> **Note**
>
> †The EF Core getting started tutorial uses PowerShell commands to execute database migrations when using Visual Studio. An alternative approach in Visual Studio is to use the Connected Services UI:
>
> In **Solution Explorer**, double-click **Connected Services**. In **Service Dependencies** > **SQL Server Express LocalDB**, select the ellipsis (`...`) followed by either **Add**

migration to create a migration or **Update database** to update the database.

Configure Identity to use the EF Core database and expose the Identity endpoints via the calls to AddIdentityCore, AddEntityFrameworkStores, and AddApiEndpoints.

A Cross-Origin Resource Sharing (CORS) policy is established to permit requests from the frontend and backend apps. Fallback URLs are configured for the CORS policy if app settings don't provide them:

- `Backend` app (`BackendUrl`): `https://localhost:5001`
- `BlazorWasmAuth` app (`FrontendUrl`): `https://localhost:5002`

Services and endpoints for Swagger/OpenAPI are included for web API documentation and development testing. For more information on NSwag, see Get started with NSwag and ASP.NET Core.

User role claims are sent from a Minimal API at the `/roles` endpoint.

Routes are mapped for Identity endpoints by calling `MapIdentityApi<AppUser>()`.

A logout endpoint (`/Logout`) is configured in the middleware pipeline to sign users out.

To secure an endpoint, add the RequireAuthorization extension method to the route definition. For a controller, add the [Authorize] attribute to the controller or action.

For more information on basic patterns for initialization and configuration of a DbContext instance, see DbContext Lifetime, Configuration, and Initialization in the EF Core documentation.

Frontend standalone Blazor WebAssembly app packages and code

A standalone Blazor WebAssembly frontend app demonstrates user authentication and authorization to access a private webpage.

Packages

The app uses the following NuGet packages:

- Microsoft.AspNetCore.Components.WebAssembly.Authentication
- Microsoft.Extensions.Http
- Microsoft.AspNetCore.Components.WebAssembly

- Microsoft.AspNetCore.Components.WebAssembly.DevServer

Sample app code

The `Models` folder contains the app's models:

- FormResult (Identity/Models/FormResult.cs): Response for login and registration.
- UserInfo (Identity/Models/UserInfo.cs): User info from identity endpoint to establish claims.

The IAccountManagement interface (Identity/CookieHandler.cs) provides account management services.

The CookieAuthenticationStateProvider class (Identity/CookieAuthenticationStateProvider.cs) handles state for cookie-based authentication and provides account management service implementations described by the `IAccountManagement` interface. The `LoginAsync` method explicitly enables cookie authentication via the `useCookies` query string value of `true`. The class also manages creating role claims for authenticated users.

The CookieHandler class (Identity/CookieHandler.cs) ensures cookie credentials are sent with each request to the backend web API, which handles Identity and maintains the Identity data store.

The wwwroot/appsettings.file provides backend and frontend URL endpoints.

The App component exposes the authentication state as a cascading parameter. For more information, see ASP.NET Core Blazor authentication and authorization.

The MainLayout component and NavMenu component use the AuthorizeView component to selectively display content based on the user's authentication status.

The following components handle common user authentication tasks, making use of `IAccountManagement` services:

- Register component (Components/Identity/Register.razor)
- Login component (Components/Identity/Login.razor)
- Logout component (Components/Identity/Logout.razor)

The PrivatePage component (Components/Pages/PrivatePage.razor) requires authentication and shows the user's claims.

Services and configuration is provided in the Program file (Program.cs):

- The cookie handler is registered as a scoped service.

- Authorization services are registered.
- The custom authentication state provider is registered as a scoped service.
- The account management interface (`IAccountManagement`) is registered.
- The base host URL is configured for a registered HTTP client instance.
- The base backend URL is configured for a registered HTTP client instance that's used for auth interactions with the backend web API. The HTTP client uses the cookie handler to ensure that cookie credentials are sent with each request.

Call AuthenticationStateProvider.NotifyAuthenticationStateChanged when the user's authentication state changes. For an example, see the `LoginAsync` and `LogoutAsync` methods of the CookieAuthenticationStateProvider class (Identity/CookieAuthenticationStateProvider.cs).

> ⚠ **Warning**
>
> The **AuthorizeView** component selectively displays UI content depending on whether the user is authorized. All content within a Blazor WebAssembly app placed in an **AuthorizeView** component is discoverable without authentication, so sensitive content should be obtained from a backend server-based web API after authentication succeeds. For more information, see the following resources:
>
> - ASP.NET Core Blazor authentication and authorization
> - Call a web API from an ASP.NET Core Blazor app
> - ASP.NET Core Blazor WebAssembly additional security scenarios

Test user seeding demonstration

The `SeedData` class (SeedData.cs) demonstrates how to create test users for development. The test user, named Leela, signs into the app with the email address `leela@contoso.com`. The user's password is set to `Passw0rd!`. Leela is given `Administrator` and `Manager` roles for authorization, which enables the user to access the manager page at `/private-manager-page` but not the editor page at `/private-editor-page`.

> ⚠ **Warning**
>
> Never allow test user code to run in a production environment. `SeedData.InitializeAsync` is only called in the `Development` environment in the `Program` file:

C#

```
if (builder.Environment.IsDevelopment())
{
    await using var scope = app.Services.CreateAsyncScope();
    await SeedData.InitializeAsync(scope.ServiceProvider);
}
```

Roles

Role claims aren't sent back from the `manage/info` endpoint to create user claims for users of the `BlazorWasmAuth` app. Role claims are managed independently via a separate request in the `GetAuthenticationStateAsync` method of the CookieAuthenticationStateProvider class (Identity/CookieAuthenticationStateProvider.cs) after the user is authenticated in the `Backend` project.

In the `CookieAuthenticationStateProvider`, a roles request is made to the `/roles` endpoint of the `Backend` server API project. The response is read into a string by calling ReadAsStringAsync(). JsonSerializer.Deserialize deserializes the string into a custom `RoleClaim` array. Finally, the claims are added to the user's claims collection.

In the `Backend` server API's `Program` file, a Minimal API manages the `/roles` endpoint. Claims of RoleClaimType are selected into an anonymous type and serialized for return to the `BlazorWasmAuth` project with TypedResults.Json.

The roles endpoint requires authorization by calling RequireAuthorization. If you decide not to use Minimal APIs in favor of controllers for secure server API endpoints, be sure to set the [Authorize] attribute on controllers or actions.

Cross-domain hosting (same-site configuration)

The sample apps are configured for hosting both apps at the same domain. If you host the `Backend` app at a different domain than the `BlazorWasmAuth` app, uncomment the code that configures the cookie (ConfigureApplicationCookie) in the `Backend` app's `Program` file. The default values are:

- Same-site mode: SameSiteMode.Lax
- Secure policy: CookieSecurePolicy.SameAsRequest

Change the values to:

- Same-site mode: SameSiteMode.None
- Secure policy: CookieSecurePolicy.Always

diff

```
- options.Cookie.SameSite = SameSiteMode.Lax;
- options.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;
+ options.Cookie.SameSite = SameSiteMode.None;
+ options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
```

For more information on same-site cookie settings, see the following resources:

- Set-Cookie: SameSite=<samesite-value> (MDN documentation)
- Cookies: HTTP State Management Mechanism (RFC Draft 6265, Section 4.1)

Antiforgery support

Only the logout endpoint (`/logout`) in the `Backend` app requires attention to mitigate the threat of Cross-Site Request Forgery (CSRF).

The logout endpoint checks for an empty body to prevent CSRF attacks. By requiring a body, the request must be made from JavaScript, which is the only way to access the authentication cookie. The logout endpoint can't be accessed by a form-based POST. This prevents a malicious site from logging the user out.

Furthermore, the endpoint is protected by authorization (RequireAuthorization) to prevent anonymous access.

The `BlazorWasmAuth` client app is simply required to pass an empty object `{}` in the body of the request.

Outside of the logout endpoint, antiforgery mitigation is only required when submitting form data to the server encoded as `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`. Blazor manages CSRF mitigation for forms in most cases. For more information, see ASP.NET Core Blazor authentication and authorization and ASP.NET Core Blazor forms overview.

Requests to other server API endpoints (web API) with `application/json`-encoded content and CORS enabled doesn't require CSRF protection. This is why no CSRF protection is required for the `Backend` app's data processing (`/data-processing`)

endpoint. The roles (`/roles`) endpoint doesn't need CSRF protection because it's a GET endpoint that doesn't modify any state.

Troubleshoot

Logging

To enable debug or trace logging for Blazor WebAssembly authentication, see ASP.NET Core Blazor logging.

Common errors

Check each project's configuration. Confirm that the URLs are correct:

- `Backend` project
  - `appsettings.json`
    - `BackendUrl`
    - `FrontendUrl`
  - `Backend.http`: `Backend_HostAddress`
- `BlazorWasmAuth` project: `wwwroot/appsettings.json`
  - `BackendUrl`
  - `FrontendUrl`

If the configuration appears correct:

- Analyze application logs.
- Examine the network traffic between the `BlazorWasmAuth` app and `Backend` app with the browser's developer tools. Often, an exact error message or a message with a clue to what's causing the problem is returned to the client by the backend app after making a request. Developer tools guidance is found in the following articles:
- Google Chrome (Google documentation)
- Microsoft Edge
- Mozilla Firefox (Mozilla documentation)

The documentation team responds to document feedback and bugs in articles. Open an issue using the **Open a documentation issue** link at the bottom of the article. The team

isn't able to provide product support. Several public support forums are available to assist with troubleshooting an app. We recommend the following:

- Stack Overflow (tag: blazor)
- ASP.NET Core Slack Team
- Blazor Gitter

The preceding forums are not owned or controlled by Microsoft.

For non-security, non-sensitive, and non-confidential reproducible framework bug reports, open an issue with the ASP.NET Core product unit. Don't open an issue with the product unit until you've thoroughly investigated the cause of a problem and can't resolve it on your own and with the help of the community on a public support forum. The product unit isn't able to troubleshoot individual apps that are broken due to simple misconfiguration or use cases involving third-party services. If a report is sensitive or confidential in nature or describes a potential security flaw in the product that cyberattackers may exploit, see Reporting security issues and bugs (dotnet/aspnetcore GitHub repository).

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes, or app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE for any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in InPrivate or Incognito mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is

installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in InPrivate or Incognito mode. Some browsers require the URL of the app.
    - Microsoft Edge: Use `-inprivate`.
    - Google Chrome: Use `--incognito --new-window {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
    - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE for any change to the app, test user, or provider configuration.

App upgrades

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Clear the local system's NuGet package caches by executing dotnet nuget locals all --clear from a command shell.
2. Delete the project's `bin` and `obj` folders.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

> **Note**
>
> Use of package versions incompatible with the app's target framework isn't supported. For information on a package, use the NuGet Gallery or FuGet

Package Explorer.

Inspect the user's claims

To troubleshoot problems with user claims, the following `UserClaims` component can be used directly in apps or serve as the basis for further customization.

`UserClaims.razor`:

razor

```
@page "/user-claims"
@using System.Security.Claims
@attribute [Authorize]

<PageTitle>User Claims</PageTitle>

<h1>User Claims</h1>

**Name**: @AuthenticatedUser?.Identity?.Name

<h2>Claims</h2>

@foreach (var claim in AuthenticatedUser?.Claims ?? Array.Empty<Claim>())
{
    <p class="claim">@(claim.Type): @claim.Value</p>
}

@code {
    [CascadingParameter]
    private Task<AuthenticationState>? AuthenticationState { get; set; }

    public ClaimsPrincipal? AuthenticatedUser { get; set; }

    protected override async Task OnInitializedAsync()
    {
        if (AuthenticationState is not null)
        {
            var state = await AuthenticationState;
            AuthenticatedUser = state.User;
        }
    }
}
```

Additional resources

- AuthenticationStateProvider service
- Client-side SignalR cross-origin negotiation for authentication

Account confirmation and password recovery in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

Article • 12/11/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure an ASP.NET Core Blazor WebAssembly app with ASP.NET Core Identity with email confirmation and password recovery.

> **Note**
>
> This article only applies standalone Blazor WebAssembly apps with ASP.NET Core Identity. To implement email confirmation and password recovery for Blazor Web Apps, see Account confirmation and password recovery in ASP.NET Core Blazor.

Namespaces and article code examples

The namespaces used by the examples in this article are:

- `Backend` for the backend server web API project ("server project" in this article).
- `BlazorWasmAuth` for the front-end client standalone Blazor WebAssembly app ("client project" in this article).

These namespaces correspond to the projects in the `BlazorWebAssemblyStandaloneWithIdentity` sample solution in the dotnet/blazor-samples GitHub repository. For more information, see Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

If you aren't using the `BlazorWebAssemblyStandaloneWithIdentity` sample solution, change the namespaces in the code examples to use the namespaces of your projects.

Select and configure an email provider for the server project

In this article, Mailchimp's Transactional API is used via Mandrill.net to send email. We recommend using an email service to send email rather than SMTP. SMTP is difficult to configure and secure properly. Whichever email service you use, access their guidance for .NET apps, create an account, configure an API key for their service, and install any NuGet packages required.

In the server project, create a class to hold the secret email provider API key. The example in this article uses a class named `AuthMessageSenderOptions` with an `EmailAuthKey` property to hold the key.

`AuthMessageSenderOptions.cs`:

C#

```
namespace Backend;

public class AuthMessageSenderOptions
{
    public string? EmailAuthKey { get; set; }
}
```

Register the `AuthMessageSenderOptions` configuration instance in the server project's `Program` file:

C#

```
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);
```

Configure a user secret for the provider's security key

If the server project has already been initialized for the Secret Manager tool, it will already have a app secrets identifier (`<AppSecretsId>`) in its project file (`.csproj`). In Visual Studio, you can tell if the app secrets ID is present by looking at the **Properties** panel when the project is selected in **Solution Explorer**. If the app hasn't been initialized, execute the following command in a command shell opened to the server project's directory. In Visual Studio, you can use the Developer PowerShell command prompt (use

the `cd` command to change the directory to the server project after you open the command shell).

.NET CLI

```
dotnet user-secrets init
```

Set the API key with the Secret Manager tool. In the following example, the key name is `EmailAuthKey` to match `AuthMessageSenderOptions.EmailAuthKey`, and the key is represented by the `{KEY}` placeholder. Execute the following command with the API key:

.NET CLI

```
dotnet user-secrets set "EmailAuthKey" "{KEY}"
```

If using Visual Studio, you can confirm the secret is set by right-clicking the server project in **Solution Explorer** and selecting **Manage User Secrets**.

For more information, see Safe storage of app secrets in development in ASP.NET Core.

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

Implement `IEmailSender` in the server project

The following example is based on Mailchimp's Transactional API using Mandrill.net. For a different provider, refer to their documentation on how to implement sending an email message.

Add the Mandrill.net NuGet package to the server project.

Add the following `EmailSender` class to implement IEmailSender<TUser>. In the following example, `AppUser` is an IdentityUser. The message HTML markup can be further customized. As long as the `message` passed to `MandrillMessage` starts with the `<` character, the Mandrill.net API assumes that the message body is composed in HTML.

`EmailSender.cs`:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using Mandrill;
using Mandrill.Model;

namespace Backend;

public class EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor,
    ILogger<EmailSender> logger) : IEmailSender<AppUser>
{
    private readonly ILogger logger = logger;

    public AuthMessageSenderOptions Options { get; } =
optionsAccessor.Value;

    public Task SendConfirmationLinkAsync(AppUser user, string email,
        string confirmationLink) => SendEmailAsync(email, "Confirm your
email",
        "<html lang=\"en\"><head></head><body>Please confirm your account by
" +
        $"<a href='{confirmationLink}'>clicking here</a>.</body></html>");

    public Task SendPasswordResetLinkAsync(AppUser user, string email,
        string resetLink) => SendEmailAsync(email, "Reset your password",
        "<html lang=\"en\"><head></head><body>Please reset your password by
" +
        $"<a href='{resetLink}'>clicking here</a>.</body></html>");

    public Task SendPasswordResetCodeAsync(AppUser user, string email,
        string resetCode) => SendEmailAsync(email, "Reset your password",
        "<html lang=\"en\"><head></head><body>Please reset your password " +
        $"using the following code:<br>{resetCode}</body></html>");

    public async Task SendEmailAsync(string toEmail, string subject, string
message)
    {
        if (string.IsNullOrEmpty(Options.EmailAuthKey))
        {
            throw new Exception("Null EmailAuthKey");
        }

        await Execute(Options.EmailAuthKey, subject, message, toEmail);
    }
```

```
    public async Task Execute(string apiKey, string subject, string message,
        string toEmail)
    {
        var api = new MandrillApi(apiKey);
        var mandrillMessage = new MandrillMessage("sarah@contoso.com",
toEmail,
            subject, message);
        await api.Messages.SendAsync(mandrillMessage);

        logger.LogInformation("Email to {EmailAddress} sent!", toEmail);
    }
}
```

> **Note**
>
> Body content for messages might require special encoding for the email service provider. If links in the message body can't be followed in the email message, consult the service provider's documentation to troubleshoot the problem.

Add the following IEmailSender<TUser> service registration to the server project's `Program` file:

C#

```
builder.Services.AddTransient<IEmailSender<AppUser>, EmailSender>();
```

Configure the server project to require email confirmation

In the server project's `Program` file, require a confirmed email address to sign in to the app.

Locate the line that calls AddIdentityCore and set the RequireConfirmedEmail property to `true`:

diff

```
- builder.Services.AddIdentityCore<AppUser>()
+ builder.Services.AddIdentityCore<AppUser>(o =>
o.SignIn.RequireConfirmedEmail = true)
```

Update the client project's account registration response

In the client project's `Register` component (`Components/Identity/Register.razor`), change the message to users on a successful account registration to instruct them to confirm their account. The following example includes a link to trigger Identity on the server to resend the confirmation email.

diff

```
- You successfully registered and can <a href="login">login</a> to the app.
+ You successfully registered. You must now confirm your account by clicking
+ the link in the email that was sent to you. After confirming your account,
+ you can <a href="login">login</a> to the app.
+ <a href="resendConfirmationEmail">Resend confirmation email</a>
```

Update seed data code to confirm seeded accounts

In the server project's seed data class (`SeedData.cs`), change the code in the `InitializeAsync` method to confirm the seeded accounts, which avoids requiring email address confirmation for each test run of the solution with one of the accounts:

diff

```
- if (appUser is not null && user.RoleList is not null)
- {
-     await userManager.AddToRolesAsync(appUser, user.RoleList);
- }
+ if (appUser is not null)
+ {
+     if (user.RoleList is not null)
+     {
+         await userManager.AddToRolesAsync(appUser, user.RoleList);
+     }
+
+     var token = await
userManager.GenerateEmailConfirmationTokenAsync(appUser);
+     await userManager.ConfirmEmailAsync(appUser, token);
+ }
```

Enable account confirmation after a site has users

Enabling account confirmation on a site with users locks out all the existing users. Existing users are locked out because their accounts aren't confirmed. Use one of the following approaches, which are beyond the scope of this article:

- Update the database to mark all existing users as confirmed.
- Batch-send emails with confirmation links to all existing users, which requires each user to confirm their own account.

Password recovery

Password recovery requires the server app to adopt an email provider in order to send password reset codes to users. Therefore, follow the guidance earlier in this article to adopt an email provider:

- Select and configure an email provider for the server project
- Configure a user secret for the provider's security key
- Implement IEmailSender in the server project

Password recovery is a two-step process:

1. A POST request is made to the `/forgotPassword` endpoint provided by MapIdentityApi in the server project. A message in the UI instructs the user to check their email for a reset code.
2. A POST request is made to the `/resetPassword` endpoint of the server project with the user's email address, password reset code, and new password.

The preceding steps are demonstrated by the following implementation guidance for the sample solution.

In the client project, add the following method signatures to the `IAccountManagement` class (`Identity/IAccountManagement.cs`):

C#

```
public Task<bool> ForgotPasswordAsync(string email);

public Task<FormResult> ResetPasswordAsync(string email, string resetCode,
    string newPassword);
```

In the client project, add implementations for the preceding methods in the `CookieAuthenticationStateProvider` class (`Identity/CookieAuthenticationStateProvider.cs`):

C#

```
public async Task<bool> ForgotPasswordAsync(string email)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "forgotPassword", new
            {
                email
            });

        if (result.IsSuccessStatusCode)
        {
            return true;
        }
    }
    catch { }

    return false;
}

public async Task<FormResult> ResetPasswordAsync(string email, string
resetCode,
    string newPassword)
{
    string[] defaultDetail = ["An unknown error prevented resetting the
password."];

    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "resetPassword", new
            {
                email,
                resetCode,
                newPassword
            });

        if (result.IsSuccessStatusCode)
        {
            return new FormResult { Succeeded = true };
        }

        var details = await result.Content.ReadAsStringAsync();
        var problemDetails = JsonDocument.Parse(details);
        var errors = new List<string>();
        var errorList = problemDetails.RootElement.GetProperty("errors");
```

```
        foreach (var errorEntry in errorList.EnumerateObject())
        {
            if (errorEntry.Value.ValueKind == JsonValueKind.String)
            {
                errors.Add(errorEntry.Value.GetString()!);
            }
            else if (errorEntry.Value.ValueKind == JsonValueKind.Array)
            {
                errors.AddRange(
                    errorEntry.Value.EnumerateArray().Select(
                        e => e.GetString() ?? string.Empty)
                    .Where(e => !string.IsNullOrEmpty(e)));
            }
        }

        return new FormResult
        {
            Succeeded = false,
            ErrorList = problemDetails == null ? defaultDetail : [.. errors]
        };
    }
    catch { }

    return new FormResult
    {
        Succeeded = false,
        ErrorList = defaultDetail
    };
}
```

In the client project, add the following `ForgotPassword` component.

`Components/Identity/ForgotPassword.razor`:

razor

```
@page "/forgot-password"
@using System.ComponentModel.DataAnnotations
@using BlazorWasmAuth.Identity
@inject IAccountManagement Acct

<PageTitle>Forgot your password?</PageTitle>

<h1>Forgot your password?</h1>
<p>Provide your email address and select the <b>Reset password</b> button.
</p>
<hr />
<div class="row">
    <div class="col-md-4">
        @if (!passwordResetCodeSent)
        {
            <EditForm Model="Input" FormName="forgot-password"
                      OnValidSubmit="OnValidSubmitStep1Async" method="post">
```

```
                <DataAnnotationsValidator />
                <ValidationSummary class="text-danger" role="alert" />

                <div class="form-floating mb-3">
                    <InputText @bind-Value="Input.Email"
                        id="Input.Email" class="form-control"
                        autocomplete="username" aria-required="true"
                        placeholder="name@example.com" />
                    <label for="Input.Email" class="form-label">
                        Email
                    </label>
                    <ValidationMessage For="() => Input.Email"
                        class="text-danger" />
                </div>
                <button type="submit" class="w-100 btn btn-lg btn-primary">
                    Request reset code
                </button>
            </EditForm>
        }
        else
        {
            if (passwordResetSuccess)
            {
                if (errors)
                {
                    foreach (var error in errorList)
                    {
                        <div class="alert alert-danger">@error</div>
                    }
                }
                else
                {
                    <div>
                        Your password was reset. You may <a
href="login">login</a>
                        to the app with your new password.
                    </div>
                }
            }
            else
            {
                <div>
                    A password reset code has been sent to your email
address.
                    Obtain the code from the email for this form.
                </div>
                <EditForm Model="Reset" FormName="reset-password"
                    OnValidSubmit="OnValidSubmitStep2Async" method="post">
                    <DataAnnotationsValidator />
                    <ValidationSummary class="text-danger" role="alert" />

                    <div class="form-floating mb-3">
                        <InputText @bind-Value="Reset.ResetCode"
                            id="Reset.ResetCode" class="form-control"
                            autocomplete="username" aria-required="true" />
```

```
                        <label for="Reset.ResetCode" class="form-label">
                            Reset code
                        </label>
                        <ValidationMessage For="() => Reset.ResetCode"
                            class="text-danger" />
                    </div>
                    <div class="form-floating mb-3">
                        <InputText type="password" @bind-
Value="Reset.NewPassword"
                            id="Reset.NewPassword" class="form-control"
                            autocomplete="new-password" aria-required="true"
                            placeholder="password" />
                        <label for="Reset.NewPassword" class="form-label">
                            New Password
                        </label>
                        <ValidationMessage For="() => Reset.NewPassword"
                            class="text-danger" />
                    </div>
                    <div class="form-floating mb-3">
                        <InputText type="password"
                            @bind-Value="Reset.ConfirmPassword"
                            id="Reset.ConfirmPassword" class="form-control"
                            autocomplete="new-password" aria-required="true"
                            placeholder="password" />
                        <label for="Reset.ConfirmPassword" class="form-
label">
                            Confirm Password
                        </label>
                        <ValidationMessage For="() => Reset.ConfirmPassword"
                            class="text-danger" />
                    </div>
                    <button type="submit" class="w-100 btn btn-lg btn-
primary">
                        Reset password
                    </button>
                </EditForm>
            }
        }
    </div>
</div>

@code {
    private bool passwordResetCodeSent, passwordResetSuccess, errors;
    private string[] errorList = [];

    [SupplyParameterFromForm(FormName = "forgot-password")]
    private InputModel Input { get; set; } = new();

    [SupplyParameterFromForm(FormName = "reset-password")]
    private ResetModel Reset { get; set; } = new();

    private async Task OnValidSubmitStep1Async()
    {
        passwordResetCodeSent = await Acct.ForgotPasswordAsync(Input.Email);
    }
```

```
    private async Task OnValidSubmitStep2Async()
    {
        var result = await Acct.ResetPasswordAsync(Input.Email,
Reset.ResetCode,
            Reset.NewPassword);

        if (result.Succeeded)
        {
            passwordResetSuccess = true;

        }
        else
        {
            errors = true;
            errorList = result.ErrorList;
        }
    }

    private sealed class InputModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; } = string.Empty;
    }

    private sealed class ResetModel
    {
        [Required]
        [Base64String]
        public string ResetCode { get; set; } = string.Empty;

        [StringLength(100, ErrorMessage = "The {0} must be at least {2} and
at " +
            "max {1} characters long.", MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string NewPassword { get; set; } = string.Empty;

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("NewPassword", ErrorMessage = "The new password and
confirmation " +
            "password don't match.")]
        public string ConfirmPassword { get; set; } = string.Empty;
    }
}
```

In the `Login` component (`Components/Identity/Login.razor`) of the client project immediately before the closing `</NotAuthorized>` tag, add a forgot password link to reach the `ForgotPassword` component:

HTML

```
<div>
    <a href="forgot-password">Forgot password</a>
</div>
```

Email and activity timeout

The default inactivity timeout is 14 days. In the server project, the following code sets the inactivity timeout to five days with sliding expiration:

C#

```
builder.Services.ConfigureApplicationCookie(options => {
    options.ExpireTimeSpan = TimeSpan.FromDays(5);
    options.SlidingExpiration = true;
});
```

Change all ASP.NET Core Data Protection token lifespans

In the server project, the following code changes Data Protection tokens' timeout period to three hours:

C#

```
builder.Services.Configure<DataProtectionTokenProviderOptions>(options =>
    options.TokenLifespan = TimeSpan.FromHours(3));
```

The built-in Identity user tokens (AspNetCore/src/Identity/Extensions.Core/src/TokenOptions.cs) have a one day timeout.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Change the email token lifespan

The default token lifespan of the Identity user tokens is one day.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

To change the email token lifespan, add a custom DataProtectorTokenProvider<TUser> and DataProtectionTokenProviderOptions to the server project.

`CustomTokenProvider.cs`:

C#

```
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;

namespace BlazorSample;

public class CustomEmailConfirmationTokenProvider<TUser>
    : DataProtectorTokenProvider<TUser> where TUser : class
{
    public CustomEmailConfirmationTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<EmailConfirmationTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
        : base(dataProtectionProvider, options, logger)
    {
    }
}

public class EmailConfirmationTokenProviderOptions
    : DataProtectionTokenProviderOptions
{
    public EmailConfirmationTokenProviderOptions()
    {
        Name = "EmailDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(4);
    }
}

public class CustomPasswordResetTokenProvider<TUser>
    : DataProtectorTokenProvider<TUser> where TUser : class
```

```
{
    public CustomPasswordResetTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<PasswordResetTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
        : base(dataProtectionProvider, options, logger)
    {
    }
}

public class PasswordResetTokenProviderOptions :
    DataProtectionTokenProviderOptions
{
    public PasswordResetTokenProviderOptions()
    {
        Name = "PasswordResetDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(3);
    }
}
```

Configure the services to use the custom token provider in the server project's `Program` file:

C#

```
builder.Services.AddIdentityCore<AppUser>(options =>
    {
        options.SignIn.RequireConfirmedAccount = true;
        options.Tokens.ProviderMap.Add("CustomEmailConfirmation",
            new TokenProviderDescriptor(
                typeof(CustomEmailConfirmationTokenProvider<AppUser>)));
        options.Tokens.EmailConfirmationTokenProvider =
            "CustomEmailConfirmation";
    })
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddSignInManager()
    .AddDefaultTokenProviders();

builder.Services
    .AddTransient<CustomEmailConfirmationTokenProvider<AppUser>>();
```

Troubleshoot

If you can't get email working:

- Set a breakpoint in `EmailSender.Execute` to verify `SendEmailAsync` is called.
- Create a console app to send email using code similar to `EmailSender.Execute` to debug the problem.
- Review the account email history pages at the email provider's website.

- Check your spam folder for messages.
- Try another email alias on a different email provider, such as Microsoft, Yahoo, or Gmail.
- Try sending to different email accounts.

Additional resources

- Mandrill.net (GitHub repository)
- Mailchimp developer: Transactional API

Enable QR code generation for TOTP authenticator apps in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

Article • 12/11/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure an ASP.NET Core Blazor WebAssembly app with Identity for two-factor authentication (2FA) with QR codes generated by Time-based One-time Password Algorithm (TOTP) authenticator apps.

For an introduction to 2FA with TOTP authenticator apps, see Enable QR code generation for TOTP authenticator apps in ASP.NET Core.

> **Warning**
>
> TOTP codes should be kept secret because they can be used to authenticate multiple times before they expire.

Namespaces and article code examples

The namespaces used by the examples in this article are:

- `Backend` for the backend server web API project, described as the "server project" in this article.
- `BlazorWasmAuth` for the front-end client standalone Blazor WebAssembly app, described as the "client project" in this article.

These namespaces correspond to the projects in the `BlazorWebAssemblyStandaloneWithIdentity` sample solution in the dotnet/blazor-samples GitHub repository. For more information, see Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

If you aren't using the `BlazorWebAssemblyStandaloneWithIdentity` sample, change the namespaces in the code examples to use the namespaces of your projects.

All of the changes to the solution covered by this article take place in the `BlazorWasmAuth` project of the `BlazorWebAssemblyStandaloneWithIdentity` solution.

In article examples, code lines are split to reduce horizontal scrolling. These breaks don't affect execution but can be removed when pasting into your project.

Optional account confirmation and password recovery

Although apps that implement 2FA usually adopt account confirmation and password recovery features, 2FA doesn't require it. The guidance in this article can be followed to implement 2FA without following the guidance in Account confirmation and password recovery in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

Add a QR code library to the app

A QR code generated by the app to set up 2FA with an TOTP authenticator app must be generated by a QR code library.

The guidance in this article uses manuelbl/QrCodeGenerator, but you can use any QR code generation library.

Add a package reference to the client project for the Net.Codecrete.QrCodeGenerator NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Set the TOTP organization name

Set the site name in the app settings file of the client project. Use a meaningful site name that users can identify easily in their authenticator app. Developers usually set a site name that matches the company's name. We recommend limiting the site name

length to 30 characters or less to allow the site name to display on narrow mobile device screens.

In the following example, the the company name is `Weyland-Yutani Corporation` (©1986 20th Century Studios *Aliens*).

Added to `wwwroot/appsettings.json`:

JSON

```
"TotpOrganizationName": "Weyland-Yutani Corporation"
```

The app settings file after the TOTP organization name configuration is added:

JSON

```
{
  "BackendUrl": "https://localhost:7211",
  "FrontendUrl": "https://localhost:7171",
  "TotpOrganizationName": "Weyland-Yutani Corporation"
}
```

Add model classes

Add the following `LoginResponse` class to the `Models` folder. This class is populated for requests to the `/login` endpoint of MapIdentityApi in the server app.

`Identity/Models/LoginResponse.cs`:

C#

```
namespace BlazorWasmAuth.Identity.Models;

public class LoginResponse
{
    public string? Type { get; set; }
    public string? Title { get; set; }
    public int Status { get; set; }
    public string? Detail { get; set; }
}
```

Add the following `TwoFactorRequest` class to the `Models` folder. This class is populated for requests to the `/manage/2fa` endpoint of MapIdentityApi in the server app.

`Identity/Models/TwoFactorRequest.cs`:

C#

```
namespace BlazorWasmAuth.Identity.Models;

public class TwoFactorRequest
{
    public bool? Enable { get; set; }
    public string? TwoFactorCode { get; set; }
    public bool? ResetSharedKey { get; set; }
    public bool? ResetRecoveryCodes { get; set; }
    public bool? ForgetMachine {  get; set; }
}
```

Add the following `TwoFactorResponse` class to the `Models` folder. This class is populated by the response to a 2FA request made to the `/manage/2fa` endpoint of MapIdentityApi in the server app.

`Identity/Models/TwoFactorResponse.cs`:

C#

```
namespace BlazorWasmAuth.Identity.Models;

public class TwoFactorResponse
{
    public string SharedKey { get; set; } = string.Empty;
    public int RecoveryCodesLeft { get; set; } = 0;
    public string[] RecoveryCodes { get; set; } = [];
    public bool IsTwoFactorEnabled { get; set; }
    public bool IsMachineRemembered { get; set; }
    public string[] ErrorList { get; set; } = [];
}
```

`IAccountManagement` interface

Add the following class signatures to the `IAccountManagement` interface. The class signatures represent methods added to the cookie authentication state provider for the following client requests:

- Log in with a 2FA TOTP code (`/login` endpoint): `LoginTwoFactorCodeAsync`
- Log in with a 2FA recovery code (`/login` endpoint): `LoginTwoFactorRecoveryCodeAsync`
- Make a 2FA management request (`/manage/2fa` endpoint): `TwoFactorRequestAsync`

`Identity/IAccountManagement.cs` (paste the following code at the bottom of the file):

C#

```
public Task<FormResult> LoginTwoFactorCodeAsync(
    string email,
    string password,
    string twoFactorCode);

public Task<FormResult> LoginTwoFactorRecoveryCodeAsync(
    string email,
    string password,
    string twoFactorRecoveryCode);

public Task<TwoFactorResponse> TwoFactorRequestAsync(
    TwoFactorRequest twoFactorRequest);
```

Update the cookie authentication state provider

Update the `CookieAuthenticationStateProvider` with features to add the following features:

- Authenticate users with either a TOTP authenticator app code or a recovery code.
- Manage 2FA in the app.

At the top of the `CookieAuthenticationStateProvider.cs` file, add a `using` statement for System.Text.Json.Serialization:

C#

```
using System.Text.Json.Serialization;
```

In the JsonSerializerOptions, add the DefaultIgnoreCondition option set to JsonIgnoreCondition.WhenWritingNull, which avoids serializing null properties:

diff

```
private readonly JsonSerializerOptions jsonSerializerOptions =
    new()
    {
        PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
+       DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull,
    };
```

The `LoginAsync` method is updated with the following logic:

- Attempt a normal login at the `/login` endpoint with an email address and password.
- If the server responds with a success status code, the method returns a `FormResult` with the `Succeeded` property set to `true`.
- If the server responds with the *401 - Unauthorized* status code and a detail code of "`RequiresTwoFactor`," a `FormResult` is returned with `Succeeded` set to `false` and the `RequiresTwoFactor` detail in the error list.

In `Identity/CookieAuthenticationStateProvider.cs`, replace the `LoginAsync` method with the following code:

C#

```
public async Task<FormResult> LoginAsync(string email, string password)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "login?useCookies=true", new
            {
                email,
                password
            });

        if (result.IsSuccessStatusCode)
        {
            NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

            return new FormResult { Succeeded = true };
        }
        else if (result.StatusCode == HttpStatusCode.Unauthorized)
        {
            var responseJson = await result.Content.ReadAsStringAsync();
            var response = JsonSerializer.Deserialize<LoginResponse>(
                responseJson, jsonSerializerOptions);

            if (response?.Detail == "RequiresTwoFactor")
            {
                return new FormResult
                {
                    Succeeded = false,
                    ErrorList = [ "RequiresTwoFactor" ]
                };
            }
        }
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "App error");
    }
```

```
    return new FormResult
    {
        Succeeded = false,
        ErrorList = [ "Invalid email and/or password." ]
    };
}
```

A `LoginTwoFactorCodeAsync` method is added, which sends a request to the `/login` endpoint with a 2FA TOTP code (`twoFactorCode`). The method processes the response in a similar fashion to a normal, non-2FA login request.

Add the following method and class to `Identity/CookieAuthenticationStateProvider.cs` (paste the following code at the bottom of the class file):

C#

```
public async Task<FormResult> LoginTwoFactorCodeAsync(
    string email, string password, string twoFactorCode)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "login?useCookies=true", new
            {
                email,
                password,
                twoFactorCode
            });

        if (result.IsSuccessStatusCode)
        {
            NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

            return new FormResult { Succeeded = true };
        }
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "App error");
    }

    return new FormResult
    {
        Succeeded = false,
        ErrorList = [ "Invalid two-factor code." ]
    };
}
```

A `LoginTwoFactorRecoveryCodeAsync` method is added, which sends a request to the `/login` endpoint with a 2FA recovery code (`twoFactorRecoveryCode`). The method

processes the response in a similar fashion to a normal, non-2FA login request.

Add the following method and class to `Identity/CookieAuthenticationStateProvider.cs` (paste the following code at the bottom of the class file):

C#

```
public async Task<FormResult> LoginTwoFactorRecoveryCodeAsync(string email,
    string password, string twoFactorRecoveryCode)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "login?useCookies=true", new
            {
                email,
                password,
                twoFactorRecoveryCode
            });

        if (result.IsSuccessStatusCode)
        {
            NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

            return new FormResult { Succeeded = true };
        }
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "App error");
    }

    return new FormResult
    {
        Succeeded = false,
        ErrorList = [ "Invalid recovery code." ]
    };
}
```

A `TwoFactorRequestAsync` method is added, which is used to manage 2FA for the user:

- Reset the shared 2FA key when `TwoFactorRequest.ResetSharedKey` is `true`. Resetting the shared key implicitly disables 2FA. This forces the user to prove that they can provide a valid TOTP code from their authenticator app to enable 2FA after receiving a new shared key.
- Reset the user's recovery codes when `TwoFactorRequest.ResetRecoveryCodes` is `true`.
- Forget the machine when `TwoFactorRequest.ForgetMachine` is `true`, which means that a new 2FA TOTP code is required on the next login attempt.

- Enable 2FA using a TOTP code from a TOTP authenticator app when `TwoFactorRequest.Enable` is `true` and `TwoFactorRequest.TwoFactorCode` has a valid TOTP value.
- Obtain 2FA status with an empty request when all of `TwoFactorRequest`'s properties are `null`.

Add the following `TwoFactorRequestAsync` method to `Identity/CookieAuthenticationStateProvider.cs` (paste the following code at the bottom of the class file):

C#

```
public async Task<TwoFactorResponse> TwoFactorRequestAsync(TwoFactorRequest
twoFactorRequest)
{
    string[] defaultDetail =
        [ "An unknown error prevented two-factor authentication." ];

    var response = await httpClient.PostAsJsonAsync("manage/2fa",
twoFactorRequest,
        jsonSerializerOptions);

    // successful?
    if (response.IsSuccessStatusCode)
    {
        return await response.Content
            .ReadFromJsonAsync<TwoFactorResponse>() ??
            new()
            {
                ErrorList = [ "There was an error processing the request." ]
            };
    }

    // body should contain details about why it failed
    var details = await response.Content.ReadAsStringAsync();
    var problemDetails = JsonDocument.Parse(details);
    var errors = new List<string>();
    var errorList = problemDetails.RootElement.GetProperty("errors");

    foreach (var errorEntry in errorList.EnumerateObject())
    {
        if (errorEntry.Value.ValueKind == JsonValueKind.String)
        {
            errors.Add(errorEntry.Value.GetString()!);
        }
        else if (errorEntry.Value.ValueKind == JsonValueKind.Array)
        {
            errors.AddRange(
                errorEntry.Value.EnumerateArray().Select(
                    e => e.GetString() ?? string.Empty)
                .Where(e => !string.IsNullOrEmpty(e)));
```

```
        }
    }

    // return the error list
    return new TwoFactorResponse
    {
        ErrorList = problemDetails == null ? defaultDetail : [.. errors]
    };
}
```

Replace `Login` component

Replace the `Login` component. The following version of the `Login` component:

- Accepts a user's email address and password for an initial login attempt.
- If login is successful (2FA is disabled), the component notifies the user that they're authenticated.
- If the login attempt results in a response indicating that 2FA is required, a 2FA input element appears to receive either a 2FA TOTP code from an authenticator app or a recovery code. Depending on which code the user enters, login is attempted again by calling either `LoginTwoFactorCodeAsync` for a TOTP code or `LoginTwoFactorRecoveryCodeAsync` for a recovery code.

`Components/Identity/Login.razor`:

razor

```
@page "/login"
@using System.ComponentModel.DataAnnotations
@using BlazorWasmAuth.Identity
@using BlazorWasmAuth.Identity.Models
@inject IAccountManagement Acct
@inject ILogger<Login> Logger
@inject NavigationManager Navigation

<PageTitle>Login</PageTitle>

<h1>Login</h1>

<AuthorizeView>
    <Authorized>
        <div class="alert alert-success">
            You're logged in as @context.User.Identity?.Name.
        </div>
    </Authorized>
    <NotAuthorized>
        @foreach (var error in formResult.ErrorList)
        {
```

```
            <div class="alert alert-danger">@error</div>
        }
        <div class="row">
            <div class="col">
                <section>
                    <EditForm Model="Input" method="post"
OnValidSubmit="LoginUser"
                              FormName="login" Context="editform_context">
                        <DataAnnotationsValidator />
                        <h2>Use a local account to log in.</h2>
                        <hr />
                        <div style="display:@(requiresTwoFactor ? "none" :
"block")">
                            <div class="form-floating mb-3">
                                <InputText @bind-Value="Input.Email"
                                    id="Input.Email"
                                    class="form-control"
                                    autocomplete="username"
                                    aria-required="true"
                                    placeholder="name@example.com" />
                                <label for="Input.Email" class="form-label">
                                    Email
                                </label>
                                <ValidationMessage For="() => Input.Email"
                                    class="text-danger" />
                            </div>
                            <div class="form-floating mb-3">
                                <InputText type="password"
                                    @bind-Value="Input.Password"
                                    id="Input.Password"
                                    class="form-control"
                                    autocomplete="current-password"
                                    aria-required="true"
                                    placeholder="password" />
                                <label for="Input.Password" class="form-
label">
                                    Password
                                </label>
                                <ValidationMessage For="() =>
Input.Password"
                                    class="text-danger" />
                            </div>
                        </div>
                        <div style="display:@(requiresTwoFactor ? "block" :
"none")">
                            <div class="form-floating mb-3">
                                <InputText @bind-
Value="Input.TwoFactorCodeOrRecoveryCode"
                                    id="Input.TwoFactorCodeOrRecoveryCode"
                                    class="form-control"
                                    autocomplete="off"
                                    placeholder="###### or #####-#####" />
                                <label
for="Input.TwoFactorCodeOrRecoveryCode" class="form-label">
                                    Two-factor Code or Recovery Code
```

```
                                </label>
                                <ValidationMessage For="() =>
Input.TwoFactorCodeOrRecoveryCode"
                                    class="text-danger" />
                            </div>
                        </div>
                        <div>
                            <button type="submit" class="w-100 btn btn-lg
btn-primary">
                                Log in
                            </button>
                        </div>
                        <div class="mt-3">
                            <p>
                                <a href="forgot-password">Forgot
password</a>
                            </p>
                            <p>
                                <a href="register">Register as a new
user</a>
                            </p>
                        </div>
                    </EditForm>
                </section>
            </div>
        </div>
    </NotAuthorized>
</AuthorizeView>

@code {
    private FormResult formResult = new();
    private bool requiresTwoFactor;

    [SupplyParameterFromForm]
    private InputModel Input { get; set; } = new();

    [SupplyParameterFromQuery]
    private string? ReturnUrl { get; set; }

    public async Task LoginUser()
    {
        if (requiresTwoFactor)
        {
            if (!string.IsNullOrEmpty(Input.TwoFactorCodeOrRecoveryCode))
            {
                // The [RegularExpression] data annotation ensures that the
input
                // is either a six-digit authenticator code (######) or an
                // eleven-character alphanumeric recovery code (#####-#####)
                if (Input.TwoFactorCodeOrRecoveryCode.Length == 6)
                {
                    formResult = await Acct.LoginTwoFactorCodeAsync(
                        Input.Email, Input.Password,
                        Input.TwoFactorCodeOrRecoveryCode);
                }
```

```
                else
                {
                    formResult = await Acct.LoginTwoFactorRecoveryCodeAsync(
                        Input.Email, Input.Password,
                        Input.TwoFactorCodeOrRecoveryCode);

                    if (formResult.Succeeded)
                    {
                        var twoFactorResponse = await
Acct.TwoFactorRequestAsync(new());
                    }
                }
            }
            else
            {
                formResult =
                    new FormResult
                    {
                        Succeeded = false,
                        ErrorList = [ "Invalid two-factor code." ]
                    };
            }
        }
        else
        {
            formResult = await Acct.LoginAsync(Input.Email, Input.Password);
            requiresTwoFactor =
formResult.ErrorList.Contains("RequiresTwoFactor");
            Input.TwoFactorCodeOrRecoveryCode = string.Empty;

            if (requiresTwoFactor)
            {
                formResult.ErrorList = [];
            }
        }

        if (formResult.Succeeded && !string.IsNullOrEmpty(ReturnUrl))
        {
            Navigation.NavigateTo(ReturnUrl);
        }
    }

    private sealed class InputModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; } = string.Empty;

        [Required]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; } = string.Empty;

        [RegularExpression(@"^([0-9]{6})|([A-Z0-9]{5}[-]{1}[A-Z0-9]{5})$",
```

```
            ErrorMessage = "Must be a six-digit authenticator code (######)
or " +
            "eleven-character alphanumeric recovery code (#####-#####, dash
" +
            "required)")]
        [Display(Name = "Two-factor Code or Recovery Code")]
        public string TwoFactorCodeOrRecoveryCode { get; set; } =
string.Empty;
    }
}
```

Using the preceding component, the user is remembered after a successful login with a valid TOTP code from an authenticator app. If you want to always require a TOTP code for login and not remember the machine, call the `TwoFactorRequestAsync` method with `TwoFactorRequest.ForgetMachine` set to `true` immediately after a successful two-factor login:

diff

```
if (Input.TwoFactorCodeOrRecoveryCode.Length == 6)
{
    formResult = await Acct.LoginTwoFactorCodeAsync(Input.Email,
Input.Password,
        Input.TwoFactorCodeOrRecoveryCode);

+    if (formResult.Succeeded)
+    {
+        var forgetMachine =
+            await Acct.TwoFactorRequestAsync(new() { ForgetMachine = true
});
+    }
}
```

Add a component to display recovery codes

Add the following `ShowRecoveryCodes` component to the app to display recovery codes to the user.

`Components/Identity/ShowRecoveryCodes.razor`:

razor

```
<h3>Recovery codes</h3>

<div class="alert alert-warning" role="alert">
    <p>
        <strong>Put these codes in a safe place.</strong>
    </p>
```

```
    <p>
        If you lose your device and don't have an unused
        recovery code, you can't access your account.
    </p>
</div>
<div class="row">
    <div class="col-md-12">
        @foreach (var recoveryCode in RecoveryCodes)
        {
            <div>
                <code class="recovery-code">@recoveryCode</code>
            </div>
        }
    </div>
</div>

@code {
    [Parameter]
    public string[] RecoveryCodes { get; set; } = [];
}
```

Manage 2FA page

Add the following `Manage2fa` component to the app to manage 2FA for users.

If 2FA isn't enabled, the component loads a form with a QR code to enable 2FA with a TOTP authenticator app. The user adds the app to their authenticator app and then verifies the authenticator app and enables 2FA by providing a TOTP code from the authenticator app.

If 2FA is enabled, buttons appear to disable 2FA and regenerate recovery codes.

`Components/Identity/Manage2fa.razor`:

razor

```
@page "/manage-2fa"
@using System.ComponentModel.DataAnnotations
@using System.Globalization
@using System.Text
@using System.Text.Encodings.Web
@using Net.Codecrete.QrCodeGenerator
@using BlazorWasmAuth.Identity
@using BlazorWasmAuth.Identity.Models
@attribute [Authorize]
@inject IAccountManagement Acct
@inject IAuthorizationService AuthorizationService
@inject IConfiguration Config
@inject ILogger<Manage2fa> Logger
```

```
<PageTitle>Manage 2FA</PageTitle>

<h1>Manage Two-factor Authentication</h1>
<hr />
<div class="row">
    <div class="col">
        @if (loading)
        {
            <p>Loading ...</p>
        }
        else
        {
            @if (twoFactorResponse is not null)
            {
                @foreach (var error in twoFactorResponse.ErrorList)
                {
                    <div class="alert alert-danger">@error</div>
                }
                @if (twoFactorResponse.IsTwoFactorEnabled)
                {
                    <div class="alert alert-success" role="alert">
                        Two-factor authentication is enabled for your
account.
                    </div>

                    <div class="m-1">
                        <button @onclick="Disable2FA" class="btn btn-lg btn-
primary">
                            Disable 2FA
                        </button>
                    </div>

                    @if (twoFactorResponse.RecoveryCodes is null)
                    {
                        <div class="m-1">
                            Recovery Codes Remaining:
                            @twoFactorResponse.RecoveryCodesLeft
                        </div>
                        <div class="m-1">
                            <button @onclick="GenerateNewCodes"
                                    class="btn btn-lg btn-primary">
                                Generate New Recovery Codes
                            </button>
                        </div>
                    }
                    else
                    {
                        <ShowRecoveryCodes
                            RecoveryCodes="twoFactorResponse.RecoveryCodes"
/>
                    }
                }
                else
                {
                    <h3>Configure authenticator app</h3>
```

```
                    <div>
                        <p>To use an authenticator app:</p>
                        <ol class="list">
                            <li>
                                <p>
                                    Download a two-factor authenticator app,
such
                                    as either of the following:
                                    <ul>
                                        <li>
                                            Microsoft Authenticator for
                                            <a
href="https://go.microsoft.com/fwlink/?Linkid=825072">
                                                Android
                                            </a> and
                                            <a
href="https://go.microsoft.com/fwlink/?Linkid=825073">
                                                iOS
                                            </a>
                                        </li>
                                        <li>
                                            Google Authenticator for
                                            <a
href="https://play.google.com/store/apps/details?
id=com.google.android.apps.authenticator2">
                                                Android
                                            </a> and
                                            <a
href="https://itunes.apple.com/us/app/google-authenticator/id388497605?
mt=8">
                                                iOS
                                            </a>
                                        </li>
                                    </ul>
                                </p>
                            </li>
                            <li>
                                <p>
                                    Scan the QR Code or enter this key
                                    <kbd>@twoFactorResponse.SharedKey</kbd>
into your
                                    two-factor authenticator app. Spaces and
casing
                                    don't matter.
                                </p>
                                <div>
                                    <svg xmlns="http://www.w3.org/2000/svg"
height="300"
                                        width="300" stroke="none"
version="1.1"
                                        viewBox="0 0 50 50">
                                        <rect width="300" height="300"
fill="#ffffff" />
                                        <path d="@svgGraphicsPath"
fill="#000000" />
```

```
                                    </svg>
                                </div>
                            </li>
                            <li>
                                <p>
                                    After you have scanned the QR code or
input the
                                    key above, your two-factor authenticator
app
                                    will provide you with a unique two-
factor code.
                                    Enter the code in the confirmation box
below.
                                </p>
                                <div class="row">
                                    <div class="col-xl-6">
                                        <EditForm Model="Input"
                                                  FormName="send-code"
OnValidSubmit="OnValidSubmitAsync"
                                                  method="post">
                                            <DataAnnotationsValidator />
                                            <div class="form-floating mb-3">
                                                <InputText
                                                    @bind-Value="Input.Code"
                                                    id="Input.Code"
                                                    class="form-control"
                                                    autocomplete="off"
                                                    placeholder="Enter the
code" />
                                                <label for="Input.Code"
                                                       class="control-label
form-label">
                                                    Verification Code
                                                </label>
                                                <ValidationMessage
                                                    For="() => Input.Code"
                                                    class="text-danger" />
                                            </div>
                                            <button type="submit"
                                                    class="w-100 btn btn-lg
btn-primary">
                                                Verify
                                            </button>
                                        </EditForm>
                                    </div>
                                </div>
                            </li>
                        </ol>
                    </div>
                }
            }
        }
    </div>
</div>
```

```
@code {
    private TwoFactorResponse twoFactorResponse = new();
    private bool loading = true;
    private string? svgGraphicsPath;

    [SupplyParameterFromForm]
    private InputModel Input { get; set; } = new();

    [CascadingParameter]
    private Task<AuthenticationState>? authenticationState { get; set; }

    protected override async Task OnInitializedAsync()
    {
        twoFactorResponse = await Acct.TwoFactorRequestAsync(new());
        svgGraphicsPath = await GetQrCode(twoFactorResponse.SharedKey);
        loading = false;
    }

    private async Task<string> GetQrCode(string sharedKey)
    {
        if (authenticationState is not null &&
!string.IsNullOrEmpty(sharedKey))
        {
            var authState = await authenticationState;
            var email = authState?.User?.Identity?.Name!;
            var uri = string.Format(
                CultureInfo.InvariantCulture,
                "otpauth://totp/{0}:{1}?secret={2}&issuer={0}&digits=6",
                UrlEncoder.Default.Encode(Config["TotpOrganizationName"]!),
                email,
                twoFactorResponse.SharedKey);
            var qr = QrCode.EncodeText(uri, QrCode.Ecc.Medium);

            return qr.ToGraphicsPath();
        }

        return string.Empty;
    }

    private async Task Disable2FA()
    {
        await Acct.TwoFactorRequestAsync(new() { ForgetMachine = true });
        twoFactorResponse =
            await Acct.TwoFactorRequestAsync(new() { ResetSharedKey = true
});
        svgGraphicsPath = await GetQrCode(twoFactorResponse.SharedKey);
    }

    private async Task GenerateNewCodes()
    {
        twoFactorResponse =
            await Acct.TwoFactorRequestAsync(new() { ResetRecoveryCodes =
true });
    }
```

```
    private async Task OnValidSubmitAsync()
    {
        twoFactorResponse = await Acct.TwoFactorRequestAsync(
            new()
            {
                Enable = true,
                TwoFactorCode = Input.Code
            });
        Input.Code = string.Empty;

        // When 2FA is first enabled, recovery codes are returned.
        // However, subsequently disabling and re-enabling 2FA
        // leaves the existing codes in place and doesn't generate
        // a new set of recovery codes. The following code ensures
        // that a new set of recovery codes is generated each
        // time 2FA is enabled.
        if (twoFactorResponse.RecoveryCodes is null ||
            twoFactorResponse.RecoveryCodes.Length == 0)
        {
            await GenerateNewCodes();
        }
    }

    private sealed class InputModel
    {
        [Required]
        [RegularExpression(@"^([0-9]{6})$",
            ErrorMessage = "Must be a six-digit authenticator code
(######)")]
        [DataType(DataType.Text)]
        [Display(Name = "Verification Code")]
        public string Code { get; set; } = string.Empty;
    }
}
```

Link to the the Manage 2FA page

Add a link to the navigation menu for users to reach the `Manage2fa` component page.

In the `<Authorized>` content of the `<AuthorizeView>` in `Components/Layout/NavMenu.razor`, add the following markup:

diff

```
<AuthorizeView>
    <Authorized>

        ...

+       <div class="nav-item px-3">
```

```
+           <NavLink class="nav-link" href="manage-2fa">
+               <span class="bi bi-key" aria-hidden="true"></span> Manage
2FA
+           </NavLink>
+       </div>

        ...

    </Authorized>
</AuthorizeView>
```

Additional resources

- nimiq/qr-creator
- MapIdentityApi

Secure an ASP.NET Core Blazor WebAssembly standalone app with the Authentication library

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to secure an ASP.NET Core Blazor WebAssembly standalone app with the Blazor WebAssembly Authentication library.

The Blazor WebAssembly Authentication library (`Authentication.js`) only supports the Proof Key for Code Exchange (PKCE) authorization code flow via the Microsoft Authentication Library (MSAL, msal.js). To implement other grant flows, access the MSAL guidance to implement MSAL directly, but we don't support or recommend the use of grant flows other than PKCE for Blazor apps.

For Microsoft Entra (ME-ID) and Azure Active Directory B2C (AAD B2C) guidance, don't follow the guidance in this topic. See Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID or Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory B2C.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Register an app
- Create the Blazor app
- Run the app

Register an app

Register an app with an OpenID Connect (OIDC) Identity Provider (IP) following the guidance provided by the maintainer of the IP.

Record the following information:

- Authority (for example, `https://accounts.google.com/`).
- Application (client) ID (for example, `2...7-e...q.apps.googleusercontent.com`).
- Additional IP configuration (see the IP's documentation).

> **Note**
>
> The IP must use OIDC. For example, Facebook's IP isn't an OIDC-compliant provider, so the guidance in this topic doesn't work with the Facebook IP. For more information, see Secure ASP.NET Core Blazor WebAssembly.

Create the Blazor app

To create a standalone Blazor WebAssembly app that uses the Microsoft.AspNetCore.Components.WebAssembly.Authentication library, follow the guidance for your choice of tooling. If adding support for authentication, see the Parts of the app section of this article for guidance on setting up and configuring the app.

Visual Studio

To create a new Blazor WebAssembly project with an authentication mechanism:

After choosing the **Blazor WebAssembly App** template, set the **Authentication type** to **Individual Accounts**.

The **Individual Accounts** selection uses ASP.NET Core's Identity system. This selection adds authentication support and doesn't result in storing users in a database. The following sections of this article provide further details.

Configure the app

Configure the app following the IP's guidance. At a minimum, the app requires the `Local:Authority` and `Local:ClientId` configuration settings in the app's `wwwroot/appsettings.json` file:

JSON

```
{
  "Local": {
    "Authority": "{AUTHORITY}",
    "ClientId": "{CLIENT ID}"
```

```
  }
}
```

Google OAuth 2.0 OIDC example for an app that runs on the `localhost` address at port 5001:

JSON

```
{
  "Local": {
    "Authority": "https://accounts.google.com/",
    "ClientId": "2...7-e...q.apps.googleusercontent.com",
    "PostLogoutRedirectUri": "https://localhost:5001/authentication/logout-
callback",
    "RedirectUri": "https://localhost:5001/authentication/login-callback",
    "ResponseType": "code"
  }
}
```

The redirect URI (`https://localhost:5001/authentication/login-callback`) is registered in the Google APIs console in **Credentials** > `{NAME}` > **Authorized redirect URIs**, where `{NAME}` is the app's client name in the **OAuth 2.0 Client IDs** app list of the Google APIs console.

> **Note**
>
> Supplying the port number for a `localhost` redirect URI isn't required for some OIDC IPs per the OAuth 2.0 specification. Some IPs permit the redirect URI for loopback addresses to omit the port. Others allow the use of a wildcard for the port number (for example, `*`). For additional information, see the IP's documentation.

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press `F5`.
- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use Individual User Accounts, the app automatically receives a package reference for the Microsoft.AspNetCore.Components.WebAssembly.Authentication package. The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Authentication service support

Support for authenticating users using OpenID Connect (OIDC) is registered in the service container with the AddOidcAuthentication extension method provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.

The AddOidcAuthentication method accepts a callback to configure the parameters required to authenticate an app using OIDC. The values required for configuring the app can be obtained from the OIDC-compliant IP. Obtain the values when you register the app, which typically occurs in their online portal.

For a new app, provide values for the `{AUTHORITY}` and `{CLIENT ID}` placeholders in the following configuration. Provide other configuration values that are required for use with the app's IP. The example is for Google, which requires `PostLogoutRedirectUri`,

`RedirectUri`, and `ResponseType`. If adding authentication to an app, manually add the following code and configuration to the app with values for the placeholders and other configuration values.

In the `Program` file:

C#

```
builder.Services.AddOidcAuthentication(options =>
{
    builder.Configuration.Bind("Local", options.ProviderOptions);
});
```

`wwwroot/appsettings.json` configuration

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```
{
  "Local": {
    "Authority": "{AUTHORITY}",
    "ClientId": "{CLIENT ID}"
  }
}
```

Access token scopes

The Blazor WebAssembly template automatically configures default scopes for `openid` and `profile`.

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default token scopes of the OidcProviderOptions. If adding authentication to an app, manually add the following code and configure the scope URI.

In the `Program` file:

C#

```
builder.Services.AddOidcAuthentication(options =>
{
    ...
```

```
    options.ProviderOptions.DefaultScopes.Add("{SCOPE URI}");
});
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

HTML

```
<script
src="_content/Microsoft.AspNetCore.Components.WebAssembly.Authentication/Aut
henticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `App` component (`App.razor`) in the generated app.

- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

  > **Note**
  >
  > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

> **Note**

> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.

- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

  > **Note**
  >
  > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

razor

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string? Action { get; set; }
}
```

> **Note**
>
> Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core in .NET 6, the `string` type appears without the null type designation (`?`).

Troubleshoot

Logging

To enable debug or trace logging for Blazor WebAssembly authentication, see the *Client-side authentication logging* section of ASP.NET Core Blazor logging with the article version selector set to ASP.NET Core 7.0 or later.

Common errors

- Misconfiguration of the app or Identity Provider (IP)

  The most common errors are caused by incorrect configuration. The following are a few examples:

- Depending on the requirements of the scenario, a missing or incorrect Authority, Instance, Tenant ID, Tenant domain, Client ID, or Redirect URI prevents an app from authenticating clients.
- Incorrect request scopes prevent clients from accessing server web API endpoints.
- Incorrect or missing server API permissions prevent clients from accessing server web API endpoints.
- Running the app at a different port than is configured in the Redirect URI of the IP's app registration. Note that a port isn't required for Microsoft Entra ID and an app running at a `localhost` development testing address, but the app's port configuration and the port where the app is running must match for non-`localhost` addresses.

Configuration sections of this article's guidance show examples of the correct configuration. Carefully check each section of the article looking for app and IP misconfiguration.

If the configuration appears correct:

- Analyze application logs.

- Examine the network traffic between the client app and the IP or server app with the browser's developer tools. Often, an exact error message or a message with a clue to what's causing the problem is returned to the client by the IP or server app after making a request. Developer tools guidance is found in the following articles:
  - Google Chrome (Google documentation)
  - Microsoft Edge
  - Mozilla Firefox (Mozilla documentation)

- For releases of Blazor where a JSON Web Token (JWT) is used, decode the contents of the token used for authenticating a client or accessing a server web API, depending on where the problem is occurring. For more information, see Inspect the content of a JSON Web Token (JWT).

The documentation team responds to document feedback and bugs in articles (open an issue from the **This page** feedback section) but is unable to provide product support. Several public support forums are available to assist with troubleshooting an app. We recommend the following:

- Stack Overflow (tag: blazor)
- ASP.NET Core Slack Team
- Blazor Gitter

The preceding forums are not owned or controlled by Microsoft.

For non-security, non-sensitive, and non-confidential reproducible framework bug reports, open an issue with the ASP.NET Core product unit. Don't open an issue with the product unit until you've thoroughly investigated the cause of a problem and can't resolve it on your own and with the help of the community on a public support forum. The product unit isn't able to troubleshoot individual apps that are broken due to simple misconfiguration or use cases involving third-party services. If a report is sensitive or confidential in nature or describes a potential security flaw in the product that cyberattackers may exploit, see Reporting security issues and bugs (dotnet/aspnetcore GitHub repository).

- Unauthorized client for ME-ID

  > info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2]
  > Authorization failed. These requirements were not met:
  > DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

  Login callback error from ME-ID:
  - Error: `unauthorized_client`
  - Description: `AADB2C90058: The provided application is not configured to allow public clients.`

  To resolve the error:

  1. In the Azure portal, access the app's manifest.
  2. Set the allowPublicClient attribute to `null` or `true`.

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser

    - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
    - Make sure that the browser is closed manually or by the IDE for any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in InPrivate or Incognito mode in Visual Studio:
    - Open **Browse With** dialog box from Visual Studio's **Run** button.
    - Select the **Add** button.
    - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
        - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
        - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
        - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
    - In the **Arguments** field, provide the command-line option that the browser uses to open in InPrivate or Incognito mode. Some browsers require the URL of the app.
        - Microsoft Edge: Use `-inprivate`.
        - Google Chrome: Use `--incognito --new-window {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
        - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
    - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
    - Select the **OK** button.
    - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
    - Make sure that the browser is closed by the IDE for any change to the app, test user, or provider configuration.

App upgrades

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Clear the local system's NuGet package caches by executing dotnet nuget locals all --clear from a command shell.
2. Delete the project's `bin` and `obj` folders.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

> **Note**
>
> Use of package versions incompatible with the app's target framework isn't supported. For information on a package, use the NuGet Gallery or FuGet Package Explorer.

Inspect the user

The following `User` component can be used directly in apps or serve as the basis for further customization.

`User.razor`:

razor

```
@page "/user"
@attribute [Authorize]
@using System.Text.Json
@using System.Security.Claims
@inject IAccessTokenProvider AuthorizationService

<h1>@AuthenticatedUser?.Identity?.Name</h1>

<h2>Claims</h2>

@foreach (var claim in AuthenticatedUser?.Claims ?? Array.Empty<Claim>())
{
    <p class="claim">@(claim.Type): @claim.Value</p>
}

<h2>Access token</h2>

<p id="access-token">@AccessToken?.Value</p>

<h2>Access token claims</h2>

@foreach (var claim in GetAccessTokenClaims())
{
    <p>@(claim.Key): @claim.Value.ToString()</p>
}
```

```
@if (AccessToken != null)
{
    <h2>Access token expires</h2>

    <p>Current time: <span id="current-time">@DateTimeOffset.Now</span></p>
    <p id="access-token-expires">@AccessToken.Expires</p>

    <h2>Access token granted scopes (as reported by the API)</h2>

    @foreach (var scope in AccessToken.GrantedScopes)
    {
        <p>Scope: @scope</p>
    }
}

@code {
    [CascadingParameter]
    private Task<AuthenticationState> AuthenticationState { get; set; }

    public ClaimsPrincipal AuthenticatedUser { get; set; }
    public AccessToken AccessToken { get; set; }

    protected override async Task OnInitializedAsync()
    {
        await base.OnInitializedAsync();
        var state = await AuthenticationState;
        var accessTokenResult = await
AuthorizationService.RequestAccessToken();

        if (!accessTokenResult.TryGetToken(out var token))
        {
            throw new InvalidOperationException(
                "Failed to provision the access token.");
        }

        AccessToken = token;

        AuthenticatedUser = state.User;
    }

    protected IDictionary<string, object> GetAccessTokenClaims()
    {
        if (AccessToken == null)
        {
            return new Dictionary<string, object>();
        }

        // header.payload.signature
        var payload = AccessToken.Value.Split(".")[1];
        var base64Payload = payload.Replace('-', '+').Replace('_', '/')
            .PadRight(payload.Length + (4 - payload.Length % 4) % 4, '=');

        return JsonSerializer.Deserialize<IDictionary<string, object>>(
            Convert.FromBase64String(base64Payload));
```

```
    }
}
```

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Example encoded JWT (shortened for display):

> eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMll0djhkbE5QNC1j ...
> bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLPr20tTS8Ct7_uwy9DWrzCMzpD-EiwT5IjXwlGX3IXVjHIlX50IVIydBoPQtadvT7saKo1G5Jmutgq41o-dmz6-yBMKV2_nXA25Q

Example JWT decoded by the tool for an app that authenticates against Azure AAD B2C:

JSON

```
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "X5eXk4xyojNFum1kl2Ytv8dlNP4-c57dO6QGTVBwaNk"
}.{
  "exp": 1610059429,
  "nbf": 1610055829,
  "ver": "1.0",
  "iss": "https://mysiteb2c.b2clogin.com/11112222-bbbb-3333-cccc-4444dddd5555/v2.0/",
  "sub": "aaaaaaaa-0000-1111-2222-bbbbbbbbbbbb",
  "aud": "00001111-aaaa-2222-bbbb-3333cccc4444",
  "nonce": "bbbb0000-cccc-1111-dddd-2222eeee3333",
  "iat": 1610055829,
  "auth_time": 1610055822,
  "idp": "idp.com",
  "tfp": "B2C_1_signupsignin"
}.[Signature]
```

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client

- Configure ASP.NET Core to work with proxy servers and load balancers: Includes guidance on:
  - Using Forwarded Headers Middleware to preserve HTTPS scheme information across proxy servers and internal networks.
  - Additional scenarios and use cases, including manual scheme configuration, request path changes for correct request routing, and forwarding the request scheme for Linux and non-IIS reverse proxies.

Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Accounts

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to create a standalone Blazor WebAssembly app that uses Microsoft Accounts with Microsoft Entra (ME-ID) for authentication.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Create a tenant in Azure
- Register an app in Azure
- Create the Blazor app
- Run the app

Create a tenant in Azure

Follow the guidance in Quickstart: Set up a tenant to create a tenant in ME-ID.

Register an app in Azure

Register an ME-ID app:

1. Navigate to **Microsoft Entra ID** in the Azure portal. Select **App registrations** in the sidebar. Select the **New registration** button.
2. Provide a **Name** for the app (for example, **Blazor Standalone ME-ID MS Accounts**).
3. In **Supported account types**, select **Accounts in any organizational directory (Any Microsoft Entra ID directory – Multitenant)**.

4. Set the **Redirect URI** dropdown list to **Single-page application (SPA)** and provide the following redirect URI: `https://localhost/authentication/login-callback`. If you know the production redirect URI for the Azure default host (for example, `azurewebsites.net`) or the custom domain host (for example, `contoso.com`), you can also add the production redirect URI at the same time that you're providing the `localhost` redirect URI. Be sure to include the port number for non-`:443` ports in any production redirect URIs that you add.
5. If you're using an unverified publisher domain, clear the **Permissions** > **Grant admin consent to openid and offline_access permissions** checkbox. If the publisher domain is verified, this checkbox isn't present.
6. Select **Register**.

> ⓘ **Note**
>
> Supplying the port number for a `localhost` ME-ID redirect URI isn't required. For more information, see Redirect URI (reply URL) restrictions and limitations: Localhost exceptions (Entra documentation).

Record the Application (client) ID (for example, `00001111-aaaa-2222-bbbb-3333cccc4444`).

In **Authentication** > **Platform configurations** > **Single-page application**:

1. Confirm the redirect URI of `https://localhost/authentication/login-callback` is present.
2. In the **Implicit grant** section, ensure that the checkboxes for **Access tokens** and **ID tokens** aren't selected. **Implicit grant isn't recommended for Blazor apps using MSAL v2.0 or later.** For more information, see Secure ASP.NET Core Blazor WebAssembly.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button if you made changes.

Create the Blazor app

Create the app. Replace the placeholders in the following command with the information recorded earlier and execute the following command in a command shell:

.NET CLI

```
dotnet new blazorwasm -au SingleOrg --client-id "{CLIENT ID}" --tenant-id
"common" -o {PROJECT NAME}
```

Expand table

Placeholder	Azure portal name	Example
`{PROJECT NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`00001111-aaaa-2222-bbbb-3333cccc4444`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the project's name.

Add a pair of MsalProviderOptions for `openid` and `offline_access` DefaultAccessTokenScopes:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
    options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
});
```

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press F5.
- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use Work or School Accounts (`SingleOrg`), the app automatically receives a package reference for the Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.Authentication.WebAssembly.Msal package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

The Microsoft.Authentication.WebAssembly.Msal package transitively adds the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the Microsoft.Authentication.WebAssembly.Msal package. This method sets up all of the services required for the app to interact with the Identity Provider (IP).

In the `Program` file:

```
C#

builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAd",
options.ProviderOptions.Authentication);
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the ME-ID configuration when you register the app.

`wwwroot/appsettings.json` configuration

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/common",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  }
}
```

Example:

JSON

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/common",
    "ClientId": "00001111-aaaa-2222-bbbb-3333cccc4444",
    "ValidateAuthority": true
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

Specify additional scopes with `AdditionalScopesToConsent`:

C#

```
options.ProviderOptions.AdditionalScopesToConsent.Add("{ADDITIONAL SCOPE
URI}");
```

> **Note**
>
> AdditionalScopesToConsent for Microsoft Graph via the Microsoft Entra ID consent UI when a user first uses an app registered in Microsoft Azure. For more information, see Use Graph API with ASP.NET Core Blazor WebAssembly.

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests
- Quickstart: Configure an application to expose web APIs

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case-sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
```

```
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

HTML

```
<script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationServic
e.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `App` component (`App.razor`) in the generated app.
- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

> **Note**

> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.
- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

razor

```razor
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string? Action { get; set; }
}
```

> **Note**
>
> Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core in .NET 6, the `string` type appears without the null type designation (`?`).

Troubleshoot

Logging

To enable debug or trace logging for Blazor WebAssembly authentication, see the *Client-side authentication logging* section of ASP.NET Core Blazor logging with the article version selector set to ASP.NET Core 7.0 or later.

Common errors

- Misconfiguration of the app or Identity Provider (IP)

  The most common errors are caused by incorrect configuration. The following are a few examples:
  - Depending on the requirements of the scenario, a missing or incorrect Authority, Instance, Tenant ID, Tenant domain, Client ID, or Redirect URI prevents an app from authenticating clients.
  - Incorrect request scopes prevent clients from accessing server web API endpoints.
  - Incorrect or missing server API permissions prevent clients from accessing server web API endpoints.
  - Running the app at a different port than is configured in the Redirect URI of the IP's app registration. Note that a port isn't required for Microsoft Entra ID and an app running at a `localhost` development testing address, but the app's port configuration and the port where the app is running must match for non-`localhost` addresses.

  Configuration sections of this article's guidance show examples of the correct configuration. Carefully check each section of the article looking for app and IP misconfiguration.

  If the configuration appears correct:

  - Analyze application logs.

- Examine the network traffic between the client app and the IP or server app with the browser's developer tools. Often, an exact error message or a message with a clue to what's causing the problem is returned to the client by the IP or server app after making a request. Developer tools guidance is found in the following articles:
  - Google Chrome (Google documentation)
  - Microsoft Edge
  - Mozilla Firefox (Mozilla documentation)

- For releases of Blazor where a JSON Web Token (JWT) is used, decode the contents of the token used for authenticating a client or accessing a server web API, depending on where the problem is occurring. For more information, see Inspect the content of a JSON Web Token (JWT).

The documentation team responds to document feedback and bugs in articles (open an issue from the **This page** feedback section) but is unable to provide product support. Several public support forums are available to assist with troubleshooting an app. We recommend the following:

- Stack Overflow (tag: blazor)
- ASP.NET Core Slack Team
- Blazor Gitter

The preceding forums are not owned or controlled by Microsoft.

For non-security, non-sensitive, and non-confidential reproducible framework bug reports, open an issue with the ASP.NET Core product unit. Don't open an issue with the product unit until you've thoroughly investigated the cause of a problem and can't resolve it on your own and with the help of the community on a public support forum. The product unit isn't able to troubleshoot individual apps that are broken due to simple misconfiguration or use cases involving third-party services. If a report is sensitive or confidential in nature or describes a potential security flaw in the product that cyberattackers may exploit, see Reporting security issues and bugs (dotnet/aspnetcore GitHub repository).

- Unauthorized client for ME-ID

> info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2]
> Authorization failed. These requirements were not met:
> DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

Login callback error from ME-ID:

- Error: `unauthorized_client`

- Description: `AADB2C90058: The provided application is not configured to allow public clients.`

To resolve the error:

1. In the Azure portal, access the app's manifest.
2. Set the allowPublicClient attribute to `null` or `true`.

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE for any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in InPrivate or Incognito mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in InPrivate or Incognito mode. Some browsers require the URL of the app.

    - Microsoft Edge: Use `-inprivate`.
    - Google Chrome: Use `--incognito --new-window {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
    - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE for any change to the app, test user, or provider configuration.

App upgrades

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Clear the local system's NuGet package caches by executing dotnet nuget locals all --clear from a command shell.
2. Delete the project's `bin` and `obj` folders.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

> **Note**
>
> Use of package versions incompatible with the app's target framework isn't supported. For information on a package, use the NuGet Gallery or FuGet Package Explorer.

Inspect the user

The following `User` component can be used directly in apps or serve as the basis for further customization.

`User.razor`:

```
razor
```

```
@page "/user"
@attribute [Authorize]
@using System.Text.Json
@using System.Security.Claims
@inject IAccessTokenProvider AuthorizationService

<h1>@AuthenticatedUser?.Identity?.Name</h1>

<h2>Claims</h2>

@foreach (var claim in AuthenticatedUser?.Claims ?? Array.Empty<Claim>())
{
    <p class="claim">@(claim.Type): @claim.Value</p>
}

<h2>Access token</h2>

<p id="access-token">@AccessToken?.Value</p>

<h2>Access token claims</h2>

@foreach (var claim in GetAccessTokenClaims())
{
    <p>@(claim.Key): @claim.Value.ToString()</p>
}

@if (AccessToken != null)
{
    <h2>Access token expires</h2>

    <p>Current time: <span id="current-time">@DateTimeOffset.Now</span></p>
    <p id="access-token-expires">@AccessToken.Expires</p>

    <h2>Access token granted scopes (as reported by the API)</h2>

    @foreach (var scope in AccessToken.GrantedScopes)
    {
        <p>Scope: @scope</p>
    }
}

@code {
    [CascadingParameter]
    private Task<AuthenticationState> AuthenticationState { get; set; }

    public ClaimsPrincipal AuthenticatedUser { get; set; }
    public AccessToken AccessToken { get; set; }

    protected override async Task OnInitializedAsync()
    {
        await base.OnInitializedAsync();
        var state = await AuthenticationState;
        var accessTokenResult = await
AuthorizationService.RequestAccessToken();
```

```
        if (!accessTokenResult.TryGetToken(out var token))
        {
            throw new InvalidOperationException(
                "Failed to provision the access token.");
        }

        AccessToken = token;

        AuthenticatedUser = state.User;
    }

    protected IDictionary<string, object> GetAccessTokenClaims()
    {
        if (AccessToken == null)
        {
            return new Dictionary<string, object>();
        }

        // header.payload.signature
        var payload = AccessToken.Value.Split(".")[1];
        var base64Payload = payload.Replace('-', '+').Replace('_', '/')
            .PadRight(payload.Length + (4 - payload.Length % 4) % 4, '=');

        return JsonSerializer.Deserialize<IDictionary<string, object>>(
            Convert.FromBase64String(base64Payload));
    }
}
```

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Example encoded JWT (shortened for display):

> eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMll0djhkbE5QNC1j ...
> bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLPr20tTS8Ct7_uwy9DWrzCMzpD-EiwT5IjXwlGX3IXVjHIlX50IVIydBoPQtadvT7saKo1G5Jmutgq41o-dmz6-yBMKV2_nXA25Q

Example JWT decoded by the tool for an app that authenticates against Azure AAD B2C:

JSON

```
{
  "typ": "JWT",
  "alg": "RS256",
```

```
  "kid": "X5eXk4xyojNFum1kl2Ytv8dlNP4-c57dO6QGTVBwaNk"
}.{
  "exp": 1610059429,
  "nbf": 1610055829,
  "ver": "1.0",
  "iss": "https://mysiteb2c.b2clogin.com/11112222-bbbb-3333-cccc-
4444dddd5555/v2.0/",
  "sub": "aaaaaaaa-0000-1111-2222-bbbbbbbbbbbb",
  "aud": "00001111-aaaa-2222-bbbb-3333cccc4444",
  "nonce": "bbbb0000-cccc-1111-dddd-2222eeee3333",
  "iat": 1610055829,
  "auth_time": 1610055822,
  "idp": "idp.com",
  "tfp": "B2C_1_signupsignin"
}.[Signature]
```


Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Build a custom version of the Authentication.MSAL JavaScript library
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles
- Quickstart: Register an application with the Microsoft identity platform
- Quickstart: Configure an application to expose web APIs

Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to create a standalone Blazor WebAssembly app that uses Microsoft Entra ID (ME-ID) for authentication.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Create a tenant in Azure
- Register an app in Azure
- Create the Blazor app
- Run the app

Create a tenant in Azure

Follow the guidance in Quickstart: Set up a tenant to create a tenant in ME-ID.

Register an app in Azure

Register an ME-ID app:

1. Navigate to **Microsoft Entra ID** in the Azure portal. Select **Applications** > **App registrations** in the sidebar. Select the **New registration** button.
2. Provide a **Name** for the app (for example, **Blazor Standalone ME-ID**).
3. Choose a **Supported account types**. You may select **Accounts in this organizational directory only** for this experience.

4. Set the **Redirect URI** dropdown list to **Single-page application (SPA)** and provide the following redirect URI: `https://localhost/authentication/login-callback`. If you know the production redirect URI for the Azure default host (for example, `azurewebsites.net`) or the custom domain host (for example, `contoso.com`), you can also add the production redirect URI at the same time that you're providing the `localhost` redirect URI. Be sure to include the port number for non-`:443` ports in any production redirect URIs that you add.
5. If you're using an unverified publisher domain, clear the **Permissions** > **Grant admin consent to openid and offline_access permissions** checkbox. If the publisher domain is verified, this checkbox isn't present.
6. Select **Register**.

> ⓘ **Note**
>
> Supplying the port number for a `localhost` ME-ID redirect URI isn't required. For more information, see Redirect URI (reply URL) restrictions and limitations: Localhost exceptions (Entra documentation).

Record the following information:

- Application (client) ID (for example, `00001111-aaaa-2222-bbbb-3333cccc4444`)
- Directory (tenant) ID (for example, `aaaabbbb-0000-cccc-1111-dddd2222eeee`)

In **Authentication** > **Platform configurations** > **Single-page application**:

1. Confirm the redirect URI of `https://localhost/authentication/login-callback` is present.
2. In the **Implicit grant** section, ensure that the checkboxes for **Access tokens** and **ID tokens** aren't selected. **Implicit grant isn't recommended for Blazor apps using MSAL v2.0 or later.** For more information, see Secure ASP.NET Core Blazor WebAssembly.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button if you made changes.

Create the Blazor app

Create the app in an empty folder. Replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

```
.NET CLI
```

```
dotnet new blazorwasm -au SingleOrg --client-id "{CLIENT ID}" -o {PROJECT
NAME} --tenant-id "{TENANT ID}"
```

Expand table

Placeholder	Azure portal name	Example
`{PROJECT NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`00001111-aaaa-2222-bbbb-3333cccc4444`
`{TENANT ID}`	Directory (tenant) ID	`aaaabbbb-0000-cccc-1111-dddd2222eeee`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the project's name.

Add a MsalProviderOptions for `User.Read` permission with DefaultAccessTokenScopes:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes
        .Add("https://graph.microsoft.com/User.Read");
});
```

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press `F5`.
- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance

in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use Work or School Accounts (`SingleOrg`), the app automatically receives a package reference for the Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.Authentication.WebAssembly.Msal package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

The Microsoft.Authentication.WebAssembly.Msal package transitively adds the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the Microsoft.Authentication.WebAssembly.Msal package. This method sets up the services required for the app to interact with the Identity Provider (IP).

In the `Program` file:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAd",
options.ProviderOptions.Authentication);
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the ME-ID configuration when you register the app.

`wwwroot/appsettings.json` configuration

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  }
}
```

Example:

JSON

```
{
  "AzureAd": {
    "Authority":
"https://login.microsoftonline.com/e86c78e2-...-918e0565a45e",
    "ClientId": "00001111-aaaa-2222-bbbb-3333cccc4444",
    "ValidateAuthority": true
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

Specify additional scopes with `AdditionalScopesToConsent`:

```
C#

options.ProviderOptions.AdditionalScopesToConsent.Add("{ADDITIONAL SCOPE
URI}");
```

> ⓘ Note
>
> AdditionalScopesToConsent isn't able to provision delegated user permissions for Microsoft Graph via the Microsoft Entra ID consent UI when a user first uses an app registered in Microsoft Azure. For more information, see Use Graph API with ASP.NET Core Blazor WebAssembly.

For more information, see the following resources:

- Request additional access tokens
- Attach tokens to outgoing requests
- Quickstart: Configure an application to expose web APIs
- Access token scopes for Microsoft Graph API

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

```
C#

builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case-sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

HTML

```
<script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationServic
e.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to

use. Inspect the `App` component (`App.razor`) in the generated app.

- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

  > **Note**
  >
  > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.

- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

  > **Note**
  >
  > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

```
razor
```

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string? Action { get; set; }
}
```

> ⓘ Note
>
> Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core in .NET 6, the `string` type appears without the null type designation (`?`).

Troubleshoot

Logging

To enable debug or trace logging for Blazor WebAssembly authentication, see the *Client-side authentication logging* section of ASP.NET Core Blazor logging with the article version selector set to ASP.NET Core 7.0 or later.

Common errors

- Misconfiguration of the app or Identity Provider (IP)

  The most common errors are caused by incorrect configuration. The following are a few examples:
  - Depending on the requirements of the scenario, a missing or incorrect Authority, Instance, Tenant ID, Tenant domain, Client ID, or Redirect URI prevents an app from authenticating clients.
  - Incorrect request scopes prevent clients from accessing server web API endpoints.
  - Incorrect or missing server API permissions prevent clients from accessing server web API endpoints.
  - Running the app at a different port than is configured in the Redirect URI of the IP's app registration. Note that a port isn't required for Microsoft Entra ID and an app running at a `localhost` development testing address, but the app's port

configuration and the port where the app is running must match for non-`localhost` addresses.

Configuration sections of this article's guidance show examples of the correct configuration. Carefully check each section of the article looking for app and IP misconfiguration.

If the configuration appears correct:

- Analyze application logs.
- Examine the network traffic between the client app and the IP or server app with the browser's developer tools. Often, an exact error message or a message with a clue to what's causing the problem is returned to the client by the IP or server app after making a request. Developer tools guidance is found in the following articles:
  - Google Chrome (Google documentation)
  - Microsoft Edge
  - Mozilla Firefox (Mozilla documentation)
- For releases of Blazor where a JSON Web Token (JWT) is used, decode the contents of the token used for authenticating a client or accessing a server web API, depending on where the problem is occurring. For more information, see Inspect the content of a JSON Web Token (JWT).

The documentation team responds to document feedback and bugs in articles (open an issue from the **This page** feedback section) but is unable to provide product support. Several public support forums are available to assist with troubleshooting an app. We recommend the following:

- Stack Overflow (tag: blazor)
- ASP.NET Core Slack Team
- Blazor Gitter

The preceding forums are not owned or controlled by Microsoft.

For non-security, non-sensitive, and non-confidential reproducible framework bug reports, open an issue with the ASP.NET Core product unit. Don't open an issue with the product unit until you've thoroughly investigated the cause of a problem and can't resolve it on your own and with the help of the community on a public support forum. The product unit isn't able to troubleshoot individual apps that are broken due to simple misconfiguration or use cases involving third-party services. If a report is sensitive or confidential in nature or describes a potential security flaw

in the product that cyberattackers may exploit, see Reporting security issues and bugs (dotnet/aspnetcore GitHub repository).

- Unauthorized client for ME-ID

  > info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2]
  > Authorization failed. These requirements were not met:
  > DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

  Login callback error from ME-ID:
  - Error: `unauthorized_client`
  - Description: `AADB2C90058: The provided application is not configured to allow public clients.`

  To resolve the error:

  1. In the Azure portal, access the app's manifest.
  2. Set the allowPublicClient attribute to `null` or `true`.

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE for any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in InPrivate or Incognito mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.

- Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
  - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
  - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
  - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
- In the **Arguments** field, provide the command-line option that the browser uses to open in InPrivate or Incognito mode. Some browsers require the URL of the app.
  - Microsoft Edge: Use `-inprivate`.
  - Google Chrome: Use `--incognito --new-window {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
  - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
- Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
- Select the **OK** button.
- To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
- Make sure that the browser is closed by the IDE for any change to the app, test user, or provider configuration.

App upgrades

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Clear the local system's NuGet package caches by executing dotnet nuget locals all --clear from a command shell.
2. Delete the project's `bin` and `obj` folders.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

> **Note**

Use of package versions incompatible with the app's target framework isn't supported. For information on a package, use the NuGet Gallery or FuGet Package Explorer.

Inspect the user

The following `User` component can be used directly in apps or serve as the basis for further customization.

`User.razor`:

razor

```razor
@page "/user"
@attribute [Authorize]
@using System.Text.Json
@using System.Security.Claims
@inject IAccessTokenProvider AuthorizationService

<h1>@AuthenticatedUser?.Identity?.Name</h1>

<h2>Claims</h2>

@foreach (var claim in AuthenticatedUser?.Claims ?? Array.Empty<Claim>())
{
    <p class="claim">@(claim.Type): @claim.Value</p>
}

<h2>Access token</h2>

<p id="access-token">@AccessToken?.Value</p>

<h2>Access token claims</h2>

@foreach (var claim in GetAccessTokenClaims())
{
    <p>@(claim.Key): @claim.Value.ToString()</p>
}

@if (AccessToken != null)
{
    <h2>Access token expires</h2>

    <p>Current time: <span id="current-time">@DateTimeOffset.Now</span></p>
    <p id="access-token-expires">@AccessToken.Expires</p>

    <h2>Access token granted scopes (as reported by the API)</h2>

    @foreach (var scope in AccessToken.GrantedScopes)
    {
```

```
        <p>Scope: @scope</p>
    }
}

@code {
    [CascadingParameter]
    private Task<AuthenticationState> AuthenticationState { get; set; }

    public ClaimsPrincipal AuthenticatedUser { get; set; }
    public AccessToken AccessToken { get; set; }

    protected override async Task OnInitializedAsync()
    {
        await base.OnInitializedAsync();
        var state = await AuthenticationState;
        var accessTokenResult = await
AuthorizationService.RequestAccessToken();

        if (!accessTokenResult.TryGetToken(out var token))
        {
            throw new InvalidOperationException(
                "Failed to provision the access token.");
        }

        AccessToken = token;

        AuthenticatedUser = state.User;
    }

    protected IDictionary<string, object> GetAccessTokenClaims()
    {
        if (AccessToken == null)
        {
            return new Dictionary<string, object>();
        }

        // header.payload.signature
        var payload = AccessToken.Value.Split(".")[1];
        var base64Payload = payload.Replace('-', '+').Replace('_', '/')
            .PadRight(payload.Length + (4 - payload.Length % 4) % 4, '=');

        return JsonSerializer.Deserialize<IDictionary<string, object>>(
            Convert.FromBase64String(base64Payload));
    }
}
```

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Example encoded JWT (shortened for display):

> eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMll0djhkbE5QNC1j ...
> bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLPr20tTS8Ct7_uwy9DWrzCMzpD-EiwT5IjXwlGX3IXVjHIlX50IVIydBoPQtadvT7saKo1G5Jmutgq41o-dmz6-yBMKV2_nXA25Q

Example JWT decoded by the tool for an app that authenticates against Azure AAD B2C:

JSON

```
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "X5eXk4xyojNFum1kl2Ytv8dlNP4-c57dO6QGTVBwaNk"
}.{
  "exp": 1610059429,
  "nbf": 1610055829,
  "ver": "1.0",
  "iss": "https://mysiteb2c.b2clogin.com/11112222-bbbb-3333-cccc-4444dddd5555/v2.0/",
  "sub": "aaaaaaaa-0000-1111-2222-bbbbbbbbbbbb",
  "aud": "00001111-aaaa-2222-bbbb-3333cccc4444",
  "nonce": "bbbb0000-cccc-1111-dddd-2222eeee3333",
  "iat": 1610055829,
  "auth_time": 1610055822,
  "idp": "idp.com",
  "tfp": "B2C_1_signupsignin"
}.[Signature]
```

Additional resources

- Identity and account types for single- and multitenant apps
- ASP.NET Core Blazor WebAssembly additional security scenarios
- Build a custom version of the Authentication.MSAL JavaScript library
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles
- Microsoft identity platform and Microsoft Entra ID with ASP.NET Core
- Microsoft identity platform documentation
- Security best practices for application properties in Microsoft Entra ID

Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory B2C

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to create a standalone Blazor WebAssembly app that uses Azure Active Directory (AAD) B2C for authentication.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Create a tenant in Azure
- Register an app in Azure
- Create the Blazor app
- Run the app

Create a tenant in Azure

Follow the guidance in Tutorial: Create an Azure Active Directory B2C tenant to create an AAD B2C tenant.

Before proceeding with this article's guidance, confirm that you've selected the correct directory for the AAD B2C tenant.

Register an app in Azure

Register an AAD B2C app:

1. Navigate to **Azure AD B2C** in the Azure portal. Select **App registrations** in the sidebar. Select the **New registration** button.

2. Provide a **Name** for the app (for example, **Blazor Standalone AAD B2C**).
3. For **Supported account types**, select the multi-tenant option: **Accounts in any organizational directory or any identity provider. For authenticating users with Azure AD B2C.**
4. Set the **Redirect URI** dropdown list to **Single-page application (SPA)** and provide the following redirect URI: `https://localhost/authentication/login-callback`. If you know the production redirect URI for the Azure default host (for example, `azurewebsites.net`) or the custom domain host (for example, `contoso.com`), you can also add the production redirect URI at the same time that you're providing the `localhost` redirect URI. Be sure to include the port number for non-`:443` ports in any production redirect URIs that you add.
5. If you're using an unverified publisher domain, confirm that **Permissions** > **Grant admin consent to openid and offline_access permissions** is selected. If the publisher domain is verified, this checkbox isn't present.
6. Select **Register**.

> **Note**
>
> Supplying the port number for a `localhost` AAD B2C redirect URI isn't required. For more information, see Redirect URI (reply URL) restrictions and limitations: Localhost exceptions (Entra documentation).

Record the following information:

- Application (client) ID (for example, `00001111-aaaa-2222-bbbb-3333cccc4444`).
- AAD B2C instance (for example, `https://contoso.b2clogin.com/`, which includes the trailing slash): The instance is the scheme and host of an Azure B2C app registration, which can be found by opening the **Endpoints** window from the **App registrations** page in the Azure portal.
- AAD B2C Primary/Publisher/Tenant domain (for example, `contoso.onmicrosoft.com`): The domain is available as the **Publisher domain** in the **Branding** blade of the Azure portal for the registered app.

In **Authentication** > **Platform configurations** > **Single-page application**:

1. Confirm the redirect URI of `https://localhost/authentication/login-callback` is present.
2. In the **Implicit grant** section, ensure that the checkboxes for **Access tokens** and **ID tokens** aren't selected. **Implicit grant isn't recommended for Blazor apps using MSAL v2.0 or later.** For more information, see Secure ASP.NET Core Blazor WebAssembly.

3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button if you made changes.

In **Home** > **Azure AD B2C** > **User flows**:

Create a sign-up and sign-in user flow

At a minimum, select the **Application claims** > **Display Name** user attribute to populate the `context.User.Identity?.Name`/`context.User.Identity.Name` in the `LoginDisplay` component (`Shared/LoginDisplay.razor`).

Record the sign-up and sign-in user flow name created for the app (for example, `B2C_1_signupsignin`).

Create the Blazor app

In an empty folder, replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

.NET CLI

```
dotnet new blazorwasm -au IndividualB2C --aad-b2c-instance "{AAD B2C
INSTANCE}" --client-id "{CLIENT ID}" --domain "{TENANT DOMAIN}" -o {PROJECT
NAME} -ssp "{SIGN UP OR SIGN IN POLICY}"
```

Expand table

Placeholder	Azure portal name	Example
`{AAD B2C INSTANCE}`	Instance	`https://contoso.b2clogin.com/` (includes the trailing slash)
`{PROJECT NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`00001111-aaaa-2222-bbbb-3333cccc4444`
`{SIGN UP OR SIGN IN POLICY}`	Sign-up/sign-in user flow	`B2C_1_signupsignin1`
`{TENANT DOMAIN}`	Primary/Publisher/Tenant domain	`contoso.onmicrosoft.com`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the project's name.

Add a pair of MsalProviderOptions for `openid` and `offline_access` DefaultAccessTokenScopes:

```
C#

builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
    options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
});
```

After creating the app, you should be able to:

- Log into the app using an Microsoft Entra ID user account.
- Request access tokens for Microsoft APIs. For more information, see:
  - Access token scopes
  - Quickstart: Configure an application to expose web APIs.

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press `F5`.
- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use an Individual B2C Account (`IndividualB2C`), the app automatically receives a package reference for the Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.Authentication.WebAssembly.Msal package to the app.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

The Microsoft.Authentication.WebAssembly.Msal package transitively adds the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the Microsoft.Authentication.WebAssembly.Msal package. This method sets up all of the services required for the app to interact with the Identity Provider (IP).

In the `Program` file:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAdB2C",
options.ProviderOptions.Authentication);
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the configuration when you register the app.

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```
{
  "AzureAdB2C": {
```

```
    "Authority": "{AAD B2C INSTANCE}{TENANT DOMAIN}/{SIGN UP OR SIGN IN
POLICY}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": false
  }
}
```

In the preceding configuration, the `{AAD B2C INSTANCE}` includes a trailing slash.

Example:

JSON

```
{
  "AzureAdB2C": {
    "Authority":
"https://contoso.b2clogin.com/contoso.onmicrosoft.com/B2C_1_signupsignin1",
    "ClientId": "00001111-aaaa-2222-bbbb-3333cccc4444",
    "ValidateAuthority": false
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

Specify additional scopes with `AdditionalScopesToConsent`:

C#

```
options.ProviderOptions.AdditionalScopesToConsent.Add("{ADDITIONAL SCOPE
URI}");
```

> **Note**

AdditionalScopesToConsent isn't able to provision delegated user permissions for Microsoft Graph via the Microsoft Entra ID consent UI when a user first uses an app registered in Microsoft Azure. For more information, see Use Graph API with ASP.NET Core Blazor WebAssembly.

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case-sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

HTML

```
<script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationServic
e.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `App` component (`App.razor`) in the generated app.

- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

  > **Note**
  >
  > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags**

dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.
- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

razor

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code {
    [Parameter]
    public string? Action { get; set; }
}
```

> **Note**
>
> Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core

in .NET 6, the `string` type appears without the null type designation (`?`).

Custom policies

The Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal, NuGet package) doesn't support AAD B2C custom policies.

Troubleshoot

Logging

To enable debug or trace logging for Blazor WebAssembly authentication, see the *Client-side authentication logging* section of ASP.NET Core Blazor logging with the article version selector set to ASP.NET Core 7.0 or later.

Common errors

- Misconfiguration of the app or Identity Provider (IP)

  The most common errors are caused by incorrect configuration. The following are a few examples:
  - Depending on the requirements of the scenario, a missing or incorrect Authority, Instance, Tenant ID, Tenant domain, Client ID, or Redirect URI prevents an app from authenticating clients.
  - Incorrect request scopes prevent clients from accessing server web API endpoints.
  - Incorrect or missing server API permissions prevent clients from accessing server web API endpoints.
  - Running the app at a different port than is configured in the Redirect URI of the IP's app registration. Note that a port isn't required for Microsoft Entra ID and an app running at a `localhost` development testing address, but the app's port configuration and the port where the app is running must match for non-`localhost` addresses.

  Configuration sections of this article's guidance show examples of the correct configuration. Carefully check each section of the article looking for app and IP misconfiguration.

  If the configuration appears correct:

    - Analyze application logs.
    - Examine the network traffic between the client app and the IP or server app with the browser's developer tools. Often, an exact error message or a message with a clue to what's causing the problem is returned to the client by the IP or server app after making a request. Developer tools guidance is found in the following articles:
        - Google Chrome (Google documentation)
        - Microsoft Edge
        - Mozilla Firefox (Mozilla documentation)
    - For releases of Blazor where a JSON Web Token (JWT) is used, decode the contents of the token used for authenticating a client or accessing a server web API, depending on where the problem is occurring. For more information, see Inspect the content of a JSON Web Token (JWT).

  The documentation team responds to document feedback and bugs in articles (open an issue from the **This page** feedback section) but is unable to provide product support. Several public support forums are available to assist with troubleshooting an app. We recommend the following:
    - Stack Overflow (tag: blazor)
    - ASP.NET Core Slack Team
    - Blazor Gitter

  The preceding forums are not owned or controlled by Microsoft.

  For non-security, non-sensitive, and non-confidential reproducible framework bug reports, open an issue with the ASP.NET Core product unit. Don't open an issue with the product unit until you've thoroughly investigated the cause of a problem and can't resolve it on your own and with the help of the community on a public support forum. The product unit isn't able to troubleshoot individual apps that are broken due to simple misconfiguration or use cases involving third-party services. If a report is sensitive or confidential in nature or describes a potential security flaw in the product that cyberattackers may exploit, see Reporting security issues and bugs (dotnet/aspnetcore GitHub repository).

- Unauthorized client for ME-ID

  > info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2]
  > Authorization failed. These requirements were not met:
  > DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

  Login callback error from ME-ID:

- Error: `unauthorized_client`
- Description: `AADB2C90058: The provided application is not configured to allow public clients.`

To resolve the error:

1. In the Azure portal, access the app's manifest.
2. Set the allowPublicClient attribute to `null` or `true`.

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE for any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in InPrivate or Incognito mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in InPrivate or Incognito mode. Some browsers require the URL of the

app.
    - Microsoft Edge: Use `-inprivate`.
    - Google Chrome: Use `--incognito --new-window {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
    - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE for any change to the app, test user, or provider configuration.

App upgrades

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Clear the local system's NuGet package caches by executing dotnet nuget locals all --clear from a command shell.
2. Delete the project's `bin` and `obj` folders.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

> **Note**
>
> Use of package versions incompatible with the app's target framework isn't supported. For information on a package, use the NuGet Gallery or FuGet Package Explorer.

Inspect the user

The following `User` component can be used directly in apps or serve as the basis for further customization.

`User.razor`:

razor

```razor
@page "/user"
@attribute [Authorize]
@using System.Text.Json
@using System.Security.Claims
@inject IAccessTokenProvider AuthorizationService

<h1>@AuthenticatedUser?.Identity?.Name</h1>

<h2>Claims</h2>

@foreach (var claim in AuthenticatedUser?.Claims ?? Array.Empty<Claim>())
{
    <p class="claim">@(claim.Type): @claim.Value</p>
}

<h2>Access token</h2>

<p id="access-token">@AccessToken?.Value</p>

<h2>Access token claims</h2>

@foreach (var claim in GetAccessTokenClaims())
{
    <p>@(claim.Key): @claim.Value.ToString()</p>
}

@if (AccessToken != null)
{
    <h2>Access token expires</h2>

    <p>Current time: <span id="current-time">@DateTimeOffset.Now</span></p>
    <p id="access-token-expires">@AccessToken.Expires</p>

    <h2>Access token granted scopes (as reported by the API)</h2>

    @foreach (var scope in AccessToken.GrantedScopes)
    {
        <p>Scope: @scope</p>
    }
}

@code {
    [CascadingParameter]
    private Task<AuthenticationState> AuthenticationState { get; set; }

    public ClaimsPrincipal AuthenticatedUser { get; set; }
    public AccessToken AccessToken { get; set; }

    protected override async Task OnInitializedAsync()
    {
        await base.OnInitializedAsync();
        var state = await AuthenticationState;
```

```
        var accessTokenResult = await
AuthorizationService.RequestAccessToken();

        if (!accessTokenResult.TryGetToken(out var token))
        {
            throw new InvalidOperationException(
                "Failed to provision the access token.");
        }

        AccessToken = token;

        AuthenticatedUser = state.User;
    }

    protected IDictionary<string, object> GetAccessTokenClaims()
    {
        if (AccessToken == null)
        {
            return new Dictionary<string, object>();
        }

        // header.payload.signature
        var payload = AccessToken.Value.Split(".")[1];
        var base64Payload = payload.Replace('-', '+').Replace('_', '/')
            .PadRight(payload.Length + (4 - payload.Length % 4) % 4, '=');

        return JsonSerializer.Deserialize<IDictionary<string, object>>(
            Convert.FromBase64String(base64Payload));
    }
}
```

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Example encoded JWT (shortened for display):

> eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMll0djhkbE5QNC1j ...
> bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLPr20tTS8Ct7_uwy9DWrzCMzpD-EiwT5IjXwlGX3IXVjHIlX50IVIydBoPQtadvT7saKo1G5Jmutgq41o-dmz6-yBMKV2_nXA25Q

Example JWT decoded by the tool for an app that authenticates against Azure AAD B2C:

```
JSON
```

```
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "X5eXk4xyojNFum1kl2Ytv8dlNP4-c57dO6QGTVBwaNk"
}.{
  "exp": 1610059429,
  "nbf": 1610055829,
  "ver": "1.0",
  "iss": "https://mysiteb2c.b2clogin.com/11112222-bbbb-3333-cccc-
4444dddd5555/v2.0/",
  "sub": "aaaaaaaa-0000-1111-2222-bbbbbbbbbbbb",
  "aud": "00001111-aaaa-2222-bbbb-3333cccc4444",
  "nonce": "bbbb0000-cccc-1111-dddd-2222eeee3333",
  "iat": 1610055829,
  "auth_time": 1610055822,
  "idp": "idp.com",
  "tfp": "B2C_1_signupsignin"
}.[Signature]
```


Additional resources

- Identity and account types for single- and multitenant apps
- ASP.NET Core Blazor WebAssembly additional security scenarios
- Build a custom version of the Authentication.MSAL JavaScript library
- Unauthenticated or unauthorized web API requests in an app with a secure default client
- Cloud authentication with Azure Active Directory B2C in ASP.NET Core
- Tutorial: Create an Azure Active Directory B2C tenant
- Tutorial: Register an application in Azure Active Directory B2C
- Microsoft identity platform documentation

ASP.NET Core Blazor WebAssembly additional security scenarios

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes additional security scenarios for Blazor WebAssembly apps.

Attach tokens to outgoing requests

AuthorizationMessageHandler is a DelegatingHandler used to process access tokens. Tokens are acquired using the IAccessTokenProvider service, which is registered by the framework. If a token can't be acquired, an AccessTokenNotAvailableException is thrown. AccessTokenNotAvailableException has a Redirect method that navigates to AccessTokenResult.InteractiveRequestUrl using the given AccessTokenResult.InteractionOptions to allow refreshing the access token.

For convenience, the framework provides the BaseAddressAuthorizationMessageHandler preconfigured with the app's base address as an authorized URL. **Access tokens are only added when the request URI is within the app's base URI.** When outgoing request URIs aren't within the app's base URI, use a custom AuthorizationMessageHandler class (*recommended*) or configure the AuthorizationMessageHandler.

> **Note**
>
> In addition to the client app configuration for server API access, the server API must also allow cross-origin requests (CORS) when the client and the server don't reside at the same base address. For more information on server-side CORS configuration, see the Cross-Origin Resource Sharing (CORS) section later in this article.

In the following example:

- AddHttpClient adds IHttpClientFactory and related services to the service collection and configures a named HttpClient (`WebAPI`). HttpClient.BaseAddress is the base

address of the resource URI when sending requests. IHttpClientFactory is provided by the Microsoft.Extensions.Http NuGet package.
- BaseAddressAuthorizationMessageHandler is the DelegatingHandler used to process access tokens. Access tokens are only added when the request URI is within the app's base URI.
- IHttpClientFactory.CreateClient creates and configures an HttpClient instance for outgoing requests using the configuration that corresponds to the named HttpClient (`WebAPI`).

In the following example, HttpClientFactoryServiceCollectionExtensions.AddHttpClient is an extension in Microsoft.Extensions.Http. Add the package to an app that doesn't already reference it.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

C#

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddHttpClient("WebAPI",
        client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>()
    .CreateClient("WebAPI"));
```

The configured HttpClient is used to make authorized requests using the try-catch pattern:

razor

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject HttpClient Http

...

protected override async Task OnInitializedAsync()
{
```

```
    try
    {
        var examples =
            await Http.GetFromJsonAsync<ExampleType[]>("ExampleAPIMethod");

        ...
    }
    catch (AccessTokenNotAvailableException exception)
    {
        exception.Redirect();
    }
}
```

Custom authentication request scenarios

The following scenarios demonstrate how to customize authentication requests and how to obtain the login path from authentication options.

Customize the login process

Manage additional parameters to a login request with the following methods one or more times on a new instance of InteractiveRequestOptions:

- TryAddAdditionalParameter
- TryRemoveAdditionalParameter
- TryGetAdditionalParameter

In the following `LoginDisplay` component example, additional parameters are added to the login request:

- `prompt` is set to `login`: Forces the user to enter their credentials on that request, negating single sign on.
- `loginHint` is set to `peter@contoso.com`: Pre-fills the username/email address field of the sign-in page for the user to `peter@contoso.com`. Apps often use this parameter during re-authentication, having already extracted the username from a previous sign in using the `preferred_username` claim.

`Shared/LoginDisplay.razor`:

C#

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation
```

```
<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity?.Name!
        <button @onclick="BeginLogOut">Log out</button>
    </Authorized>
    <NotAuthorized>
        <button @onclick="BeginLogIn">Log in</button>
    </NotAuthorized>
</AuthorizeView>

@code{
    public void BeginLogOut()
    {
        Navigation.NavigateToLogout("authentication/logout");
    }

    public void BeginLogIn()
    {
        InteractiveRequestOptions requestOptions =
            new()
            {
                Interaction = InteractionType.SignIn,
                ReturnUrl = Navigation.Uri,
            };

        requestOptions.TryAddAdditionalParameter("prompt", "login");
        requestOptions.TryAddAdditionalParameter("loginHint",
"peter@contoso.com");

        Navigation.NavigateToLogin("authentication/login", requestOptions);
    }
}
```

For more information, see the following resources:

- InteractiveRequestOptions
- Popup request parameter list

Customize options before obtaining a token interactively

If an AccessTokenNotAvailableException occurs, manage additional parameters for a new identity provider access token request with the following methods one or more times on a new instance of InteractiveRequestOptions:

- TryAddAdditionalParameter
- TryRemoveAdditionalParameter
- TryGetAdditionalParameter

In the following example that obtains JSON data via web API, additional parameters are added to the redirect request if an access token isn't available

(AccessTokenNotAvailableException is thrown):

- `prompt` is set to `login`: Forces the user to enter their credentials on that request, negating single sign on.
- `loginHint` is set to `peter@contoso.com`: Pre-fills the username/email address field of the sign-in page for the user to `peter@contoso.com`. Apps often use this parameter during re-authentication, having already extracted the username from a previous sign in using the `preferred_username` claim.

C#

```
try
{
    var examples = await Http.GetFromJsonAsync<ExampleType[]>
("ExampleAPIMethod");

    ...
}
catch (AccessTokenNotAvailableException ex)
{
    ex.Redirect(requestOptions => {
        requestOptions.TryAddAdditionalParameter("prompt", "login");
        requestOptions.TryAddAdditionalParameter("loginHint",
"peter@contoso.com");
    });
}
```

The preceding example assumes that:

- The presence of an `@using`/`using` statement for API in the Microsoft.AspNetCore.Components.WebAssembly.Authentication namespace.
- `HttpClient` injected as `Http`.

For more information, see the following resources:

- InteractiveRequestOptions
- Redirect request parameter list

Customize options when using an `IAccessTokenProvider`

If obtaining a token fails when using an IAccessTokenProvider, manage additional parameters for a new identity provider access token request with the following methods one or more times on a new instance of InteractiveRequestOptions:

- TryAddAdditionalParameter
- TryRemoveAdditionalParameter

- TryGetAdditionalParameter

In the following example that attempts to obtain an access token for the user, additional parameters are added to the login request if the attempt to obtain a token fails when TryGetToken is called:

- `prompt` is set to `login`: Forces the user to enter their credentials on that request, negating single sign on.
- `loginHint` is set to `peter@contoso.com`: Pre-fills the username/email address field of the sign-in page for the user to `peter@contoso.com`. Apps often use this parameter during re-authentication, having already extracted the username from a previous sign in using the `preferred_username` claim.

C#

```
var tokenResult = await TokenProvider.RequestAccessToken(
    new AccessTokenRequestOptions
    {
        Scopes = new[] { ... }
    });

if (!tokenResult.TryGetToken(out var token))
{
    tokenResult.InteractionOptions.TryAddAdditionalParameter("prompt",
"login");
    tokenResult.InteractionOptions.TryAddAdditionalParameter("loginHint",
        "peter@contoso.com");

    Navigation.NavigateToLogin(accessTokenResult.InteractiveRequestUrl,
        accessTokenResult.InteractionOptions);
}
```

The preceding example assumes:

- The presence of an `@using`/`using` statement for API in the Microsoft.AspNetCore.Components.WebAssembly.Authentication namespace.
- IAccessTokenProvider injected as `TokenProvider`.

For more information, see the following resources:

- InteractiveRequestOptions
- Popup request parameter list

Logout with a custom return URL

The following example logs out the user and returns the user to the `/goodbye` endpoint:

C#

```
Navigation.NavigateToLogout("authentication/logout", "goodbye");
```

Obtain the login path from authentication options

Obtain the configured login path from RemoteAuthenticationOptions:

C#

```
var loginPath =

RemoteAuthOptions.Get(Options.DefaultName).AuthenticationPaths.LogInPath;
```

The preceding example assumes:

- The presence of an `@using`/`using` statement for API in the following namespaces:
  - Microsoft.AspNetCore.Components.WebAssembly.Authentication
  - Microsoft.Extensions.Options
- `IOptionsSnapshot<RemoteAuthenticationOptions<ApiAuthorizationProviderOptions>>` injected as `RemoteAuthOptions`.

Custom `AuthorizationMessageHandler` class

This guidance in this section is recommended for client apps that make outgoing requests to URIs that aren't within the app's base URI.

In the following example, a custom class extends AuthorizationMessageHandler for use as the DelegatingHandler for an HttpClient. ConfigureHandler configures this handler to authorize outbound HTTP requests using an access token. The access token is only attached if at least one of the authorized URLs is a base of the request URI (HttpRequestMessage.RequestUri).

C#

```
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class CustomAuthorizationMessageHandler : AuthorizationMessageHandler
{
    public CustomAuthorizationMessageHandler(IAccessTokenProvider provider,
        NavigationManager navigation)
        : base(provider, navigation)
    {
```

```
        ConfigureHandler(
            authorizedUrls: [ "https://api.contoso.com/v1.0" ],
            scopes: [ "example.read", "example.write" ]);
    }
}
```

In the preceding code, the scopes `example.read` and `example.write` are generic examples not meant to reflect valid scopes for any particular provider.

In the `Program` file, `CustomAuthorizationMessageHandler` is registered as a transient service and is configured as the DelegatingHandler for outgoing HttpResponseMessage instances made by a named HttpClient.

In the following example, HttpClientFactoryServiceCollectionExtensions.AddHttpClient is an extension in Microsoft.Extensions.Http. Add the package to an app that doesn't already reference it.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

C#

```
builder.Services.AddTransient<CustomAuthorizationMessageHandler>();

builder.Services.AddHttpClient("WebAPI",
        client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler<CustomAuthorizationMessageHandler>();
```

> **Note**
>
> In the preceding example, the `CustomAuthorizationMessageHandler` DelegatingHandler is registered as a transient service for AddHttpMessageHandler. Transient registration is recommended for IHttpClientFactory, which manages its own DI scopes. For more information, see the following resources:
>
> - Utility base component classes to manage a DI scope
> - Detect client-side transient disposables

The configured HttpClient is used to make authorized requests using the try-catch pattern. Where the client is created with CreateClient (Microsoft.Extensions.Http package), the HttpClient is supplied instances that include access tokens when making requests to the server API. If the request URI is a relative URI, as it is in the following example (`ExampleAPIMethod`), it's combined with the BaseAddress when the client app makes the request:

razor

```
@inject IHttpClientFactory ClientFactory

...

@code {
    protected override async Task OnInitializedAsync()
    {
        try
        {
            var client = ClientFactory.CreateClient("WebAPI");

            var examples =
                await client.GetFromJsonAsync<ExampleType[]>
("ExampleAPIMethod");

            ...
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
}
```

Configure `AuthorizationMessageHandler`

AuthorizationMessageHandler can be configured with authorized URLs, scopes, and a return URL using the ConfigureHandler method. ConfigureHandler configures the handler to authorize outbound HTTP requests using an access token. The access token is only attached if at least one of the authorized URLs is a base of the request URI (HttpRequestMessage.RequestUri). If the request URI is a relative URI, it's combined with the BaseAddress.

In the following example, AuthorizationMessageHandler configures an HttpClient in the `Program` file:

C#

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddScoped(sp => new HttpClient(
    sp.GetRequiredService<AuthorizationMessageHandler>()
    .ConfigureHandler(
        authorizedUrls: [ "https://api.contoso.com/v1.0" ],
        scopes: [ "example.read", "example.write" ])
    .InnerHandler = new HttpClientHandler())
{
    BaseAddress = new Uri("https://api.contoso.com/v1.0")
});
```

In the preceding code, the scopes `example.read` and `example.write` are generic examples not meant to reflect valid scopes for any particular provider.

Typed `HttpClient`

A typed client can be defined that handles all of the HTTP and token acquisition concerns within a single class.

`WeatherForecastClient.cs`:

C#

```
using System.Net.Http.Json;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using static {ASSEMBLY NAME}.Data;

public class WeatherForecastClient(HttpClient http)
{
    private WeatherForecast[]? forecasts;

    public async Task<WeatherForecast[]> GetForecastAsync()
    {
        try
        {
            forecasts = await http.GetFromJsonAsync<WeatherForecast[]>(
                "WeatherForecast");
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }

        return forecasts ?? Array.Empty<WeatherForecast>();
```

```
    }
}
```

In the preceding example, the `WeatherForecast` type is a static class that holds weather forecast data. The `{ASSEMBLY NAME}` placeholder is the app's assembly name (for example, `using static BlazorSample.Data;`).

In the following example, HttpClientFactoryServiceCollectionExtensions.AddHttpClient is an extension in Microsoft.Extensions.Http. Add the package to an app that doesn't already reference it.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the `Program` file:

C#

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddHttpClient<WeatherForecastClient>(
        client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();
```

In a component that fetches weather data:

razor

```
@inject WeatherForecastClient Client

...

protected override async Task OnInitializedAsync()
{
    forecasts = await Client.GetForecastAsync();
}
```

Configure the `HttpClient` handler

The handler can be further configured with ConfigureHandler for outbound HTTP requests.

In the following example, HttpClientFactoryServiceCollectionExtensions.AddHttpClient is an extension in Microsoft.Extensions.Http. Add the package to an app that doesn't already reference it.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the `Program` file:

```
C#
```

```
builder.Services.AddHttpClient<WeatherForecastClient>(
        client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler(sp =>
sp.GetRequiredService<AuthorizationMessageHandler>()
    .ConfigureHandler(
        authorizedUrls: new [] { "https://api.contoso.com/v1.0" },
        scopes: new[] { "example.read", "example.write" }));
```

In the preceding code, the scopes `example.read` and `example.write` are generic examples not meant to reflect valid scopes for any particular provider.

Unauthenticated or unauthorized web API requests in an app with a secure default client

An app that ordinarily uses a secure default HttpClient can also make unauthenticated or unauthorized web API requests by configuring a named HttpClient.

In the following example, HttpClientFactoryServiceCollectionExtensions.AddHttpClient is an extension in Microsoft.Extensions.Http. Add the package to an app that doesn't already reference it.

> **Note**

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the `Program` file:

C#

```
builder.Services.AddHttpClient("WebAPI.NoAuthenticationClient",
    client => client.BaseAddress = new Uri("https://api.contoso.com/v1.0"));
```

The preceding registration is in addition to the existing secure default HttpClient registration.

A component creates the HttpClient from the IHttpClientFactory (Microsoft.Extensions.Http package) to make unauthenticated or unauthorized requests:

razor

```
@inject IHttpClientFactory ClientFactory

...

@code {
    protected override async Task OnInitializedAsync()
    {
        var client =
ClientFactory.CreateClient("WebAPI.NoAuthenticationClient");

        var examples = await client.GetFromJsonAsync<ExampleType[]>(
            "ExampleNoAuthentication");

        ...
    }
}
```

Note

The controller in the server API, `ExampleNoAuthenticationController` for the preceding example, isn't marked with the [Authorize] attribute.

The decision whether to use a secure client or an insecure client as the default HttpClient instance is up to the developer. One way to make this decision is to consider the number of authenticated versus unauthenticated endpoints that the app contacts. If

the majority of the app's requests are to secure API endpoints, use the authenticated HttpClient instance as the default. Otherwise, register the unauthenticated HttpClient instance as the default.

An alternative approach to using the IHttpClientFactory is to create a typed client for unauthenticated access to anonymous endpoints.

Request additional access tokens

Access tokens can be manually obtained by calling IAccessTokenProvider.RequestAccessToken. In the following example, an additional scope is required by an app for the default HttpClient. The Microsoft Authentication Library (MSAL) example configures the scope with `MsalProviderOptions`:

In the `Program` file:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...

    options.ProviderOptions.AdditionalScopesToConsent.Add("{CUSTOM SCOPE
1}");
    options.ProviderOptions.AdditionalScopesToConsent.Add("{CUSTOM SCOPE
2}");
}
```

The `{CUSTOM SCOPE 1}` and `{CUSTOM SCOPE 2}` placeholders in the preceding example are custom scopes.

> ⓘ Note
>
> AdditionalScopesToConsent isn't able to provision delegated user permissions for Microsoft Graph via the Microsoft Entra ID consent UI when a user first uses an app registered in Microsoft Azure. For more information, see Use Graph API with ASP.NET Core Blazor WebAssembly.

The IAccessTokenProvider.RequestAccessToken method provides an overload that allows an app to provision an access token with a given set of scopes.

In a Razor component:

razor

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IAccessTokenProvider TokenProvider

...

var tokenResult = await TokenProvider.RequestAccessToken(
    new AccessTokenRequestOptions
    {
        Scopes = new[] { "{CUSTOM SCOPE 1}", "{CUSTOM SCOPE 2}" }
    });

if (tokenResult.TryGetToken(out var token))
{
    ...
}
```

The `{CUSTOM SCOPE 1}` and `{CUSTOM SCOPE 2}` placeholders in the preceding example are custom scopes.

AccessTokenResult.TryGetToken returns:

- `true` with the `token` for use.
- `false` if the token isn't retrieved.

Cross-Origin Resource Sharing (CORS)

When sending credentials (authorization cookies/headers) on CORS requests, the `Authorization` header must be allowed by the CORS policy.

The following policy includes configuration for:

- Request origins (`http://localhost:5000`, `https://localhost:5001`).
- Any method (verb).
- `Content-Type` and `Authorization` headers. To allow a custom header (for example, `x-custom-header`), list the header when calling WithHeaders.
- Credentials set by client-side JavaScript code (`credentials` property set to `include`).

C#

```
app.UseCors(policy =>
    policy.WithOrigins("http://localhost:5000", "https://localhost:5001")
        .AllowAnyMethod()
        .WithHeaders(HeaderNames.ContentType, HeaderNames.Authorization,
```

```
            "x-custom-header")
        .AllowCredentials());
```

For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core and the sample app's HTTP Request Tester component (`Components/HTTPRequestTester.razor`).

Handle token request errors

When a single-page application (SPA) authenticates a user using OpenID Connect (OIDC), the authentication state is maintained locally within the SPA and in the Identity Provider (IP) in the form of a session cookie that's set as a result of the user providing their credentials.

The tokens that the IP emits for the user typically are valid for short periods of time, about one hour normally, so the client app must regularly fetch new tokens. Otherwise, the user would be logged-out after the granted tokens expire. In most cases, OIDC clients are able to provision new tokens without requiring the user to authenticate again thanks to the authentication state or "session" that is kept within the IP.

There are some cases in which the client can't get a token without user interaction, for example, when for some reason the user explicitly logs out from the IP. This scenario occurs if a user visits `https://login.microsoftonline.com` and logs out. In these scenarios, the app doesn't know immediately that the user has logged out. Any token that the client holds might no longer be valid. Also, the client isn't able to provision a new token without user interaction after the current token expires.

These scenarios aren't specific to token-based authentication. They are part of the nature of SPAs. An SPA using cookies also fails to call a server API if the authentication cookie is removed.

When an app performs API calls to protected resources, you must be aware of the following:

- To provision a new access token to call the API, the user might be required to authenticate again.
- Even if the client has a token that seems to be valid, the call to the server might fail because the token was revoked by the user.

When the app requests a token, there are two possible outcomes:

- The request succeeds, and the app has a valid token.

- The request fails, and the app must authenticate the user again to obtain a new token.

When a token request fails, you need to decide whether you want to save any current state before you perform a redirection. Several approaches exist to store state with increasing levels of complexity:

- Store the current page state in session storage. During the OnInitializedAsync lifecycle method (OnInitializedAsync), check if state can be restored before continuing.
- Add a query string parameter and use that as a way to signal the app that it needs to re-hydrate the previously saved state.
- Add a query string parameter with a unique identifier to store data in session storage without risking collisions with other items.

Save app state before an authentication operation with session storage

The following example shows how to:

- Preserve state before redirecting to the login page.
- Recover the previous state after authentication using a query string parameter.

razor

```
...
@using System.Text.Json
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IAccessTokenProvider TokenProvider
@inject IJSRuntime JS
@inject NavigationManager Navigation

<EditForm Model="User" OnSubmit="OnSaveAsync">
    <label>
        First Name:
        <InputText @bind-Value="User!.Name" />
    </label>
    <label>
        Last Name:
        <InputText @bind-Value="User!.LastName" />
    </label>
    <button type="submit">Save User</button>
</EditForm>

@code {
    public Profile User { get; set; } = new Profile();
```

```
    protected override async Task OnInitializedAsync()
    {
        var currentQuery = new Uri(Navigation.Uri).Query;

        if (currentQuery.Contains("state=resumeSavingProfile"))
        {
            var user = await JS.InvokeAsync<string>
("sessionStorage.getItem",
                "resumeSavingProfile");

            if (!string.IsNullOrEmpty(user))
            {
                User = JsonSerializer.Deserialize<Profile>(user);
            }
        }
    }

    public async Task OnSaveAsync()
    {
        var http = new HttpClient();
        http.BaseAddress = new Uri(Navigation.BaseUri);

        var resumeUri = Navigation.Uri + $"?state=resumeSavingProfile";

        var tokenResult = await TokenProvider.RequestAccessToken(
            new AccessTokenRequestOptions
            {
                ReturnUrl = resumeUri
            });

        if (tokenResult.TryGetToken(out var token))
        {
            http.DefaultRequestHeaders.Add("Authorization",
                $"Bearer {token.Value}");
            await http.PostAsJsonAsync("Save", User);
        }
        else
        {
            await JS.InvokeVoidAsync("sessionStorage.setItem",
                "resumeSavingProfile", JsonSerializer.Serialize(User));
            Navigation.NavigateTo(tokenResult.InteractiveRequestUrl);
        }
    }

    public class Profile
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
    }
}
```

Save app state before an authentication operation with session storage and a state container

During an authentication operation, there are cases where you want to save the app state before the browser is redirected to the IP. This can be the case when you're using a state container and want to restore the state after the authentication succeeds. You can use a custom authentication state object to preserve app-specific state or a reference to it and restore that state after the authentication operation successfully completes. The following example demonstrates the approach.

A state container class is created in the app with properties to hold the app's state values. In the following example, the container is used to maintain the counter value of the default Blazor project template's `Counter` component (`Counter.razor`). Methods for serializing and deserializing the container are based on System.Text.Json.

C#

```
using System.Text.Json;

public class StateContainer
{
    public int CounterValue { get; set; }

    public string GetStateForLocalStorage() =>
JsonSerializer.Serialize(this);

    public void SetStateFromLocalStorage(string locallyStoredState)
    {
        var deserializedState =
            JsonSerializer.Deserialize<StateContainer>(locallyStoredState);

        CounterValue = deserializedState.CounterValue;
    }
}
```

The `Counter` component uses the state container to maintain the `currentCount` value outside of the component:

razor

```
@page "/counter"
@inject StateContainer State

<h1>Counter</h1>
```

```
<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    protected override void OnInitialized()
    {
        if (State.CounterValue > 0)
        {
            currentCount = State.CounterValue;
        }
    }

    private void IncrementCount()
    {
        currentCount++;
        State.CounterValue = currentCount;
    }
}
```

Create an `ApplicationAuthenticationState` from RemoteAuthenticationState. Provide an `Id` property, which serves as an identifier for the locally-stored state.

`ApplicationAuthenticationState.cs`:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class ApplicationAuthenticationState : RemoteAuthenticationState
{
    public string? Id { get; set; }
}
```

The `Authentication` component (`Authentication.razor`) saves and restores the app's state using local session storage with the `StateContainer` serialization and deserialization methods, `GetStateForLocalStorage` and `SetStateFromLocalStorage`:

razor

```
@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IJSRuntime JS
@inject StateContainer State

<RemoteAuthenticatorViewCore Action="Action"

TAuthenticationState="ApplicationAuthenticationState"
```

```
                            AuthenticationState="AuthenticationState"
                            OnLogInSucceeded="RestoreState"
                            OnLogOutSucceeded="RestoreState" />

@code {
    [Parameter]
    public string? Action { get; set; }

    public ApplicationAuthenticationState AuthenticationState { get; set; }
=
        new ApplicationAuthenticationState();

    protected override async Task OnInitializedAsync()
    {
        if
(RemoteAuthenticationActions.IsAction(RemoteAuthenticationActions.LogIn,
            Action) ||

RemoteAuthenticationActions.IsAction(RemoteAuthenticationActions.LogOut,
            Action))
        {
            AuthenticationState.Id = Guid.NewGuid().ToString();

            await JS.InvokeVoidAsync("sessionStorage.setItem",
                AuthenticationState.Id, State.GetStateForLocalStorage());
        }
    }

    private async Task RestoreState(ApplicationAuthenticationState state)
    {
        if (state.Id != null)
        {
            var locallyStoredState = await JS.InvokeAsync<string>(
                "sessionStorage.getItem", state.Id);

            if (locallyStoredState != null)
            {
                State.SetStateFromLocalStorage(locallyStoredState);
                await JS.InvokeVoidAsync("sessionStorage.removeItem",
state.Id);
            }
        }
    }
}
```

This example uses Microsoft Entra (ME-ID) for authentication. In the `Program` file:

- The `ApplicationAuthenticationState` is configured as the Microsoft Authentication Library (MSAL) `RemoteAuthenticationState` type.
- The state container is registered in the service container.

C#

```
builder.Services.AddMsalAuthentication<ApplicationAuthenticationState>
(options =>
{
    builder.Configuration.Bind("AzureAd",
options.ProviderOptions.Authentication);
});

builder.Services.AddSingleton<StateContainer>();
```

Customize app routes

The Microsoft.AspNetCore.Components.WebAssembly.Authentication library uses the routes shown in the following table for representing different authentication states.

Expand table

Route	Purpose
`authentication/login`	Triggers a sign-in operation.
`authentication/login-callback`	Handles the result of any sign-in operation.
`authentication/login-failed`	Displays error messages when the sign-in operation fails for some reason.
`authentication/logout`	Triggers a sign-out operation.
`authentication/logout-callback`	Handles the result of a sign-out operation.
`authentication/logout-failed`	Displays error messages when the sign-out operation fails for some reason.
`authentication/logged-out`	Indicates that the user has successfully logout.
`authentication/profile`	Triggers an operation to edit the user profile.
`authentication/register`	Triggers an operation to register a new user.

The routes shown in the preceding table are configurable via RemoteAuthenticationOptions<TRemoteAuthenticationProviderOptions>.AuthenticationPaths. When setting options to provide custom routes, confirm that the app has a route that handles each path.

In the following example, all of the paths are prefixed with `/security`.

`Authentication` component (`Authentication.razor`):

razor

```razor
@page "/security/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code{
    [Parameter]
    public string? Action { get; set; }
}
```

In the `Program` file:

C#

```
builder.Services.AddApiAuthorization(options => {
    options.AuthenticationPaths.LogInPath = "security/login";
    options.AuthenticationPaths.LogInCallbackPath = "security/login-
callback";
    options.AuthenticationPaths.LogInFailedPath = "security/login-failed";
    options.AuthenticationPaths.LogOutPath = "security/logout";
    options.AuthenticationPaths.LogOutCallbackPath = "security/logout-
callback";
    options.AuthenticationPaths.LogOutFailedPath = "security/logout-failed";
    options.AuthenticationPaths.LogOutSucceededPath = "security/logged-out";
    options.AuthenticationPaths.ProfilePath = "security/profile";
    options.AuthenticationPaths.RegisterPath = "security/register";
});
```

If the requirement calls for completely different paths, set the routes as described previously and render the RemoteAuthenticatorView with an explicit action parameter:

razor

```razor
@page "/register"

<RemoteAuthenticatorView Action="RemoteAuthenticationActions.Register" />
```

You're allowed to break the UI into different pages if you choose to do so.

Customize the authentication user interface

RemoteAuthenticatorView includes a default set of UI fragments for each authentication state. Each state can be customized by passing in a custom RenderFragment. To

customize the displayed text during the initial login process, can change the RemoteAuthenticatorView as follows.

`Authentication` component (`Authentication.razor`):

```razor
@page "/security/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action">
    <LoggingIn>
        You are about to be redirected to https://login.microsoftonline.com.
    </LoggingIn>
</RemoteAuthenticatorView>

@code{
    [Parameter]
    public string? Action { get; set; }
}
```

The RemoteAuthenticatorView has one fragment that can be used per authentication route shown in the following table.

Expand table

Route	Fragment
`authentication/login`	`<LoggingIn>`
`authentication/login-callback`	`<CompletingLoggingIn>`
`authentication/login-failed`	`<LogInFailed>`
`authentication/logout`	`<LogOut>`
`authentication/logout-callback`	`<CompletingLogOut>`
`authentication/logout-failed`	`<LogOutFailed>`
`authentication/logged-out`	`<LogOutSucceeded>`
`authentication/profile`	`<UserProfile>`
`authentication/register`	`<Registering>`

Customize the user

Users bound to the app can be customized.

Customize the user with a payload claim

In the following example, the app's authenticated users receive an `amr` claim for each of the user's authentication methods. The `amr` claim identifies how the subject of the token was authenticated in Microsoft identity platform v1.0 payload claims. The example uses a custom user account class based on RemoteUserAccount.

Create a class that extends the RemoteUserAccount class. The following example sets the `AuthenticationMethod` property to the user's array of `amr` JSON property values. `AuthenticationMethod` is populated automatically by the framework when the user is authenticated.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class CustomUserAccount : RemoteUserAccount
{
    [JsonPropertyName("amr")]
    public string[]? AuthenticationMethod { get; set; }
}
```

Create a factory that extends AccountClaimsPrincipalFactory<TAccount> to create claims from the user's authentication methods stored in `CustomUserAccount.AuthenticationMethod`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;

public class CustomAccountFactory(NavigationManager navigation,
    IAccessTokenProviderAccessor accessor)
    : AccountClaimsPrincipalFactory<CustomUserAccount>(accessor)
{
    public override async ValueTask<ClaimsPrincipal> CreateUserAsync(
        CustomUserAccount account, RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity != null &&
initialUser.Identity.IsAuthenticated)
```

```
        {
            var userIdentity = (ClaimsIdentity)initialUser.Identity;

            if (account.AuthenticationMethod is not null)
            {
                foreach (var value in account.AuthenticationMethod)
                {
                    userIdentity.AddClaim(new Claim("amr", value));
                }
            }
        }

        return initialUser;
    }
}
```

Register the `CustomAccountFactory` for the authentication provider in use. Any of the following registrations are valid:

- AddOidcAuthentication:

  C#

  ```
  using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

  ...

  builder.Services.AddOidcAuthentication<RemoteAuthenticationState,
      CustomUserAccount>(options =>
      {
          ...
      })
      .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
          CustomUserAccount, CustomAccountFactory>();
  ```

- AddMsalAuthentication:

  C#

  ```
  using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

  ...

  builder.Services.AddMsalAuthentication<RemoteAuthenticationState,
      CustomUserAccount>(options =>
      {
          ...
      })
      .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
          CustomUserAccount, CustomAccountFactory>();
  ```

- AddApiAuthorization:

```
C#

using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddApiAuthorization<RemoteAuthenticationState,
    CustomUserAccount>(options =>
    {
        ...
    })
    .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
        CustomUserAccount, CustomAccountFactory>();
```

ME-ID security groups and roles with a custom user account class

For an additional example that works with ME-ID security groups and ME-ID Administrator Roles and a custom user account class, see ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles.

Authenticate users to only call protected third party APIs

Authenticate the user with a client-side OAuth flow against the third-party API provider:

```
C#

builder.services.AddOidcAuthentication(options => { ... });
```

In this scenario:

- The server hosting the app doesn't play a role.
- APIs on the server can't be protected.
- The app can only call protected third-party APIs.

Authenticate users with a third-party provider and call protected APIs on the host server and the third party

Configure Identity with a third-party login provider. Obtain the tokens required for third-party API access and store them.

When a user logs in, Identity collects access and refresh tokens as part of the authentication process. At that point, there are a couple of approaches available for making API calls to third-party APIs.

Use a server access token to retrieve the third-party access token

Use the access token generated on the server to retrieve the third-party access token from a server API endpoint. From there, use the third-party access token to call third-party API resources directly from Identity on the client.

We don't recommend this approach. This approach requires treating the third-party access token as if it were generated for a public client. In OAuth terms, the public app doesn't have a client secret because it can't be trusted to store secrets safely, and the access token is produced for a confidential client. A confidential client is a client that has a client secret and is assumed to be able to safely store secrets.

- The third-party access token might be granted additional scopes to perform sensitive operations based on the fact that the third-party emitted the token for a more trusted client.
- Similarly, refresh tokens shouldn't be issued to a client that isn't trusted, as doing so gives the client unlimited access unless other restrictions are put into place.

Make API calls from the client to the server API in order to call third-party APIs

Make an API call from the client to the server API. From the server, retrieve the access token for the third-party API resource and issue whatever call is necessary.

We recommend this approach. While this approach requires an extra network hop through the server to call a third-party API, it ultimately results in a safer experience:

- The server can store refresh tokens and ensure that the app doesn't lose access to third-party resources.
- The app can't leak access tokens from the server that might contain more sensitive permissions.

Use OpenID Connect (OIDC) v2.0 endpoints

The authentication library and Blazor project templates use OpenID Connect (OIDC) v1.0 endpoints. To use a v2.0 endpoint, configure the JWT Bearer JwtBearerOptions.Authority

option. In the following example, ME-ID is configured for v2.0 by appending a `v2.0` segment to the Authority property:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;

...

builder.Services.Configure<JwtBearerOptions>(
    JwtBearerDefaults.AuthenticationScheme,
    options =>
    {
        options.Authority += "/v2.0";
    });
```

Alternatively, the setting can be made in the app settings (`appsettings.json`) file:

JSON

```
{
  "Local": {
    "Authority": "https://login.microsoftonline.com/common/oauth2/v2.0/",
    ...
  }
}
```

If tacking on a segment to the authority isn't appropriate for the app's OIDC provider, such as with non-ME-ID providers, set the Authority property directly. Either set the property in JwtBearerOptions or in the app settings file (`appsettings.json`) with the `Authority` key.

The list of claims in the ID token changes for v2.0 endpoints. Microsoft documentation on the changes has been retired, but guidance on the claims in an ID token is available in the ID token claims reference.

Configure and use gRPC in components

To configure a Blazor WebAssembly app to use the ASP.NET Core gRPC framework:

- Enable gRPC-Web on the server. For more information, see gRPC-Web in ASP.NET Core gRPC apps.
- Register gRPC services for the app's message handler. The following example configures the app's authorization message handler to use the GreeterClient service from the gRPC tutorial (the `Program` file).

> **Note**
>
> Prerendering is enabled by default in Blazor Web Apps, so you must account for the component rendering first from the server and then from the client. Any prerendered state should flow to the client so that it can be reused. For more information, see Prerender ASP.NET Core Razor components.

C#

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Grpc.Net.Client;
using Grpc.Net.Client.Web;

...

builder.Services.AddScoped(sp =>
{
    var baseAddressMessageHandler =
        sp.GetRequiredService<BaseAddressAuthorizationMessageHandler>();
    baseAddressMessageHandler.InnerHandler = new HttpClientHandler();
    var grpcWebHandler =
        new GrpcWebHandler(GrpcWebMode.GrpcWeb, baseAddressMessageHandler);
    var channel =
GrpcChannel.ForAddress(builder.HostEnvironment.BaseAddress,
        new GrpcChannelOptions { HttpHandler = grpcWebHandler });

    return new Greeter.GreeterClient(channel);
});
```

A component in the client app can make gRPC calls using the gRPC client (`Grpc.razor`):

razor

```
@page "/grpc"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
@inject Greeter.GreeterClient GreeterClient

<h1>Invoke gRPC service</h1>

<p>
    <input @bind="name" placeholder="Type your name" />
    <button @onclick="GetGreeting" class="btn btn-primary">Call gRPC
service</button>
</p>

Server response: <strong>@serverResponse</strong>
```

```
@code {
    private string name = "Bert";
    private string? serverResponse;

    private async Task GetGreeting()
    {
        try
        {
            var request = new HelloRequest { Name = name };
            var reply = await GreeterClient.SayHelloAsync(request);
            serverResponse = reply.Message;
        }
        catch (Grpc.Core.RpcException ex)
            when (ex.Status.DebugException is
                AccessTokenNotAvailableException tokenEx)
        {
            tokenEx.Redirect();
        }
    }
}
```

To use the `Status.DebugException` property, use Grpc.Net.Client version 2.30.0 or later.

For more information, see gRPC-Web in ASP.NET Core gRPC apps.

Replace the `AuthenticationService` implementation

The following subsections explain how to replace:

- Any JavaScript `AuthenticationService` implementation.
- The Microsoft Authentication Library for JavaScript (`MSAL.js`).

Replace any JavaScript `AuthenticationService` implementation

Create a JavaScript library to handle your custom authentication details.

> ⚠ **Warning**
>
> The guidance in this section is an implementation detail of the default RemoteAuthenticationService<TRemoteAuthenticationState,TAccount,TProviderOptions>. The TypeScript code in this section applies specifically to ASP.NET Core

in .NET 7 and is subject to change without notice in upcoming releases of ASP.NET Core.

TypeScript

```
// .NET makes calls to an AuthenticationService object in the Window.
declare global {
  interface Window { AuthenticationService: AuthenticationService }
}

export interface AuthenticationService {
  // Init is called to initialize the AuthenticationService.
  public static init(settings: UserManagerSettings &
AuthorizeServiceSettings, logger: any) : Promise<void>;

  // Gets the currently authenticated user.
  public static getUser() : Promise<{[key: string] : string }>;

  // Tries to get an access token silently.
  public static getAccessToken(options: AccessTokenRequestOptions) :
Promise<AccessTokenResult>;

  // Tries to sign in the user or get an access token interactively.
  public static signIn(context: AuthenticationContext) :
Promise<AuthenticationResult>;

  // Handles the sign-in process when a redirect is used.
  public static async completeSignIn(url: string) :
Promise<AuthenticationResult>;

  // Signs the user out.
  public static signOut(context: AuthenticationContext) :
Promise<AuthenticationResult>;

  // Handles the signout callback when a redirect is used.
  public static async completeSignOut(url: string) :
Promise<AuthenticationResult>;
}

// The rest of these interfaces match their C# definitions.

export interface AccessTokenRequestOptions {
  scopes: string[];
  returnUrl: string;
}

export interface AccessTokenResult {
  status: AccessTokenResultStatus;
  token?: AccessToken;
}

export interface AccessToken {
  value: string;
```

```
  expires: Date;
  grantedScopes: string[];
}

export enum AccessTokenResultStatus {
  Success = 'Success',
  RequiresRedirect = 'RequiresRedirect'
}

export enum AuthenticationResultStatus {
  Redirect = 'Redirect',
  Success = 'Success',
  Failure = 'Failure',
  OperationCompleted = 'OperationCompleted'
};

export interface AuthenticationResult {
  status: AuthenticationResultStatus;
  state?: unknown;
  message?: string;
}

export interface AuthenticationContext {
  state?: unknown;
  interactiveRequest: InteractiveAuthenticationRequest;
}

export interface InteractiveAuthenticationRequest {
  scopes?: string[];
  additionalRequestParameters?: { [key: string]: any };
};
```

You can import the library by removing the original `<script>` tag and adding a `<script>` tag that loads the custom library. The following example demonstrates replacing the default `<script>` tag with one that loads a library named `CustomAuthenticationService.js` from the `wwwroot/js` folder.

In `wwwroot/index.html` before the Blazor script (`_framework/blazor.webassembly.js`) inside the closing `</body>` tag:

diff

```
- <script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationServic
e.js"></script>
+ <script src="js/CustomAuthenticationService.js"></script>
```

For more information, see AuthenticationService.ts in the dotnet/aspnetcore GitHub repository.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Replace the Microsoft Authentication Library for JavaScript (`MSAL.js`)

If an app requires a custom version of the Microsoft Authentication Library for JavaScript (MSAL.js), perform the following steps:

1. Confirm the system has the latest developer .NET SDK or obtain and install the latest developer SDK from .NET Core SDK: Installers and Binaries. Configuration of internal NuGet feeds isn't required for this scenario.
2. Set up the `dotnet/aspnetcore` GitHub repository for development following the documentation at Build ASP.NET Core from Source. Fork and clone or download a ZIP archive of the dotnet/aspnetcore GitHub repository.
3. Open the `src/Components/WebAssembly/Authentication.Msal/src/Interop/package.json` file and set the desired version of `@azure/msal-browser`. For a list of released versions, visit the @azure/msal-browser npm website and select the **Versions** tab.
4. Build the `Authentication.Msal` project in the `src/Components/WebAssembly/Authentication.Msal/src` folder with the `yarn build` command in a command shell.
5. If the app uses compressed assets (Brotli/Gzip), compress the `Interop/dist/Release/AuthenticationService.js` file.
6. Copy the `AuthenticationService.js` file and compressed versions (`.br`/`.gz`) of the file, if produced, from the `Interop/dist/Release` folder into the app's `publish/wwwroot/_content/Microsoft.Authentication.WebAssembly.Msal` folder in the app's published assets.

Pass custom provider options

Define a class for passing the data to the underlying JavaScript library.

> **ⓘ Important**
>
> The class's structure must match what the library expects when the JSON is serialized with System.Text.Json.

The following example demonstrates a `ProviderOptions` class with JsonPropertyName attributes matching a hypothetical custom provider library's expectations:

C#

```
public class ProviderOptions
{
    public string? Authority { get; set; }
    public string? MetadataUrl { get; set; }

    [JsonPropertyName("client_id")]
    public string? ClientId { get; set; }

    public IList<string> DefaultScopes { get; set; } = [ "openid", "profile"
];

    [JsonPropertyName("redirect_uri")]
    public string? RedirectUri { get; set; }

    [JsonPropertyName("post_logout_redirect_uri")]
    public string? PostLogoutRedirectUri { get; set; }

    [JsonPropertyName("response_type")]
    public string? ResponseType { get; set; }

    [JsonPropertyName("response_mode")]
    public string? ResponseMode { get; set; }
}
```

Register the provider options within the DI system and configure the appropriate values:

C#

```
builder.Services.AddRemoteAuthentication<RemoteAuthenticationState,
RemoteUserAccount,
    ProviderOptions>(options => {
        options.ProviderOptions.Authority = "...";
        options.ProviderOptions.MetadataUrl = "...";
        options.ProviderOptions.ClientId = "...";
        options.ProviderOptions.DefaultScopes = [ "openid", "profile",
"myApi" ];
        options.ProviderOptions.RedirectUri =
"https://localhost:5001/authentication/login-callback";
        options.ProviderOptions.PostLogoutRedirectUri =
```

```
"https://localhost:5001/authentication/logout-callback";
        options.ProviderOptions.ResponseType = "...";
        options.ProviderOptions.ResponseMode = "...";
    });
```

The preceding example sets redirect URIs with regular string literals. The following alternatives are available:

- TryCreate using IWebAssemblyHostEnvironment.BaseAddress:

  C#

  ```
  Uri.TryCreate(
      $"{builder.HostEnvironment.BaseAddress}authentication/login-
  callback",
      UriKind.Absolute, out var redirectUri);
  options.RedirectUri = redirectUri;
  ```

- Host builder configuration:

  C#

  ```
  options.RedirectUri = builder.Configuration["RedirectUri"];
  ```

  `wwwroot/appsettings.json`:

  JSON

  ```
  {
    "RedirectUri": "https://localhost:5001/authentication/login-callback"
  }
  ```

Additional resources

- Use Graph API with ASP.NET Core Blazor WebAssembly
- Cookie-based request credentials (*Call web API* article)
- HttpClient and HttpRequestMessage with Fetch API request options (*Call web API* article)

Microsoft Entra (ME-ID) groups, Administrator Roles, and App Roles

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure Blazor WebAssembly to use Microsoft Entra ID (ME-ID) groups and roles.

ME-ID provides several authorization approaches that can be combined with ASP.NET Core Identity:

- Groups
  - Security
  - Microsoft 365
  - Distribution
- Roles
  - ME-ID built-in Administrator Roles
  - App Roles

The guidance in this article applies to the Blazor WebAssembly ME-ID deployment scenarios described in the following articles:

- Standalone with Microsoft Accounts
- Standalone with ME-ID

The examples in this article take advantage of new .NET/C# features. When using the examples with .NET 7 or earlier, minor modifications are required. However, the text and code examples that pertain to interacting with ME-ID and Microsoft Graph are the same for all versions of ASP.NET Core.

Sample app

Access the sample app, named `BlazorWebAssemblyEntraGroupsAndRoles`, through the latest version folder from the repository's root with the following link. The sample is provided for .NET 8 or later. See the sample app's `README` file for steps on how to run the app.

The sample app includes a `UserClaims` component for displaying a user's claims. The `UserData` component displays the user's basic account properties.

View or download sample code (how to download)

Prerequisite

The guidance in this article implements the Microsoft Graph API per the *Graph SDK* guidance in Use Graph API with ASP.NET Core Blazor WebAssembly. Follow the *Graph SDK* implementation guidance to configure the app and test it to confirm that the app can obtain Graph API data for a test user account. Additionally, see the Graph API article's security article cross-links to review Microsoft Graph security concepts.

When testing with the Graph SDK locally, we recommend using a new in-private/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID.

ME-ID app registration online tools

This article refers to the Azure portal throughout when prompting you to configure the app's ME-ID app registration, but the Microsoft Entra Admin Center is also a viable option for managing ME-ID app registrations. Either interface can be used, but the guidance in this article specifically covers gestures for the Azure portal.

Scopes

Permissions and *scopes* mean the same thing and are used interchangeably in security documentation and the Azure portal. Unless the text is referring to the Azure portal, this article uses *scope/scopes* when referring to Graph permissions.

Scopes are case insensitive, so `User.Read` is the same as `user.read`. Feel free to use either format, but we recommend a consistent choice across application code.

To permit Microsoft Graph API calls for user profile, role assignment, and group membership data, the app is configured with the ***delegated*** `User.Read` scope (`https://graph.microsoft.com/User.Read`) in the Azure portal because access to read user data is determined by the scopes granted (delegated) to individual users. This scope is required in addition to the scopes required in ME-ID deployment scenarios described by the articles listed earlier (*Standalone with Microsoft Accounts* or *Standalone with ME-ID*).

Additional required scopes include:

- ***Delegated*** RoleManagement.Read.Directory scope (`https://graph.microsoft.com/RoleManagement.Read.Directory`): Allows the app to read the role-based access control (RBAC) settings for your company's directory, on behalf of the signed-in user. This includes reading directory role templates, directory roles, and memberships. Directory role memberships are used to create `directoryRole` claims in the app for ME-ID built-in Administrator Roles. Admin consent is required.
- ***Delegated*** AdministrativeUnit.Read.All scope (`https://graph.microsoft.com/AdministrativeUnit.Read.All`): Allows the app to read administrative units and administrative unit membership on behalf of the signed-in user. These memberships are used to create `administrativeUnit` claims in the app. Admin consent is required.

For more information, see Overview of permissions and consent in the Microsoft identity platform and Overview of Microsoft Graph permissions.

Custom user account

Assign users to ME-ID security groups and ME-ID Administrator Roles in the Azure portal.

The examples in this article:

- Assume that a user is assigned to the ME-ID *Billing Administrator* role in the Azure portal ME-ID tenant for authorization to access server API data.
- Use authorization policies to control access within the app.

Extend RemoteUserAccount to include properties for:

- `Roles`: ME-ID App Roles array (covered in the App Roles section)
- `Oid`: Immutable object identifier claim (oid) (uniquely identifies a user within and across tenants)

`CustomUserAccount.cs`:

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

namespace BlazorWebAssemblyEntraGroupsAndRoles;
```

```
public class CustomUserAccount : RemoteUserAccount
{
    [JsonPropertyName("roles")]
    public List<string>? Roles { get; set; }

    [JsonPropertyName("oid")]
    public string? Oid { get; set; }
}
```

Add a package reference to app for Microsoft.Graph.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Add the Graph SDK utility classes and configuration in the *Graph SDK* guidance of the Use Graph API with ASP.NET Core Blazor WebAssembly article. Specify the `User.Read`, `RoleManagement.Read.Directory`, and `AdministrativeUnit.Read.All` scopes for the access token as the article shows in its example `wwwroot/appsettings.json` file.

Add the following custom user account factory to the app. The custom user factory is used to establish:

- App Role claims (`role`) (covered in the App Roles section).
- Example user profile data claims for the user's mobile phone number (`mobilePhone`) and office location (`officeLocation`).
- ME-ID Administrator Role claims (`directoryRole`).
- ME-ID Administrative Unit claims (`administrativeUnit`).
- ME-ID Group claims (`directoryGroup`).
- An ILogger (`logger`) for convenience in case you wish to log information or errors.

`CustomAccountFactory.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;
using Microsoft.Graph;
```

```
using Microsoft.Kiota.Abstractions.Authentication;

namespace BlazorWebAssemblyEntraGroupsAndRoles;

public class CustomAccountFactory(IAccessTokenProviderAccessor accessor,
        IServiceProvider serviceProvider, ILogger<CustomAccountFactory>
logger,
        IConfiguration config)
    : AccountClaimsPrincipalFactory<CustomUserAccount>(accessor)
{
    private readonly ILogger<CustomAccountFactory> logger = logger;
    private readonly IServiceProvider serviceProvider = serviceProvider;
    private readonly string? baseUrl = string.Join("/",
        config.GetSection("MicrosoftGraph")["BaseUrl"],
        config.GetSection("MicrosoftGraph")["Version"]);

    public override async ValueTask<ClaimsPrincipal> CreateUserAsync(
        CustomUserAccount account,
        RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity is not null &&
            initialUser.Identity.IsAuthenticated)
        {
            var userIdentity = initialUser.Identity as ClaimsIdentity;

            if (userIdentity is not null && !string.IsNullOrEmpty(baseUrl)
&&
                account.Oid is not null)
            {
                account?.Roles?.ForEach((role) =>
                {
                    userIdentity.AddClaim(new Claim("role", role));
                });

                try
                {
                    var client = new GraphServiceClient(
                        new HttpClient(),
                        serviceProvider
                            .GetRequiredService<IAuthenticationProvider>(),
                        baseUrl);

                    var user = await client.Me.GetAsync();

                    if (user is not null)
                    {
                        userIdentity.AddClaim(new Claim("mobilephone",
                            user.MobilePhone ?? "(000) 000-0000"));
                        userIdentity.AddClaim(new Claim("officelocation",
                            user.OfficeLocation ?? "Not set"));
                    }

                    var memberOf = client.Users[account?.Oid].MemberOf;
```

```
                    var graphDirectoryRoles = await
memberOf.GraphDirectoryRole.GetAsync();

                    if (graphDirectoryRoles?.Value is not null)
                    {
                        foreach (var entry in graphDirectoryRoles.Value)
                        {
                            if (entry.RoleTemplateId is not null)
                            {
                                userIdentity.AddClaim(
                                    new Claim("directoryRole",
entry.RoleTemplateId));
                            }
                        }
                    }

                    var graphAdministrativeUnits = await
memberOf.GraphAdministrativeUnit.GetAsync();

                    if (graphAdministrativeUnits?.Value is not null)
                    {
                        foreach (var entry in
graphAdministrativeUnits.Value)
                        {
                            if (entry.Id is not null)
                            {
                                userIdentity.AddClaim(
                                    new Claim("administrativeUnit",
entry.Id));
                            }
                        }
                    }

                    var graphGroups = await memberOf.GraphGroup.GetAsync();

                    if (graphGroups?.Value is not null)
                    {
                        foreach (var entry in graphGroups.Value)
                        {
                            if (entry.Id is not null)
                            {
                                userIdentity.AddClaim(
                                    new Claim("directoryGroup", entry.Id));
                            }
                        }
                    }
                }
                catch (AccessTokenNotAvailableException exception)
                {
                    exception.Redirect();
                }
            }
        }
```

```
            return initialUser;
        }
    }
```

The preceding code:

- Doesn't include transitive memberships. If the app requires direct and transitive group membership claims, replace the `MemberOf` property (`IUserMemberOfCollectionWithReferencesRequestBuilder`) with `TransitiveMemberOf` (`IUserTransitiveMemberOfCollectionWithReferencesRequestBuilder`).
- Sets GUID values in `directoryRole` claims are ME-ID Administrator Role *Template IDs* (`Microsoft.Graph.Models.DirectoryRole.RoleTemplateId`). Template IDs are stable identifiers for creating user authorization policies in apps, which is covered later in this article. Don't use `entry.Id` for directory role claim values, as they aren't stable across tenants.

Next, configure the MSAL authentication to use the custom user account factory.

Confirm that the `Program` file uses the Microsoft.AspNetCore.Components.WebAssembly.Authentication namespace:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
```

Update the AddMsalAuthentication call to the following. Note that the Blazor framework's RemoteUserAccount is replaced by the app's `CustomUserAccount` for the MSAL authentication and account claims principal factory:

C#

```
builder.Services.AddMsalAuthentication<RemoteAuthenticationState,
    CustomUserAccount>(options =>
    {
        builder.Configuration.Bind("AzureAd",
            options.ProviderOptions.Authentication);
        options.UserOptions.RoleClaim = "role";
    })
    .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
CustomUserAccount,
        CustomAccountFactory>();
```

Confirm the presence of the *Graph SDK* code in the `Program` file described by the Use Graph API with ASP.NET Core Blazor WebAssembly article:

C#

```
var baseUrl =
    string.Join("/",
        builder.Configuration.GetSection("MicrosoftGraph")["BaseUrl"] ??
            "https://graph.microsoft.com",
        builder.Configuration.GetSection("MicrosoftGraph")["Version"] ??
            "v1.0");
var scopes = builder.Configuration.GetSection("MicrosoftGraph:Scopes")
    .Get<List<string>>() ?? [ "user.read" ];

builder.Services.AddGraphClient(baseUrl, scopes);
```

ⓘ Important

Confirm in the app's registration in the Azure portal that the following permissions are granted:

- `User.Read`
- `RoleManagement.Read.Directory` (Requires admin consent)
- `AdministrativeUnit.Read.All` (Requires admin consent)

Confirm that the `wwwroot/appsettings.json` configuration is correct per the *Graph SDK* guidance.

`wwwroot/appsettings.json`:

JSON

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  },
  "MicrosoftGraph": {
    "BaseUrl": "https://graph.microsoft.com",
    "Version": "v1.0",
    "Scopes": [
      "User.Read",
      "RoleManagement.Read.Directory",
      "AdministrativeUnit.Read.All"
    ]
  }
}
```

Provide values for the following placeholders from the app's ME-ID registration in the Azure portal:

- `{TENANT ID}`: The Directory (Tenant) Id GUID value.
- `{CLIENT ID}`: The Application (Client) Id GUID value.

Authorization configuration

Create a policy for each App Role (by role name), ME-ID built-in Administrator Role (by Role Template Id/GUID), or security group (by Object Id/GUID) in the `Program` file. The following example creates a policy for the ME-ID built-in *Billing Administrator* role:

C#

```
builder.Services.AddAuthorizationCore(options =>
{
    options.AddPolicy("BillingAdministrator", policy =>
        policy.RequireClaim("directoryRole",
            "b0f54661-2d74-4c50-afa3-1ec803f12efe"));
});
```

For the complete list of IDs (GUIDs) for ME-ID Administrator Roles, see Role template IDs in the ME-ID documentation. For an Azure security or O365 group ID (GUID), see the **Object Id** for the group in the Azure portal **Groups** pane of the app's registration. For more information on authorization policies, see Policy-based authorization in ASP.NET Core.

In the following examples, the app uses the preceding policy to authorize the user.

The AuthorizeView component works with the policy:

razor

```
<AuthorizeView Policy="BillingAdministrator">
    <Authorized>
        <p>
            The user is in the 'Billing Administrator' ME-ID Administrator
Role
            and can see this content.
        </p>
    </Authorized>
    <NotAuthorized>
        <p>
            The user is NOT in the 'Billing Administrator' role and sees
this
            content.
        </p>
```

```
    </NotAuthorized>
</AuthorizeView>
```

Access to an entire component can be based on the policy using an [Authorize] attribute directive (AuthorizeAttribute):

razor

```
@page "/"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize(Policy = "BillingAdministrator")]
```

If the user isn't authorized, they're redirected to the ME-ID sign-in page.

A policy check can also be performed in code with procedural logic.

`CheckPolicy.razor`:

razor

```
@page "/checkpolicy"
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<h1>Check Policy</h1>

<p>This component checks a policy in code.</p>

<button @onclick="CheckPolicy">Check 'BillingAdministrator' policy</button>

<p>Policy Message: @policyMessage</p>

@code {
    private string policyMessage = "Check hasn't been made yet.";

    [CascadingParameter]
    private Task<AuthenticationState> authenticationStateTask { get; set; }

    private async Task CheckPolicy()
    {
        var user = (await authenticationStateTask).User;

        if ((await AuthorizationService.AuthorizeAsync(user,
            "BillingAdministrator")).Succeeded)
        {
            policyMessage = "Yes! The 'BillingAdministrator' policy is
met.";
        }
        else
        {
            policyMessage = "No! 'BillingAdministrator' policy is NOT met.";
```

```
        }
    }
}
```

Using the preceding approaches, you can also create policy-based access for security groups, where the GUID used for the policy matches the

App Roles

To configure the app in the Azure portal to provide App Roles membership claims, see Add app roles to your application and receive them in the token in the ME-ID documentation.

The following example assumes that the app is configured with two roles, and the roles are assigned to a test user:

- `Admin`
- `Developer`

Although you can't assign roles to groups without an ME-ID Premium account, you can assign roles to users and receive role claims for users with a standard Azure account. The guidance in this section doesn't require an ME-ID Premium account.

Take either of the following approaches add app roles in ME-ID:

- When working with the ***default directory***, follow the guidance in Add app roles to your application and receive them in the token to create ME-ID roles.
- If you ***aren't working with the default directory***, edit the app's manifest in the Azure portal to establish the app's roles manually in the `appRoles` entry of the manifest file. The following is an example `appRoles` entry that creates `Admin` and `Developer` roles. These example roles are used later at the component level to implement access restrictions:

  > **ⓘ Important**
  >
  > The following approach is only recommended for apps that aren't registered in the Azure account's default directory. For apps registered in the default directory, see the preceding bullet of this list.

  JSON

```
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Administrators manage developers.",
    "displayName": "Admin",
    "id": "{ADMIN GUID}",
    "isEnabled": true,
    "lang": null,
    "origin": "Application",
    "value": "Admin"
  },
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Developers write code.",
    "displayName": "Developer",
    "id": "{DEVELOPER GUID}",
    "isEnabled": true,
    "lang": null,
    "origin": "Application",
    "value": "Developer"
  }
],
```

For the `{ADMIN GUID}` and `{DEVELOPER GUID}` placeholders in the preceding example, you can generate GUIDs with an online GUID generator (Google search result for "guid generator").

To assign a role to a user (or group if you have a Premium tier Azure account):

1. Navigate to **Enterprise applications** in the ME-ID area of the Azure portal.
2. Select the app. Select **Manage** > **Users and groups** from the sidebar.
3. Select the checkbox for one or more user accounts.
4. From the menu above the list of users, select **Edit assignment**.
5. For the **Select a role** entry, select **None selected**.
6. Choose a role from the list and use the **Select** button to select it.
7. Use the **Assign** button at the bottom of the screen to assign the role.

Multiple roles are assigned in the Azure portal by ***re-adding a user*** for each additional role assignment. Use the **Add user/group** button at the top of the list of users to re-add a user. Use the preceding steps to assign another role to the user. You can repeat this process as many times as needed to add additional roles to a user (or group).

The `CustomAccountFactory` shown in the Custom user account section is set up to act on a `role` claim with a JSON array value. Add and register the `CustomAccountFactory` in the app as shown in the Custom user account section. There's no need to provide code to remove the original `role` claim because it's automatically removed by the framework.

In the `Program` file, add or confirm the claim named "`role`" as the role claim for ClaimsPrincipal.IsInRole checks:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...

    options.UserOptions.RoleClaim = "role";
});
```

> ⓘ **Note**
>
> If you prefer to use the `directoryRoles` claim (ME-ID Administrator Roles), assign "`directoryRoles`" to the RemoteAuthenticationUserOptions.RoleClaim.

After you've completed the preceding steps to create and assign roles to users (or groups if you have a Premium tier Azure account) and implemented the `CustomAccountFactory` with the Graph SDK, as explained earlier in this article and in Use Graph API with ASP.NET Core Blazor WebAssembly, you should see an `role` claim for each assigned role that a signed-in user is assigned (or roles assigned to groups that they are members of). Run the app with a test user to confirm the claims are present as expected. When testing with the Graph SDK locally, we recommend using a new in-private/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID.

Component authorization approaches are functional at this point. Any of the authorization mechanisms in components of the app can use the `Admin` role to authorize the user:

- AuthorizeView component

  razor

  ```
  <AuthorizeView Roles="Admin">
  ```

- [Authorize] attribute directive (AuthorizeAttribute)

```razor
@attribute [Authorize(Roles = "Admin")]
```

- Procedural logic

```C#
var authState = await
AuthenticationStateProvider.GetAuthenticationStateAsync();
var user = authState.User;

if (user.IsInRole("Admin")) { ... }
```

Multiple role tests are supported:

- Require that the user be in **either** the `Admin` **or** `Developer` role with the `AuthorizeView` component:

```razor
<AuthorizeView Roles="Admin, Developer">
    ...
</AuthorizeView>
```

- Require that the user be in **both** the `Admin` **and** `Developer` roles with the `AuthorizeView` component:

```razor
<AuthorizeView Roles="Admin">
    <AuthorizeView Roles="Developer" Context="innerContext">
        ...
    </AuthorizeView>
</AuthorizeView>
```

  For more information on the `Context` for the inner AuthorizeView, see ASP.NET Core Blazor authentication and authorization.

- Require that the user be in **either** the `Admin` **or** `Developer` role with the `[Authorize]` attribute:

```razor
```

```
@attribute [Authorize(Roles = "Admin, Developer")]
```

- Require that the user be in **both** the `Admin` **and** `Developer` roles with the `[Authorize]` attribute:

```razor
@attribute [Authorize(Roles = "Admin")]
@attribute [Authorize(Roles = "Developer")]
```

- Require that the user be in **either** the `Admin` **or** `Developer` role with procedural code:

```razor
@code {
    private async Task DoSomething()
    {
        var authState = await AuthenticationStateProvider
            .GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.IsInRole("Admin") || user.IsInRole("Developer"))
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

- Require that the user be in **both** the `Admin` **and** `Developer` roles with procedural code by changing the conditional OR (||) to a conditional AND (&&) in the preceding example:

```C#
if (user.IsInRole("Admin") && user.IsInRole("Developer"))
```

Multiple role tests are supported:

- Require that the user be in **either** the `Admin` **or** `Developer` role with the `[Authorize]` attribute:

```
C#

[Authorize(Roles = "Admin, Developer")]
```

- Require that the user be in **both** the `Admin` **and** `Developer` roles with the `[Authorize]` attribute:

```
C#

[Authorize(Roles = "Admin")]
[Authorize(Roles = "Developer")]
```

- Require that the user be in **either** the `Admin` **or** `Developer` role with procedural code:

```
C#

static readonly string[] scopeRequiredByApi = new string[] {
"API.Access" };

...

[HttpGet]
public IEnumerable<ReturnType> Get()
{
    HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);

    if (User.IsInRole("Admin") || User.IsInRole("Developer"))
    {
        ...
    }
    else
    {
        ...
    }

    return ...
}
```

- Require that the user be in **both** the `Admin` **and** `Developer` roles with procedural code by changing the conditional OR (||) to a conditional AND (&&) in the preceding example:

```
C#

if (User.IsInRole("Admin") && User.IsInRole("Developer"))
```

Because .NET string comparisons are case-sensitive, matching role names is also case-sensitive. For example, `Admin` (uppercase `A`) is not treated as the same role as `admin` (lowercase `a`).

Pascal case is typically used for role names (for example, `BillingAdministrator`), but the use of Pascal case isn't a strict requirement. Different casing schemes, such as camel case, kebab case, and snake case, are permitted. Using spaces in role names is also unusual but permitted. For example, `billing administrator` is an unusual role name format in .NET apps but valid.

Additional resources

- Role template IDs (ME-ID documentation)
- groupMembershipClaims attribute (ME-ID documentation)
- Add app roles to your application and receive them in the token (ME-ID documentation)
- Application roles (Azure documentation)
- Claims-based authorization in ASP.NET Core
- Role-based authorization in ASP.NET Core
- ASP.NET Core Blazor authentication and authorization

Use Graph API with ASP.NET Core Blazor WebAssembly

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use Microsoft Graph in Blazor WebAssembly apps, which enables apps to access Microsoft Cloud resources.

Two approaches are covered:

- **Graph SDK**: The Microsoft Graph SDK simplifies building high-quality, efficient, and resilient apps that access Microsoft Graph. Select the **Graph SDK** button at the top of this article to adopt this approach.
- **Named HttpClient with Graph API**: A named HttpClient can issue Microsoft Graph API requests directly to Microsoft Graph. Select the **Named HttpClient with Graph API** button at the top of this article to adopt this approach.

The guidance in this article isn't meant to replace the Microsoft Graph documentation and Azure security guidance in other Microsoft documentation sets. Assess the security guidance in the Additional resources section of this article before implementing Microsoft Graph in a production environment. Follow Microsoft's best practices to limit the vulnerabilities of your apps.

Additional approaches for working with Microsoft Graph and Blazor WebAssembly are provided by the following Microsoft Graph and Azure samples:

- Microsoft Graph sample Blazor WebAssembly app: The sample adopts a factory-based approach with the Microsoft Graph SDK to obtain Office 365 data. The sample uses a default HttpClient to make Graph client requests. If you need to use the default HttpClient with the base address of the app (`BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)`) for other purposes, for example to load data from the web root of the app, consider refactoring the Graph client factory to use a named HttpClient dedicated to Graph requests.
- ASP.NET Core 8.0 Blazor WebAssembly | standalone app | user sign-in, protected web API access (Microsoft Graph) | Microsoft identity platform: The sample is a

Progressive Web Application (PWA) that uses an authorization message handler and HttpClient to obtain Graph data. As with the preceding sample, this sample also uses a default HttpClient to make Graph client requests. Instead of using the Microsoft Graph SDK, the sample retrieves user account data as JSON via a Microsoft Graph API request in a component. This is a similar technique to the **Named HttpClient with Graph API** approach in this article (use the button at the top of this article to see the guidance), except that this article's guidance uses a named HttpClient dedicated to Graph requests.

To provide feedback on either of the preceding two samples, open an issue on the sample's GitHub repository. If you're opening an issue for the Azure sample, provide a link to the sample in your opening comment because the Azure sample repository (`Azure-Samples`) contains many samples. Describe the problem in detail and include sample code as needed. Place a minimal app into GitHub that reproduces the problem or error. Be sure to remove Azure account configuration data from the sample before you commit it to the public repository.

To provide feedback or seek assistance with this article or ASP.NET Core, see ASP.NET Core Blazor fundamentals.

> **ⓘ Important**
>
> The scenarios described in this article apply to using Microsoft Entra (ME-ID) as the identity provider, not AAD B2C. Using Microsoft Graph with a client-side Blazor WebAssembly app and the AAD B2C identity provider isn't supported at this time because the app would require a client secret, which can't be secured in the client-side Blazor app. For an AAD B2C standalone Blazor WebAssembly app use Graph API, create a backend server (web) API to access Graph API on behalf of users. The client-side app authenticates and authorizes users to call the web API to securely access Microsoft Graph and return data to the client-side Blazor app from your server-based web API. The client secret is safely maintained in the server-based web API, not in the Blazor app on the client. **Never store a client secret in a client-side Blazor app.**

The examples in this article take advantage of new .NET/C# features. When using the examples with .NET 7 or earlier, minor modifications are required. However, the text and code examples that pertain to interacting with Microsoft Graph are the same for all versions of ASP.NET Core.

The following guidance applies to Microsoft Graph v4. If you're upgrading an app from SDK v4 to v5, see the Microsoft Graph .NET SDK v5 changelog and upgrade guide.

The Microsoft Graph SDK for use in Blazor apps is called the *Microsoft Graph .NET Client Library*.

The Graph SDK examples require the following package references in the standalone Blazor WebAssembly app. The first two packages are already referenced if the app has been enabled for MSAL authentication, for example when creating the app by following the guidance in Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID.

- Microsoft.AspNetCore.Components.WebAssembly.Authentication
- Microsoft.Authentication.WebAssembly.Msal
- Microsoft.Extensions.Http
- Microsoft.Graph

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

In the Azure portal, grant delegated permissions (scopes)† for Microsoft Graph data that the app should be able to access on behalf of a user. For the example in this article, the app's registration should include delegated permission to read user data (`Microsoft.Graph` > `User.Read` scope in **API permissions**, Type: Delegated). The `User.Read` scope allows users to sign in to the app and allows the app to read the profile and company information of signed-in users. For more information, see Overview of permissions and consent in the Microsoft identity platform and Overview of Microsoft Graph permissions.

†*Permissions* and *scopes* mean the same thing and are used interchangeably in security documentation and the Azure portal. Unless the text is referring to the Azure portal, this article uses *scope/scopes* when referring to Graph permissions.

Scopes are case insensitive, so `User.Read` is the same as `user.read`. Feel free to use either format, but we recommend a consistent choice across application code.

After adding the Microsoft Graph API scopes to the app's registration in the Azure portal, add the following app settings configuration to the `wwwroot/appsettings.json` file in the app, which includes the Graph base URL with the Microsoft Graph version and scopes. In the following example, the `User.Read` scope is specified for the examples in later sections of this article. Scopes aren't case sensitive.

JSON

```
"MicrosoftGraph": {
  "BaseUrl": "https://graph.microsoft.com",
  "Version": "{VERSION}",
  "Scopes": [
    "user.read"
  ]
}
```

In the preceding example, the `{VERSION}` placeholder is the version of the Microsoft Graph API (for example: `v1.0`).

The following is an example of a complete `wwwroot/appsettings.json` configuration file for an app that uses ME-ID as its identity provider, where reading user data (`user.read` scope) is specified for Microsoft Graph:

JSON

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  },
  "MicrosoftGraph": {
    "BaseUrl": "https://graph.microsoft.com",
    "Version": "v1.0",
    "Scopes": [
      "user.read"
    ]
  }
}
```

In the preceding example, the `{TENANT ID}` placeholder is the Directory (tenant) ID, and the `{CLIENT ID}` placeholder is the Application (client) ID. For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID.

Add the following `GraphClientExtensions` class to the standalone app. The scopes are provided to the Scopes property of the AccessTokenRequestOptions in the `AuthenticateRequestAsync` method. The IHttpProvider.OverallTimeout is extended from the default value of 100 seconds to 300 seconds to give the HttpClient more time to receive a response from Microsoft Graph.

When an access token isn't obtained, the following code doesn't set a Bearer authorization header for Graph requests.

`GraphClientExtensions.cs`:

C#

```
using System.Net.Http.Headers;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.Authentication.WebAssembly.Msal.Models;
using Microsoft.Graph;

namespace BlazorSample;

internal static class GraphClientExtensions
{
    public static IServiceCollection AddGraphClient(
        this IServiceCollection services, string? baseUrl, List<string>?
scopes)
    {
        if (string.IsNullOrEmpty(baseUrl) || scopes?.Count == 0)
        {
            return services;
        }

        services.Configure<RemoteAuthenticationOptions<MsalProviderOptions>>
(
            options =>
            {
                scopes?.ForEach((scope) =>
                {

options.ProviderOptions.DefaultAccessTokenScopes.Add(scope);
                });
            });

        services.AddScoped<IAuthenticationProvider,
GraphAuthenticationProvider>();

        services.AddScoped<IHttpProvider, HttpClientHttpProvider>(sp =>
            new HttpClientHttpProvider(new HttpClient()));

        services.AddScoped(sp =>
        {
            return new GraphServiceClient(
                baseUrl,
                sp.GetRequiredService<IAuthenticationProvider>(),
                sp.GetRequiredService<IHttpProvider>());
        });

        return services;
    }

    private class GraphAuthenticationProvider(IAccessTokenProvider
tokenProvider,
        IConfiguration config) : IAuthenticationProvider
    {
```

```
        private readonly IConfiguration config = config;

        public IAccessTokenProvider TokenProvider { get; } = tokenProvider;

        public async Task AuthenticateRequestAsync(HttpRequestMessage
request)
        {
            var result = await TokenProvider.RequestAccessToken(
                new AccessTokenRequestOptions()
                {
                    Scopes =
config.GetSection("MicrosoftGraph:Scopes").Get<string[]>()
                });

            if (result.TryGetToken(out var token))
            {
                request.Headers.Authorization ??= new
AuthenticationHeaderValue(
                    "Bearer", token.Value);
            }
        }
    }

    private class HttpClientHttpProvider(HttpClient client) : IHttpProvider
    {
        private readonly HttpClient client = client;

        public ISerializer Serializer { get; } = new Serializer();

        public TimeSpan OverallTimeout { get; set; } =
TimeSpan.FromSeconds(300);

        public Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request)
        {
            return client.SendAsync(request);
        }

        public Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request,
            HttpCompletionOption completionOption,
            CancellationToken cancellationToken)
        {
            return client.SendAsync(request, completionOption,
cancellationToken);
        }

        public void Dispose()
        {
        }
    }
}
```

> **ⓘ Important**
>
> See the DefaultAccessTokenScopes versus AdditionalScopesToConsent section for an explanation on why the preceding code uses DefaultAccessTokenScopes to add the scopes rather than AdditionalScopesToConsent.

In the `Program` file, add the Graph client services and configuration with the `AddGraphClient` extension method:

C#

```
var baseUrl = string.Join("/",
    builder.Configuration.GetSection("MicrosoftGraph")["BaseUrl"] ??
        "https://graph.microsoft.com",
    builder.Configuration.GetSection("MicrosoftGraph")["Version"] ??
        "v1.0");
var scopes = builder.Configuration.GetSection("MicrosoftGraph:Scopes")
    .Get<List<string>>() ?? [ "user.read" ];

builder.Services.AddGraphClient(baseUrl, scopes);
```

Call Graph API from a component using the Graph SDK

The following `UserData` component uses an injected `GraphServiceClient` to obtain the user's ME-ID profile data and display their mobile phone number. For any test user that you create in ME-ID, make sure that you give the user's ME-ID profile a mobile phone number in the Azure portal.

`UserData.razor`:

razor

```
@page "/user-data"
@using Microsoft.AspNetCore.Authorization
@using Microsoft.Graph
@attribute [Authorize]
@inject GraphServiceClient Client

<PageTitle>User Data</PageTitle>

<h1>Microsoft Graph User Data</h1>

@if (!string.IsNullOrEmpty(user?.MobilePhone))
{
```

```
    <p>Mobile Phone: @user.MobilePhone</p>
}

@code {
    private Microsoft.Graph.User? user;

    protected override async Task OnInitializedAsync()
    {
        var request = Client.Me.Request();
        user = await request.GetAsync();
    }
}
```

Add a link to the component's page in the `NavMenu` component (`Layout/NavMenu.razor`):

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="user-data">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
User Data
    </NavLink>
</div>
```

> **Tip**
>
> To add users to an app, see the **Assign users to an app registration with or without app roles** section.

When testing with the Graph SDK locally, we recommend using a new InPrivate/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID.

Customize user claims using the Graph SDK

In the following example, the app creates mobile phone number and office location claims for a user from their ME-ID user profile's data. The app must have the `User.Read` Graph API scope configured in ME-ID. Any test users for this scenario must have a mobile phone number and office location in their ME-ID profile, which can be added via the Azure portal.

In the following custom user account factory:

- An ILogger (`logger`) is included for convenience in case you wish to log information or errors in the `CreateUserAsync` method.
- In the event that an AccessTokenNotAvailableException is thrown, the user is redirected to the identity provider to sign into their account. Additional or different actions can be taken when requesting an access token fails. For example, the app can log the AccessTokenNotAvailableException and create a support ticket for further investigation.
- The framework's RemoteUserAccount represents the user's account. If the app requires a custom user account class that extends RemoteUserAccount, swap your custom user account class for RemoteUserAccount in the following code.

`CustomAccountFactory.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;
using Microsoft.Graph;

public class CustomAccountFactory(IAccessTokenProviderAccessor accessor,
        IServiceProvider serviceProvider, ILogger<CustomAccountFactory>
logger)
    : AccountClaimsPrincipalFactory<RemoteUserAccount>(accessor)
{
    private readonly ILogger<CustomAccountFactory> logger = logger;
    private readonly IServiceProvider serviceProvider = serviceProvider;

    public override async ValueTask<ClaimsPrincipal> CreateUserAsync(
        RemoteUserAccount account,
        RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity is not null &&
            initialUser.Identity.IsAuthenticated)
        {
            var userIdentity = initialUser.Identity as ClaimsIdentity;

            if (userIdentity is not null)
            {
                try
                {
                    var client = ActivatorUtilities
                        .CreateInstance<GraphServiceClient>
(serviceProvider);
                    var request = client.Me.Request();
                    var user = await request.GetAsync();

                    if (user is not null)
```

```
                    {
                        userIdentity.AddClaim(new Claim("mobilephone",
                            user.MobilePhone ?? "(000) 000-0000"));
                        userIdentity.AddClaim(new Claim("officelocation",
                            user.OfficeLocation ?? "Not set"));
                    }
                }
                catch (AccessTokenNotAvailableException exception)
                {
                    exception.Redirect();
                }
            }
        }

        return initialUser;
    }
}
```

Configure the MSAL authentication to use the custom user account factory.

Confirm that the `Program` file uses the Microsoft.AspNetCore.Components.WebAssembly.Authentication namespace:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
```

The example in this section builds on the approach of reading the base URL with version and scopes from app configuration via the `MicrosoftGraph` section in `wwwroot/appsettings.json` file. The following lines should already be present in the `Program` file from following the guidance earlier in this article:

C#

```
var baseUrl = string.Join("/",
    builder.Configuration.GetSection("MicrosoftGraph")["BaseUrl"] ??
        "https://graph.microsoft.com",
    builder.Configuration.GetSection("MicrosoftGraph")["Version"] ??
        "v1.0");
var scopes = builder.Configuration.GetSection("MicrosoftGraph:Scopes")
    .Get<List<string>>() ?? [ "user.read" ];

builder.Services.AddGraphClient(baseUrl, scopes);
```

In the `Program` file, find the call to the AddMsalAuthentication extension method. Update the code to the following, which includes a call to

AddAccountClaimsPrincipalFactory that adds an account claims principal factory with the `CustomAccountFactory`.

If the app uses a custom user account class that extends RemoteUserAccount, swap the custom user account class for RemoteUserAccount in the following code.

C#

```
builder.Services.AddMsalAuthentication<RemoteAuthenticationState,
    RemoteUserAccount>(options =>
    {
        builder.Configuration.Bind("AzureAd",
            options.ProviderOptions.Authentication);
    })
    .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
RemoteUserAccount,
        CustomAccountFactory>();
```

You can use the following `UserClaims` component to study the user's claims after the user authenticates with ME-ID:

`UserClaims.razor`:

razor

```
@page "/user-claims"
@using System.Security.Claims
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
@inject AuthenticationStateProvider AuthenticationStateProvider

<h1>User Claims</h1>

@if (claims.Any())
{
    <ul>
        @foreach (var claim in claims)
        {
            <li>@claim.Type: @claim.Value</li>
        }
    </ul>
}
else
{
    <p>No claims found.</p>
}

@code {
    private IEnumerable<Claim> claims = Enumerable.Empty<Claim>();

    protected override async Task OnInitializedAsync()
```

```
    {
        var authState = await AuthenticationStateProvider
            .GetAuthenticationStateAsync();
        var user = authState.User;

        claims = user.Claims;
    }
}
```

Add a link to the component's page in the `NavMenu` component (`Layout/NavMenu.razor`):

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="user-claims">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
User Claims
    </NavLink>
</div>
```

When testing with the Graph SDK locally, we recommend using a new InPrivate/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID.

Assign users to an app registration with or without app roles

You can add users to an app registration and assign roles to users with the following steps in the Azure portal.

To add a user, select **Users** from the ME-ID area of the Azure portal:

1. Select **New user** > **Create new user**.
2. Use the **Create user** template.
3. Supply the user's information in the **Identity** area.
4. You can generate an initial password or assign an initial password that the user changes when they sign in for the first time. If you use the password generated by the portal, make a note of it now.
5. Select **Create** to create the user. When **Create new user** interface closes, select **Refresh** to update the user list and show the new user.
6. For the examples in this article, assign a mobile phone number to the new user by selecting their name from the users list, selecting **Properties**, and editing the contact information to provide a mobile phone number.

To assign users to the app *without app roles*:

1. In the ME-ID area of the Azure portal, open **Enterprise applications**.
2. Select the app from the list.
3. Select **Users and groups**.
4. Select **Add user/group**.
5. Select a user.
6. Select the **Assign** button.

To assign users to the app *with app roles*:

1. Add roles to the app's registration in the Azure portal following the guidance in ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles.
2. In the ME-ID area of the Azure portal, open **Enterprise applications**.
3. Select the app from the list.
4. Select **Users and groups**.
5. Select **Add user/group**.
6. Select a user and select their role for accessing the app. Multiple roles are assigned to a user by repeating the process of adding the user to the app until all of the roles for a user are assigned. Users with multiple roles are listed once for each assigned role in the **Users and groups** list of users for the app.
7. Select the **Assign** button.

`DefaultAccessTokenScopes` versus `AdditionalScopesToConsent`

The examples in this article provision Graph API scopes with DefaultAccessTokenScopes, not AdditionalScopesToConsent.

AdditionalScopesToConsent isn't used because it's unable to provision Graph API scopes for users when they sign in to the app for the first time with MSAL via the Azure consent UI. When the user attempts to access Graph API for the first time with the Graph SDK, they're confronted with an exception:

> Microsoft.Graph.Models.ODataErrors.ODataError: Access token is empty.

After a user provisions Graph API scopes provided via DefaultAccessTokenScopes, the app can use AdditionalScopesToConsent for a subsequent user sign in. However, changing app code makes no sense for a production app that requires the periodic addition of new users with delegated Graph scopes or adding new delegated Graph API scopes to the app.

The preceding discussion of how to provision scopes for Graph API access when the user first signs into the app only applies to:

- Apps that adopt the Graph SDK.
- Apps that use a named HttpClient for Graph API access that asks users to consent to Graph scopes on their first sign in to the app.

When using a named HttpClient that doesn't ask users to consent to Graph scopes on their first sign in, users are redirected to the Azure consent UI for Graph API scopes consent *when they first request access to Graph API* via the DelegatingHandler of the preconfigured, named HttpClient. When Graph scopes aren't consented initially with the named HttpClient approach, neither DefaultAccessTokenScopes nor AdditionalScopesToConsent are called by the app. For more information, see the named HttpClient coverage in this article.

Additional resources

General guidance

- Microsoft Graph documentation
- Microsoft Graph sample Blazor WebAssembly app : This sample demonstrates how to use the Microsoft Graph .NET SDK to access data in Office 365 from Blazor WebAssembly apps.
- Build .NET apps with Microsoft Graph tutorial and Microsoft Graph sample ASP.NET Core app : Although these resources don't directly apply to calling Graph from *client-side* Blazor WebAssembly apps, the ME-ID app configuration and Microsoft Graph coding practices in the linked resources are relevant for standalone Blazor WebAssembly apps and should be consulted for general best practices.

Security guidance

- Microsoft Graph auth overview
- Overview of Microsoft Graph permissions
- Microsoft Graph permissions reference
- Overview of permissions and consent in the Microsoft identity platform
- Enhance security with the principle of least privilege
- Azure privilege escalation articles on the Internet (Google search result)
- Microsoft Security Best Practices: Securing privileged access

Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)

Article • 12/20/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes how to secure a Blazor Web App with OpenID Connect (OIDC) using a sample app in the dotnet/blazor-samples GitHub repository (.NET 8 or later) (how to download).

This version of the article covers implementing OIDC without adopting the Backend for Frontend (BFF) pattern. The BFF pattern is useful for making authenticated requests to external services. Change the article version selector to **OIDC with BFF pattern** if the app's specification calls for adopting the BFF pattern.

The following specification is covered:

- The Blazor Web App uses the Auto render mode with global interactivity.
- Custom auth state provider services are used by the server and client apps to capture the user's authentication state and flow it between the server and client.
- This app is a starting point for any OIDC authentication flow. OIDC is configured manually in the app and doesn't rely upon Microsoft Entra ID or Microsoft Identity Web packages, nor does the sample app require Microsoft Azure hosting. However, the sample app can be used with Entra, Microsoft Identity Web, and hosted in Azure.
- Automatic non-interactive token refresh.
- Securely calls a (web) API in the server project for data.

For an alternative experience using Microsoft Authentication Library for .NET, Microsoft Identity Web, and Microsoft Entra ID, see Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID.

Sample app

The sample app consists of two projects:

- `BlazorWebAppOidc`: Server-side project of the Blazor Web App, containing an example Minimal API endpoint for weather data.
- `BlazorWebAppOidc.Client`: Client-side project of the Blazor Web App.

Access the sample apps through the latest version folder from the repository's root with the following link. The projects are in the `BlazorWebAppOidc` folder for .NET 8 or later.

View or download sample code (how to download)

Server-side Blazor Web App project (`BlazorWebAppOidc`)

The `BlazorWebAppOidc` project is the server-side project of the Blazor Web App.

The `BlazorWebAppOidc.http` file can be used for testing the weather data request. Note that the `BlazorWebAppOidc` project must be running to test the endpoint, and the endpoint is hardcoded into the file. For more information, see Use .http files in Visual Studio 2022.

Configuration

This section explains how to configure the sample app.

> **Note**
>
> For Microsoft Entra ID or Azure AD B2C, you can use AddMicrosoftIdentityWebApp from Microsoft Identity Web (Microsoft.Identity.Web NuGet package, API documentation), which adds both the OIDC and Cookie authentication handlers with the appropriate defaults. The sample app and the guidance in this section doesn't use Microsoft Identity Web. The guidance demonstrates how to configure the OIDC handler *manually* for any OIDC provider. For more information on implementing Microsoft Identity Web, see the linked resources.

Establish the client secret

> **Warning**

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

For local development testing, use the Secret Manager tool to store the server app's client secret under the configuration key `Authentication:Schemes:MicrosoftOidc:ClientSecret`.

> **Note**
>
> If the app uses Microsoft Entra ID or Azure AD B2C, create a client secret in the app's registration in the Entra or Azure portal (**Manage** > **Certificates & secrets** > **New client secret**). Use the **Value** of the new secret in the following guidance.

The sample app hasn't been initialized for the Secret Manager tool. Use a command shell, such as the Developer PowerShell command shell in Visual Studio, to execute the following command. Before executing the command, change the directory with the `cd` command to the server project's directory. The command establishes a user secrets identifier (`<UserSecretsId>` in the server app's project file):

.NET CLI

```
dotnet user-secrets init
```

Execute the following command to set the client secret. The `{SECRET}` placeholder is the client secret obtained from the app's registration:

.NET CLI

```
dotnet user-secrets set "Authentication:Schemes:MicrosoftOidc:ClientSecret"
"{SECRET}"
```

If using Visual Studio, you can confirm the secret is set by right-clicking the server project in **Solution Explorer** and selecting **Manage User Secrets**.

Configure the app

The following OpenIdConnectOptions configuration is found in the project's `Program` file on the call to AddOpenIdConnect:

- SignInScheme: Sets the authentication scheme corresponding to the middleware responsible of persisting user's identity after a successful authentication. The OIDC handler needs to use a sign-in scheme that's capable of persisting user credentials across requests. The following line is present merely for demonstration purposes. If omitted, DefaultSignInScheme is used as a fallback value.

  C#

  ```
  oidcOptions.SignInScheme =
  CookieAuthenticationDefaults.AuthenticationScheme;
  ```

- Scopes for `openid` and `profile` (Scope) (Optional): The `openid` and `profile` scopes are also configured by default because they're required for the OIDC handler to work, but these may need to be re-added if scopes are included in the `Authentication:Schemes:MicrosoftOidc:Scope` configuration. For general configuration guidance, see Configuration in ASP.NET Core and ASP.NET Core Blazor configuration.

  C#

  ```
  oidcOptions.Scope.Add(OpenIdConnectScope.OpenIdProfile);
  ```

- SaveTokens: Defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. This property is set to `false` to reduce the size of the final authentication cookie.

  C#

  ```
  oidcOptions.SaveTokens = false;
  ```

- Scope for offline access (Scope): The `offline_access` scope is required for the refresh token.

  C#

  ```
  oidcOptions.Scope.Add(OpenIdConnectScope.OfflineAccess);
  ```

- Authority and ClientId: Sets the Authority and Client ID for OIDC calls.

```
C#

oidcOptions.Authority = "{AUTHORITY}";
oidcOptions.ClientId = "{CLIENT ID}";
```

Example:

- Authority (`{AUTHORITY}`): `https://login.microsoftonline.com/aaaabbbb-0000-cccc-1111-dddd2222eeee/v2.0/` (uses Tenant ID `aaaabbbb-0000-cccc-1111-dddd2222eeee`)
- Client Id (`{CLIENT ID}`): `00001111-aaaa-2222-bbbb-3333cccc4444`

```
C#

oidcOptions.Authority = "https://login.microsoftonline.com/aaaabbbb-0000-cccc-1111-dddd2222eeee/v2.0/";
oidcOptions.ClientId = "00001111-aaaa-2222-bbbb-3333cccc4444";
```

Example for Microsoft Azure "common" authority:

The "common" authority should be used for multi-tenant apps. You can also use the "common" authority for single-tenant apps, but a custom IssuerValidator is required, as shown later in this section.

```
C#

oidcOptions.Authority =
"https://login.microsoftonline.com/common/v2.0/";
```

- ResponseType: Configures the OIDC handler to only perform authorization code flow. Implicit grants and hybrid flows are unnecessary in this mode. The OIDC handler automatically requests the appropriate tokens using the code returned from the authorization endpoint.

```
C#

oidcOptions.ResponseType = OpenIdConnectResponseType.Code;
```

> **Note**
>
> In the Entra or Azure portal's **Implicit grant and hybrid flows** app registration configuration, do **not** select either checkbox for the authorization endpoint to return **Access tokens** or **ID tokens**.

- MapInboundClaims and configuration of NameClaimType and RoleClaimType: Many OIDC servers use "`name`" and "`role`" rather than the SOAP/WS-Fed defaults in ClaimTypes. When MapInboundClaims is set to `false`, the handler doesn't perform claims mappings, and the claim names from the JWT are used directly by the app. The following example sets the role claim type to "`roles`," which is appropriate for Microsoft Entra ID (ME-ID). Consult your identity provider's documentation for more information.

  > **Note**
  >
  > MapInboundClaims must be set to `false` for most OIDC providers, which prevents renaming claims.

  C#

  ```
  oidcOptions.MapInboundClaims = false;
  oidcOptions.TokenValidationParameters.NameClaimType = "name";
  oidcOptions.TokenValidationParameters.RoleClaimType = "roles";
  ```

- Path configuration: Paths must match the redirect URI (login callback path) and post logout redirect (signed-out callback path) paths configured when registering the application with the OIDC provider. In the Azure portal, paths are configured in the **Authentication** blade of the app's registration. Both the sign-in and sign-out paths must be registered as redirect URIs. The default values are `/signin-oidc` and `/signout-callback-oidc`.

  - CallbackPath: The request path within the app's base path where the user-agent is returned.

    Configure the signed-out callback path in the app's OIDC provider registration. In the following example, the `{PORT}` placeholder is the app's port:

    > https://localhost:{PORT}/signin-oidc

    > **Note**
    >
    > A port isn't required for `localhost` addresses when using Microsoft Entra ID. Most other OIDC providers require the correct port.

- SignedOutCallbackPath (configuration key: "`SignedOutCallbackPath`"): The request path within the app's base path intercepted by the OIDC handler where the user agent is first returned after signing out from the identity provider. The sample app doesn't set a value for the path because the default value of "`/signout-callback-oidc`" is used. After intercepting the request, the OIDC handler redirects to the SignedOutRedirectUri or RedirectUri, if specified.

  Configure the signed-out callback path in the app's OIDC provider registration. In the following example, the `{PORT}` placeholder is the app's port:

  > https://localhost:{PORT}/signout-callback-oidc

  > **Note**
  >
  > When using Microsoft Entra ID, set the path in the **Web** platform configuration's **Redirect URI** entries in the Entra or Azure portal. A port isn't required for `localhost` addresses when using Entra. Most other OIDC providers require the correct port. If you don't add the signed-out callback path URI to the app's registration in Entra, Entra refuses to redirect the user back to the app and merely asks them to close their browser window.

- RemoteSignOutPath: Requests received on this path cause the handler to invoke sign-out using the sign-out scheme.

  In the following example, the `{PORT}` placeholder is the app's port:

  > https://localhost/signout-oidc

  > **Note**
  >
  > When using Microsoft Entra ID, set the **Front-channel logout URL** in the Entra or Azure portal. A port isn't required for `localhost` addresses when using Entra. Most other OIDC providers require the correct port.

  C#

  ```
  oidcOptions.CallbackPath = new PathString("{PATH}");
  oidcOptions.SignedOutCallbackPath = new PathString("{PATH}");
  oidcOptions.RemoteSignOutPath = new PathString("{PATH}");
  ```

Examples (default values):

```
C#

oidcOptions.CallbackPath = new PathString("/signin-oidc");
oidcOptions.SignedOutCallbackPath = new PathString("/signout-
callback-oidc");
oidcOptions.RemoteSignOutPath = new PathString("/signout-oidc");
```

- (*Microsoft Azure only with the "common" endpoint*) TokenValidationParameters.IssuerValidator: Many OIDC providers work with the default issuer validator, but we need to account for the issuer parameterized with the Tenant ID (`{TENANT ID}`) returned by `https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration`. For more information, see SecurityTokenInvalidIssuerException with OpenID Connect and the Azure AD "common" endpoint (AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet #1731).

  Only for apps using Microsoft Entra ID or Azure AD B2C with the "common" endpoint:

  ```
  C#

  var microsoftIssuerValidator =
  AadIssuerValidator.GetAadIssuerValidator(oidcOptions.Authority);
  oidcOptions.TokenValidationParameters.IssuerValidator =
  microsoftIssuerValidator.Validate;
  ```

Sample app code

Inspect the sample app for the following features:

- Automatic non-interactive token refresh with the help of a custom cookie refresher (`CookieOidcRefresher.cs`).
- The `PersistingAuthenticationStateProvider` class (`PersistingAuthenticationStateProvider.cs`) is a server-side AuthenticationStateProvider that uses PersistentComponentState to flow the authentication state to the client, which is then fixed for the lifetime of the WebAssembly application.
- An example requests to the Blazor Web App for weather data is handled by a Minimal API endpoint (`/weather-forecast`) in the `Program` file (`Program.cs`). The endpoint requires authorization by calling RequireAuthorization. For any

controllers that you add to the project, add the [Authorize] attribute to the controller or action.

- The app securely calls a (web) API in the server project for weather data:
  - When rendering the `Weather` component on the server, the component uses the `ServerWeatherForecaster` on the server to obtain weather data directly (not via a web API call).
  - When the component is rendered on the client, the component uses the `ClientWeatherForecaster` service implementation, which uses a preconfigured HttpClient (in the client project's `Program` file) to make a web API call to the server project. A Minimal API endpoint (`/weather-forecast`) defined in the server project's `Program` file obtains the weather data from the `ServerWeatherForecaster` and returns the data to the client.

For more information on (web) API calls using a service abstractions in Blazor Web Apps, see Call a web API from an ASP.NET Core Blazor app.

Client-side Blazor Web App project (`BlazorWebAppOidc.Client`)

The `BlazorWebAppOidc.Client` project is the client-side project of the Blazor Web App.

The `PersistentAuthenticationStateProvider` class (`PersistentAuthenticationStateProvider.cs`) is a client-side AuthenticationStateProvider that determines the user's authentication state by looking for data persisted in the page when it was rendered on the server. The authentication state is fixed for the lifetime of the WebAssembly application.

If the user needs to log in or out, a full page reload is required.

The sample app only provides a user name and email for display purposes. It doesn't include tokens that authenticate to the server when making subsequent requests, which works separately using a cookie that's included on HttpClient requests to the server.

Redirect to the home page on logout

The `LogInOrOut` component (`Layout/LogInOrOut.razor`) sets a hidden field for the return URL (`ReturnUrl`) to the current URL (`currentURL`). When the user signs out of the app, the identity provider returns the user to the page from which they logged out. If the user logs out from a secure page, they're returned to the same secure page and sent

back through the authentication process. This authentication flow is reasonable when users need to change accounts regularly.

Alternatively, use the following `LogInOrOut` component, which doesn't supply a return URL when logging out.

`Layout/LogInOrOut.razor`:

razor

```razor
<div class="nav-item px-3">
    <AuthorizeView>
        <Authorized>
            <form action="authentication/logout" method="post">
                <AntiforgeryToken />
                <button type="submit" class="nav-link">
                    <span class="bi bi-arrow-bar-left-nav-menu" aria-
hidden="true">
                    </span> Logout @context.User.Identity?.Name
                </button>
            </form>
        </Authorized>
        <NotAuthorized>
            <a class="nav-link" href="authentication/login">
                <span class="bi bi-person-badge-nav-menu" aria-
hidden="true"></span>
                Login
            </a>
        </NotAuthorized>
    </AuthorizeView>
</div>
```

Token refresh

The custom cookie refresher (`CookieOidcRefresher.cs`) implementation updates the user's claims automatically when they expire. The current implementation expects to receive an ID token from the token endpoint in exchange for the refresh token. The claims in this ID token are then used to overwrite the user's claims.

The sample implementation doesn't include code for requesting claims from the UserInfo endpoint on token refresh. For more information, see BlazorWebAppOidc AddOpenIdConnect with GetClaimsFromUserInfoEndpoint = true doesn't propogate role claims to client (dotnet/aspnetcore #58826).

> ⓘ Note

Some identity providers only return an access token when using a refresh token. The `CookieOidcRefresher` can be updated with additional logic to continue to use the prior set of claims stored in the authentication cookie or use the access token to request claims from the UserInfo endpoint.

Cryptographic nonce

A *nonce* is a string value that associates a client's session with an ID token to mitigate replay attacks.

If you receive a nonce error during authentication development and testing, use a new InPrivate/incognito browser session for each test run, no matter how small the change made to the app or test user because stale cookie data can lead to a nonce error. For more information, see the Cookies and site data section.

A nonce isn't required or used when a refresh token is exchanged for a new access token. In the sample app, the `CookieOidcRefresher` (`CookieOidcRefresher.cs`) deliberately sets OpenIdConnectProtocolValidator.RequireNonce to `false`.

Application roles for apps not registered with Microsoft Entra (ME-ID)

This section pertains to apps that don't use Microsoft Entra ID (ME-ID) as the identity provider. For apps registered with ME-ID, see the Application roles for apps registered with Microsoft Entra (ME-ID) section.

Configure the role claim type (TokenValidationParameters.RoleClaimType) in the OpenIdConnectOptions of `Program.cs`:

C#

```
oidcOptions.TokenValidationParameters.RoleClaimType = "{ROLE CLAIM TYPE}";
```

For many OIDC identity providers, the role claim type is `role`. Check your identity provider's documentation for the correct value.

Replace the `UserInfo` class in the `BlazorWebAppOidc.Client` project with the following class.

`UserInfo.cs`:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using System.Security.Claims;

namespace BlazorWebAppOidc.Client;

// Add properties to this class and update the server and client
// AuthenticationStateProviders to expose more information about
// the authenticated user to the client.
public sealed class UserInfo
{
    public required string UserId { get; init; }
    public required string Name { get; init; }
    public required string[] Roles { get; init; }

    public const string UserIdClaimType = "sub";
    public const string NameClaimType = "name";
    private const string RoleClaimType = "role";

    public static UserInfo FromClaimsPrincipal(ClaimsPrincipal principal) =>
        new()
        {
            UserId = GetRequiredClaim(principal, UserIdClaimType),
            Name = GetRequiredClaim(principal, NameClaimType),
            Roles = principal.FindAll(RoleClaimType).Select(c => c.Value)
                .ToArray(),
        };

    public ClaimsPrincipal ToClaimsPrincipal() =>
        new(new ClaimsIdentity(
            Roles.Select(role => new Claim(RoleClaimType, role))
                .Concat([
                    new Claim(UserIdClaimType, UserId),
                    new Claim(NameClaimType, Name),
                ]),
            authenticationType: nameof(UserInfo),
            nameType: NameClaimType,
            roleType: RoleClaimType));

    private static string GetRequiredClaim(ClaimsPrincipal principal,
        string claimType) =>
            principal.FindFirst(claimType)?.Value ??
            throw new InvalidOperationException(
                $"Could not find required '{claimType}' claim.");
}
```

At this point, Razor components can adopt role-based and policy-based authorization. Application roles appear in `role` claims, one claim per role.

Application roles for apps registered with Microsoft Entra (ME-ID)

Use the guidance in this section to implement application roles, ME-ID security groups, and ME-ID built-in administrator roles for apps using Microsoft Entra ID (ME-ID).

The approach described in this section configures ME-ID to send groups and roles in the authentication cookie header. When users are only a member of a few security groups and roles, the following approach should work for most hosting platforms without running into a problem where headers are too long, for example with IIS hosting that has a default header length limit of 16 KB (`MaxRequestBytes`). If header length is a problem due to high group or role membership, we recommend not following the guidance in this section in favor of implementing Microsoft Graph to obtain a user's groups and roles from ME-ID separately, an approach that doesn't inflate the size of the authentication cookie. For more information, see Bad Request - Request Too Long - IIS Server (dotnet/aspnetcore #57545).

Configure the role claim type (TokenValidationParameters.RoleClaimType) in OpenIdConnectOptions of `Program.cs`. Set the value to `roles`:

```
C#

oidcOptions.TokenValidationParameters.RoleClaimType = "roles";
```

Although you can't assign roles to groups without an ME-ID Premium account, you can assign roles to users and receive role claims for users with a standard Azure account. The guidance in this section doesn't require an ME-ID Premium account.

When working with the default directory, follow the guidance in Add app roles to your application and receive them in the token (ME-ID documentation) to configure and assign roles. If you aren't working with the default directory, edit the app's manifest in the Azure portal to establish the app's roles manually in the `appRoles` entry of the manifest file. For more information, see Configure the role claim (ME-ID documentation).

A user's Azure security groups arrive in `groups` claims, and a user's built-in ME-ID administrator role assignments arrive in well-known IDs (wids) claims. Values for both claim types are GUIDs. When received by the app, these claims can be used to establish role and policy authorization in Razor components.

In the app's manifest in the Azure portal, set the groupMembershipClaims attribute to `All`. A value of `All` results in ME-ID sending all of the security/distribution groups

(`groups` claims) and roles (`wids` claims) of the signed-in user. To set the `groupMembershipClaims` attribute:

1. Open the app's registration in the Azure portal.
2. Select **Manage** > **Manifest** in the sidebar.
3. Find the `groupMembershipClaims` attribute.
4. Set the value to `All` (`"groupMembershipClaims": "All"`).
5. Select the **Save** button.

Replace the `UserInfo` class in the `BlazorWebAppOidc.Client` project with the following class.

`UserInfo.cs`:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using System.Security.Claims;

namespace BlazorWebAppOidc.Client;

// Add properties to this class and update the server and client
// AuthenticationStateProviders to expose more information about
// the authenticated user to the client.
public sealed class UserInfo
{
    public required string UserId { get; init; }
    public required string Name { get; init; }
    public required string[] Roles { get; init; }
    public required string[] Groups { get; init; }
    public required string[] Wids { get; init; }

    public const string UserIdClaimType = "sub";
    public const string NameClaimType = "name";
    private const string RoleClaimType = "roles";
    private const string GroupsClaimType = "groups";
    private const string WidsClaimType = "wids";

    public static UserInfo FromClaimsPrincipal(ClaimsPrincipal principal) =>
        new()
        {
            UserId = GetRequiredClaim(principal, UserIdClaimType),
            Name = GetRequiredClaim(principal, NameClaimType),
            Roles = principal.FindAll(RoleClaimType).Select(c => c.Value)
                .ToArray(),
            Groups = principal.FindAll(GroupsClaimType).Select(c => c.Value)
                .ToArray(),
            Wids = principal.FindAll(WidsClaimType).Select(c => c.Value)
                .ToArray(),
        };
```

```
    public ClaimsPrincipal ToClaimsPrincipal() =>
        new(new ClaimsIdentity(
            Roles.Select(role => new Claim(RoleClaimType, role))
                .Concat(Groups.Select(role => new Claim(GroupsClaimType,
role)))
                .Concat(Wids.Select(role => new Claim(WidsClaimType, role)))
                .Concat([
                    new Claim(UserIdClaimType, UserId),
                    new Claim(NameClaimType, Name),
                ]),
            authenticationType: nameof(UserInfo),
            nameType: NameClaimType,
            roleType: RoleClaimType));

    private static string GetRequiredClaim(ClaimsPrincipal principal,
        string claimType) =>
            principal.FindFirst(claimType)?.Value ??
            throw new InvalidOperationException(
                $"Could not find required '{claimType}' claim.");
}
```

At this point, Razor components can adopt role-based and policy-based authorization:

- Application roles appear in `roles` claims, one claim per role.
- Security groups appear in `groups` claims, one claim per group. The security group GUIDs appear in the Azure portal when you create a security group and are listed when selecting **Identity** > **Overview** > **Groups** > **View**.
- Built-in ME-ID administrator roles appear in `wids` claims, one claim per role. The `wids` claim with a value of `b79fbf4d-3ef9-4689-8143-76b194e85509` is always sent by ME-ID for non-guest accounts of the tenant and doesn't refer to an administrator role. Administrator role GUIDs (*role template IDs*) appear in the Azure portal when selecting **Roles & admins**, followed by the ellipsis (...) > **Description** for the listed role. The role template IDs are also listed in Microsoft Entra built-in roles (Entra documentation).

Troubleshoot

Logging

The server app is a standard ASP.NET Core app. See the ASP.NET Core logging guidance to enable a lower logging level in the server app.

To enable debug or trace logging for Blazor WebAssembly authentication, see the *Client-side authentication logging* section of ASP.NET Core Blazor logging with the article version selector set to ASP.NET Core 7.0 or later.

Common errors

- Misconfiguration of the app or Identity Provider (IP)

  The most common errors are caused by incorrect configuration. The following are a few examples:

  - Depending on the requirements of the scenario, a missing or incorrect Authority, Instance, Tenant ID, Tenant domain, Client ID, or Redirect URI prevents an app from authenticating clients.
  - Incorrect request scopes prevent clients from accessing server web API endpoints.
  - Incorrect or missing server API permissions prevent clients from accessing server web API endpoints.
  - Running the app at a different port than is configured in the Redirect URI of the IP's app registration. Note that a port isn't required for Microsoft Entra ID and an app running at a `localhost` development testing address, but the app's port configuration and the port where the app is running must match for non-`localhost` addresses.

  Configuration coverage in this article shows examples of the correct configuration. Carefully check the configuration looking for app and IP misconfiguration.

  If the configuration appears correct:

  - Analyze application logs.

  - Examine the network traffic between the client app and the IP or server app with the browser's developer tools. Often, an exact error message or a message with a clue to what's causing the problem is returned to the client by the IP or server app after making a request. Developer tools guidance is found in the following articles:
    - Google Chrome (Google documentation)
    - Microsoft Edge
    - Mozilla Firefox (Mozilla documentation)

  The documentation team responds to document feedback and bugs in articles (open an issue from the **This page** feedback section) but is unable to provide product support. Several public support forums are available to assist with troubleshooting an app. We recommend the following:

  - Stack Overflow (tag: blazor)
  - ASP.NET Core Slack Team
  - Blazor Gitter

The preceding forums are not owned or controlled by Microsoft.

For non-security, non-sensitive, and non-confidential reproducible framework bug reports, open an issue with the ASP.NET Core product unit. Don't open an issue with the product unit until you've thoroughly investigated the cause of a problem and can't resolve it on your own and with the help of the community on a public support forum. The product unit isn't able to troubleshoot individual apps that are broken due to simple misconfiguration or use cases involving third-party services. If a report is sensitive or confidential in nature or describes a potential security flaw in the product that cyberattackers may exploit, see Reporting security issues and bugs (dotnet/aspnetcore GitHub repository).

- Unauthorized client for ME-ID

  > info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2]
  > Authorization failed. These requirements were not met:
  > DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

  Login callback error from ME-ID:
  - Error: `unauthorized_client`
  - Description: `AADB2C90058: The provided application is not configured to allow public clients.`

  To resolve the error:

  1. In the Azure portal, access the app's manifest.
  2. Set the allowPublicClient attribute to `null` or `true`.

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser

    - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
    - Make sure that the browser is closed manually or by the IDE for any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in InPrivate or Incognito mode in Visual Studio:
    - Open **Browse With** dialog box from Visual Studio's **Run** button.
    - Select the **Add** button.
    - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
        - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
        - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
        - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
    - In the **Arguments** field, provide the command-line option that the browser uses to open in InPrivate or Incognito mode. Some browsers require the URL of the app.
        - Microsoft Edge: Use `-inprivate`.
        - Google Chrome: Use `--incognito --new-window {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
        - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
    - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
    - Select the **OK** button.
    - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
    - Make sure that the browser is closed by the IDE for any change to the app, test user, or provider configuration.

App upgrades

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Clear the local system's NuGet package caches by executing dotnet nuget locals all --clear from a command shell.
2. Delete the project's `bin` and `obj` folders.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

> **Note**
>
> Use of package versions incompatible with the app's target framework isn't supported. For information on a package, use the NuGet Gallery or FuGet Package Explorer.

Run the server app

When testing and troubleshooting Blazor Web App, make sure that you're running the app from the server project.

Inspect the user

The following `UserClaims` component can be used directly in apps or serve as the basis for further customization.

`UserClaims.razor`:

razor

```
@page "/user-claims"
@using System.Security.Claims
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]

<PageTitle>User Claims</PageTitle>

<h1>User Claims</h1>

@if (claims.Any())
{
    <ul>
        @foreach (var claim in claims)
        {
            <li><b>@claim.Type:</b> @claim.Value</li>
        }
    </ul>
}
```

```
@code {
    private IEnumerable<Claim> claims = Enumerable.Empty<Claim>();

    [CascadingParameter]
    private Task<AuthenticationState>? AuthState { get; set; }

    protected override async Task OnInitializedAsync()
    {
        if (AuthState == null)
        {
            return;
        }

        var authState = await AuthState;
        claims = authState.User.Claims;
    }
}
```

Additional resources

- AzureAD/microsoft-identity-web GitHub repository: Helpful guidance on implementing Microsoft Identity Web for Microsoft Entra ID and Azure Active Directory B2C for ASP.NET Core apps, including links to sample apps and related Azure documentation. Currently, Blazor Web Apps aren't explicitly addressed by the Azure documentation, but the setup and configuration of a Blazor Web App for ME-ID and Azure hosting is the same as it is for any ASP.NET Core web app.
- AuthenticationStateProvider service
- Manage authentication state in Blazor Web Apps
- Refresh token during http request in Blazor Interactive Server with OIDC (dotnet/aspnetcore #55213)

Threat mitigation guidance for ASP.NET Core Blazor static server-side rendering

Article • 11/19/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains the security considerations that developers should take into account when developing Blazor Web Apps with static server-side rendering.

Blazor combines three different models in one for writing interactive web apps. Traditional server-side rendering, which is a request/response model based on HTTP. Interactive server-side rendering, which is a rendering model based on SignalR. Finally, client-side rendering, which is a rendering model based on WebAssembly.

All of the general security considerations defined for the interactive rendering modes apply to Blazor Web Apps when there are interactive components rendering in one of the supported render modes. The following sections explain the security considerations specific to non-interactive server-side rendering in Blazor Web Apps and the specific aspects that apply when render modes interact with each other.

General considerations for server-side rendering

The server-side rendering (SSR) model is based on the traditional request/response model of HTTP. As such, there are common areas of concern between SSR and request/response HTTP. General security considerations and specific threats must be successfully mitigated. The framework provides built-in mechanisms for managing some of these threats, but other threats are specific to app code and must be handled by the app. These threats can be categorized as follows:

- Authentication and authorization: The app must ensure that the user is authenticated and authorized to access the app and the resources it exposes. The framework provides built-in mechanisms for authentication and authorization, but the app must ensure that the mechanisms are properly configured and used. The built-in mechanisms for authentication and authorization are covered in the Blazor

documentation's *Server* security node and in the ASP.NET Core documentation's *Security and Identity* node, so they won't be covered here.

- Input validation and sanitization: All input arriving from a client must be validated and sanitized before use. Otherwise, the app might be exposed to attacks, such as SQL injection, cross-site scripting, cross-site request forgery, open redirection, and other forms of attacks. The input might come from anywhere in the request.
- Session management: Properly managing user sessions is critical to ensure that the app isn't exposed to attacks, such as session fixation, session hijacking, and other attacks. Information stored in the session must be properly protected and encrypted, and the app's code must prevent a malicious user from guessing or manipulating sessions.
- Error handling and logging: The app must ensure that errors are properly handled and logged. Otherwise, the app might be exposed to attacks, such as information disclosure. This can happen when the app returns sensitive information in the response or when the app returns detailed error messages with data that can be used to attack the app.
- Data protection: Sensitive data must be properly protected, which includes app logic when running on WebAssembly, since it can be easily reverse-engineered.
- Denial of service: The app must ensure that it isn't exposed to attacks, such as denial of service. This happens for example, when the app isn't properly protected against brute force attacks or when an action can cause the app to consume too many resources.

Input validation and sanitization

All input arriving from the client must be considered untrusted unless its information was generated and protected on the server, such as a CSRF token, an authentication cookie, a session identifier, or any other payload that's protected with authenticated encryption.

Input is normally available to the app through a binding process, for example via the [SupplyParameterFromQuery] attribute or [SupplyParameterFromForm] attribute. Before processing this input, the app must make sure that the data is valid. For example, the app must confirm that there were no binding errors when mapping the form data to a component property. Otherwise, the app might process invalid data.

If the input is used to perform a redirect, the app must make sure that the input is valid and that it isn't pointing to a domain considered invalid or to an invalid subpath within

the app base path. Otherwise, the app may be exposed to open redirection attacks, where a cyberattacker can craft a link that redirects the user to a malicious site.

If the input is used to perform a database query, app must confirm that the input is valid and that it isn't exposing the app to SQL injection attacks. Otherwise, a cyberattacker might be able to craft a malicious query that can be used to extract information from the database or to modify the database.

Data that might have come from user input also must be sanitized before included in a response. For example, the input might contain HTML or JavaScript that can be used to perform cross-site scripting attacks, which can be used to extract information from the user or to perform actions on behalf of the user.

The framework provides the following mechanisms to help with input validation and sanitization:

- All bound form data is validated for basic correctness. If an input can't be parsed, the binding process reports an error that the app can discover before taking any action with the data. The built-in EditForm component takes this into account before invoking the OnValidSubmit form callback. Blazor avoids executing the callback if there are one or more binding errors.
- The framework uses an antiforgery token to protect against cross-site request forgery attacks. For more information, see ASP.NET Core Blazor authentication and authorization and ASP.NET Core Blazor forms overview.

All input and permissions must be validated on the server at the time of performing a given action to ensure that the data is valid and accurate at that time and that the user is allowed to perform the action. This approach is consistent with the security guidance provided for interactive server-side rendering.

Session management

Session management is handled by the framework. The framework uses a session cookie to identify the user session. The session cookie is protected using the ASP.NET Core Data Protection APIs. The session cookie isn't accessible to JavaScript code running on the browser and it can't be easily guessed or manipulated by a user.

With regard to other session data, such as data stored within services, the session data should be stored within scoped services, as scoped services are unique per a given user session, as opposed to singleton services which are shared across all user sessions in a given process instance.

When it comes to SSR, there's not much difference between scoped and transient services in most cases, as the lifetime of the service is limited to a single request. There's a difference in two scenarios:

- If the service is injected in more than one location or at different times during the request.
- If the service might be used in an interactive server context, where it survives multiple renders and its fundamental that the service is scoped to the user session.

Error handling and logging

The framework provides built-in logging for the app at the framework level. The framework logs important events, such as when the antiforgery token for a form fails to validate, when a root component starts to render, and when an action is dispatched. The app is responsible for logging any other events that might be important to record.

The framework provides built-in error handling for the app at the framework level. The framework handles errors that happen during the rendering of a component and either uses the error boundary mechanism to display a friendly error message or allows the error to bubble up to the exception handling middleware, which is configured to render the error page.

Errors that occur during streaming rendering after the response has started to be sent to the client are displayed in the final response as a generic error message. Details about the cause of the error are only included during development.

ASP.NET Core Data Protection

The framework offers mechanisms for protecting sensitive information for a given user session and ensures that the built-in components use these mechanisms to protect sensitive information, such as protecting user identity when using cookie authentication. Outside of scenarios handled by the framework, developer code is responsible for protecting other app-specific information. The most common way of doing this is via the ASP.NET Core Data Protection (DP) APIs or any other form of encryption. As a general rule, the app is responsible for:

- Making sure that a user can't inspect or modify the private information of another user.
- Making sure that a user can't modify user data of another user, such as an internal identifier.

With regard to DP, you must clearly understand where the code is executing. For the static server-side rendering (static SSR) and interactive server-side rendering (interactive SSR), code is stored on the server and never reaches the client. For the Interactive WebAssembly render mode, the app code *always reaches the client*, which means that any sensitive information stored in the app code is available to anyone with access to the app. Obfuscation and other similar technique to "protect" the code isn't effective. Once the code reaches the client, it can be reverse-engineered to extract the sensitive information.

Denial of service

At the server level, the framework provides limits on request/response parameters, such as the maximum size of the request and the header size. In regard to app code, Blazor's form mapping system defines limits similar to those defined by the MVC model binding system:

- Limit on the maximum number of errors.
- Limit on the maximum recursion depth for the binder.
- Limit on the maximum number of elements bound in a collection.

In addition, there are limits defined for the form, such as the maximum form key size and value size and the maximum number of entries.

In general, the app must evaluate when there's a chance that a request triggers an asymmetric amount of work by the server. Examples of this include when the user sends a request parameterized by N and the server performs an operation in response that is N times as expensive, where N is a parameter that a user controls and can grow indefinitely. Normally, the app must either impose a limit on the maximum N that it's willing to process or ensure that any operation is either less, equal, or more expensive than the request by a constant factor.

This aspect has more to do with the difference in growth between the work the client performs and the work the server performs than with a specific 1→N comparison. For example, a client might submit a work item (inserting elements into a list) that takes N units of time to perform, but the server needs N^2 to process (because it might be doing something very naive). It's the difference between N and N^2 that matters.

As such, there's a limit on how much work the server must be willing to do, which is specific to the app. This aspect applies to server-side workloads, since the resources are on the server, but doesn't necessarily apply to WebAssembly workloads on the client in most cases.

The other important aspect is that this isn't only reserved to CPU time. It also applies to any resources, such as memory, network, and space on disk.

For WebAssembly workloads, there's usually little concern over the amount of work the client performs, since the client is normally limited by the resources available on the client. However, there are some scenarios where the client might be impacted, if for example, an app displays data from other users and one user is capable of adding data to the system that forces the clients that display the data to perform an amount of work that isn't proportional to the amount of data added by the user.

Recommended (non-exhaustive) check list

- Ensure that the user is authenticated and authorized to access the app and the resources it exposes.
- Validate and sanitize all input coming from a client before using it.
- Properly manage user sessions to ensure that state isn't mistakenly shared across users.
- Handle and log errors properly to avoid exposing sensitive information.
- Log important events in the app to identify potential issues and audit actions performed by users.
- Protect sensitive information using the ASP.NET Core Data Protection APIs or one of the available components (Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage, PersistentComponentState).
- Ensure that the app understands the resources that can be consumed by a given request and has limits in place to avoid denial of service attacks.

Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering

Article • 11/19/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to mitigate security threats in interactive server-side Blazor.

Apps adopt a *stateful* data processing model, where the server and client maintain a long-lived relationship. The persistent state is maintained by a circuit, which can span connections that are also potentially long-lived.

When a user visits a site, the server creates a circuit in the server's memory. The circuit indicates to the browser what content to render and responds to events, such as when the user selects a button in the UI. To perform these actions, a circuit invokes JavaScript functions in the user's browser and .NET methods on the server. This two-way JavaScript-based interaction is referred to as JavaScript interop (JS interop).

Because JS interop occurs over the Internet and the client uses a remote browser, apps share most web app security concerns. This topic describes common threats to server-side Blazor apps and provides threat mitigation guidance focused on Internet-facing apps.

In constrained environments, such as inside corporate networks or intranets, some of the mitigation guidance either:

- Doesn't apply in the constrained environment.
- Isn't worth the cost to implement because the security risk is low in a constrained environment.

Shared state

Server-side Blazor apps live in server memory, and multiple app sessions are hosted within the same process. For each app session, Blazor starts a circuit with its own

dependency injection container scope, thus scoped services are unique per Blazor session.

> ⚠ **Warning**
>
> We don't recommend apps on the same server share state using singleton services unless extreme care is taken, as this can introduce security vulnerabilities, such as leaking user state across circuits.

You can use stateful singleton services in Blazor apps if they're specifically designed for it. For example, use of a singleton memory cache is acceptable because a memory cache requires a key to access a given entry. Assuming users don't have control over the cache keys that are used with the cache, state stored in the cache doesn't leak across circuits.

For general guidance on state management, see ASP.NET Core Blazor state management.

`IHttpContextAccessor` / `HttpContext` in Razor components

IHttpContextAccessor must be avoided with interactive rendering because there isn't a valid `HttpContext` available.

IHttpContextAccessor can be used for components that are statically rendered on the server. **However, we recommend avoiding it if possible.**

HttpContext can be used as a cascading parameter only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

C#

```
[CascadingParameter]
public HttpContext? HttpContext { get; set; }
```

For scenarios where the HttpContext is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Resource exhaustion

Resource exhaustion can occur when a client interacts with the server and causes the server to consume excessive resources. Excessive resource consumption primarily affects:

- CPU
- Memory
- Client connections

Denial of Service (DoS) attacks usually seek to exhaust an app or server's resources. However, resource exhaustion isn't necessarily the result of an attack on the system. For example, finite resources can be exhausted due to high user demand. DoS is covered further in the DoS section.

Resources external to the Blazor framework, such as databases and file handles (used to read and write files), may also experience resource exhaustion. For more information, see ASP.NET Core Best Practices.

CPU

CPU exhaustion can occur when one or more clients force the server to perform intensive CPU work.

For example, consider an app that calculates a *Fibonnacci number*. A Fibonnacci number is produced from a Fibonnacci sequence, where each number in the sequence is the sum of the two preceding numbers. The amount of work required to reach the answer depends on the length of the sequence and the size of the initial value. If the app doesn't place limits on a client's request, the CPU-intensive calculations may dominate the CPU's time and diminish the performance of other tasks. Excessive resource consumption is a security concern impacting availability.

CPU exhaustion is a concern for all public-facing apps. In regular web apps, requests and connections time out as a safeguard, but Blazor apps don't provide the same safeguards. Blazor apps must include appropriate checks and limits before performing potentially CPU-intensive work.

Memory

Memory exhaustion can occur when one or more clients force the server to consume a large amount of memory.

For example, consider an app with a component that accepts and displays a list of items. If the Blazor app doesn't place limits on the number of items allowed or the number of items rendered back to the client, the memory-intensive processing and rendering may dominate the server's memory to the point where performance of the server suffers. The server may crash or slow to the point that it appears to have crashed.

Consider the following scenario for maintaining and displaying a list of items that pertain to a potential memory exhaustion scenario on the server:

- The items in a `List<T>` property or field use the server's memory. If the app allows the list of items to grow unbounded, there's a risk of the server running out of memory. Running out of memory causes the current session to end (crash) and all of the concurrent sessions in that server instance receive an out-of-memory exception. To prevent this scenario from occurring, the app must use a data structure that imposes an item limit on concurrent users.
- If a paging scheme isn't used for rendering, the server uses additional memory for objects that aren't visible in the UI. Without a limit on the number of items, memory demands may exhaust the available server memory. To prevent this scenario, use one of the following approaches:
  - Use paginated lists when rendering.
  - Only display the first 100 to 1,000 items and require the user to enter search criteria to find items beyond the items displayed.
  - For a more advanced rendering scenario, implement lists or grids that support *virtualization*. Using virtualization, lists only render a subset of items currently visible to the user. When the user interacts with the scrollbar in the UI, the component renders only those items required for display. The items that aren't currently required for display can be held in secondary storage, which is the ideal approach. Undisplayed items can also be held in memory, which is less ideal.

> ⓘ **Note**
>
> Blazor has built-in support for virtualization. For more information, see ASP.NET Core Razor component virtualization.

Blazor apps offer a similar programming model to other UI frameworks for stateful apps, such as WPF, Windows Forms, or Blazor WebAssembly. The main difference is that in several of the UI frameworks the memory consumed by the app belongs to the client and only affects that individual client. For example, a Blazor WebAssembly app runs entirely on the client and only uses client memory resources. For a server-side Blazor

app, the memory consumed by the app belongs to the server and is shared among clients on the server instance.

Server-side memory demands are a consideration for all server-side Blazor apps. However, most web apps are stateless, and the memory used while processing a request is released when the response is returned. As a general recommendation, don't permit clients to allocate an unbound amount of memory as in any other server-side app that persists client connections. The memory consumed by a server-side Blazor app persists for a longer time than a single request.

> ⓘ **Note**
>
> During development, a profiler can be used or a trace captured to assess memory demands of clients. A profiler or trace won't capture the memory allocated to a specific client. To capture the memory use of a specific client during development, capture a dump and examine the memory demand of all the objects rooted at a user's circuit.

Client connections

Connection exhaustion can occur when one or more clients open too many concurrent connections to the server, preventing other clients from establishing new connections.

Blazor clients establish a single connection per session and keep the connection open for as long as the browser window is open. Given the persistent nature of the connections and the stateful nature of server-side Blazor apps, connection exhaustion is a greater risk to availability of the app.

There's no limit on the number of connections per user for an app. If the app requires a connection limit, take one or more of the following approaches:

- Require authentication, which naturally limits the ability of unauthorized users to connect to the app. For this scenario to be effective, users must be prevented from provisioning new users on demand.
- Limit the number of connections per user. Limiting connections can be accomplished via the following approaches. Exercise care to allow legitimate users to access the app (for example, when a connection limit is established based on the client's IP address).
  - Application level
    - Endpoint routing extensibility.

        - Require authentication to connect to the app and keep track of the active sessions per user.
        - Reject new sessions upon reaching a limit.
        - Proxy WebSocket connections to an app through the use of a proxy, such as the Azure SignalR Service that multiplexes connections from clients to an app. This provides an app with greater connection capacity than a single client can establish, preventing a client from exhausting the connections to the server.
    - Server level
        - Use a proxy/gateway in front of the app. For example, Azure Application Gateway is a web traffic (OSI layer 7) load balancer that enables you to manage traffic to your web applications. For more information, see Overview of WebSocket support in Application Gateway.
        - Although Long Polling is supported for Blazor apps, which would permit the adoption of Azure Front Door, WebSockets is the recommended transport protocol. As of September, 2024, Azure Front Door doesn't support WebSockets, but support for WebSockets is under consideration. For more information, see Support WebSocket connections on Azure Front Door.

Denial of Service (DoS) attacks

Denial of Service (DoS) attacks involve a client causing the server to exhaust one or more of its resources making the app unavailable. Blazor apps include default limits and rely on other ASP.NET Core and SignalR limits that are set on CircuitOptions to protect against DoS attacks:

- CircuitOptions.DisconnectedCircuitMaxRetained
- CircuitOptions.DisconnectedCircuitRetentionPeriod
- CircuitOptions.JSInteropDefaultCallTimeout
- CircuitOptions.MaxBufferedUnacknowledgedRenderBatches
- HubConnectionContextOptions.MaximumReceiveMessageSize

For more information and configuration coding examples, see the following articles:

- ASP.NET Core Blazor SignalR guidance
- ASP.NET Core SignalR configuration

Interactions with the browser (client)

A client interacts with the server through JS interop event dispatching and render completion. JS interop communication goes both ways between JavaScript and .NET:

- Browser events are dispatched from the client to the server in an asynchronous fashion.
- The server responds asynchronously rerendering the UI as necessary.

JavaScript functions invoked from .NET

For calls from .NET methods to JavaScript:

- All invocations have a configurable timeout after which they fail, returning a OperationCanceledException to the caller.
  - There's a default timeout for the calls (CircuitOptions.JSInteropDefaultCallTimeout) of one minute. To configure this limit, see Call JavaScript functions from .NET methods in ASP.NET Core Blazor.
  - A cancellation token can be provided to control the cancellation on a per-call basis. Rely on the default call timeout where possible and time-bound any call to the client if a cancellation token is provided.
- The result of a JavaScript call can't be trusted. The Blazor app client running in the browser searches for the JavaScript function to invoke. The function is invoked, and either the result or an error is produced. A malicious client can attempt to:
  - Cause an issue in the app by returning an error from the JavaScript function.
  - Induce an unintended behavior on the server by returning an unexpected result from the JavaScript function.

Take the following precautions to protect against the preceding scenarios:

- Wrap JS interop calls within try-catch statements to account for errors that might occur during the invocations. For more information, see Handle errors in ASP.NET Core Blazor apps.
- Validate data returned from JS interop invocations, including error messages, before taking any action.

.NET methods invoked from the browser

Don't trust calls from JavaScript to .NET methods. When a .NET method is exposed to JavaScript, consider how the .NET method is invoked:

- Treat any .NET method exposed to JavaScript as you would a public endpoint to the app.
  - Validate input.
    - Ensure that values are within expected ranges.
    - Ensure that the user has permission to perform the action requested.

    - Don't allocate an excessive quantity of resources as part of the .NET method invocation. For example, perform checks and place limits on CPU and memory use.
    - Take into account that static and instance methods can be exposed to JavaScript clients. Avoid sharing state across sessions unless the design calls for sharing state with appropriate constraints.
        - For instance methods exposed through DotNetObjectReference objects that are originally created through dependency injection (DI), the objects should be registered as scoped objects. This applies to any DI service that the app uses.
        - For static methods, avoid establishing state that can't be scoped to the client unless the app is explicitly sharing state by-design across all users on a server instance.
    - Avoid passing user-supplied data in parameters to JavaScript calls. If passing data in parameters is absolutely required, ensure that the JavaScript code handles passing the data without introducing Cross-site scripting (XSS) vulnerabilities. For example, don't write user-supplied data to the DOM by setting the `innerHTML` property of an element. Consider using Content Security Policy (CSP) to disable `eval` and other unsafe JavaScript primitives. For more information, see Enforce a Content Security Policy for ASP.NET Core Blazor.
- Avoid implementing custom dispatching of .NET invocations on top of the framework's dispatching implementation. Exposing .NET methods to the browser is an advanced scenario, not recommended for general Blazor development.

Events

Events provide an entry point to an app. The same rules for safeguarding endpoints in web apps apply to event handling in Blazor apps. A malicious client can send any data it wishes to send as the payload for an event.

For example:

- A change event for a `<select>` could send a value that isn't within the options that the app presented to the client.
- An `<input>` could send any text data to the server, bypassing client-side validation.

The app must validate the data for any event that the app handles. The Blazor framework forms components perform basic validations. If the app uses custom forms components, custom code must be written to validate event data as appropriate.

Events are asynchronous, so multiple events can be dispatched to the server before the app has time to react by producing a new render. This has some security implications to

consider. Limiting client actions in the app must be performed inside event handlers and not depend on the current rendered view state.

Consider a counter component that should allow a user to increment a counter a maximum of three times. The button to increment the counter is conditionally based on the value of `count`:

razor

```
<p>Count: @count</p>

@if (count < 3)
{
    <button @onclick="IncrementCount" value="Increment count" />
}

@code
{
    private int count = 0;

    private void IncrementCount()
    {
        count++;
    }
}
```

A client can dispatch one or more increment events before the framework produces a new render of this component. The result is that the `count` can be incremented *over three times* by the user because the button isn't removed by the UI quickly enough. The correct way to achieve the limit of three `count` increments is shown in the following example:

razor

```
<p>Count: @count</p>

@if (count < 3)
{
    <button @onclick="IncrementCount" value="Increment count" />
}

@code
{
    private int count = 0;

    private void IncrementCount()
    {
        if (count < 3)
        {
            count++;
```

```
        }
    }
}
```

By adding the `if (count < 3) { ... }` check inside the handler, the decision to increment `count` is based on the current app state. The decision isn't based on the state of the UI as it was in the previous example, which might be temporarily stale.

Protect against multiple dispatches

If an event callback invokes a long running operation asynchronously, such as fetching data from an external service or database, consider using a safeguard. The safeguard can prevent the user from enqueueing multiple operations while the operation is in progress with visual feedback. The following component code sets `isLoading` to `true` while `DataService.GetDataAsync` obtains data from the server. While `isLoading` is `true`, the button is disabled in the UI:

razor

```
<button disabled="@isLoading" @onclick="UpdateData">Update</button>

@code {
    private bool isLoading;
    private Data[] data = Array.Empty<Data>();

    private async Task UpdateData()
    {
        if (!isLoading)
        {
            isLoading = true;
            data = await DataService.GetDataAsync(DateTime.Now);
            isLoading = false;
        }
    }
}
```

The safeguard pattern demonstrated in the preceding example works if the background operation is executed asynchronously with the `async`-`await` pattern.

Cancel early and avoid use-after-dispose

In addition to using a safeguard as described in the Protect against multiple dispatches section, consider using a CancellationToken to cancel long-running operations when the component is disposed. This approach has the added benefit of avoiding *use-after-dispose* in components:

razor

```
@implements IDisposable

...

@code {
    private readonly CancellationTokenSource TokenSource =
        new CancellationTokenSource();

    private async Task UpdateData()
    {
        ...

        data = await DataService.GetDataAsync(DateTime.Now,
TokenSource.Token);

        if (TokenSource.Token.IsCancellationRequested)
        {
           return;
        }

        ...
    }

    public void Dispose()
    {
        TokenSource.Cancel();
    }
}
```

Avoid events that produce large amounts of data

Some DOM events, such as `oninput` or `onscroll`, can produce a large amount of data. Avoid using these events in server-side Blazor server.

Additional security guidance

The guidance for securing ASP.NET Core apps apply to server-side Blazor apps and are covered in the following sections of this article:

- Logging and sensitive data
- Protect information in transit with HTTPS
- Cross-site scripting (XSS)
- Cross-origin protection
- Click-jacking
- Open redirects

Logging and sensitive data

JS interop interactions between the client and server are recorded in the server's logs with ILogger instances. Blazor avoids logging sensitive information, such as actual events or JS interop inputs and outputs.

When an error occurs on the server, the framework notifies the client and tears down the session. The client receives a generic error message that can be seen in the browser's developer tools.

The client-side error doesn't include the call stack and doesn't provide detail on the cause of the error, but server logs do contain such information. For development purposes, sensitive error information can be made available to the client by enabling detailed errors.

> ⚠ **Warning**
>
> Exposing error information to clients on the Internet is a security risk that should always be avoided.

Protect information in transit with HTTPS

Blazor uses SignalR for communication between the client and the server. Blazor normally uses the transport that SignalR negotiates, which is typically WebSockets.

Blazor doesn't ensure the integrity and confidentiality of the data sent between the server and the client. Always use HTTPS.

Cross-site scripting (XSS)

Cross-site scripting (XSS) allows an unauthorized party to execute arbitrary logic in the context of the browser. A compromised app could potentially run arbitrary code on the client. The vulnerability could be used to potentially perform a number of malicious actions against the server:

- Dispatch fake/invalid events to the server.
- Dispatch fail/invalid render completions.
- Avoid dispatching render completions.
- Dispatch interop calls from JavaScript to .NET.
- Modify the response of interop calls from .NET to JavaScript.
- Avoid dispatching .NET to JS interop results.

The Blazor framework takes steps to protect against some of the preceding threats:

- Stops producing new UI updates if the client isn't acknowledging render batches. Configured with CircuitOptions.MaxBufferedUnacknowledgedRenderBatches.
- Times out any .NET to JavaScript call after one minute without receiving a response from the client. Configured with CircuitOptions.JSInteropDefaultCallTimeout.
- Performs basic validation on all input coming from the browser during JS interop:
  - .NET references are valid and of the type expected by the .NET method.
  - The data isn't malformed.
  - The correct number of arguments for the method are present in the payload.
  - The arguments or result can be deserialized correctly before invoking the method.
- Performs basic validation in all input coming from the browser from dispatched events:
  - The event has a valid type.
  - The data for the event can be deserialized.
  - There's an event handler associated with the event.

In addition to the safeguards that the framework implements, the app must be coded by the developer to safeguard against threats and take appropriate actions:

- Always validate data when handling events.
- Take appropriate action upon receiving invalid data:
  - Ignore the data and return. This allows the app to continue processing requests.
  - If the app determines that the input is illegitimate and couldn't be produced by legitimate client, throw an exception. Throwing an exception tears down the circuit and ends the session.
- Don't trust the error message provided by render batch completions included in the logs. The error is ***provided by the client*** and can't generally be trusted, as the client might be compromised.
- Don't trust the input on JS interop calls in either direction between JavaScript and .NET methods.
- The app is responsible for validating that the content of arguments and results are valid, even if the arguments or results are correctly deserialized.

For a XSS vulnerability to exist, the app must incorporate user input in the rendered page. Blazor executes a compile-time step where the markup in a `.razor` file is transformed into procedural C# logic. At runtime, the C# logic builds a *render tree* describing the elements, text, and child components. This is applied to the browser's DOM via a sequence of JavaScript instructions (or is serialized to HTML in the case of prerendering):

- User input rendered via normal Razor syntax (for example, `@someStringValue`) doesn't expose a XSS vulnerability because the Razor syntax is added to the DOM via commands that can only write text. Even if the value includes HTML markup, the value is displayed as static text. When prerendering, the output is HTML-encoded, which also displays the content as static text.
- Component authors can author components in C# without using Razor. The component author is responsible for using the correct APIs when emitting output. For example, use `builder.AddContent(0, someUserSuppliedString)` and *not* `builder.AddMarkupContent(0, someUserSuppliedString)`, as the latter could create a XSS vulnerability.

Consider further mitigating XSS vulnerabilities. For example, implement a restrictive Content Security Policy (CSP). For more information, see Enforce a Content Security Policy for ASP.NET Core Blazor.

For more information, see Prevent Cross-Site Scripting (XSS) in ASP.NET Core.

Cross-origin protection

Cross-origin attacks involve a client from a different origin performing an action against the server. The malicious action is typically a GET request or a form POST (Cross-Site Request Forgery, CSRF), but opening a malicious WebSocket is also possible. Blazor apps offer the same guarantees that any other SignalR app using the hub protocol offer:

- Apps can be accessed cross-origin unless additional measures are taken to prevent it. To disable cross-origin access, either disable CORS in the endpoint by adding the CORS Middleware to the pipeline and adding the DisableCorsAttribute to the Blazor endpoint metadata or limit the set of allowed origins by configuring SignalR for Cross-Origin Resource Sharing. For guidance on WebSocket origin restrictions, see WebSockets support in ASP.NET Core.
- If CORS is enabled, extra steps might be required to protect the app depending on the CORS configuration. If CORS is globally enabled, CORS can be disabled for the Blazor SignalR hub by adding the DisableCorsAttribute metadata to the endpoint metadata after calling MapBlazorHub on the endpoint route builder.

For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Click-jacking

Click-jacking involves rendering a site as an `<iframe>` inside a site from a different origin in order to trick the user into performing actions on the site under attack.

To protect an app from rendering inside of an `<iframe>`, use Content Security Policy (CSP) and the `X-Frame-Options` header.

For more information, see the following resources:

- Enforce a Content Security Policy for ASP.NET Core Blazor
- MDN web docs: X-Frame-Options

Open redirects

When an app session starts, the server performs basic validation of the URLs sent as part of starting the session. The framework checks that the base URL is a parent of the current URL before establishing the circuit. No additional checks are performed by the framework.

When a user selects a link on the client, the URL for the link is sent to the server, which determines what action to take. For example, the app may perform a client-side navigation or indicate to the browser to go to the new location.

Components can also trigger navigation requests programmatically through the use of NavigationManager. In such scenarios, the app might perform a client-side navigation or indicate to the browser to go to the new location.

Components must:

- Avoid using user input as part of the navigation call arguments.
- Validate arguments to ensure that the target is allowed by the app.

Otherwise, a malicious user can force the browser to go to a cyberattacker-controlled site. In this scenario, the cyberattacker tricks the app into using some user input as part of the invocation of the NavigationManager.NavigateTo method.

This advice also applies when rendering links as part of the app:

- If possible, use relative links.
- Validate that absolute link destinations are valid before including them in a page.

For more information, see Prevent open redirect attacks in ASP.NET Core.

Security checklist

The following list of security considerations isn't comprehensive:

- Validate arguments from events.

- Validate inputs and results from JS interop calls.
- Avoid using (or validate beforehand) user input for .NET to JS interop calls.
- Prevent the client from allocating an unbound amount of memory.
  - Data within the component.
  - DotNetObjectReference objects returned to the client.
- Protect against multiple dispatches.
- Cancel long-running operations when the component is disposed.
- Avoid events that produce large amounts of data.
- Avoid using user input as part of calls to NavigationManager.NavigateTo and validate user input for URLs against a set of allowed origins first if unavoidable.
- Don't make authorization decisions based on the state of the UI but only from component state.
- Consider using Content Security Policy (CSP) to protect against XSS attacks. For more information, see Enforce a Content Security Policy for ASP.NET Core Blazor.
- Consider using CSP and X-Frame-Options to protect against click-jacking.
- Ensure CORS settings are appropriate when enabling CORS or explicitly disable CORS for Blazor apps.
- Test to ensure that the server-side limits for the Blazor app provide an acceptable user experience without unacceptable levels of risk.

Account confirmation and password recovery in ASP.NET Core Blazor

Article • 11/19/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure an ASP.NET Core Blazor Web App with email confirmation and password recovery.

> **Note**
>
> This article only applies to Blazor Web Apps. To implement email confirmation and password recovery for standalone Blazor WebAssembly apps with ASP.NET Core Identity, see Account confirmation and password recovery in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

Namespace

The app's namespace used by the example in this article is `BlazorSample`. Update the code examples to use the namespace of your app.

Select and configure an email provider

In this article, Mailchimp's Transactional API is used via Mandrill.net to send email. We recommend using an email service to send email rather than SMTP. SMTP is difficult to configure and secure properly. Whichever email service you use, access their guidance for .NET apps, create an account, configure an API key for their service, and install any NuGet packages required.

Create a class to hold the secret email provider API key. The example in this article uses a class named `AuthMessageSenderOptions` with an `EmailAuthKey` property to hold the key.

`AuthMessageSenderOptions.cs`:

```
C#
```

```
namespace BlazorSample;

public class AuthMessageSenderOptions
{
    public string? EmailAuthKey { get; set; }
}
```

Register the `AuthMessageSenderOptions` configuration instance in the `Program` file:

C#

```
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);
```

Configure a user secret for the provider's security key

If the project has already been initialized for the Secret Manager tool, it will already have an app secrets identifier (`<AppSecretsId>`) in its project file (`.csproj`). In Visual Studio, you can tell if the app secrets ID is present by looking at the **Properties** panel when the project is selected in **Solution Explorer**. If the app hasn't been initialized, execute the following command in a command shell opened to the project's directory. In Visual Studio, you can use the Developer PowerShell command prompt.

.NET CLI

```
dotnet user-secrets init
```

Set the API key with the Secret Manager tool. In the following example, the key name is `EmailAuthKey` to match `AuthMessageSenderOptions.EmailAuthKey`, and the key is represented by the `{KEY}` placeholder. Execute the following command with the API key:

.NET CLI

```
dotnet user-secrets set "EmailAuthKey" "{KEY}"
```

If using Visual Studio, you can confirm the secret is set by right-clicking the server project in **Solution Explorer** and selecting **Manage User Secrets**.

For more information, see Safe storage of app secrets in development in ASP.NET Core.

> ⚠ **Warning**

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

Implement `IEmailSender`

The following example is based on Mailchimp's Transactional API using Mandrill.net. For a different provider, refer to their documentation on how to implement sending an email message.

Add the Mandrill.net NuGet package to the project.

Add the following `EmailSender` class to implement IEmailSender<TUser>. In the following example, `ApplicationUser` is an IdentityUser. The message HTML markup can be further customized. As long as the `message` passed to `MandrillMessage` starts with the `<` character, the Mandrill.net API assumes that the message body is composed in HTML.

`Components/Account/EmailSender.cs`:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using Mandrill;
using Mandrill.Model;
using BlazorSample.Data;

namespace BlazorSample.Components.Account;

public class EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor,
    ILogger<EmailSender> logger) : IEmailSender<ApplicationUser>
{
    private readonly ILogger logger = logger;

    public AuthMessageSenderOptions Options { get; } =
optionsAccessor.Value;

    public Task SendConfirmationLinkAsync(AppUser user, string email,
```

```
        string confirmationLink) => SendEmailAsync(email, "Confirm your
email",
        "<html lang=\"en\"><head></head><body>Please confirm your account by
" +
        $"<a href='{confirmationLink}'>clicking here</a>.</body></html>");

    public Task SendPasswordResetLinkAsync(AppUser user, string email,
        string resetLink) => SendEmailAsync(email, "Reset your password",
        "<html lang=\"en\"><head></head><body>Please reset your password by
" +
        $"<a href='{resetLink}'>clicking here</a>.</body></html>");

    public Task SendPasswordResetCodeAsync(AppUser user, string email,
        string resetCode) => SendEmailAsync(email, "Reset your password",
        "<html lang=\"en\"><head></head><body>Please reset your password " +
        $"using the following code:<br>{resetCode}</body></html>");

    public async Task SendEmailAsync(string toEmail, string subject, string
message)
    {
        if (string.IsNullOrEmpty(Options.EmailAuthKey))
        {
            throw new Exception("Null EmailAuthKey");
        }

        await Execute(Options.EmailAuthKey, subject, message, toEmail);
    }

    public async Task Execute(string apiKey, string subject, string message,
        string toEmail)
    {
        var api = new MandrillApi(apiKey);
        var mandrillMessage = new MandrillMessage("sarah@contoso.com",
toEmail,
            subject, message);
        await api.Messages.SendAsync(mandrillMessage);

        logger.LogInformation("Email to {EmailAddress} sent!", toEmail);
    }
}
```

> ⓘ Note
>
> Body content for messages might require special encoding for the email service provider. If links in the message body can't be followed in the email message, consult the service provider's documentation to troubleshoot the problem.

Configure app to support email

In the `Program` file, change the email sender implementation to the `EmailSender`:

diff

```
- builder.Services.AddSingleton<IEmailSender<ApplicationUser>,
IdentityNoOpEmailSender>();
+ builder.Services.AddSingleton<IEmailSender<ApplicationUser>, EmailSender>
();
```

Remove the `IdentityNoOpEmailSender` (`Components/Account/IdentityNoOpEmailSender.cs`) from the app.

In the `RegisterConfirmation` component (`Components/Account/Pages/RegisterConfirmation.razor`), remove the conditional block in the `@code` block that checks if the `EmailSender` is an `IdentityNoOpEmailSender`:

diff

```
- else if (EmailSender is IdentityNoOpEmailSender)
- {
-     ...
- }
```

Also in the `RegisterConfirmation` component, remove the Razor markup and code for checking the `emailConfirmationLink` field, leaving just the line instructing the user to check their email ...

diff

```
- @if (emailConfirmationLink is not null)
- {
-     ...
- }
- else
- {
      <p>Please check your email to confirm your account.</p>
- }

@code {
-     private string? emailConfirmationLink;

      ...
}
```

Enable account confirmation after a site has users

Enabling account confirmation on a site with users locks out all the existing users. Existing users are locked out because their accounts aren't confirmed. To work around existing user lockout, use one of the following approaches:

- Update the database to mark all existing users as confirmed.
- Confirm existing users. For example, batch-send emails with confirmation links.

Email and activity timeout

The default inactivity timeout is 14 days. The following code sets the inactivity timeout to five days with sliding expiration:

C#

```
builder.Services.ConfigureApplicationCookie(options => {
    options.ExpireTimeSpan = TimeSpan.FromDays(5);
    options.SlidingExpiration = true;
});
```

Change all ASP.NET Core Data Protection token lifespans

The following code changes Data Protection tokens' timeout period to three hours:

C#

```
builder.Services.Configure<DataProtectionTokenProviderOptions>(options =>
    options.TokenLifespan = TimeSpan.FromHours(3));
```

The built-in Identity user tokens (AspNetCore/src/Identity/Extensions.Core/src/TokenOptions.cs) have a one day timeout .

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list.

For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Change the email token lifespan

The default token lifespan of the Identity user tokens is one day.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

To change the email token lifespan, add a custom DataProtectorTokenProvider<TUser> and DataProtectionTokenProviderOptions.

`CustomTokenProvider.cs`:

C#

```
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;

namespace BlazorSample;

public class CustomEmailConfirmationTokenProvider<TUser>
    : DataProtectorTokenProvider<TUser> where TUser : class
{
    public CustomEmailConfirmationTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<EmailConfirmationTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
        : base(dataProtectionProvider, options, logger)
    {
    }
}

public class EmailConfirmationTokenProviderOptions
    : DataProtectionTokenProviderOptions
{
    public EmailConfirmationTokenProviderOptions()
    {
        Name = "EmailDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(4);
```

```
    }
}

public class CustomPasswordResetTokenProvider<TUser>
    : DataProtectorTokenProvider<TUser> where TUser : class
{
    public CustomPasswordResetTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<PasswordResetTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
        : base(dataProtectionProvider, options, logger)
    {
    }
}

public class PasswordResetTokenProviderOptions :
    DataProtectionTokenProviderOptions
{
    public PasswordResetTokenProviderOptions()
    {
        Name = "PasswordResetDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(3);
    }
}
```

Configure the services to use the custom token provider in the `Program` file:

C#

```
builder.Services.AddIdentityCore<ApplicationUser>(options =>
    {
        options.SignIn.RequireConfirmedAccount = true;
        options.Tokens.ProviderMap.Add("CustomEmailConfirmation",
            new TokenProviderDescriptor(

typeof(CustomEmailConfirmationTokenProvider<ApplicationUser>)));
        options.Tokens.EmailConfirmationTokenProvider =
            "CustomEmailConfirmation";
    })
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddSignInManager()
    .AddDefaultTokenProviders();

builder.Services
    .AddTransient<CustomEmailConfirmationTokenProvider<ApplicationUser>>();
```

Troubleshoot

If you can't get email working:

- Set a breakpoint in `EmailSender.Execute` to verify `SendEmailAsync` is called.
- Create a console app to send email using code similar to `EmailSender.Execute` to debug the problem.
- Review the account email history pages at the email provider's website.
- Check your spam folder for messages.
- Try another email alias on a different email provider, such as Microsoft, Yahoo, or Gmail.
- Try sending to different email accounts.

Additional resources

- Mandrill.net (GitHub repository)
- Mailchimp developer: Transactional API

Enable QR code generation for TOTP authenticator apps in an ASP.NET Core Blazor Web App

Article • 01/16/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure an ASP.NET Core Blazor Web App for two-factor authentication (2FA) with QR codes generated by Time-based One-time Password Algorithm (TOTP) authenticator apps.

For an introduction to 2FA with TOTP authenticator apps, see Enable QR code generation for TOTP authenticator apps in ASP.NET Core.

The guidance in this article relies upon either creating the app with **Individual Accounts** for the new app's **Authentication type** or scaffolding Identity into an existing app. For guidance on using the .NET CLI instead of Visual Studio for scaffolding Identity into an existing app, see ASP.NET Core code generator tool (`aspnet-codegenerator`).

> **Warning**
>
> TOTP codes should be kept secret because they can be used to authenticate multiple times before they expire.

Adding QR codes to the 2FA configuration page

A QR code generated by the app to set up 2FA with an TOTP authenticator app must be generated by a QR code library.

The guidance in this article uses manuelbl/QrCodeGenerator, but you can use any QR code generation library.

Add a package reference for the Net.Codecrete.QrCodeGenerator NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Open the `EnableAuthenticator` component in the `Components/Account/Pages/Manage` folder. At the top of the file under the `@page` directive, add an `@using` directive for the QrCodeGenerator namespace:

razor

```razor
@using Net.Codecrete.QrCodeGenerator
```

Delete the `<div>` element that contains the QR code instructions and the two `<div>` elements where the QR code should appear and where the QR code data is stored in the page:

diff

```diff
- <div class="alert alert-info">
-     Learn how to <a href="https://go.microsoft.com/fwlink/?
Linkid=852423">enable
-     QR code generation</a>.
- </div>
- <div></div>
- <div data-url="@authenticatorUri"></div>
```

Replace the deleted elements with the following markup:

razor

```razor
<div>
    <svg xmlns="http://www.w3.org/2000/svg" height="300" width="300"
stroke="none"
            version="1.1" viewBox="0 0 50 50">
        <rect width="300" height="300" fill="#ffffff" />
        <path d="@svgGraphicsPath" fill="#000000" />
    </svg>
</div>
```

Just inside the opening `@code` block, change the variable declaration for `authenticatorUri` to `svgGraphicsPath`:

diff

```
- private string? authenticatorUri;
+ private string? svgGraphicsPath;
```

Change the site name in the `GenerateQrCodeUri` method. The default value is `Microsoft.AspNetCore.Identity.UI`. Change the value to a meaningful site name that users can identify easily in their authenticator app. Developers usually set a site name that matches the company's name. We recommend limiting the site name length to 30 characters or less to allow the site name to display on narrow mobile device screens.

In the following example, the default value `Microsoft.AspNetCore.Identity.UI` is changed to the company name `Weyland-Yutani Corporation` (©1986 20th Century Studios *Aliens*).

In the `GenerateQrCodeUri` method:

diff

```
- UrlEncoder.Encode("Microsoft.AspNetCore.Identity.UI"),
+ UrlEncoder.Encode("Weyland-Yutani Corporation"),
```

At the bottom of the `LoadSharedKeyAndQrCodeUriAsync` method, add the var keyword to the line that sets `authenticatorUri`, making it an implicitly-typed local variable:

diff

```
- authenticatorUri = GenerateQrCodeUri(email!, unformattedKey!);
+ var authenticatorUri = GenerateQrCodeUri(email!, unformattedKey!);
```

Add the following lines of code at the bottom of the `LoadSharedKeyAndQrCodeUriAsync` method:

C#

```
var qr = QrCode.EncodeText(authenticatorUri, QrCode.Ecc.Medium);
svgGraphicsPath = qr.ToGraphicsPath();
```

Run the app and ensure that the QR code is scannable and that the code validates.

> ⚠ **Warning**

An ASP.NET Core TOTP code should be kept secret because it can be used to authenticate successfully multiple times before it expires.

`EnableAuthenticator` component in reference source

The `EnableAuthenticator` component can be inspected in reference source:

EnableAuthenticator component in reference source

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Failures due to TOTP time skew

TOTP authentication depends on accurate time keeping on the TOTP authenticator app device and the app's host. TOTP tokens are only valid for 30 seconds. If logins are failing due to rejected TOTP codes, confirm accurate time is maintained, preferably synchronized to an accurate NTP service.

Additional resources

- Using a different QR code library
- TOTP client and server time skew

Enforce a Content Security Policy for ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use a Content Security Policy (CSP) with ASP.NET Core Blazor apps to help protect against Cross-Site Scripting (XSS) attacks.

Cross-Site Scripting (XSS) is a security vulnerability where a cyberattacker places one or more malicious client-side scripts into an app's rendered content. A CSP helps protect against XSS attacks by informing the browser of valid:

- Sources for loaded content, including scripts, stylesheets, images, and plugins.
- Actions taken by a page, specifying permitted URL targets of forms.

To apply a CSP to an app, the developer specifies several CSP content security *directives* in one or more `Content-Security-Policy` headers or `<meta>` tags. For guidance on applying a CSP to an app in C# code at startup, see ASP.NET Core Blazor startup.

Policies are evaluated by the browser while a page is loading. The browser inspects the page's sources and determines if they meet the requirements of the content security directives. When policy directives aren't met for a resource, the browser doesn't load the resource. For example, consider a policy that doesn't allow third-party scripts. When a page contains a `<script>` tag with a third-party origin in the `src` attribute, the browser prevents the script from loading.

CSP is supported in most modern desktop and mobile browsers, including Chrome, Edge, Firefox, Opera, and Safari. CSP is recommended for Blazor apps.

Policy directives

Minimally, specify the following directives and sources for Blazor apps. Add additional directives and sources as needed. The following directives are used in the *Apply the policy* section of this article, where example security policies for Blazor apps are provided:

- base-uri: Restricts the URLs for a page's `<base>` tag. Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
- default-src: Indicates a fallback for source directives that aren't explicitly specified by the policy. Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
- img-src: Indicates valid sources for images.
  - Specify `data:` to permit loading images from `data:` URLs.
  - Specify `https:` to permit loading images from HTTPS endpoints.
- object-src: Indicates valid sources for the `<object>`, `<embed>`, and `<applet>` tags. Specify `none` to prevent all URL sources.
- script-src: Indicates valid sources for scripts.
  - Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
  - In a client-side Blazor app:
    - Specify wasm-unsafe-eval to permit the client-side Blazor Mono runtime to function.
    - Specify any additional hashes to permit your required *non-framework scripts* to load.
  - In a server-side Blazor app, specify hashes to permit required scripts to load.
- style-src: Indicates valid sources for stylesheets.
  - Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
  - If the app uses inline styles, specify `unsafe-inline` to allow the use of your inline styles.
- upgrade-insecure-requests: Indicates that content URLs from insecure (HTTP) sources should be acquired securely over HTTPS.

The preceding directives are supported by all browsers except Microsoft Internet Explorer.

To obtain SHA hashes for additional inline scripts:

- Apply the CSP shown in the *Apply the policy* section.
- Access the browser's developer tools console while running the app locally. The browser calculates and displays hashes for blocked scripts when a CSP header or `meta` tag is present.
- Copy the hashes provided by the browser to the `script-src` sources. Use single quotes around each hash.

For a Content Security Policy Level 2 browser support matrix, see Can I use: Content Security Policy Level 2.

Apply the policy

Use a `<meta>` tag to apply the policy:

- Set the value of the `http-equiv` attribute to `Content-Security-Policy`.
- Place the directives in the `content` attribute value. Separate directives with a semicolon (`;`).
- Always place the `meta` tag in the <head> content.

The following sections show example policies. These examples are versioned with this article for each release of Blazor. To use a version appropriate for your release, select the document version with the **Version** dropdown selector on this webpage.

Server-side Blazor apps

In the <head> content, apply the directives described in the *Policy directives* section:

HTML

```
<meta http-equiv="Content-Security-Policy"
      content="base-uri 'self';
               default-src 'self';
               img-src data: https:;
               object-src 'none';
               script-src 'self';
               style-src 'self';
               upgrade-insecure-requests;">
```

Add additional `script-src` and `style-src` hashes as required by the app. During development, use an online tool or browser developer tools to have the hashes calculated for you. For example, the following browser tools console error reports the hash for a required script not covered by the policy:

> Refused to execute inline script because it violates the following Content Security Policy directive: " ... ". Either the 'unsafe-inline' keyword, a hash ('sha256-v8v3RKRPmN4odZ1CWM5gw80QKPCCWMcpNeOmimNL2AA='), or a nonce ('nonce-...') is required to enable inline execution.

The particular script associated with the error is displayed in the console next to the error.

Client-side Blazor apps

In the <head> content, apply the directives described in the *Policy directives* section:

HTML

```
<meta http-equiv="Content-Security-Policy"
      content="base-uri 'self';
               default-src 'self';
               img-src data: https:;
               object-src 'none';
               script-src 'self'
                          'wasm-unsafe-eval';
               style-src 'self';
               upgrade-insecure-requests;">
```

Add additional `script-src` and `style-src` hashes as required by the app. During development, use an online tool or browser developer tools to have the hashes calculated for you. For example, the following browser tools console error reports the hash for a required script not covered by the policy:

> Refused to execute inline script because it violates the following Content Security Policy directive: " ... ". Either the 'unsafe-inline' keyword, a hash ('sha256-v8v3RKRPmN4odZ1CWM5gw80QKPCCWMcpNeOmimNL2AA='), or a nonce ('nonce-...') is required to enable inline execution.

The particular script associated with the error is displayed in the console next to the error.

Apply a CSP in non-`Development` environments

When a CSP is applied to a Blazor app's `<head>` content, it interferes with local testing in the `Development` environment. For example, Browser Link and the browser refresh script fail to load. The following examples demonstrate how to apply the CSP's `<meta>` tag in non-`Development` environments.

> **Note**
>
> The examples in this section don't show the full `<meta>` tag for the CSPs. The complete `<meta>` tags are found in the subsections of the Apply the policy section earlier in this article.

Three general approaches are available:

- Apply the CSP via the `App` component, which applies the CSP to all layouts of the app.
- If you need to apply CSPs to different areas of the app, for example a custom CSP for only the admin pages, apply the CSPs on a per-layout basis using the `<HeadContent>` tag. For complete effectiveness, every app layout file must adopt the approach.
- The hosting service or server can provide a CSP via a Content-Security-Policy header added an app's outgoing responses. Because this approach varies by hosting service or server, it isn't addressed in the following examples. If you wish to adopt this approach, consult the documentation for your hosting service provider or server.

Blazor Web App approaches

In the `App` component (`Components/App.razor`), inject IHostEnvironment:

razor

```
@inject IHostEnvironment Env
```

In the `App` component's `<head>` content, apply the CSP when not in the `Development` environment:

razor

```
@if (!Env.IsDevelopment())
{
    <meta ...>
}
```

Alternatively, apply CSPs on a per-layout basis in the `Components/Layout` folder, as the following example demonstrates. Make sure that every layout specifies a CSP.

razor

```
@inject IHostEnvironment Env

@if (!Env.IsDevelopment())
{
    <HeadContent>
        <meta ...>
    </HeadContent>
}
```

Blazor WebAssembly app approaches

In the `App` component (`App.razor`), inject IWebAssemblyHostEnvironment:

```razor
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
@inject IWebAssemblyHostEnvironment Env
```

In the `App` component's `<head>` content, apply the CSP when not in the `Development` environment:

```razor
@if (!Env.IsDevelopment())
{
    <HeadContent>
        <meta ...>
    </HeadContent>
}
```

Alternatively, use the preceding code but apply CSPs on a per-layout basis in the `Layout` folder. Make sure that every layout specifies a CSP.

Meta tag limitations

A `<meta>` tag policy doesn't support the following directives:

- frame-ancestors
- report-to
- report-uri
- sandbox

To support the preceding directives, use a header named `Content-Security-Policy`. The directive string is the header's value.

Test a policy and receive violation reports

Testing helps confirm that third-party scripts aren't inadvertently blocked when building an initial policy.

To test a policy over a period of time without enforcing the policy directives, set the `<meta>` tag's `http-equiv` attribute or header name of a header-based policy to `Content-`

`Security-Policy-Report-Only`. Failure reports are sent as JSON documents to a specified URL. For more information, see MDN web docs: Content-Security-Policy-Report-Only.

For reporting on violations while a policy is active, see the following articles:

- report-to
- report-uri

Although `report-uri` is no longer recommended for use, both directives should be used until `report-to` is supported by all of the major browsers. Don't exclusively use `report-uri` because support for `report-uri` is subject to being dropped *at any time* from browsers. Remove support for `report-uri` in your policies when `report-to` is fully supported. To track adoption of `report-to`, see Can I use: report-to.

Test and update an app's policy every release.

Troubleshoot

- Errors appear in the browser's developer tools console. Browsers provide information about:
  - Elements that don't comply with the policy.
  - How to modify the policy to allow for a blocked item.
- A policy is only completely effective when the client's browser supports all of the included directives. For a current browser support matrix, see Can I use: Content-Security-Policy.

Additional resources

- Apply a CSP in C# code at startup
- MDN web docs: Content Security Policy (CSP)
- MDN web docs: Content-Security-Policy response header
- Content Security Policy Level 2
- Google CSP Evaluator

EU General Data Protection Regulation (GDPR) support in ASP.NET Core Blazor

Article • 01/16/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to implement support for EU General Data Protection Regulation (GDPR) requirements.

In the `Program` file:

- Add CookiePolicyOptions configuration to require user consent for non-essential cookies and set the same-site policy to none. For more information, see Work with SameSite cookies in ASP.NET Core.
- Add the default implementation for the IHttpContextAccessor service by calling AddHttpContextAccessor.

C#

```
builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.CheckConsentNeeded = context => true;

    options.MinimumSameSitePolicy = SameSiteMode.None;
});

builder.Services.AddHttpContextAccessor();
```

In the `Program` file before the call to MapRazorComponents, add Cookie Policy Middleware by calling UseCookiePolicy:

C#

```
app.UseCookiePolicy();
```

Add the following `CookieConsent` component to handle cookie policy consent.

The component uses a collocated JavaScript file, named `CookieConsent.razor.js`, to load a module. Confirm or adjust the path to the collocated file in the

`OnAfterRenderAsync` method. The following component assumes that the component and its companion JavaScript file are in the `Components` folder of the app.

`CookieConsent.razor`:

razor

```
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Http
@implements IAsyncDisposable
@inject IHttpContextAccessor Http
@inject IJSRuntime JS

@if (showBanner)
{
    <div id="cookieConsent" class="alert alert-info alert-dismissible fade
show"
        role="alert">
        Use this space to summarize your privacy and cookie use policy.
        <a href="/privacy">Privacy Policy</a>
        <button type="button" @onclick="AcceptPolicy" class="accept-policy
close"
            data-bs-dismiss="alert" aria-label="Close"
            data-cookie-string="@cookieString">
            Accept
        </button>
    </div>
}
@code {
    private IJSObjectReference? module;
    private ITrackingConsentFeature? consentFeature;
    private bool showBanner;
    private string? cookieString;

    protected override void OnInitialized()
    {
        consentFeature =
Http.HttpContext?.Features.Get<ITrackingConsentFeature>();
        showBanner = !consentFeature?.CanTrack ?? false;
        cookieString = consentFeature?.CreateConsentCookie();
    }

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSObjectReference>("import",
                "./Components/CookieConsent.razor.js");
        }
    }

    private async Task AcceptPolicy()
    {
```

```
        if (module is not null)
        {
            await module.InvokeVoidAsync("acceptPolicy", cookieString);
            showBanner = false;
        }
    }

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            try
            {
                await module.DisposeAsync();
            }
            catch (JSDisconnectedException)
            {
            }
        }
    }
}
```

Add the following collocated JavaScript file to maintain the `acceptPolicy` function in a JavaScript module.

`CookieConsent.razor.js`:

JavaScript

```
export function acceptPolicy(cookieString) {
  document.cookie = cookieString;
}
```

Within `<main>` Razor markup of the `MainLayout` component (`MainLayout.razor`), add the `CookieConsent` component:

razor

```
<CookieConsent />
```

Customize the cookie consent value

Specify the cookie consent value by assigning a custom string to CookiePolicyOptions.ConsentCookieValue. The following example changes the default value of "`yes`" to "`true`":

C#

```
options.ConsentCookieValue = "true";
```

Additional resources

- Microsoft Trust Center: Safeguard individual privacy with cloud services from Microsoft: GDPR
- European Commission: Data protection explained

ASP.NET Core server-side and Blazor Web App additional security scenarios

Article • 11/19/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to configure server-side Blazor for additional security scenarios, including how to pass tokens to a Blazor app.

> **Note**
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from the `string?`, `TodoItem[]?`, `WeatherForecast[]?`, and `IEnumerable<GitHubBranch>?` types in the article's examples.

Pass tokens to a server-side Blazor app

Updating this section for Blazor Web Apps is pending Update section on passing tokens in Blazor Web Apps (dotnet/AspNetCore.Docs #31691). For more information, see Problem providing Access Token to HttpClient in Interactive Server mode (dotnet/aspnetcore #52390).

For Blazor Server, view the 7.0 version of this article section.

Set the authentication scheme

For an app that uses more than one Authentication Middleware and thus has more than one authentication scheme, the scheme that Blazor uses can be explicitly set in the endpoint configuration of the `Program` file. The following example sets the OpenID Connect (OIDC) scheme:

C#

```
using Microsoft.AspNetCore.Authentication.OpenIdConnect;

...

app.MapRazorComponents<App>().RequireAuthorization(
    new AuthorizeAttribute
    {
        AuthenticationSchemes = OpenIdConnectDefaults.AuthenticationScheme
    })
    .AddInteractiveServerRenderMode();
```

Circuit handler to capture users for custom services

Use a CircuitHandler to capture a user from the AuthenticationStateProvider and set the user in a service. If you want to update the user, register a callback to AuthenticationStateChanged and enqueue a Task to obtain the new user and update the service. The following example demonstrates the approach.

In the following example:

- OnConnectionUpAsync is called every time the circuit reconnects, setting the user for the lifetime of the connection. Only the OnConnectionUpAsync method is required unless you implement updates via a handler for authentication changes (`AuthenticationChanged` in the following example).
- OnCircuitOpenedAsync is called to attach the authentication changed handler, `AuthenticationChanged`, to update the user.
- The `catch` block of the `UpdateAuthentication` task takes no action on exceptions because there's no way to report the exceptions at this point in code execution. If an exception is thrown from the task, the exception is reported elsewhere in app.

`UserService.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Components.Server.Circuits;

public class UserService
{
    private ClaimsPrincipal currentUser = new(new ClaimsIdentity());

    public ClaimsPrincipal GetUser() => currentUser;
```

```
    internal void SetUser(ClaimsPrincipal user)
    {
        if (currentUser != user)
        {
            currentUser = user;
        }
    }
}

internal sealed class UserCircuitHandler(
        AuthenticationStateProvider authenticationStateProvider,
        UserService userService)
        : CircuitHandler, IDisposable
{
    public override Task OnCircuitOpenedAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        authenticationStateProvider.AuthenticationStateChanged +=
            AuthenticationChanged;

        return base.OnCircuitOpenedAsync(circuit, cancellationToken);
    }

    private void AuthenticationChanged(Task<AuthenticationState> task)
    {
        _ = UpdateAuthentication(task);

        async Task UpdateAuthentication(Task<AuthenticationState> task)
        {
            try
            {
                var state = await task;
                userService.SetUser(state.User);
            }
            catch
            {
            }
        }
    }

    public override async Task OnConnectionUpAsync(Circuit circuit,
        CancellationToken cancellationToken)
    {
        var state = await
authenticationStateProvider.GetAuthenticationStateAsync();
        userService.SetUser(state.User);
    }

    public void Dispose()
    {
        authenticationStateProvider.AuthenticationStateChanged -=
            AuthenticationChanged;
    }
}
```

In the `Program` file:

C#

```
using Microsoft.AspNetCore.Components.Server.Circuits;
using Microsoft.Extensions.DependencyInjection.Extensions;

...

builder.Services.AddScoped<UserService>();
builder.Services.TryAddEnumerable(
    ServiceDescriptor.Scoped<CircuitHandler, UserCircuitHandler>());
```

Use the service in a component to obtain the user:

razor

```
@inject UserService UserService

<h1>Hello, @(UserService.GetUser().Identity?.Name ?? "world")!</h1>
```

To set the user in middleware for MVC, Razor Pages, and in other ASP.NET Core scenarios, call `SetUser` on the `UserService` in custom middleware after the Authentication Middleware runs, or set the user with an IClaimsTransformation implementation. The following example adopts the middleware approach.

`UserServiceMiddleware.cs`:

C#

```
public class UserServiceMiddleware
{
    private readonly RequestDelegate next;

    public UserServiceMiddleware(RequestDelegate next)
    {
        this.next = next ?? throw new ArgumentNullException(nameof(next));
    }

    public async Task InvokeAsync(HttpContext context, UserService service)
    {
        service.SetUser(context.User);
        await next(context);
    }
}
```

Immediately before the call to `app.MapRazorComponents<App>()` in the `Program` file, call the middleware:

C#

```
app.UseMiddleware<UserServiceMiddleware>();
```

Access `AuthenticationStateProvider` in outgoing request middleware

The AuthenticationStateProvider from a DelegatingHandler for HttpClient created with IHttpClientFactory can be accessed in outgoing request middleware using a circuit activity handler.

> **Note**
>
> For general guidance on defining delegating handlers for HTTP requests by HttpClient instances created using IHttpClientFactory in ASP.NET Core apps, see the following sections of Make HTTP requests using IHttpClientFactory in ASP.NET Core:
>
> - Outgoing request middleware
> - Use DI in outgoing request middleware

The following example uses AuthenticationStateProvider to attach a custom user name header for authenticated users to outgoing requests.

First, implement the `CircuitServicesAccessor` class in the following section of the Blazor dependency injection (DI) article:

Access server-side Blazor services from a different DI scope

Use the `CircuitServicesAccessor` to access the AuthenticationStateProvider in the DelegatingHandler implementation.

`AuthenticationStateHandler.cs`:

C#

```
public class AuthenticationStateHandler(
    CircuitServicesAccessor circuitServicesAccessor)
    : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
```

```
        var authStateProvider = circuitServicesAccessor.Services
            .GetRequiredService<AuthenticationStateProvider>();
        var authState = await
authStateProvider.GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity is not null && user.Identity.IsAuthenticated)
        {
            request.Headers.Add("X-USER-IDENTITY-NAME", user.Identity.Name);
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

In the `Program` file, register the `AuthenticationStateHandler` and add the handler to the IHttpClientFactory that creates HttpClient instances:

C#

```
builder.Services.AddTransient<AuthenticationStateHandler>();

builder.Services.AddHttpClient("HttpMessageHandler")
    .AddHttpMessageHandler<AuthenticationStateHandler>();
```

ASP.NET Core Blazor state management

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes common approaches for maintaining a user's data (state) while they use an app and across browser sessions.

> **Note**
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from types in the article's examples.

Maintain user state

Server-side Blazor is a stateful app framework. Most of the time, the app maintains a connection to the server. The user's state is held in the server's memory in a *circuit*.

Examples of user state held in a circuit include:

- The hierarchy of component instances and their most recent render output in the rendered UI.
- The values of fields and properties in component instances.
- Data held in dependency injection (DI) service instances that are scoped to the circuit.

User state might also be found in JavaScript variables in the browser's memory set via JavaScript interop calls.

If a user experiences a temporary network connection loss, Blazor attempts to reconnect the user to their original circuit with their original state. However, reconnecting a user to their original circuit in the server's memory isn't always possible:

- The server can't retain a disconnected circuit forever. The server must release a disconnected circuit after a timeout or when the server is under memory pressure.

- In multi-server, load-balanced deployment environments, individual servers may fail or be automatically removed when no longer required to handle the overall volume of requests. The original server processing requests for a user may become unavailable when the user attempts to reconnect.
- The user might close and reopen their browser or reload the page, which removes any state held in the browser's memory. For example, JavaScript variable values set through JavaScript interop calls are lost.

When a user can't be reconnected to their original circuit, the user receives a new circuit with an empty state. This is equivalent to closing and reopening a desktop app.

Persist state across circuits

Generally, maintain state across circuits where users are actively creating data, not simply reading data that already exists.

To preserve state across circuits, the app must persist the data to some other storage location than the server's memory. State persistence isn't automatic. You must take steps when developing the app to implement stateful data persistence.

Data persistence is typically only required for high-value state that users expended effort to create. In the following examples, persisting state either saves time or aids in commercial activities:

- Multi-step web forms: It's time-consuming for a user to re-enter data for several completed steps of a multi-step web form if their state is lost. A user loses state in this scenario if they navigate away from the form and return later.
- Shopping carts: Any commercially important component of an app that represents potential revenue can be maintained. A user who loses their state, and thus their shopping cart, may purchase fewer products or services when they return to the site later.

An app can only persist *app state*. UIs can't be persisted, such as component instances and their render trees. Components and render trees aren't generally serializable. To persist UI state, such as the expanded nodes of a tree view control, the app must use custom code to model the behavior of the UI state as serializable app state.

Where to persist state

Common locations exist for persisting state:

- Server-side storage

- URL
- Browser storage
- In-memory state container service

Server-side storage

For permanent data persistence that spans multiple users and devices, the app can use server-side storage. Options include:

- Blob storage
- Key-value storage
- Relational database
- Table storage

After data is saved, the user's state is retained and available in any new circuit.

For more information on Azure data storage options, see the following:

- Azure Databases
- Azure Storage Documentation

URL

For transient data representing navigation state, model the data as a part of the URL. Examples of user state modeled in the URL include:

- The ID of a viewed entity.
- The current page number in a paged grid.

The contents of the browser's address bar are retained:

- If the user manually reloads the page.
- If the web server becomes unavailable, and the user is forced to reload the page in order to connect to a different server.

For information on defining URL patterns with the @page directive, see ASP.NET Core Blazor routing and navigation.

Browser storage

For transient data that the user is actively creating, a commonly used storage location is the browser's localStorage and sessionStorage collections:

- `localStorage` is scoped to the browser's instance. If the user reloads the page or closes and reopens the browser, the state persists. If the user opens multiple browser tabs, the state is shared across the tabs. Data persists in `localStorage` until explicitly cleared. The `localStorage` data for a document loaded in a "private browsing" or "incognito" session is cleared when the last "private" tab is closed.
- `sessionStorage` is scoped to the browser tab. If the user reloads the tab, the state persists. If the user closes the tab or the browser, the state is lost. If the user opens multiple browser tabs, each tab has its own independent version of the data.

Generally, `sessionStorage` is safer to use. `sessionStorage` avoids the risk that a user opens multiple tabs and encounters the following:

- Bugs in state storage across tabs.
- Confusing behavior when a tab overwrites the state of other tabs.

`localStorage` is the better choice if the app must persist state across closing and reopening the browser.

Caveats for using browser storage:

- Similar to the use of a server-side database, loading and saving data are asynchronous.
- The requested page doesn't exist in the browser during prerendering, so local storage isn't available during prerendering.
- Storage of a few kilobytes of data is reasonable to persist for server-side Blazor apps. Beyond a few kilobytes, you must consider the performance implications because the data is loaded and saved across the network.
- Users may view or tamper with the data. ASP.NET Core Data Protection can mitigate the risk. For example, ASP.NET Core Protected Browser Storage uses ASP.NET Core Data Protection.

Third-party NuGet packages provide APIs for working with `localStorage` and `sessionStorage`. It's worth considering choosing a package that transparently uses ASP.NET Core Data Protection. Data Protection encrypts stored data and reduces the potential risk of tampering with stored data. If JSON-serialized data is stored in plain text, users can see the data using browser developer tools and also modify the stored data. Securing trivial data isn't a problem. For example, reading or modifying the stored color of a UI element isn't a significant security risk to the user or the organization. Avoid allowing users to inspect or tamper with *sensitive data*.

ASP.NET Core Protected Browser Storage

ASP.NET Core Protected Browser Storage leverages ASP.NET Core Data Protection for localStorage and sessionStorage.

> **Note**
>
> Protected Browser Storage relies on ASP.NET Core Data Protection and is only supported for server-side Blazor apps.

Save and load data within a component

In any component that requires loading or saving data to browser storage, use the @inject directive to inject an instance of either of the following:

- `ProtectedLocalStorage`
- `ProtectedSessionStorage`

The choice depends on which browser storage location you wish to use. In the following example, `sessionStorage` is used:

razor

```
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedSessionStorage ProtectedSessionStore
```

The `@using` directive can be placed in the app's `_Imports.razor` file instead of in the component. Use of the `_Imports.razor` file makes the namespace available to larger segments of the app or the whole app.

To persist the `currentCount` value in the `Counter` component of an app based on the Blazor project template, modify the `IncrementCount` method to use `ProtectedSessionStore.SetAsync`:

C#

```
private async Task IncrementCount()
{
    currentCount++;
    await ProtectedSessionStore.SetAsync("count", currentCount);
}
```

In larger, more realistic apps, storage of individual fields is an unlikely scenario. Apps are more likely to store entire model objects that include complex state.

`ProtectedSessionStore` automatically serializes and deserializes JSON data to store complex state objects.

In the preceding code example, the `currentCount` data is stored as `sessionStorage['count']` in the user's browser. The data isn't stored in plain text but rather is protected using ASP.NET Core Data Protection. The encrypted data can be inspected if `sessionStorage['count']` is evaluated in the browser's developer console.

To recover the `currentCount` data if the user returns to the `Counter` component later, including if the user is on a new circuit, use `ProtectedSessionStore.GetAsync`:

C#

```
protected override async Task OnInitializedAsync()
{
    var result = await ProtectedSessionStore.GetAsync<int>("count");
    currentCount = result.Success ? result.Value : 0;
}
```

If the component's parameters include navigation state, call `ProtectedSessionStore.GetAsync` and assign a non-`null` result in OnParametersSetAsync, not OnInitializedAsync. OnInitializedAsync is only called once when the component is first instantiated. OnInitializedAsync isn't called again later if the user navigates to a different URL while remaining on the same page. For more information, see ASP.NET Core Razor component lifecycle.

> ⚠ **Warning**
>
> The examples in this section only work if the server doesn't have prerendering enabled. With prerendering enabled, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.
>
> Either disable prerendering or add additional code to work with prerendering. To learn more about writing code that works with prerendering, see the Handle prerendering section.

Handle the loading state

Since browser storage is accessed asynchronously over a network connection, there's always a period of time before the data is loaded and available to a component. For the

best results, render a message while loading is in progress instead of displaying blank or default data.

One approach is to track whether the data is `null`, which means that the data is still loading. In the default `Counter` component, the count is held in an `int`. Make currentCount nullable by adding a question mark (`?`) to the type (`int`):

C#

```
private int? currentCount;
```

Instead of unconditionally displaying the count and **`Increment`** button, display these elements only if the data is loaded by checking HasValue:

razor

```
@if (currentCount.HasValue)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}
```

Handle prerendering

During prerendering:

- An interactive connection to the user's browser doesn't exist.
- The browser doesn't yet have a page in which it can run JavaScript code.

`localStorage` or `sessionStorage` aren't available during prerendering. If the component attempts to interact with storage, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.

One way to resolve the error is to disable prerendering. This is usually the best choice if the app makes heavy use of browser-based storage. Prerendering adds complexity and doesn't benefit the app because the app can't prerender any useful content until `localStorage` or `sessionStorage` are available.

To disable prerendering, indicate the render mode with the `prerender` parameter set to `false` at the highest-level component in the app's component hierarchy that isn't a root

component.

> **Note**
>
> Making a root component interactive, such as the `App` component, isn't supported. Therefore, prerendering can't be disabled directly by the `App` component.

For apps based on the Blazor Web App project template, prerendering is typically disabled where the `Routes` component is used in the `App` component (`Components/App.razor`):

razor

```
<Routes @rendermode="new InteractiveServerRenderMode(prerender: false)" />
```

Also, disable prerendering for the `HeadOutlet` component:

razor

```
<HeadOutlet @rendermode="new InteractiveServerRenderMode(prerender: false)"
/>
```

For more information, see ASP.NET Core Blazor render modes.

When prerendering is disabled, prerendering of <head> content is disabled.

Prerendering might be useful for other pages that don't use `localStorage` or `sessionStorage`. To retain prerendering, defer the loading operation until the browser is connected to the circuit. The following is an example for storing a counter value:

razor

```
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedLocalStorage ProtectedLocalStore

@if (isConnected)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}

@code {
```

```
    private int currentCount;
    private bool isConnected;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            isConnected = true;
            await LoadStateAsync();
            StateHasChanged();
        }
    }

    private async Task LoadStateAsync()
    {
        var result = await ProtectedLocalStore.GetAsync<int>("count");
        currentCount = result.Success ? result.Value : 0;
    }

    private async Task IncrementCount()
    {
        currentCount++;
        await ProtectedLocalStore.SetAsync("count", currentCount);
    }
}
```

Factor out the state preservation to a common location

If many components rely on browser-based storage, implementing state provider code many times creates code duplication. One option for avoiding code duplication is to create a *state provider parent component* that encapsulates the state provider logic. Child components can work with persisted data without regard to the state persistence mechanism.

In the following example of a `CounterStateProvider` component, counter data is persisted to `sessionStorage`:

razor

```
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedSessionStorage ProtectedSessionStore

@if (isLoaded)
{
    <CascadingValue Value="this">
        @ChildContent
    </CascadingValue>
}
else
{
```

```
    <p>Loading...</p>
}

@code {
    private bool isLoaded;

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    public int CurrentCount { get; set; }

    protected override async Task OnInitializedAsync()
    {
        var result = await ProtectedSessionStore.GetAsync<int>("count");
        CurrentCount = result.Success ? result.Value : 0;
        isLoaded = true;
    }

    public async Task SaveChangesAsync()
    {
        await ProtectedSessionStore.SetAsync("count", CurrentCount);
    }
}
```

> ⓘ Note
>
> For more information on RenderFragment, see ASP.NET Core Razor components.

The `CounterStateProvider` component handles the loading phase by not rendering its child content until state loading is complete.

To make the state accessible to all components in an app, wrap the `CounterStateProvider` component around the Router (`<Router>...</Router>`) in the `Routes` component with global interactive server-side rendering (interactive SSR).

In the `App` component (`Components/App.razor`):

razor

```
<Routes @rendermode="InteractiveServer" />
```

In the `Routes` component (`Components/Routes.razor`):

razor

```
<CounterStateProvider>
    <Router ...>
        ...
```

```
    </Router>
</CounterStateProvider>
```

Wrapped components receive and can modify the persisted counter state. The following `Counter` component implements the pattern:

razor

```razor
@page "/counter"

<p>Current count: <strong>@CounterStateProvider?.CurrentCount</strong></p>
<button @onclick="IncrementCount">Increment</button>

@code {
    [CascadingParameter]
    private CounterStateProvider? CounterStateProvider { get; set; }

    private async Task IncrementCount()
    {
        if (CounterStateProvider is not null)
        {
            CounterStateProvider.CurrentCount++;
            await CounterStateProvider.SaveChangesAsync();
        }
    }
}
```

The preceding component isn't required to interact with `ProtectedBrowserStorage`, nor does it deal with a "loading" phase.

To deal with prerendering as described earlier, `CounterStateProvider` can be amended so that all of the components that consume the counter data automatically work with prerendering. For more information, see the Handle prerendering section.

In general, the *state provider parent component* pattern is recommended:

- To consume state across many components.
- If there's just one top-level state object to persist.

To persist many different state objects and consume different subsets of objects in different places, it's better to avoid persisting state globally.

User state created in a Blazor WebAssembly app is held in the browser's memory.

Examples of user state held in browser memory include:

- The hierarchy of component instances and their most recent render output in the rendered UI.

- The values of fields and properties in component instances.
- Data held in dependency injection (DI) service instances.
- Values set through JavaScript interop calls.

When a user closes and reopens their browser or reloads the page, user state held in the browser's memory is lost.

> **Note**
>
> Protected Browser Storage (Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage namespace) relies on ASP.NET Core Data Protection and is only supported for server-side Blazor apps.

Persist state across browser sessions

Generally, maintain state across browser sessions where users are actively creating data, not simply reading data that already exists.

To preserve state across browser sessions, the app must persist the data to some other storage location than the browser's memory. State persistence isn't automatic. You must take steps when developing the app to implement stateful data persistence.

Data persistence is typically only required for high-value state that users expended effort to create. In the following examples, persisting state either saves time or aids in commercial activities:

- Multi-step web forms: It's time-consuming for a user to re-enter data for several completed steps of a multi-step web form if their state is lost. A user loses state in this scenario if they navigate away from the form and return later.
- Shopping carts: Any commercially important component of an app that represents potential revenue can be maintained. A user who loses their state, and thus their shopping cart, may purchase fewer products or services when they return to the site later.

An app can only persist *app state*. UIs can't be persisted, such as component instances and their render trees. Components and render trees aren't generally serializable. To persist UI state, such as the expanded nodes of a tree view control, the app must use custom code to model the behavior of the UI state as serializable app state.

Where to persist state

Common locations exist for persisting state:

- Server-side storage
- URL
- Browser storage
- In-memory state container service

Server-side storage

For permanent data persistence that spans multiple users and devices, the app can use independent server-side storage accessed via a web API. Options include:

- Blob storage
- Key-value storage
- Relational database
- Table storage

After data is saved, the user's state is retained and available in any new browser session.

Because Blazor WebAssembly apps run entirely in the user's browser, they require additional measures to access secure external systems, such as storage services and databases. Blazor WebAssembly apps are secured in the same manner as single-page applications (SPAs). Typically, an app authenticates a user via OAuth /OpenID Connect (OIDC) and then interacts with storage services and databases through web API calls to a server-side app. The server-side app mediates the transfer of data between the Blazor WebAssembly app and the storage service or database. The Blazor WebAssembly app maintains an ephemeral connection to the server-side app, while the server-side app has a persistent connection to storage.

For more information, see the following resources:

- Call a web API from an ASP.NET Core Blazor app
- Secure ASP.NET Core Blazor WebAssembly
- Blazor *Security and Identity* articles

For more information on Azure data storage options, see the following:

- Azure Databases
- Azure Storage Documentation

URL

For transient data representing navigation state, model the data as a part of the URL. Examples of user state modeled in the URL include:

- The ID of a viewed entity.
- The current page number in a paged grid.

The contents of the browser's address bar are retained if the user manually reloads the page.

For information on defining URL patterns with the @page directive, see ASP.NET Core Blazor routing and navigation.

Browser storage

For transient data that the user is actively creating, a commonly used storage location is the browser's localStorage and sessionStorage collections:

- `localStorage` is scoped to the browser's instance. If the user reloads the page or closes and reopens the browser, the state persists. If the user opens multiple browser tabs, the state is shared across the tabs. Data persists in `localStorage` until explicitly cleared. The `localStorage` data for a document loaded in a "private browsing" or "incognito" session is cleared when the last "private" tab is closed.
- `sessionStorage` is scoped to the browser tab. If the user reloads the tab, the state persists. If the user closes the tab or the browser, the state is lost. If the user opens multiple browser tabs, each tab has its own independent version of the data.

> **Note**
>
> `localStorage` and `sessionStorage` can be used in Blazor WebAssembly apps but only by writing custom code or using a third-party package.

Generally, `sessionStorage` is safer to use. `sessionStorage` avoids the risk that a user opens multiple tabs and encounters the following:

- Bugs in state storage across tabs.
- Confusing behavior when a tab overwrites the state of other tabs.

`localStorage` is the better choice if the app must persist state across closing and reopening the browser.

> **Warning**

Users may view or tamper with the data stored in `localStorage` and `sessionStorage`.

In-memory state container service

Nested components typically bind data using *chained bind* as described in ASP.NET Core Blazor data binding. Nested and unnested components can share access to data using a registered in-memory state container. A custom state container class can use an assignable Action to notify components in different parts of the app of state changes. In the following example:

- A pair of components uses a state container to track a property.
- One component in the following example is nested in the other component, but nesting isn't required for this approach to work.

> **ⓘ Important**
>
> The example in this section demonstrates how to create an in-memory state container service, register the service, and use the service in components. The example doesn't persist data without further development. For persistent storage of data, the state container must adopt an underlying storage mechanism that survives when browser memory is cleared. This can be accomplished with `localStorage`/`sessionStorage` or some other technology.

`StateContainer.cs`:

C#

```
public class StateContainer
{
    private string? savedString;

    public string Property
    {
        get => savedString ?? string.Empty;
        set
        {
            savedString = value;
            NotifyStateChanged();
        }
    }

    public event Action? OnChange;
```

```
    private void NotifyStateChanged() => OnChange?.Invoke();
}
```

Client-side apps (`Program` file):

C#

```
builder.Services.AddSingleton<StateContainer>();
```

Server-side apps (`Program` file, ASP.NET Core in .NET 6 or later):

C#

```
builder.Services.AddScoped<StateContainer>();
```

Server-side apps (`Startup.ConfigureServices` of `Startup.cs`, ASP.NET Core earlier than 6.0):

C#

```
services.AddScoped<StateContainer>();
```

`Shared/Nested.razor`:

razor

```
@implements IDisposable
@inject StateContainer StateContainer

<h2>Nested component</h2>

<p>Nested component Property: <b>@StateContainer.Property</b></p>

<p>
    <button @onclick="ChangePropertyValue">
        Change the Property from the Nested component
    </button>
</p>

@code {
    protected override void OnInitialized()
    {
        StateContainer.OnChange += StateHasChanged;
    }

    private void ChangePropertyValue()
    {
        StateContainer.Property =
```

```
            $"New value set in the Nested component: {DateTime.Now}";
    }

    public void Dispose()
    {
        StateContainer.OnChange -= StateHasChanged;
    }
}
```

`StateContainerExample.razor`:

razor

```
@page "/state-container-example"
@implements IDisposable
@inject StateContainer StateContainer

<h1>State Container Example component</h1>

<p>State Container component Property: <b>@StateContainer.Property</b></p>

<p>
    <button @onclick="ChangePropertyValue">
        Change the Property from the State Container Example component
    </button>
</p>

<Nested />

@code {
    protected override void OnInitialized()
    {
        StateContainer.OnChange += StateHasChanged;
    }

    private void ChangePropertyValue()
    {
        StateContainer.Property = "New value set in the State " +
            $"Container Example component: {DateTime.Now}";
    }

    public void Dispose()
    {
        StateContainer.OnChange -= StateHasChanged;
    }
}
```

The preceding components implement IDisposable, and the `OnChange` delegates are unsubscribed in the `Dispose` methods, which are called by the framework when the components are disposed. For more information, see ASP.NET Core Razor component lifecycle.

Additional approaches

When implementing custom state storage, a useful approach is to adopt cascading values and parameters:

- To consume state across many components.
- If there's just one top-level state object to persist.

Troubleshoot

In a custom state management service, a callback invoked outside of Blazor's synchronization context must wrap the logic of the callback in ComponentBase.InvokeAsync to move it onto the renderer's synchronization context.

When the state management service doesn't call StateHasChanged on Blazor's synchronization context, the following error is thrown:

> System.InvalidOperationException: 'The current thread is not associated with the Dispatcher. Use InvokeAsync() to switch execution to the Dispatcher when triggering rendering or component state.'

For more information and an example of how to address this error, see ASP.NET Core Razor component rendering.

Additional resources

- Save app state before an authentication operation (Blazor WebAssembly)
- Managing state via an external server API
  - Call a web API from an ASP.NET Core Blazor app
  - Secure ASP.NET Core Blazor WebAssembly

Debug ASP.NET Core apps

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes how to debug Blazor apps, including debugging Blazor WebAssembly apps with browser developer tools or an integrated development environment (IDE).

Blazor Web Apps can be debugged in Visual Studio or Visual Studio Code.

Blazor WebAssembly apps can be debugged:

- In Visual Studio or Visual Studio Code.
- Using browser developer tools in Chromium-based browsers, including Microsoft Edge, Google Chrome, and Firefox.

Available scenarios for Blazor WebAssembly debugging include:

- Set and remove breakpoints.
- Run the app with debugging support in IDEs.
- Single-step through the code.
- Resume code execution with a keyboard shortcut in IDEs.
- In the *Locals* window, observe the values of local variables.
- See the call stack, including call chains between JavaScript and .NET.
- Use a symbol server for debugging, configured by Visual Studio preferences.

Unsupported scenarios include:

- Debug in non-local scenarios (for example, Windows Subsystem for Linux (WSL) or Visual Studio Codespaces).
- Debug in Firefox from Visual Studio or Visual Studio Code.

Prerequisites

This section explains the prerequisites for debugging.

Browser prerequisites

The latest version of the following browsers:

- Google Chrome
- Microsoft Edge
- Firefox (browser developer tools only)

Ensure that firewalls or proxies don't block communication with the debug proxy (`NodeJS` process). For more information, see the Firewall configuration section.

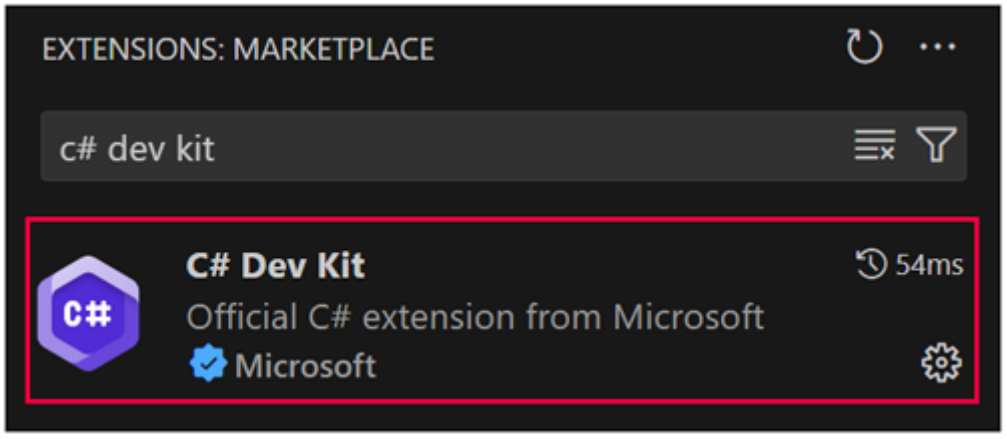
> **Note**
>
> Apple Safari on macOS isn't currently supported.

IDE prerequisites

The latest version of Visual Studio or Visual Studio Code is required.

Visual Studio Code prerequisites

Visual Studio Code requires the C# Dev Kit for Visual Studio Code (Getting Started with C# in VS Code). In the Visual Studio Code Extensions Marketplace, filter the extension list with "`c# dev kit`" to locate the extension:



Installing the C# Dev Kit automatically installs the following additional extensions:

- .NET Install Tool
- C#
- IntelliCode for C# Dev Kit

If you encounter warnings or errors, you can open an issue (microsoft/vscode-dotnettools GitHub repository) describing the problem.

App configuration prerequisites

The guidance in this subsection applies to client-side debugging.

Open the `Properties/launchSettings.json` file of the startup project. Confirm the presence of the following `inspectUri` property in each launch profile of the file's `profiles` node. If the following property isn't present, add it to each profile:

JSON

```
"inspectUri": "{wsProtocol}://{url.hostname}:{url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}"
```

The `inspectUri` property:

- Enables the IDE to detect that the app is a Blazor app.
- Instructs the script debugging infrastructure to connect to the browser through Blazor's debugging proxy.

The placeholder values for the WebSocket protocol (`wsProtocol`), host (`url.hostname`), port (`url.port`), and inspector URI on the launched browser (`browserInspectUri`) are provided by the framework.

Packages

Blazor Web Apps: Microsoft.AspNetCore.Components.WebAssembly.Server: References an internal package (Microsoft.NETCore.BrowserDebugHost.Transport) for assemblies that share the browser debug host.

Standalone Blazor WebAssembly: Microsoft.AspNetCore.Components.WebAssembly.DevServer: Development server for use when building Blazor apps. Calls UseWebAssemblyDebugging internally to add middleware for debugging Blazor WebAssembly apps inside Chromium developer tools.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Debug a Blazor Web App in an IDE

Visual Studio

The example in this section assumes that you've created a Blazor Web App with an interactive render mode of Auto (Server and WebAssembly) and per-component interactivity location.

1. Open the app.
2. Set a breakpoint on the `currentCount++;` line in the `Counter` component (`Pages/Counter.razor`) of the client project (`.Client`).
3. With the server project selected in **Solution Explorer**, press `F5` to run the app in the debugger.
4. In the browser, navigate to `Counter` page at `/counter`. Wait a few seconds for the debug proxy to load and run. Select the **Click me** button to hit the breakpoint.
5. In Visual Studio, inspect the value of the `currentCount` field in the **Locals** window.
6. Press `F5` to continue execution.

Breakpoints can also be hit in the server project in statically-rendered and interactively-rendered server-side components.

1. Stop the debugger.
2. In the server app, open the statically-rendered `Weather` component (`Components/Pages/Weather.razor`) and set a breakpoint anywhere in the `OnInitializedAsync` method.
3. Press `F5` to run the app in the debugger.
4. In the browser, navigate to the `Weather` page at `/weather`. Wait a few seconds for the debug proxy to load and run. Application execution stops at the breakpoint.
5. Press `F5` to continue execution.

Breakpoints are **not** hit during app startup before the debug proxy is running. This includes breakpoints in the `Program` file and breakpoints in the OnInitialized{Async} lifecycle methods of components that are loaded by the first page requested from the app.

Debug a Blazor WebAssembly app in an IDE

Visual Studio

1. Open the app.
2. Set a breakpoint on the `currentCount++;` line in the `Counter` component (`Pages/Counter.razor`).
3. Press `F5` to run the app in the debugger.
4. In the browser, navigate to `Counter` page at `/counter`. Wait a few seconds for the debug proxy to load and run. Select the **Click me** button to hit the breakpoint.
5. In Visual Studio, inspect the value of the `currentCount` field in the **Locals** window.
6. Press `F5` to continue execution.

Breakpoints are **not** hit during app startup before the debug proxy is running. This includes breakpoints in the `Program` file and breakpoints in the OnInitialized{Async} lifecycle methods of components that are loaded by the first page requested from the app.

Attach to an existing Visual Studio Code debugging session

To attach to a running Blazor app, open the `.vscode/launch.json` file and replace the `{URL}` placeholder with the URL where the app is running:

JSON

```
{
  "name": "Attach and Debug",
  "type": "blazorwasm",
  "request": "attach",
  "url": "{URL}"
}
```

Visual Studio Code launch options

The launch configuration options in the following table are supported for the `blazorwasm` debug type (`.vscode/launch.json`).

Expand table

Option	Description
`browser`	The browser to launch for the debugging session. Set to `edge` or `chrome`. Defaults to `edge`.
`cwd`	The working directory to launch the app under.
`request`	Use `launch` to launch and attach a debugging session to a Blazor WebAssembly app or `attach` to attach a debugging session to an already-running app.
`timeout`	The number of milliseconds to wait for the debugging session to attach. Defaults to 30,000 milliseconds (30 seconds).
`trace`	Used to generate logs from the JS debugger. Set to `true` to generate logs.
`url`	The URL to open in the browser when debugging.
`webRoot`	Specifies the absolute path of the web server. Should be set if an app is served from a sub-route.

Debug Blazor WebAssembly with Google Chrome or Microsoft Edge

The guidance in this section applies debugging Blazor WebAssembly apps in:

- ***Google Chrome** running on Windows or macOS.*
- ***Microsoft Edge** running on Windows.*

1. Run the app in a command shell with `dotnet watch` (or `dotnet run`).

2. Launch a browser and navigate to the app's URL.

3. Start remote debugging by pressing:

   - `Shift`+`Alt`+`d` on Windows.
   - `Shift`+`⌘`+`d` on macOS.

   The browser must be running with remote debugging enabled, which isn't the default. If remote debugging is disabled, an **Unable to find debuggable browser tab** error page is rendered with instructions for launching the browser with the debugging port open. Follow the instructions for your browser.

   After following the instructions to enable remote debugging, the app opens in a new browser window. Start remote debugging by pressing the HotKey combination in the new browser window:

- `Shift`+`Alt`+`d` on Windows.
- `Shift`+`⌘`+`d` on macOS.

A new window developer tools browser tab opens showing a ghosted image of the app.

> ⓘ Note
>
> If you followed the instructions to open a new browser tab with remote debugging enabled, you can close the original browser window, leaving the second window open with the first tab running the app and the second tab running the debugger.

4. After a moment, the **Sources** tab shows a list of the app's .NET assemblies and pages.

5. Open the `file://` node. In component code (`.razor` files) and C# code files (`.cs`), breakpoints that you set are hit when code executes in the app's browser tab (the initial tab opened after starting remote debugging). After a breakpoint is hit, single-step (`F10`) through the code or resume (`F8`) code execution normally in the debugging tab.

For Chromium-based browser debugging, Blazor provides a debugging proxy that implements the Chrome DevTools Protocol and augments the protocol with .NET-specific information. When debugging keyboard shortcut is pressed, Blazor points the Chrome DevTools at the proxy. The proxy connects to the browser window you're seeking to debug (hence the need to enable remote debugging).

Debug a Blazor WebAssembly app with Firefox

The guidance in this section applies debugging Blazor WebAssembly apps in Firefox running on Windows.

Debugging a Blazor WebAssembly app with Firefox requires configuring the browser for remote debugging and connecting to the browser using the browser developer tools through the .NET WebAssembly debugging proxy.

> ⓘ Note
>
> Debugging in Firefox from Visual Studio isn't supported at this time.

To debug a Blazor WebAssembly app in Firefox during development:

1. Configure Firefox:
    - Open `about:config` in a new browser tab. Read and dismiss the warning that appears.
    - Enable `devtools.debugger.remote-enabled` by setting its value to `True`.
    - Enable `devtools.chrome.enabled` by setting its value to `True`.
    - Disable `devtools.debugger.prompt-connection` by setting its value to `False`.
2. Close all Firefox instances.
3. Run the app in a command shell with `dotnet watch` (or `dotnet run`).
4. Relaunch the Firefox browser and navigate to the app.
5. Open `about:debugging` in a new browser tab. **Leave this tab open**.
6. Go back to the tab where the app is running. Start remote debugging by pressing `Shift`+`Alt`+`d`.
7. In the `Debugger` tab, open the app source file you wish to debug under the `file://` node and set a breakpoint. For example, set a breakpoint on the `currentCount++;` line in the `IncrementCount` method of the `Counter` component (`Pages/Counter.razor`).
8. Navigate to the `Counter` component page (`/counter`) in the app's browser tab and select the counter button to hit the breakpoint.
9. Press `F5` to continue execution in the debugging tab.

Break on unhandled exceptions

The debugger doesn't break on unhandled exceptions because Blazor catches exceptions that are unhandled by developer code.

To break on unhandled exceptions:

- Open the debugger's exception settings (**Debug** > **Windows** > **Exception Settings**) in Visual Studio.
- Set the following **JavaScript Exceptions** settings:
    - **All Exceptions**
    - **Uncaught Exceptions**

Browser source maps

Browser source maps allow the browser to map compiled files back to their original source files and are commonly used for client-side debugging. However, Blazor doesn't

currently map C# directly to JavaScript/WASM. Instead, Blazor does IL interpretation within the browser, so source maps aren't relevant.

Firewall configuration

If a firewall blocks communication with the debug proxy, create a firewall exception rule that permits communication between the browser and the `NodeJS` process.

> ⚠ **Warning**
>
> Modification of a firewall configuration must be made with care to avoid creating security vulnerabilities. Carefully apply security guidance, follow best security practices, and respect warnings issued by the firewall's manufacturer.
>
> Permitting open communication with the `NodeJS` process:
>
> - Opens up the Node server to any connection, depending on the firewall's capabilities and configuration.
> - Might be risky depending on your network.
> - **Is only recommended on developer machines.**
>
> If possible, only allow open communication with the `NodeJS` process **on trusted or private networks**.

For Windows Firewall configuration guidance, see Create an Inbound Program or Service Rule. For more information, see Windows Defender Firewall with Advanced Security and related articles in the Windows Firewall documentation set.

Troubleshoot

If you're running into errors, the following tips may help:

- Remove breakpoints:
  - Google Chrome: In the **Debugger** tab, open the developer tools in your browser. In the console, execute `localStorage.clear()` to remove any breakpoints.
  - Microsoft Edge: In the **Application** tab, open **Local storage**. Right-click the site and select **Clear**.
- Confirm that you've installed and trusted the ASP.NET Core HTTPS development certificate. For more information, see Enforce HTTPS in ASP.NET Core.

- Visual Studio requires the **Enable JavaScript debugging for ASP.NET (Chrome and Edge)** option in **Tools** > **Options** > **Debugging** > **General**. This is the default setting for Visual Studio. If debugging isn't working, confirm that the option is selected.
- If your environment uses an HTTP proxy, make sure that `localhost` is included in the proxy bypass settings. This can be done by setting the `NO_PROXY` environment variable in either:
  - The `launchSettings.json` file for the project.
  - At the user or system environment variables level for it to apply to all apps. When using an environment variable, restart Visual Studio for the change to take effect.
- Ensure that firewalls or proxies don't block communication with the debug proxy (`NodeJS` process). For more information, see the Firewall configuration section.

Breakpoints in `OnInitialized{Async}` not hit

The Blazor framework's debugging proxy doesn't launch instantly on app startup, so breakpoints in the OnInitialized{Async} lifecycle methods might not be hit. We recommend adding a delay at the start of the method body to give the debug proxy some time to launch before the breakpoint is hit. You can include the delay based on an if compiler directive to ensure that the delay isn't present for a release build of the app.

OnInitialized:

C#

```
protected override void OnInitialized()
{
#if DEBUG
    Thread.Sleep(10000);
#endif

    ...
}
```

OnInitializedAsync:

C#

```
protected override async Task OnInitializedAsync()
{
#if DEBUG
    await Task.Delay(10000);
#endif
```

```
    ...
}
```

Visual Studio (Windows) timeout

If Visual Studio throws an exception that the debug adapter failed to launch mentioning that the timeout was reached, you can adjust the timeout with a Registry setting:

Console

```
VsRegEdit.exe set "<VSInstallFolder>" HKCU JSDebugger\Options\Debugging
"BlazorTimeoutInMilliseconds" dword {TIMEOUT}
```

The `{TIMEOUT}` placeholder in the preceding command is in milliseconds. For example, one minute is assigned as `60000`.

Lazy load assemblies in ASP.NET Core Blazor WebAssembly

Article • 12/02/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Blazor WebAssembly app startup performance can be improved by waiting to load developer-created app assemblies until the assemblies are required, which is called *lazy loading*.

This article's initial sections cover the app configuration. For a working demonstration, see the Complete example section at the end of this article.

This article only applies to Blazor WebAssembly apps. Assembly lazy loading doesn't benefit server-side apps because server-rendered apps don't download assemblies to the client.

Lazy loading shouldn't be used for core runtime assemblies, which might be trimmed on publish and unavailable on the client when the app loads.

File extension placeholder (`{FILE EXTENSION}`) for assembly files

Assembly files use the Webcil packaging format for .NET assemblies with a `.wasm` file extension.

Throughout the article, the `{FILE EXTENSION}` placeholder represents "`wasm`".

Project file configuration

Mark assemblies for lazy loading in the app's project file (`.csproj`) using the `BlazorWebAssemblyLazyLoad` item. Use the assembly name with file extension. The Blazor framework prevents the assembly from loading at app launch.

XML

```
<ItemGroup>
  <BlazorWebAssemblyLazyLoad Include="{ASSEMBLY NAME}.{FILE EXTENSION}" />
</ItemGroup>
```

The `{ASSEMBLY NAME}` placeholder is the name of the assembly, and the `{FILE EXTENSION}` placeholder is the file extension. The file extension is required.

Include one `BlazorWebAssemblyLazyLoad` item for each assembly. If an assembly has dependencies, include a `BlazorWebAssemblyLazyLoad` entry for each dependency.

`Router` component configuration

The Blazor framework automatically registers a singleton service for lazy loading assemblies in client-side Blazor WebAssembly apps, LazyAssemblyLoader. The LazyAssemblyLoader.LoadAssembliesAsync method:

- Uses JS interop to fetch assemblies via a network call.
- Loads assemblies into the runtime executing on WebAssembly in the browser.

Blazor's Router component designates the assemblies that Blazor searches for routable components and is also responsible for rendering the component for the route where the user navigates. The Router component's OnNavigateAsync method is used in conjunction with lazy loading to load the correct assemblies for endpoints that a user requests.

Logic is implemented inside OnNavigateAsync to determine the assemblies to load with LazyAssemblyLoader. Options for how to structure the logic include:

- Conditional checks inside the OnNavigateAsync method.
- A lookup table that maps routes to assembly names, either injected into the component or implemented within the component's code.

In the following example:

- The namespace for Microsoft.AspNetCore.Components.WebAssembly.Services is specified.
- The LazyAssemblyLoader service is injected (`AssemblyLoader`).
- The `{PATH}` placeholder is the path where the list of assemblies should load. The example uses a conditional check for a single path that loads a single set of assemblies.
- The `{LIST OF ASSEMBLIES}` placeholder is the comma-separated list of assembly file name strings, including their file extensions (for example, `"Assembly1.{FILE`

`EXTENSION}", "Assembly2.{FILE EXTENSION}"`).

`App.razor`:

razor

```
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject LazyAssemblyLoader AssemblyLoader
@inject ILogger<App> Logger

<Router AppAssembly="typeof(App).Assembly"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
           {
               if (args.Path == "{PATH}")
               {
                   var assemblies = await
AssemblyLoader.LoadAssembliesAsync(
                       new[] { {LIST OF ASSEMBLIES} });
               }
           }
           catch (Exception ex)
           {
               Logger.LogError("Error: {Message}", ex.Message);
           }
    }
}
```

> **Note**
>
> The preceding example doesn't show the contents of the Router component's Razor markup (`...`). For a demonstration with complete code, see the Complete example section of this article.

Assemblies that include routable components

When the list of assemblies includes routable components, the assembly list for a given path is passed to the Router component's AdditionalAssemblies collection.

In the following example:

- The List<Assembly> in `lazyLoadedAssemblies` passes the assembly list to AdditionalAssemblies. The framework searches the assemblies for routes and updates the route collection if new routes are found. To access the Assembly type, the namespace for System.Reflection is included at the top of the `App.razor` file.
- The `{PATH}` placeholder is the path where the list of assemblies should load. The example uses a conditional check for a single path that loads a single set of assemblies.
- The `{LIST OF ASSEMBLIES}` placeholder is the comma-separated list of assembly file name strings, including their file extensions (for example, `"Assembly1.{FILE EXTENSION}", "Assembly2.{FILE EXTENSION}"`).

`App.razor`:

razor

```
@using System.Reflection
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject ILogger<App> Logger
@inject LazyAssemblyLoader AssemblyLoader

<Router AppAssembly="typeof(App).Assembly"
    AdditionalAssemblies="lazyLoadedAssemblies"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private List<Assembly> lazyLoadedAssemblies = new();

    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
           {
               if (args.Path == "{PATH}")
               {
                   var assemblies = await
AssemblyLoader.LoadAssembliesAsync(
                       new[] { {LIST OF ASSEMBLIES} });
                   lazyLoadedAssemblies.AddRange(assemblies);
               }
           }
           catch (Exception ex)
           {
               Logger.LogError("Error: {Message}", ex.Message);
           }
```

```
    }
}
```

> **Note**
>
> The preceding example doesn't show the contents of the **Router** component's Razor markup (`...`). For a demonstration with complete code, see the **Complete example** section of this article.

For more information, see ASP.NET Core Blazor routing and navigation.

User interaction with `<Navigating>` content

While loading assemblies, which can take several seconds, the Router component can indicate to the user that a page transition is occurring with the router's Navigating property.

For more information, see ASP.NET Core Blazor routing and navigation.

Handle cancellations in `OnNavigateAsync`

The NavigationContext object passed to the OnNavigateAsync callback contains a CancellationToken that's set when a new navigation event occurs. The OnNavigateAsync callback must throw when the cancellation token is set to avoid continuing to run the OnNavigateAsync callback on an outdated navigation.

For more information, see ASP.NET Core Blazor routing and navigation.

`OnNavigateAsync` events and renamed assembly files

The resource loader relies on the assembly names that are defined in the `blazor.boot.json` file. If assemblies are renamed, the assembly names used in an OnNavigateAsync callback and the assembly names in the `blazor.boot.json` file are out of sync.

To rectify this:

- Check to see if the app is running in the `Production` environment when determining which assembly names to use.

- Store the renamed assembly names in a separate file and read from that file to determine what assembly name to use with the LazyAssemblyLoader service and OnNavigateAsync callback.

Complete example

The demonstration in this section:

- Creates a robot controls assembly (`GrantImaharaRobotControls.{FILE EXTENSION}`) as a Razor class library (RCL) that includes a `Robot` component (`Robot.razor` with a route template of `/robot`).
- Lazily loads the RCL's assembly to render its `Robot` component when the `/robot` URL is requested by the user.

Create a standalone Blazor WebAssembly app to demonstrate lazy loading of a Razor class library's assembly. Name the project `LazyLoadTest`.

Add an ASP.NET Core class library project to the solution:

- Visual Studio: Right-click the solution file in **Solution Explorer** and select **Add** > **New project**. From the dialog of new project types, select **Razor Class Library**. Name the project `GrantImaharaRobotControls`. Do **not** select the **Support pages and views** checkbox.
- Visual Studio Code/.NET CLI: Execute `dotnet new razorclasslib -o GrantImaharaRobotControls` from a command prompt. The `-o|--output` option creates a folder and names the project `GrantImaharaRobotControls`.

Create a `HandGesture` class in the RCL with a `ThumbUp` method that hypothetically makes a robot perform a thumbs-up gesture. The method accepts an argument for the axis, `Left` or `Right`, as an enum. The method returns `true` on success.

`HandGesture.cs`:

C#

```
using Microsoft.Extensions.Logging;

namespace GrantImaharaRobotControls;

public static class HandGesture
{
    public static bool ThumbUp(Axis axis, ILogger logger)
    {
        logger.LogInformation("Thumb up gesture. Axis: {Axis}", axis);
```

```
        // Code to make robot perform gesture

        return true;
    }
}

public enum Axis { Left, Right }
```

Add the following component to the root of the RCL project. The component permits the user to submit a left or right hand thumb-up gesture request.

`Robot.razor`:

razor

```
@page "/robot"
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.Extensions.Logging
@inject ILogger<Robot> Logger

<h1>Robot</h1>

<EditForm FormName="RobotForm" Model="robotModel"
OnValidSubmit="HandleValidSubmit">
    <InputRadioGroup @bind-Value="robotModel.AxisSelection">
        @foreach (var entry in Enum.GetValues<Axis>())
        {
            <InputRadio Value="entry" />
            <text> </text>@entry<br>
        }
    </InputRadioGroup>

    <button type="submit">Submit</button>
</EditForm>

<p>
    @message
</p>

@code {
    private RobotModel robotModel = new() { AxisSelection = Axis.Left };
    private string? message;

    private void HandleValidSubmit()
    {
        Logger.LogInformation("HandleValidSubmit called");

        var result = HandGesture.ThumbUp(robotModel.AxisSelection, Logger);

        message = $"ThumbUp returned {result} at {DateTime.Now}.";
    }

    public class RobotModel
```

```
    {
        public Axis AxisSelection { get; set; }
    }
}
```

In the `LazyLoadTest` project, create a project reference for the `GrantImaharaRobotControls` RCL:

- Visual Studio: Right-click the `LazyLoadTest` project and select **Add** > **Project Reference** to add a project reference for the `GrantImaharaRobotControls` RCL.
- Visual Studio Code/.NET CLI: Execute `dotnet add reference {PATH}` in a command shell from the project's folder. The `{PATH}` placeholder is the path to the RCL project.

Specify the RCL's assembly for lazy loading in the `LazyLoadTest` app's project file (`.csproj`):

XML

```
<ItemGroup>
    <BlazorWebAssemblyLazyLoad Include="GrantImaharaRobotControls.{FILE
EXTENSION}" />
</ItemGroup>
```

The following Router component demonstrates loading the `GrantImaharaRobotControls.{FILE EXTENSION}` assembly when the user navigates to `/robot`. Replace the app's default `App` component with the following `App` component.

During page transitions, a styled message is displayed to the user with the `<Navigating>` element. For more information, see the User interaction with <Navigating> content section.

The assembly is assigned to AdditionalAssemblies, which results in the router searching the assembly for routable components, where it finds the `Robot` component. The `Robot` component's route is added to the app's route collection. For more information, see the ASP.NET Core Blazor routing and navigation article and the Assemblies that include routable components section of this article.

`App.razor`:

razor

```
@using System.Reflection
@using Microsoft.AspNetCore.Components.Routing
```

```
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject ILogger<App> Logger
@inject LazyAssemblyLoader AssemblyLoader

<Router AppAssembly="typeof(App).Assembly"
        AdditionalAssemblies="lazyLoadedAssemblies"
        OnNavigateAsync="OnNavigateAsync">
    <Navigating>
        <div style="padding:20px;background-color:blue;color:white">
            <p>Loading the requested page&hellip;</p>
        </div>
    </Navigating>
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)"
/>
    </Found>
    <NotFound>
        <LayoutView Layout="typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>

@code {
    private List<Assembly> lazyLoadedAssemblies = new();
    private bool grantImaharaRobotControlsAssemblyLoaded;

    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
        {
            if ((args.Path == "robot") &&
!grantImaharaRobotControlsAssemblyLoaded)
            {
                var assemblies = await AssemblyLoader.LoadAssembliesAsync(
                    new[] { "GrantImaharaRobotControls.{FILE EXTENSION}" });
                lazyLoadedAssemblies.AddRange(assemblies);
                grantImaharaRobotControlsAssemblyLoaded = true;
            }
        }
        catch (Exception ex)
        {
            Logger.LogError("Error: {Message}", ex.Message);
        }
    }
}
```

Build and run the app.

When the `Robot` component from the RCL is requested at `/robot`, the `GrantImaharaRobotControls.{FILE EXTENSION}` assembly is loaded and the `Robot`

component is rendered. You can inspect the assembly loading in the **Network** tab of the browser's developer tools.

Troubleshoot

- If unexpected rendering occurs, such as rendering a component from a previous navigation, confirm that the code throws if the cancellation token is set.
- If assemblies configured for lazy loading unexpectedly load at app start, check that the assembly is marked for lazy loading in the project file.

Additional resources

- Handle asynchronous navigation events with OnNavigateAsync
- ASP.NET Core Blazor performance best practices

ASP.NET Core Blazor WebAssembly native dependencies

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Blazor WebAssembly apps can use native dependencies built to run on WebAssembly. You can statically link native dependencies into the .NET WebAssembly runtime using the **.NET WebAssembly build tools**, the same tools used to ahead-of-time (AOT) compile a Blazor app to WebAssembly and to relink the runtime to remove unused features.

This article only applies to Blazor WebAssembly.

.NET WebAssembly build tools

The .NET WebAssembly build tools are based on Emscripten, a compiler toolchain for the web platform. For more information on the build tools, including installation, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Add native dependencies to a Blazor WebAssembly app by adding `NativeFileReference` items in the app's project file. When the project is built, each `NativeFileReference` is passed to Emscripten by the .NET WebAssembly build tools so that they are compiled and linked into the runtime. Next, p/invoke into the native code from the app's .NET code.

Generally, any portable native code can be used as a native dependency with Blazor WebAssembly. You can add native dependencies to C/C++ code or code previously compiled using Emscripten:

- Object files (`.o`)
- Archive files (`.a`)
- Bitcode (`.bc`)
- Standalone WebAssembly modules (`.wasm`)

Prebuilt dependencies typically must be built using the same version of Emscripten used to build the .NET WebAssembly runtime.

> **Note**
>
> For Mono /WebAssembly MSBuild properties and targets, see WasmApp.targets (dotnet/runtime GitHub repository). Official documentation for common MSBuild properties is planned per Document blazor msbuild configuration options (dotnet/docs #27395).

Use native code

Add a simple native C function to a Blazor WebAssembly app:

1. Create a new Blazor WebAssembly project.
2. Add a `Test.c` file to the project.
3. Add a C function for computing factorials.

   `Test.c`:

   C

   ```c
   int fact(int n)
   {
       if (n == 0) return 1;
       return n * fact(n - 1);
   }
   ```

4. Add a `NativeFileReference` for `Test.c` in the app's project file:

   XML

   ```xml
   <ItemGroup>
     <NativeFileReference Include="Test.c" />
   </ItemGroup>
   ```

5. In a Razor component, add a DllImportAttribute for the `fact` function in the generated `Test` library and call the `fact` method from .NET code in the component.

   `Pages/NativeCTest.razor`:

razor

```razor
@page "/native-c-test"
@using System.Runtime.InteropServices

<PageTitle>Native C</PageTitle>

<h1>Native C Test</h1>

<p>
    @@fact(3) result: @fact(3)
</p>

@code {
    [DllImport("Test")]
    static extern int fact(int n);
}
```

When you build the app with the .NET WebAssembly build tools installed, the native C code is compiled and linked into the .NET WebAssembly runtime (`dotnet.wasm`). After the app is built, run the app to see the rendered factorial value.

C++ managed method callbacks

Label managed methods that are passed to C++ with the `[UnmanagedCallersOnly]` attribute.

The method marked with the `[UnmanagedCallersOnly]` attribute must be `static`. To call an instance method in a Razor component, pass a `GCHandle` for the instance to C++ and then pass it back to native. Alternatively, use some other method to identify the instance of the component.

The method marked with `[DllImport]` must use a C# 9.0 function pointer rather than a delegate type for the callback argument.

> ⓘ **Note**
>
> For C# function pointer types in `[DllImport]` methods, use `IntPtr` in the method signature on the managed side instead of `delegate *unmanaged<int, void>`. For more information, see [[WASM] callback from native code to .NET: Parsing function pointer types in signatures is not supported (dotnet/runtime #56145)].

Package native dependencies in a NuGet package

NuGet packages can contain native dependencies for use on WebAssembly. These libraries and their native functionality are then available to any Blazor WebAssembly app. The files for the native dependencies should be built for WebAssembly and packaged in the `browser-wasm` architecture-specific folder. WebAssembly-specific dependencies aren't referenced automatically and must be referenced manually as `NativeFileReference`s. Package authors can choose to add the native references by including a `.props` file in the package with the references.

SkiaSharp example library use

SkiaSharp is a cross-platform 2D graphics library for .NET based on the native Skia graphics library with support for Blazor WebAssembly.

To use SkiaSharp in a Blazor WebAssembly app:

1. Add a package reference to the SkiaSharp.Views.Blazor package in a Blazor WebAssembly project. Use Visual Studio's process for adding packages to an app (**Manage NuGet Packages** with **Include prerelease** selected) or execute the dotnet add package command in a command shell:

   .NET CLI

   ```
   dotnet add package --prerelease SkiaSharp.Views.Blazor
   ```

   > **Note**
   >
   > For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

2. Add a `SKCanvasView` component to the app with the following:

   - `SkiaSharp` and `SkiaSharp.Views.Blazor` namespaces.
   - Logic to draw in the SkiaSharp Canvas View component (`SKCanvasView`).

   `Pages/NativeDependencyExample.razor`:

   razor

```razor
@page "/native-dependency-example"
@using SkiaSharp
@using SkiaSharp.Views.Blazor

<PageTitle>Native dependency</PageTitle>

<h1>Native dependency example with SkiaSharp</h1>

<SKCanvasView OnPaintSurface="OnPaintSurface" />

@code {
    private void OnPaintSurface(SKPaintSurfaceEventArgs e)
    {
        var canvas = e.Surface.Canvas;

        canvas.Clear(SKColors.White);

        using var paint = new SKPaint
        {
            Color = SKColors.Black,
            IsAntialias = true,
            TextSize = 24
        };

        canvas.DrawText("SkiaSharp", 0, 24, paint);
    }
}
```

3. Build the app, which might take several minutes. Run the app and navigate to the `NativeDependencyExample` component at `/native-dependency-example`.

Additional resources

.NET WebAssembly build tools

ASP.NET Core Blazor performance best practices

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Blazor is optimized for high performance in most realistic application UI scenarios. However, the best performance depends on developers adopting the correct patterns and features.

> **Note**
>
> The code examples in this article adopt nullable reference types (NRTs) and .NET compiler null-state static analysis, which are supported in ASP.NET Core in .NET 6 or later.

Optimize rendering speed

Optimize rendering speed to minimize rendering workload and improve UI responsiveness, which can yield a *ten-fold or higher improvement* in UI rendering speed.

Avoid unnecessary rendering of component subtrees

You might be able to remove the majority of a parent component's rendering cost by skipping the rerendering of child component subtrees when an event occurs. You should only be concerned about skipping the rerendering subtrees that are particularly expensive to render and are causing UI lag.

At runtime, components exist in a hierarchy. A root component (the first component loaded) has child components. In turn, the root's children have their own child components, and so on. When an event occurs, such as a user selecting a button, the following process determines which components to rerender:

1. The event is dispatched to the component that rendered the event's handler. After executing the event handler, the component is rerendered.

2. When a component is rerendered, it supplies a new copy of parameter values to each of its child components.
3. After a new set of parameter values is received, each component decides whether to rerender. Components rerender if the parameter values may have changed, for example, if they're mutable objects.

The last two steps of the preceding sequence continue recursively down the component hierarchy. In many cases, the entire subtree is rerendered. Events targeting high-level components can cause expensive rerendering because every component below the high-level component must rerender.

To prevent rendering recursion into a particular subtree, use either of the following approaches:

- Ensure that child component parameters are of primitive immutable types, such as `string`, `int`, `bool`, `DateTime`, and other similar types. The built-in logic for detecting changes automatically skips rerendering if the primitive immutable parameter values haven't changed. If you render a child component with `<Customer CustomerId="item.CustomerId" />`, where `CustomerId` is an `int` type, then the `Customer` component isn't rerendered unless `item.CustomerId` changes.
- Override ShouldRender:
  - To accept nonprimitive parameter values, such as complex custom model types, event callbacks, or RenderFragment values.
  - If authoring a UI-only component that doesn't change after the initial render, regardless of parameter value changes.

The following airline flight search tool example uses private fields to track the necessary information to detect changes. The previous inbound flight identifier (`prevInboundFlightId`) and previous outbound flight identifier (`prevOutboundFlightId`) track information for the next potential component update. If either of the flight identifiers change when the component's parameters are set in OnParametersSet, the component is rerendered because `shouldRender` is set to `true`. If `shouldRender` evaluates to `false` after checking the flight identifiers, an expensive rerender is avoided:

razor

```
@code {
    private int prevInboundFlightId = 0;
    private int prevOutboundFlightId = 0;
    private bool shouldRender;

    [Parameter]
    public FlightInfo? InboundFlight { get; set; }
```

```
    [Parameter]
    public FlightInfo? OutboundFlight { get; set; }

    protected override void OnParametersSet()
    {
        shouldRender = InboundFlight?.FlightId != prevInboundFlightId
            || OutboundFlight?.FlightId != prevOutboundFlightId;

        prevInboundFlightId = InboundFlight?.FlightId ?? 0;
        prevOutboundFlightId = OutboundFlight?.FlightId ?? 0;
    }

    protected override bool ShouldRender() => shouldRender;
}
```

An event handler can also set `shouldRender` to `true`. For most components, determining rerendering at the level of individual event handlers usually isn't necessary.

For more information, see the following resources:

- ASP.NET Core Razor component lifecycle
- ShouldRender
- ASP.NET Core Razor component rendering

Virtualization

When rendering large amounts of UI within a loop, for example, a list or grid with thousands of entries, the sheer quantity of rendering operations can lead to a lag in UI rendering. Given that the user can only see a small number of elements at once without scrolling, it's often wasteful to spend time rendering elements that aren't currently visible.

Blazor provides the Virtualize<TItem> component to create the appearance and scroll behaviors of an arbitrarily-large list while only rendering the list items that are within the current scroll viewport. For example, a component can render a list with 100,000 entries but only pay the rendering cost of 20 items that are visible.

For more information, see ASP.NET Core Razor component virtualization.

Create lightweight, optimized components

Most Razor components don't require aggressive optimization efforts because most components don't repeat in the UI and don't rerender at high frequency. For example, routable components with an `@page` directive and components used to render high-level pieces of the UI, such as dialogs or forms, most likely appear only one at a time

and only rerender in response to a user gesture. These components don't usually create high rendering workload, so you can freely use any combination of framework features without much concern about rendering performance.

However, there are common scenarios where components are repeated at scale and often result in poor UI performance:

- Large nested forms with hundreds of individual elements, such as inputs or labels.
- Grids with hundreds of rows or thousands of cells.
- Scatter plots with millions of data points.

If modelling each element, cell, or data point as a separate component instance, there are often so many of them that their rendering performance becomes critical. This section provides advice on making such components lightweight so that the UI remains fast and responsive.

Avoid thousands of component instances

Each component is a separate island that can render independently of its parents and children. By choosing how to split the UI into a hierarchy of components, you are taking control over the granularity of UI rendering. This can result in either good or poor performance.

By splitting the UI into separate components, you can have smaller portions of the UI rerender when events occur. In a table with many rows that have a button in each row, you may be able to have only that single row rerender by using a child component instead of the whole page or table. However, each component requires additional memory and CPU overhead to deal with its independent state and rendering lifecycle.

In a test performed by the ASP.NET Core product unit engineers, a rendering overhead of around 0.06 ms per component instance was seen in a Blazor WebAssembly app. The test app rendered a simple component that accepts three parameters. Internally, the overhead is largely due to retrieving per-component state from dictionaries and passing and receiving parameters. By multiplication, you can see that adding 2,000 extra component instances would add 0.12 seconds to the rendering time and the UI would begin feeling slow to users.

It's possible to make components more lightweight so that you can have more of them. However, a more powerful technique is often to avoid having so many components to render. The following sections describe two approaches that you can take.

For more information on memory management, see Host and deploy ASP.NET Core server-side Blazor apps.

Inline child components into their parents

Consider the following portion of a parent component that renders child components in a loop:

razor

```razor
<div class="chat">
    @foreach (var message in messages)
    {
        <ChatMessageDisplay Message="message" />
    }
</div>
```

`ChatMessageDisplay.razor`:

razor

```razor
<div class="chat-message">
    <span class="author">@Message.Author</span>
    <span class="text">@Message.Text</span>
</div>

@code {
    [Parameter]
    public ChatMessage? Message { get; set; }
}
```

The preceding example performs well if thousands of messages aren't shown at once. To show thousands of messages at once, consider *not* factoring out the separate `ChatMessageDisplay` component. Instead, inline the child component into the parent. The following approach avoids the per-component overhead of rendering so many child components at the cost of losing the ability to rerender each child component's markup independently:

razor

```razor
<div class="chat">
    @foreach (var message in messages)
    {
        <div class="chat-message">
            <span class="author">@message.Author</span>
            <span class="text">@message.Text</span>
        </div>
    }
</div>
```

Define reusable `RenderFragments` in code

You might be factoring out child components purely as a way of reusing rendering logic. If that's the case, you can create reusable rendering logic without implementing additional components. In any component's `@code` block, define a RenderFragment. Render the fragment from any location as many times as needed:

razor

```
@RenderWelcomeInfo

<p>Render the welcome content a second time:</p>

@RenderWelcomeInfo

@code {
    private RenderFragment RenderWelcomeInfo = @<p>Welcome to your new app!
</p>;
}
```

To make RenderTreeBuilder code reusable across multiple components, declare the RenderFragment public and static:

razor

```
public static RenderFragment SayHello = @<h1>Hello!</h1>;
```

`SayHello` in the preceding example can be invoked from an unrelated component. This technique is useful for building libraries of reusable markup snippets that render without per-component overhead.

RenderFragment delegates can accept parameters. The following component passes the message (`message`) to the RenderFragment delegate:

razor

```
<div class="chat">
    @foreach (var message in messages)
    {
        @ChatMessageDisplay(message)
    }
</div>

@code {
    private RenderFragment<ChatMessage> ChatMessageDisplay = message =>
        @<div class="chat-message">
            <span class="author">@message.Author</span>
            <span class="text">@message.Text</span>
```

```
        </div>;
}
```

The preceding approach reuses rendering logic without per-component overhead. However, the approach doesn't permit refreshing the subtree of the UI independently, nor does it have the ability to skip rendering the subtree of the UI when its parent renders because there's no component boundary. Assignment to a RenderFragment delegate is only supported in Razor component files (`.razor`).

For a non-static field, method, or property that can't be referenced by a field initializer, such as `TitleTemplate` in the following example, use a property instead of a field for the RenderFragment:

C#

```
protected RenderFragment DisplayTitle =>
    @<div>
        @TitleTemplate
    </div>;
```

Don't receive too many parameters

If a component repeats extremely often, for example, hundreds or thousands of times, the overhead of passing and receiving each parameter builds up.

It's rare that too many parameters severely restricts performance, but it can be a factor. For a `TableCell` component that renders 4,000 times within a grid, each parameter passed to the component adds around 15 ms to the total rendering cost. Passing ten parameters requires around 150 ms and causes a UI rendering lag.

To reduce parameter load, bundle multiple parameters in a custom class. For example, a table cell component might accept a common object. In the following example, `Data` is different for every cell, but `Options` is common across all cell instances:

razor

```
@typeparam TItem

...

@code {
    [Parameter]
    public TItem? Data { get; set; }

    [Parameter]
```

```
    public GridOptions? Options { get; set; }
}
```

However, consider that it might be an improvement not to have a table cell component, as shown in the preceding example, and instead inline its logic into the parent component.

> ⓘ Note
>
> When multiple approaches are available for improving performance, benchmarking the approaches is usually required to determine which approach yields the best results.

For more information on generic type parameters (`@typeparam`), see the following resources:

- Razor syntax reference for ASP.NET Core
- ASP.NET Core Razor components
- ASP.NET Core Blazor templated components

Ensure cascading parameters are fixed

The CascadingValue component has an optional `IsFixed` parameter:

- If `IsFixed` is `false` (the default), every recipient of the cascaded value sets up a subscription to receive change notifications. Each `[CascadingParameter]` is **substantially more expensive** than a regular `[Parameter]` due to the subscription tracking.
- If `IsFixed` is `true` (for example, `<CascadingValue Value="someValue" IsFixed="true">`), recipients receive the initial value but don't set up a subscription to receive updates. Each `[CascadingParameter]` is lightweight and no more expensive than a regular `[Parameter]`.

Setting `IsFixed` to `true` improves performance if there are a large number of other components that receive the cascaded value. Wherever possible, set `IsFixed` to `true` on cascaded values. You can set `IsFixed` to `true` when the supplied value doesn't change over time.

Where a component passes `this` as a cascaded value, `IsFixed` can also be set to `true`, because `this` never changes during the component's lifecycle:

razor

```razor
<CascadingValue Value="this" IsFixed="true">
    <SomeOtherComponents>
</CascadingValue>
```

For more information, see ASP.NET Core Blazor cascading values and parameters.

Avoid attribute splatting with `CaptureUnmatchedValues`

Components can elect to receive "unmatched" parameter values using the CaptureUnmatchedValues flag:

razor

```razor
<div @attributes="OtherAttributes">...</div>

@code {
    [Parameter(CaptureUnmatchedValues = true)]
    public IDictionary<string, object>? OtherAttributes { get; set; }
}
```

This approach allows passing arbitrary additional attributes to the element. However, this approach is expensive because the renderer must:

- Match all of the supplied parameters against the set of known parameters to build a dictionary.
- Keep track of how multiple copies of the same attribute overwrite each other.

Use CaptureUnmatchedValues where component rendering performance isn't critical, such as components that aren't repeated frequently. For components that render at scale, such as each item in a large list or in the cells of a grid, try to avoid attribute splatting.

For more information, see ASP.NET Core Blazor attribute splatting and arbitrary parameters.

Implement `SetParametersAsync` manually

A significant source of per-component rendering overhead is writing incoming parameter values to `[Parameter]` properties. The renderer uses reflection to write the parameter values, which can lead to poor performance at scale.

In some extreme cases, you may wish to avoid the reflection and implement your own parameter-setting logic manually. This may be applicable when:

- A component renders extremely often, for example, when there are hundreds or thousands of copies of the component in the UI.
- A component accepts many parameters.
- You find that the overhead of receiving parameters has an observable impact on UI responsiveness.

In extreme cases, you can override the component's virtual SetParametersAsync method and implement your own component-specific logic. The following example deliberately avoids dictionary lookups:

razor

```
@code {
    [Parameter]
    public int MessageId { get; set; }

    [Parameter]
    public string? Text { get; set; }

    [Parameter]
    public EventCallback<string> TextChanged { get; set; }

    [Parameter]
    public Theme CurrentTheme { get; set; }

    public override Task SetParametersAsync(ParameterView parameters)
    {
        foreach (var parameter in parameters)
        {
            switch (parameter.Name)
            {
                case nameof(MessageId):
                    MessageId = (int)parameter.Value;
                    break;
                case nameof(Text):
                    Text = (string)parameter.Value;
                    break;
                case nameof(TextChanged):
                    TextChanged = (EventCallback<string>)parameter.Value;
                    break;
                case nameof(CurrentTheme):
                    CurrentTheme = (Theme)parameter.Value;
                    break;
                default:
                    throw new ArgumentException($"Unknown parameter:
{parameter.Name}");
            }
        }
```

```
        return base.SetParametersAsync(ParameterView.Empty);
    }
}
```

In the preceding code, returning the base class SetParametersAsync runs the normal lifecycle method without assigning parameters again.

As you can see in the preceding code, overriding SetParametersAsync and supplying custom logic is complicated and laborious, so we don't generally recommend adopting this approach. In extreme cases, it can improve rendering performance by 20-25%, but you should only consider this approach in the extreme scenarios listed earlier in this section.

Don't trigger events too rapidly

Some browser events fire extremely frequently. For example, `onmousemove` and `onscroll` can fire tens or hundreds of times per second. In most cases, you don't need to perform UI updates this frequently. If events are triggered too rapidly, you may harm UI responsiveness or consume excessive CPU time.

Rather than use native events that rapidly fire, consider the use of JS interop to register a callback that fires less frequently. For example, the following component displays the position of the mouse but only updates at most once every 500 ms:

razor

```
@implements IDisposable
@inject IJSRuntime JS

<h1>@message</h1>

<div @ref="mouseMoveElement" style="border:1px dashed red;height:200px;">
    Move mouse here
</div>

@code {
    private ElementReference mouseMoveElement;
    private DotNetObjectReference<MyComponent>? selfReference;
    private string message = "Move the mouse in the box";

    [JSInvokable]
    public void HandleMouseMove(int x, int y)
    {
        message = $"Mouse move at {x}, {y}";
        StateHasChanged();
    }
```

```
    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            selfReference = DotNetObjectReference.Create(this);
            var minInterval = 500;

            await JS.InvokeVoidAsync("onThrottledMouseMove",
                mouseMoveElement, selfReference, minInterval);
        }
    }

    public void Dispose() => selfReference?.Dispose();
}
```

The corresponding JavaScript code registers the DOM event listener for mouse movement. In this example, the event listener uses Lodash's throttle function to limit the rate of invocations:

HTML

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.20/lodash.min.js"
></script>
<script>
  function onThrottledMouseMove(elem, component, interval) {
    elem.addEventListener('mousemove', _.throttle(e => {
      component.invokeMethodAsync('HandleMouseMove', e.offsetX, e.offsetY);
    }, interval));
  }
</script>
```

Avoid rerendering after handling events without state changes

Components inherit from ComponentBase, which automatically invokes StateHasChanged after the component's event handlers are invoked. In some cases, it might be unnecessary or undesirable to trigger a rerender after an event handler is invoked. For example, an event handler might not modify component state. In these scenarios, the app can leverage the IHandleEvent interface to control the behavior of Blazor's event handling.

> **Note**
>
> The approach in this section doesn't flow exceptions to error boundaries. For more information and demonstration code that supports error boundaries by calling

ComponentBase.DispatchExceptionAsync, see AsNonRenderingEventHandler + ErrorBoundary = unexpected behavior (dotnet/aspnetcore #54543).

To prevent rerenders for all of a component's event handlers, implement IHandleEvent and provide a IHandleEvent.HandleEventAsync task that invokes the event handler without calling StateHasChanged.

In the following example, no event handler added to the component triggers a rerender, so `HandleSelect` doesn't result in a rerender when invoked.

`HandleSelect1.razor`:

razor

```
@page "/handle-select-1"
@using Microsoft.Extensions.Logging
@implements IHandleEvent
@inject ILogger<HandleSelect1> Logger

<p>
    Last render DateTime: @dt
</p>

<button @onclick="HandleSelect">
    Select me (Avoids Rerender)
</button>

@code {
    private DateTime dt = DateTime.Now;

    private void HandleSelect()
    {
        dt = DateTime.Now;

        Logger.LogInformation("This event handler doesn't trigger a
rerender.");
    }

    Task IHandleEvent.HandleEventAsync(
        EventCallbackWorkItem callback, object? arg) =>
callback.InvokeAsync(arg);
}
```

In addition to preventing rerenders after event handlers fire in a component in a global fashion, it's possible to prevent rerenders after a single event handler by employing the following utility method.

Add the following `EventUtil` class to a Blazor app. The static actions and functions at the top of the `EventUtil` class provide handlers that cover several combinations of arguments and return types that Blazor uses when handling events.

`EventUtil.cs`:

C#

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Components;

public static class EventUtil
{
    public static Action AsNonRenderingEventHandler(Action callback)
        => new SyncReceiver(callback).Invoke;
    public static Action<TValue> AsNonRenderingEventHandler<TValue>(
            Action<TValue> callback)
        => new SyncReceiver<TValue>(callback).Invoke;
    public static Func<Task> AsNonRenderingEventHandler(Func<Task> callback)
        => new AsyncReceiver(callback).Invoke;
    public static Func<TValue, Task> AsNonRenderingEventHandler<TValue>(
            Func<TValue, Task> callback)
        => new AsyncReceiver<TValue>(callback).Invoke;

    private record SyncReceiver(Action callback)
        : ReceiverBase { public void Invoke() => callback(); }
    private record SyncReceiver<T>(Action<T> callback)
        : ReceiverBase { public void Invoke(T arg) => callback(arg); }
    private record AsyncReceiver(Func<Task> callback)
        : ReceiverBase { public Task Invoke() => callback(); }
    private record AsyncReceiver<T>(Func<T, Task> callback)
        : ReceiverBase { public Task Invoke(T arg) => callback(arg); }

    private record ReceiverBase : IHandleEvent
    {
        public Task HandleEventAsync(EventCallbackWorkItem item, object arg)
=>
            item.InvokeAsync(arg);
    }
}
```

Call `EventUtil.AsNonRenderingEventHandler` to call an event handler that doesn't trigger a render when invoked.

In the following example:

- Selecting the first button, which calls `HandleClick1`, triggers a rerender.
- Selecting the second button, which calls `HandleClick2`, doesn't trigger a rerender.

- Selecting the third button, which calls `HandleClick3`, doesn't trigger a rerender and uses event arguments (MouseEventArgs).

`HandleSelect2.razor`:

razor

```razor
@page "/handle-select-2"
@using Microsoft.Extensions.Logging
@inject ILogger<HandleSelect2> Logger

<p>
    Last render DateTime: @dt
</p>

<button @onclick="HandleClick1">
    Select me (Rerenders)
</button>

<button @onclick="EventUtil.AsNonRenderingEventHandler(HandleClick2)">
    Select me (Avoids Rerender)
</button>

<button @onclick="EventUtil.AsNonRenderingEventHandler<MouseEventArgs>
(HandleClick3)">
    Select me (Avoids Rerender and uses <code>MouseEventArgs</code>)
</button>

@code {
    private DateTime dt = DateTime.Now;

    private void HandleClick1()
    {
        dt = DateTime.Now;

        Logger.LogInformation("This event handler triggers a rerender.");
    }

    private void HandleClick2()
    {
        dt = DateTime.Now;

        Logger.LogInformation("This event handler doesn't trigger a
rerender.");
    }

    private void HandleClick3(MouseEventArgs args)
    {
        dt = DateTime.Now;

        Logger.LogInformation(
            "This event handler doesn't trigger a rerender. " +
            "Mouse coordinates: {ScreenX}:{ScreenY}",
```

```
            args.ScreenX, args.ScreenY);
    }
}
```

In addition to implementing the IHandleEvent interface, leveraging the other best practices described in this article can also help reduce unwanted renders after events are handled. For example, overriding ShouldRender in child components of the target component can be used to control rerendering.

Avoid recreating delegates for many repeated elements or components

Blazor's recreation of lambda expression delegates for elements or components in a loop can lead to poor performance.

The following component shown in the event handling article renders a set of buttons. Each button assigns a delegate to its `@onclick` event, which is fine if there aren't many buttons to render.

`EventHandlerExample5.razor`:

razor

```
@page "/event-handler-example-5"

<h1>@heading</h1>

@for (var i = 1; i < 4; i++)
{
    var buttonNumber = i;

    <p>
        <button @onclick="@(e => UpdateHeading(e, buttonNumber))">
            Button #@i
        </button>
    </p>
}

@code {
    private string heading = "Select a button to learn its position";

    private void UpdateHeading(MouseEventArgs e, int buttonNumber)
    {
        heading = $"Selected #{buttonNumber} at {e.ClientX}:{e.ClientY}";
    }
}
```

If a large number of buttons are rendered using the preceding approach, rendering speed is adversely impacted leading to a poor user experience. To render a large number of buttons with a callback for click events, the following example uses a collection of button objects that assign each button's `@onclick` delegate to an Action. The following approach doesn't require Blazor to rebuild all of the button delegates each time the buttons are rendered:

`LambdaEventPerformance.razor`:

razor

```razor
@page "/lambda-event-performance"

<h1>@heading</h1>

@foreach (var button in Buttons)
{
    <p>
        <button @key="button.Id" @onclick="button.Action">
            Button #@button.Id
        </button>
    </p>
}

@code {
    private string heading = "Select a button to learn its position";

    private List<Button> Buttons { get; set; } = new();

    protected override void OnInitialized()
    {
        for (var i = 0; i < 100; i++)
        {
            var button = new Button();

            button.Id = Guid.NewGuid().ToString();

            button.Action = (e) =>
            {
                UpdateHeading(button, e);
            };

            Buttons.Add(button);
        }
    }

    private void UpdateHeading(Button button, MouseEventArgs e)
    {
        heading = $"Selected #{button.Id} at {e.ClientX}:{e.ClientY}";
    }

    private class Button
```

```
    {
        public string? Id { get; set; }
        public Action<MouseEventArgs> Action { get; set; } = e => { };
    }
}
```

Optimize JavaScript interop speed

Calls between .NET and JavaScript require additional overhead because:

- Calls are asynchronous.
- Parameters and return values are JSON-serialized to provide an easy-to-understand conversion mechanism between .NET and JavaScript types.

Additionally for server-side Blazor apps, these calls are passed across the network.

Avoid excessively fine-grained calls

Since each call involves some overhead, it can be valuable to reduce the number of calls. Consider the following code, which stores a collection of items in the browser's localStorage:

C#

```
private async Task StoreAllInLocalStorage(IEnumerable<TodoItem> items)
{
    foreach (var item in items)
    {
        await JS.InvokeVoidAsync("localStorage.setItem", item.Id,
            JsonSerializer.Serialize(item));
    }
}
```

The preceding example makes a separate JS interop call for each item. Instead, the following approach reduces the JS interop to a single call:

C#

```
private async Task StoreAllInLocalStorage(IEnumerable<TodoItem> items)
{
    await JS.InvokeVoidAsync("storeAllInLocalStorage", items);
}
```

The corresponding JavaScript function stores the whole collection of items on the client:

JavaScript

```
function storeAllInLocalStorage(items) {
  items.forEach(item => {
    localStorage.setItem(item.id, JSON.stringify(item));
  });
}
```

For Blazor WebAssembly apps, rolling individual JS interop calls into a single call usually only improves performance significantly if the component makes a large number of JS interop calls.

Consider the use of synchronous calls

Call JavaScript from .NET

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from .NET to JavaScript in a client-side component, cast IJSRuntime to IJSInProcessRuntime to make the JS interop call:

razor

```
@inject IJSRuntime JS

...

@code {
    protected override void HandleSomeEvent()
    {
        var jsInProcess = (IJSInProcessRuntime)JS;
        var value = jsInProcess.Invoke<string>
("javascriptFunctionIdentifier");
    }
}
```

When working with IJSObjectReference in ASP.NET Core 5.0 or later client-side components, you can use IJSInProcessObjectReference synchronously instead. IJSInProcessObjectReference implements IAsyncDisposable/IDisposable and should be disposed for garbage collection to prevent a memory leak, as the following example demonstrates:

razor

```
@inject IJSRuntime JS
@implements IAsyncDisposable

...

@code {
    ...
    private IJSInProcessObjectReference? module;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            module = await JS.InvokeAsync<IJSInProcessObjectReference>
("import",
            "./scripts.js");
        }
    }

    ...

    async ValueTask IAsyncDisposable.DisposeAsync()
    {
        if (module is not null)
        {
            await module.DisposeAsync();
        }
    }
}
```

In the preceding example, a JSDisconnectedException isn't trapped during module disposal because there's no Blazor-SignalR circuit in a Blazor WebAssembly app to lose. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

Call .NET from JavaScript

This section only applies to client-side components.

JS interop calls are asynchronous, regardless of whether the called code is synchronous or asynchronous. Calls are asynchronous to ensure that components are compatible

across server-side and client-side render modes. On the server, all JS interop calls must be asynchronous because they're sent over a network connection.

If you know for certain that your component only runs on WebAssembly, you can choose to make synchronous JS interop calls. This has slightly less overhead than making asynchronous calls and can result in fewer render cycles because there's no intermediate state while awaiting results.

To make a synchronous call from JavaScript to .NET in a client-side component, use `DotNet.invokeMethod` instead of `DotNet.invokeMethodAsync`.

Synchronous calls work if:

- The component is only rendered for execution on WebAssembly.
- The called function returns a value synchronously. The function isn't an `async` method and doesn't return a .NET Task or JavaScript Promise.

Use JavaScript `[JSImport]`/`[JSExport]` interop

JavaScript `[JSImport]`/`[JSExport]` interop for Blazor WebAssembly apps offers improved performance and stability over the JS interop API in framework releases prior to ASP.NET Core in .NET 7.

For more information, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

Ahead-of-time (AOT) compilation

Ahead-of-time (AOT) compilation compiles a Blazor app's .NET code directly into native WebAssembly for direct execution by the browser. AOT-compiled apps result in larger apps that take longer to download, but AOT-compiled apps usually provide better runtime performance, especially for apps that execute CPU-intensive tasks. For more information, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Minimize app download size

Runtime relinking

For information on how runtime relinking minimizes an app's download size, see ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation.

Use `System.Text.Json`

Blazor's JS interop implementation relies on System.Text.Json, which is a high-performance JSON serialization library with low memory allocation. Using System.Text.Json shouldn't result in additional app payload size over adding one or more alternate JSON libraries.

For migration guidance, see How to migrate from Newtonsoft.Json to System.Text.Json.

Intermediate Language (IL) trimming

This section only applies to client-side Blazor scenarios.

Trimming unused assemblies from a Blazor WebAssembly app reduces the app's size by removing unused code in the app's binaries. For more information, see Configure the Trimmer for ASP.NET Core Blazor.

Lazy load assemblies

This section only applies to client-side Blazor scenarios.

Load assemblies at runtime when the assemblies are required by a route. For more information, see Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Compression

This section only applies to Blazor WebAssembly apps.

When a Blazor WebAssembly app is published, the output is statically compressed during publish to reduce the app's size and remove the overhead for runtime compression. Blazor relies on the server to perform content negotiation and serve statically-compressed files.

After an app is deployed, verify that the app serves compressed files. Inspect the **Network** tab in a browser's developer tools and verify that the files are served with `Content-Encoding: br` (Brotli compression) or `Content-Encoding: gz` (Gzip compression). If the host isn't serving compressed files, follow the instructions in Host and deploy ASP.NET Core Blazor WebAssembly.

Disable unused features

This section only applies to client-side Blazor scenarios.

Blazor WebAssembly's runtime includes the following .NET features that can be disabled for a smaller payload size:

- Blazor WebAssembly carries globalization resources required to display values, such as dates and currency, in the user's culture. If the app doesn't require localization, you may configure the app to support the invariant culture, which is based on the `en-US` culture.

- Adopting invariant globalization only results in using non-localized timezone names. To trim timezone code and data from the app, apply the `<InvariantTimezone>` MSBuild property with a value of `true` in the app's project file:

  XML

  ```
  <PropertyGroup>
    <InvariantTimezone>true</InvariantTimezone>
  </PropertyGroup>
  ```

  > **Note**
  >
  > <BlazorEnableTimeZoneSupport> overrides an earlier `<InvariantTimezone>` setting. We recommend removing the `<BlazorEnableTimeZoneSupport>` setting.

Test Razor components in ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Egil Hansen

Testing Razor components is an important aspect of releasing stable and maintainable Blazor apps.

To test a Razor component, the *component under test* (CUT) is:

- Rendered with relevant input for the test.
- Depending on the type of test performed, possibly subject to interaction or modification. For example, event handlers can be triggered, such as an `onclick` event for a button.
- Inspected for expected values. A test passes when one or more inspected values matches the expected values for the test.

Test approaches

Two common approaches for testing Razor components are end-to-end (E2E) testing and unit testing:

- **Unit testing**: Unit tests are written with a unit testing library that provides:
  - Component rendering.
  - Inspection of component output and state.
  - Triggering of event handlers and life cycle methods.
  - Assertions that component behavior is correct.

  bUnit is an example of a library that enables Razor component unit testing.

- **E2E testing**: A test runner runs a Blazor app containing the CUT and automates a browser instance. The testing tool inspects and interacts with the CUT through the browser. Playwright for .NET is an example of an E2E testing framework that can be used with Blazor apps.

In unit testing, only the Razor component (Razor/C#) is involved. External dependencies, such as services and JS interop, must be mocked. In E2E testing, the Razor component and all of its auxiliary infrastructure are part of the test, including CSS, JS, and the DOM and browser APIs.

Test scope describes how extensive the tests are. Test scope typically has an influence on the speed of the tests. Unit tests run on a subset of the app's subsystems and usually execute in milliseconds. E2E tests, which test a broad group of the app's subsystems, can take several seconds to complete.

Unit testing also provides access to the instance of the CUT, allowing for inspection and verification of the component's internal state. This normally isn't possible in E2E testing.

With regard to the component's environment, E2E tests must make sure that the expected environmental state has been reached before verification starts. Otherwise, the result is unpredictable. In unit testing, the rendering of the CUT and the life cycle of the test are more integrated, which improves test stability.

E2E testing involves launching multiple processes, network and disk I/O, and other subsystem activity that often lead to poor test reliability. Unit tests are typically insulated from these sorts of issues.

The following table summarizes the difference between the two testing approaches.

Expand table

Capability	Unit testing	E2E testing
Test scope	Razor component (Razor/C#) only	Razor component (Razor/C#) with CSS/JS
Test execution time	Milliseconds	Seconds
Access to the component instance	Yes	No
Sensitive to the environment	No	Yes
Reliability	More reliable	Less reliable

Choose the most appropriate test approach

Consider the scenario when choosing the type of testing to perform. Some considerations are described in the following table.

Expand table

Scenario	Suggested approach	Remarks
Component without JS interop logic	Unit testing	When there's no dependency on JS interop in a Razor component, the component can be tested without access to JS or the DOM API. In this scenario, there are no disadvantages to choosing unit testing.
Component with simple JS interop logic	Unit testing	It's common for components to query the DOM or trigger animations through JS interop. Unit testing is usually preferred in this scenario, since it's straightforward to mock the JS interaction through the IJSRuntime interface.
Component that depends on complex JS code	Unit testing and separate JS testing	If a component uses JS interop to call a large or complex JS library but the interaction between the Razor component and JS library is simple, then the best approach is likely to treat the component and JS library or code as two separate parts and test each individually. Test the Razor component with a unit testing library, and test the JS with a JS testing library.
Component with logic that depends on JS manipulation of the browser DOM	E2E testing	When a component's functionality is dependent on JS and its manipulation of the DOM, verify both the JS and Blazor code together in an E2E test. This is the approach that the Blazor framework developers have taken with Blazor's browser rendering logic, which has tightly-coupled C# and JS code. The C# and JS code must work together to correctly render Razor components in a browser.
Component that depends on 3rd party class library with hard-to-mock dependencies	E2E testing	When a component's functionality is dependent on a 3rd party class library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component.

Test components with bUnit

There's no official Microsoft testing framework for Blazor, but the community-driven project bUnit provides a convenient way to unit test Razor components.

> **Note**
>
> bUnit is a third-party testing library and isn't supported or maintained by Microsoft.

bUnit works with general-purpose testing frameworks, such as MSTest, NUnit, and xUnit. These testing frameworks make bUnit tests look and feel like regular unit tests. bUnit tests integrated with a general-purpose testing framework are ordinarily executed with:

- Visual Studio's Test Explorer.
- dotnet test CLI command in a command shell.
- An automated DevOps testing pipeline.

> **Note**
>
> Test concepts and test implementations across different test frameworks are similar but not identical. Refer to the test framework's documentation for guidance.

The following demonstrates the structure of a bUnit test on the `Counter` component in an app based on a Blazor project template. The `Counter` component displays and increments a counter based on the user selecting a button in the page:

razor

```razor
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

The following bUnit test verifies that the CUT's counter is incremented correctly when the button is selected:

razor

```razor
@code {
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
```

```
        using var ctx = new TestContext();
        var cut = ctx.Render(@<Counter />);
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElm.MarkupMatches("Current count: 1");
    }
}
```

Tests can also be written in a C# class file:

C#

```
public class CounterTests
{
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
        using var ctx = new TestContext();
        var cut = ctx.RenderComponent<Counter>();
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElmText.MarkupMatches("Current count: 1");
    }
}
```

The following actions take place at each step of the test:

- *Arrange*: The `Counter` component is rendered using bUnit's `TestContext`. The CUT's paragraph element (`<p>`) is found and assigned to `paraElm`. In Razor syntax, a component can be passed as a RenderFragment to bUnit.
- *Act*: The button's element (`<button>`) is located and selected by calling `Click`, which should increment the counter and update the content of the paragraph tag (`<p>`). The paragraph element text content is obtained by calling `TextContent`.
- *Assert*: `MarkupMatches` is called on the text content to verify that it matches the expected string, which is `Current count: 1`.

> **Note**
>
> The `MarkupMatches` assert method differs from a regular string comparison assertion (for example, `Assert.Equal("Current count: 1", paraElmText);`). `MarkupMatches` performs a semantic comparison of the input and expected HTML markup. A semantic comparison is aware of HTML semantics, meaning things like insignificant whitespace is ignored. This results in more stable tests. For more information, see **Customizing the Semantic HTML Comparison**.

Additional resources

- Getting Started with bUnit: bUnit instructions include guidance on creating a test project, referencing testing framework packages, and building and running tests.
- Blazor Testing from A to Z - Egil Hansen - Swetugg Stockholm 2024 (Swetugg)

ASP.NET Core Blazor Progressive Web Application (PWA)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

A Blazor Progressive Web Application (PWA) is a single-page application (SPA) that uses modern browser APIs and capabilities to behave like a desktop app.

Blazor WebAssembly is a standards-based client-side web app platform, so it can use any browser API, including PWA APIs required for the following capabilities:

- Working offline and loading instantly, independent of network speed.
- Running in its own app window, not just a browser window.
- Being launched from the host's operating system start menu, dock, or home screen.
- Receiving push notifications from a backend server, even while the user isn't using the app.
- Automatically updating in the background.

The word *progressive* is used to describe these apps because:

- A user might first discover and use the app within their web browser like any other SPA.
- Later, the user progresses to installing it in their OS and enabling push notifications.

Create a project from the PWA template

Visual Studio

When creating a new **Blazor WebAssembly App**, select the **Progressive Web Application** checkbox.

Convert an existing Blazor WebAssembly app into a PWA

Convert an existing Blazor WebAssembly app into a PWA following the guidance in this section.

In the app's project file:

- Add the following `ServiceWorkerAssetsManifest` property to a `PropertyGroup`:

  XML

  ```
    ...
    <ServiceWorkerAssetsManifest>service-worker-
  assets.js</ServiceWorkerAssetsManifest>
  </PropertyGroup>
  ```

- Add the following `ServiceWorker` item to an `ItemGroup`:

  XML

  ```
  <ItemGroup>
    <ServiceWorker Include="wwwroot\service-worker.js"
      PublishedContent="wwwroot\service-worker.published.js" />
  </ItemGroup>
  ```

To obtain static assets, use **one** of the following approaches:

- Create a separate, new PWA project with the dotnet new command in a command shell:

  .NET CLI

  ```
  dotnet new blazorwasm -o MyBlazorPwa --pwa
  ```

  In the preceding command, the `-o|--output` option creates a new folder for the app named `MyBlazorPwa`.

  If you aren't converting an app for the latest release, pass the `-f|--framework` option. The following example creates the app for ASP.NET Core version 5.0:

  .NET CLI

  ```
  dotnet new blazorwasm -o MyBlazorPwa --pwa -f net5.0
  ```

- Navigate to the ASP.NET Core GitHub repository at the following URL, which links to `main` branch reference source and assets. Select the release that you're working with from the **Switch branches or tags** dropdown list that applies to your app.

  Blazor WebAssembly project template wwwroot folder (dotnet/aspnetcore GitHub repository main branch)

  > **Note**
  >
  > Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

  From the source `wwwroot` folder either in the app that you created or from the reference assets in the `dotnet/aspnetcore` GitHub repository, copy the following files into the app's `wwwroot` folder:

  - `icon-192.png`
  - `icon-512.png`
  - `manifest.webmanifest`
  - `service-worker.js`
  - `service-worker.published.js`

In the app's `wwwroot/index.html` file:

- Add `<link>` elements for the manifest and app icon:

  HTML

  ```
  <link href="manifest.webmanifest" rel="manifest" />
  <link rel="apple-touch-icon" sizes="512x512" href="icon-512.png" />
  <link rel="apple-touch-icon" sizes="192x192" href="icon-192.png" />
  ```

- Add the following `<script>` tag inside the closing `</body>` tag immediately after the `blazor.webassembly.js` script tag:

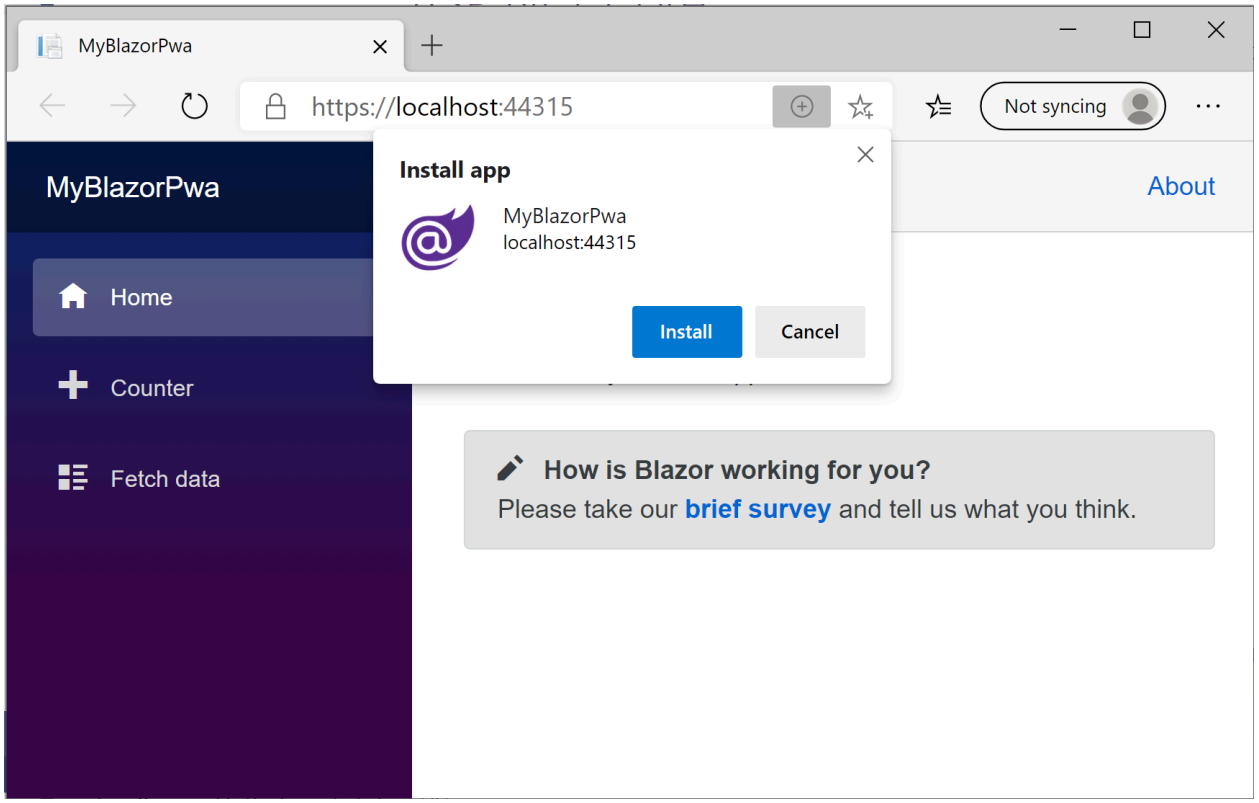
  HTML

  ```
      ...
      <script>navigator.serviceWorker.register('service-worker.js');
  ```

```
</script>
</body>
```

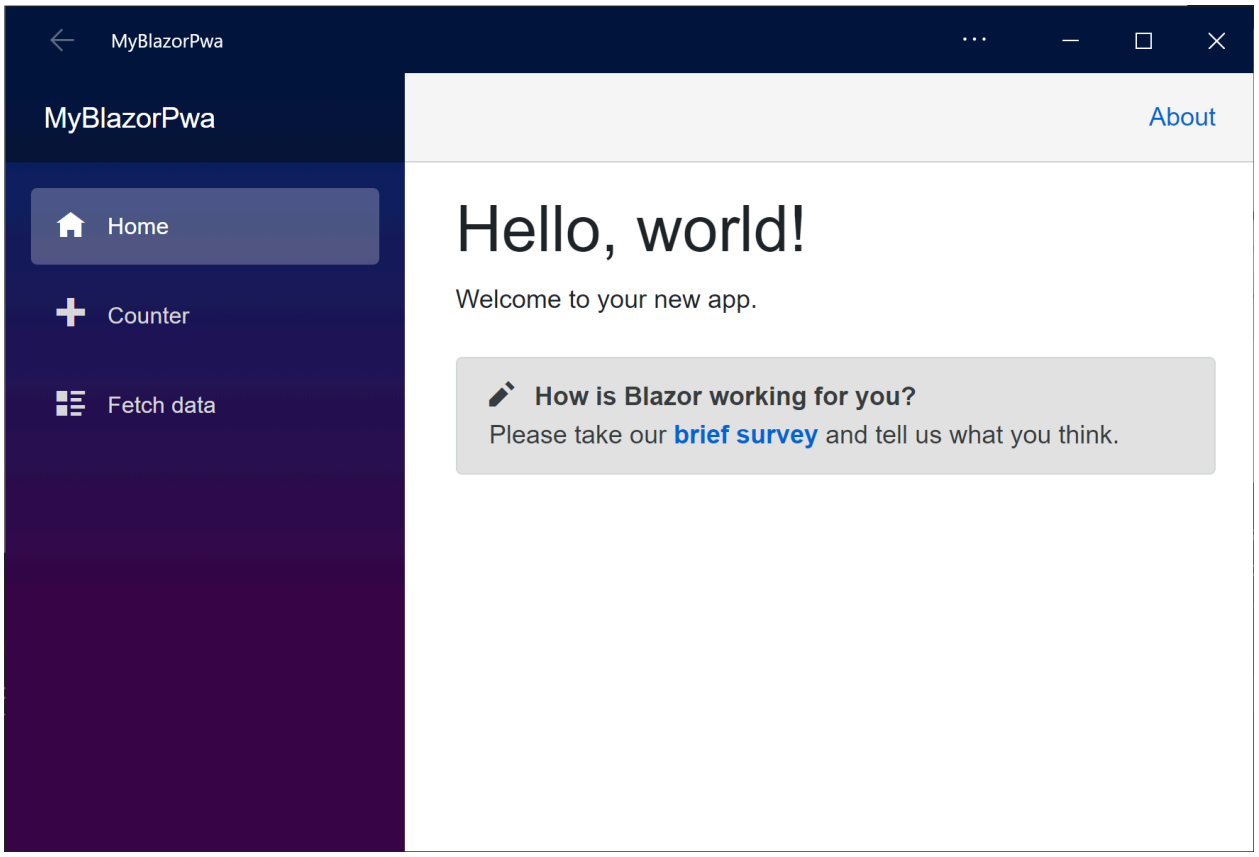
Installation and app manifest

When visiting an app created using the PWA template, users have the option of installing the app into their OS's start menu, dock, or home screen. The way this option is presented depends on the user's browser. When using desktop Chromium-based browsers, such as Edge or Chrome, an **Add** button appears within the URL bar. After the user selects the **Add** button, they receive a confirmation dialog:



On iOS, visitors can install the PWA using Safari's **Share** button and its **Add to Homescreen** option. On Chrome for Android, users should select the **Menu** button in the upper-right corner, followed by **Add to Home screen**.

Once installed, the app appears in its own window without an address bar:



To customize the window's title, color scheme, icon, or other details, see the `manifest.json` file in the project's `wwwroot` directory. The schema of this file is defined by web standards. For more information, see MDN web docs: Web App Manifest.

Offline support

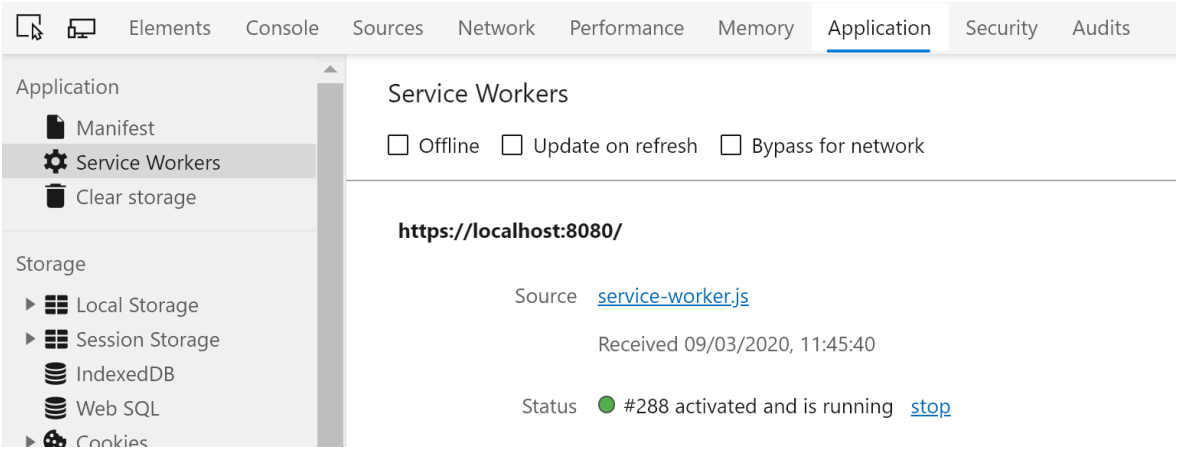
Apps created using the PWA template option have support for running offline. A user must first visit the app while they're online. The browser automatically downloads and caches all of the resources required to operate offline.

> **ⓘ Important**
>
> Development support would interfere with the usual development cycle of making changes and testing them. Therefore, offline support is only enabled for *published* apps.

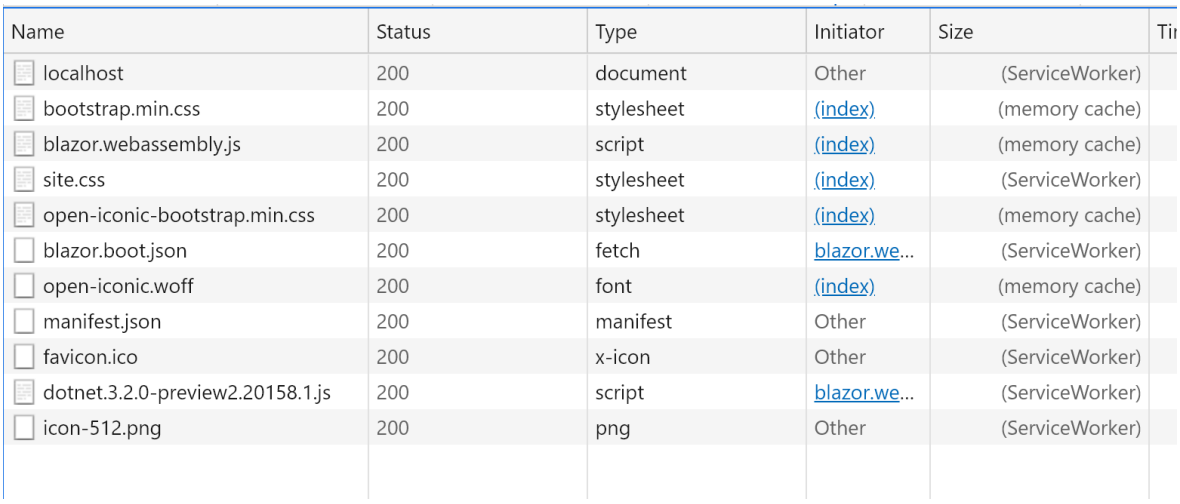
> **⚠ Warning**
>
> If you intend to distribute an offline-enabled PWA, there are several important warnings and caveats. These scenarios are inherent to offline PWAs and not specific to Blazor. Be sure to read and understand these caveats before making assumptions about how your offline-enabled app works.

To see how offline support works:

1. Publish the app. For more information, see Host and deploy ASP.NET Core Blazor.
2. Deploy the app to a server that supports HTTPS, and access the app in a browser at its secure HTTPS address.
3. Open the browser's dev tools and verify that a *Service Worker* is registered for the host on the **Application** tab:

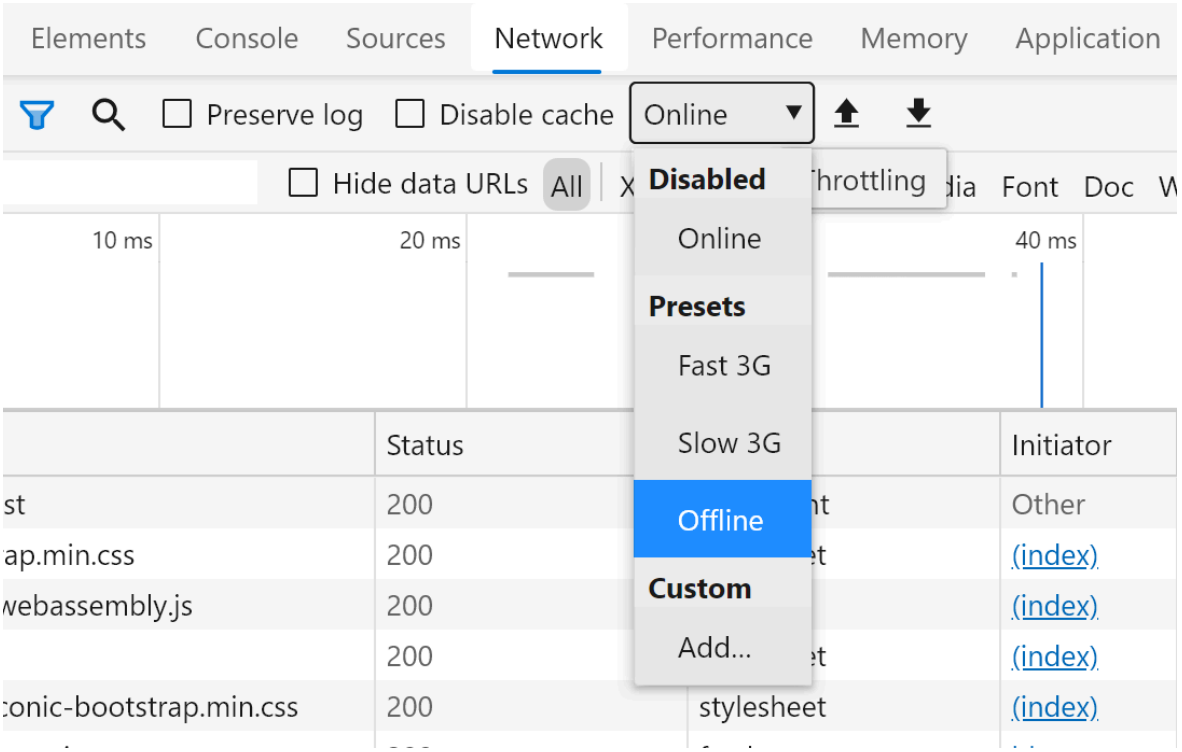


4. Reload the page and examine the **Network** tab. **Service Worker** or **memory cache** are listed as the sources for all of the page's assets:



Name	Status	Type	Initiator	Size	Ti
localhost	200	document	Other	(ServiceWorker)	
bootstrap.min.css	200	stylesheet	(index)	(memory cache)	
blazor.webassembly.js	200	script	(index)	(memory cache)	
site.css	200	stylesheet	(index)	(ServiceWorker)	
open-iconic-bootstrap.min.css	200	stylesheet	(index)	(memory cache)	
blazor.boot.json	200	fetch	blazor.we...	(ServiceWorker)	
open-iconic.woff	200	font	(index)	(memory cache)	
manifest.json	200	manifest	Other	(ServiceWorker)	
favicon.ico	200	x-icon	Other	(ServiceWorker)	
dotnet.3.2.0-preview2.20158.1.js	200	script	blazor.we...	(ServiceWorker)	
icon-512.png	200	png	Other	(ServiceWorker)	

5. To verify that the browser isn't dependent on network access to load the app, either:
   - Shut down the web server and see how the app continues to function normally, which includes page reloads. Likewise, the app continues to function normally when there's a slow network connection.
   - Instruct the browser to simulate offline mode in the **Network** tab:



Offline support using a service worker is a web standard, not specific to Blazor. For more information on service workers, see MDN web docs: Service Worker API. To learn more about common usage patterns for service workers, see Google Web: The Service Worker Lifecycle.

Blazor's PWA template produces two service worker files:

- `wwwroot/service-worker.js`, which is used during development.
- `wwwroot/service-worker.published.js`, which is used after the app is published.

To share logic between the two service worker files, consider the following approach:

- Add a third JavaScript file to hold the common logic.
- Use self.importScripts to load the common logic into both service worker files.

Cache-first fetch strategy

The built-in `service-worker.published.js` service worker resolves requests using a *cache-first* strategy. This means that the service worker prefers to return cached content, regardless of whether the user has network access or newer content is available on the server.

The cache-first strategy is valuable because:

- **It ensures reliability.** Network access isn't a boolean state. A user isn't simply online or offline:

    - The user's device may assume it's online, but the network might be so slow as to be impractical to wait for.
    - The network might return invalid results for certain URLs, such as when there's a captive WIFI portal that's currently blocking or redirecting certain requests.

    This is why the browser's `navigator.onLine` API isn't reliable and shouldn't be depended upon.

- **It ensures correctness.** When building a cache of offline resources, the service worker uses content hashing to guarantee it has fetched a complete and self-consistent snapshot of resources at a single instant in time. This cache is then used as an atomic unit. There's no point asking the network for newer resources, since the only versions required are the ones already cached. Anything else risks inconsistency and incompatibility (for example, trying to use versions of .NET assemblies that weren't compiled together).

If you must prevent the browser from fetching `service-worker-assets.js` from its HTTP cache, for example to resolve temporary integrity check failures when deploying a new version of the service worker, update the service worker registration in `wwwroot/index.html` with updateViaCache set to 'none':

HTML

```
<script>
  navigator.serviceWorker.register('/service-worker.js', {updateViaCache:
'none'});
</script>
```

Background updates

As a mental model, you can think of an offline-first PWA as behaving like a mobile app that can be installed. The app starts up immediately regardless of network connectivity, but the installed app logic comes from a point-in-time snapshot that might not be the latest version.

The Blazor PWA template produces apps that automatically try to update themselves in the background whenever the user visits and has a working network connection. The way this works is as follows:

- During compilation, the project generates a *service worker assets manifest*, which is named `service-worker-assets.js`. The manifest lists all the static resources that the app requires to function offline, such as .NET assemblies, JavaScript files, and

CSS, including their content hashes. The resource list is loaded by the service worker so that it knows which resources to cache.

- Each time the user visits the app, the browser re-requests `service-worker.js` and `service-worker-assets.js` in the background. The files are compared byte-for-byte with the existing installed service worker. If the server returns changed content for either of these files, the service worker attempts to install a new version of itself.
- When installing a new version of itself, the service worker creates a new, separate cache for offline resources and starts populating the cache with resources listed in `service-worker-assets.js`. This logic is implemented in the `onInstall` function inside `service-worker.published.js`.
- The process completes successfully when all of the resources are loaded without error and all content hashes match. If successful, the new service worker enters a *waiting for activation* state. As soon as the user closes the app (no remaining app tabs or windows), the new service worker becomes *active* and is used for subsequent app visits. The old service worker and its cache are deleted.
- If the process doesn't complete successfully, the new service worker instance is discarded. The update process is attempted again on the user's next visit, when hopefully the client has a better network connection that can complete the requests.

Customize this process by editing the service worker logic. None of the preceding behavior is specific to Blazor but is merely the default experience provided by the PWA template option. For more information, see MDN web docs: Service Worker API.

How requests are resolved

As described in the Cache-first fetch strategy section, the default service worker uses a *cache-first* strategy, meaning that it tries to serve cached content when available. If there is no content cached for a certain URL, for example when requesting data from a backend API, the service worker falls back on a regular network request. The network request succeeds if the server is reachable. This logic is implemented inside `onFetch` function within `service-worker.published.js`.

If the app's Razor components rely on requesting data from backend APIs and you want to provide a friendly user experience for failed requests due to network unavailability, implement logic within the app's components. For example, use try/catch around HttpClient requests.

Support server-rendered pages

Consider what happens when the user first navigates to a URL such as `/counter` or any other deep link in the app. In these cases, you don't want to return content cached as `/counter`, but instead need the browser to load the content cached as `/index.html` to start up your Blazor WebAssembly app. These initial requests are known as *navigation* requests, as opposed to:

- `subresource` requests for images, stylesheets, or other files.
- `fetch/XHR` requests for API data.

The default service worker contains special-case logic for navigation requests. The service worker resolves the requests by returning the cached content for `/index.html`, regardless of the requested URL. This logic is implemented in the `onFetch` function inside `service-worker.published.js`.

If your app has certain URLs that must return server-rendered HTML, and not serve `/index.html` from the cache, then you need to edit the logic in your service worker. If all URLs containing `/Identity/` need to be handled as regular online-only requests to the server, then modify `service-worker.published.js` `onFetch` logic. Locate the following code:

JavaScript

```
const shouldServeIndexHtml = event.request.mode === 'navigate';
```

Change the code to the following:

JavaScript

```
const shouldServeIndexHtml = event.request.mode === 'navigate'
  && !event.request.url.includes('/Identity/');
```

If you don't do this, then regardless of network connectivity, the service worker intercepts requests for such URLs and resolves them using `/index.html`.

Add additional endpoints for external authentication providers to the check. In the following example, `/signin-google` for Google authentication is added to the check:

JavaScript

```
const shouldServeIndexHtml = event.request.mode === 'navigate'
  && !event.request.url.includes('/Identity/')
  && !event.request.url.includes('/signin-google');
```

No action is required for the `Development` environment, where content is always fetched from the network.

Control asset caching

If your project defines the `ServiceWorkerAssetsManifest` MSBuild property, Blazor's build tooling generates a service worker assets manifest with the specified name. The default PWA template produces a project file containing the following property:

XML

```
<ServiceWorkerAssetsManifest>service-worker-
assets.js</ServiceWorkerAssetsManifest>
```

The file is placed in the `wwwroot` output directory, so the browser can retrieve this file by requesting `/service-worker-assets.js`. To see the contents of this file, open `/bin/Debug/{TARGET FRAMEWORK}/wwwroot/service-worker-assets.js` in a text editor. However, don't edit the file, as it's regenerated on each build.

The manifest lists:

- Any Blazor-managed resources, such as .NET assemblies and the .NET WebAssembly runtime files required to function offline.
- All resources for publishing to the app's `wwwroot` directory, such as images, stylesheets, and JavaScript files, including static web assets supplied by external projects and NuGet packages.

You can control which of these resources are fetched and cached by the service worker by editing the logic in `onInstall` in `service-worker.published.js`. The service worker fetches and caches files matching typical web file name extensions such as `.html`, `.css`, `.js`, and `.wasm`, plus file types specific to Blazor WebAssembly, such as `.pdb` files (all versions) and `.dll` files (ASP.NET Core in .NET 7 or earlier).

To include additional resources that aren't present in the app's `wwwroot` directory, define extra MSBuild `ItemGroup` entries, as shown in the following example:

XML

```
<ItemGroup>
  <ServiceWorkerAssetsManifestItem Include="MyDirectory\AnotherFile.json"
    RelativePath="MyDirectory\AnotherFile.json"
AssetUrl="files/AnotherFile.json" />
</ItemGroup>
```

The `AssetUrl` metadata specifies the base-relative URL that the browser should use when fetching the resource to cache. This can be independent of its original source file name on disk.

> **ⓘ Important**
>
> Adding a `ServiceWorkerAssetsManifestItem` doesn't cause the file to be published in the app's `wwwroot` directory. The publish output must be controlled separately. The `ServiceWorkerAssetsManifestItem` only causes an additional entry to appear in the service worker assets manifest.

Push notifications

Like any other PWA, a Blazor WebAssembly PWA can receive push notifications from a backend server. The server can send push notifications at any time, even when the user isn't actively using the app. For example, push notifications can be sent when a different user performs a relevant action.

The mechanism for sending a push notification is entirely independent of Blazor WebAssembly, since it's implemented by the backend server which can use any technology. If you want to send push notifications from an ASP.NET Core server, consider using a technique similar to the approach taken in the Blazing Pizza workshop.

The mechanism for receiving and displaying a push notification on the client is also independent of Blazor WebAssembly, since it's implemented in the service worker JavaScript file. For an example, see the approach used in the Blazing Pizza workshop.

Caveats for offline PWAs

Not all apps should attempt to support offline use. Offline support adds significant complexity, while not always being relevant for the use cases required.

Offline support is usually relevant only:

- If the primary data store is local to the browser. For example, the approach is relevant in an app with a UI for an IoT device that stores data in `localStorage` or IndexedDB.
- If the app performs a significant amount of work to fetch and cache the backend API data relevant to each user so that they can navigate through the data offline. If

the app must support editing, a system for tracking changes and synchronizing data with the backend must be built.
- If the goal is to guarantee that the app loads immediately regardless of network conditions. Implement a suitable user experience around backend API requests to show the progress of requests and behave gracefully when requests fail due to network unavailability.

Additionally, offline-capable PWAs must deal with a range of additional complications. Developers should carefully familiarize themselves with the caveats in the following sections.

Offline support only when published

During development you typically want to see each change reflected immediately in the browser without going through a background update process. Therefore, Blazor's PWA template enables offline support only when published.

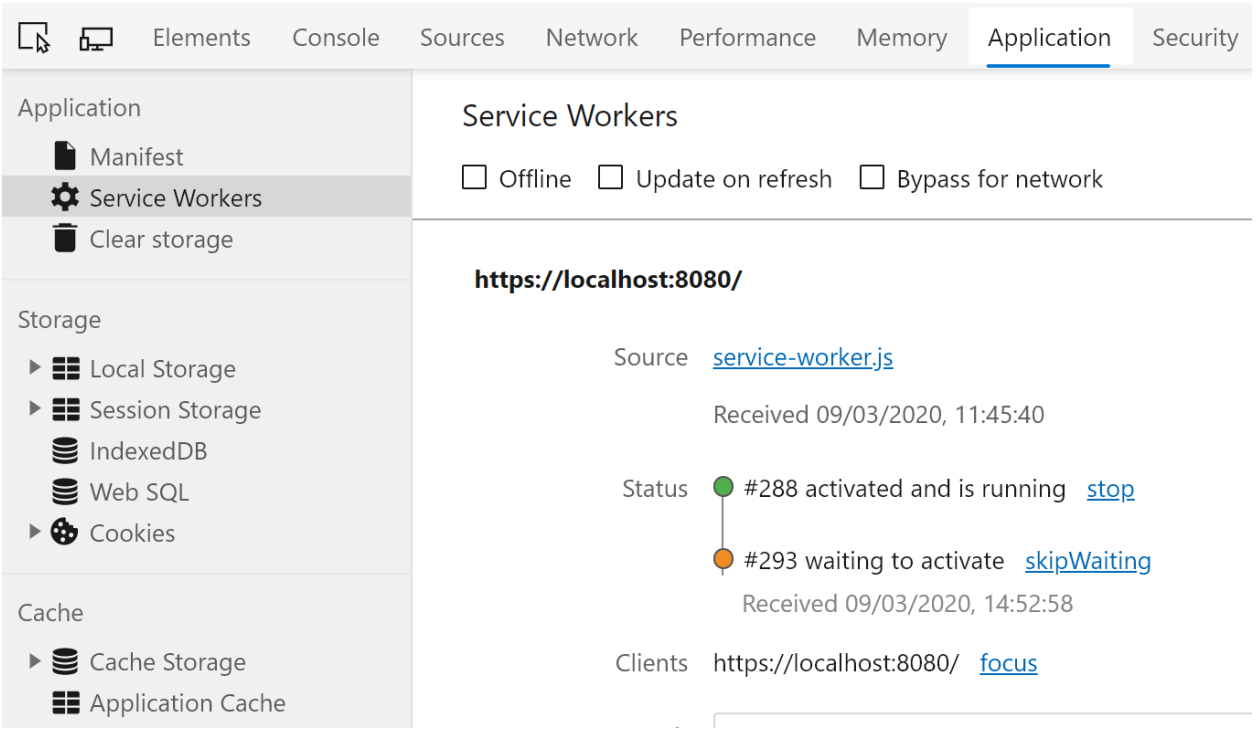
When building an offline-capable app, it's not enough to test the app in the `Development` environment. You must test the app in its published state to understand how it responds to different network conditions.

Update completion after user navigation away from app

Updates don't complete until the user has navigated away from the app in all tabs. As explained in the Background updates section, after you deploy an update to the app, the browser fetches the updated service worker files to begin the update process.

What surprises many developers is that, even when this update completes, it doesn't take effect until the user has navigated away in all tabs. It isn't sufficient to refresh the tab displaying the app, even if it's the only tab displaying the app. Until your app is completely closed, the new service worker remains in the *waiting to activate* status. This isn't specific to Blazor, but rather is a standard web platform behavior.

This commonly troubles developers who are trying to test updates to their service worker or offline cached resources. If you check in the browser's developer tools, you may see something like the following:



For as long as the list of "clients," which are tabs or windows displaying your app, is nonempty, the worker continues waiting. The reason service workers do this is to guarantee consistency. Consistency means that all resources are fetched from the same atomic cache.

When testing changes, you may find it convenient to select the "skipWaiting" link as shown in the preceding screenshot, then reload the page. You can automate this for all users by coding your service worker to skip the "waiting" phase and immediately activate on update. If you skip the waiting phase, you're giving up the guarantee that resources are always fetched consistently from the same cache instance.

Users may run any historical version of the app

Web developers habitually expect that users only run the latest deployed version of their web app, since that's normal within the traditional web distribution model. However, an offline-first PWA is more akin to a native mobile app, where users aren't necessarily running the latest version.

As explained in the Background updates section, after you deploy an update to your app, each existing user continues to use a previous version for at least one further visit because the update occurs in the background and isn't activated until the user thereafter navigates away. Plus, the previous version being used isn't necessarily the previous one you deployed. The previous version can be *any* historical version, depending on when the user last completed an update.

This can be an issue if the frontend and backend parts of your app require agreement about the schema for API requests. You must not deploy backward-incompatible API

schema changes until you can be sure that all users have upgraded. Alternatively, block users from using incompatible older versions of the app. This scenario requirement is the same as for native mobile apps. If you deploy a breaking change in server APIs, the client app is broken for users who haven't yet updated.

If possible, don't deploy breaking changes to your backend APIs. If you must do so, consider using standard Service Worker APIs such as ServiceWorkerRegistration to determine whether the app is up-to-date, and if not, to prevent usage.

Interference with server-rendered pages

As described in the Support server-rendered pages section, if you want to bypass the service worker's behavior of returning `/index.html` contents for all navigation requests, edit the logic in your service worker.

All service worker asset manifest contents are cached

As described in the Control asset caching section, the file `service-worker-assets.js` is generated during build and lists all assets the service worker should fetch and cache.

Since this list includes everything emitted to `wwwroot`, including content supplied by external packages and projects, you must be careful not to put too much content there. If the `wwwroot` directory contains millions of images, the service worker tries to fetch and cache them all, consuming excessive bandwidth and most likely not completing successfully.

Implement arbitrary logic to control which subset of the manifest's contents should be fetched and cached by editing the `onInstall` function in `service-worker.published.js`.

Interaction with authentication

The PWA template can be used in conjunction with authentication. An offline-capable PWA can also support authentication when the user has initial network connectivity.

When a user doesn't have network connectivity, they can't authenticate or obtain access tokens. Attempting to visit the login page without network access results in a "network error" message. You must design a UI flow that allows the user perform useful tasks while offline without attempting to authenticate the user or obtain access tokens. Alternatively, you can design the app to gracefully fail when the network isn't available. If the app can't be designed to handle these scenarios, you might not want to enable offline support.

When an app that's designed for online and offline use is online again:

- The app might need to provision a new access token.
- The app must detect if a different user is signed into the service so that it can apply operations to the user's account that were made while they were offline.

To create an offline PWA app that interacts with authentication:

- Replace the AccountClaimsPrincipalFactory<TAccount> with a factory that stores the last signed-in user and uses the stored user when the app is offline.
- Enqueue operations while the app is offline and apply them when the app returns online.
- During sign out, clear the stored user.

The CarChecker sample app demonstrates the preceding approaches. See the following parts of the app:

- `OfflineAccountClaimsPrincipalFactory` (`Client/Data/OfflineAccountClaimsPrincipalFactory.cs`)
- `LocalVehiclesStore` (`Client/Data/LocalVehiclesStore.cs`)
- `LoginStatus` component (`Client/Shared/LoginStatus.razor`)

Additional resources

- Troubleshoot integrity PowerShell script
- Client-side SignalR cross-origin negotiation for authentication

Host and deploy ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to host and deploy Blazor apps.

Publish the app

Apps are published for deployment in Release configuration.

Visual Studio

1. Select the **Publish {APPLICATION}** command from the **Build** menu, where the `{APPLICATION}` placeholder the app's name.
2. Select the *publish target*. To publish locally, select **Folder**.
3. Accept the default location in the **Choose a folder** field or specify a different location. Select the **`Publish`** button.

Publishing the app triggers a restore of the project's dependencies and builds the project before creating the assets for deployment. As part of the build process, unused methods and assemblies are removed to reduce app download size and load times.

Publish locations:

- Blazor Web App: The app is published into the `/bin/Release/{TARGET FRAMEWORK}/publish` folder, where the `{TARGET FRAMEWORK}` placeholder is the target framework. Deploy the contents of the `publish` folder to the host.
- Standalone Blazor WebAssembly: The app is published into the `bin\Release\{TARGET FRAMEWORK}\browser-wasm\publish\` folder. To deploy the app as a static site, copy the contents of the `wwwroot` folder to the static site host.

IIS

To host a Blazor app in IIS, see the following resources:

- IIS hosting
  - Publish an ASP.NET Core app to IIS
  - Host ASP.NET Core on Windows with IIS
- Host and deploy ASP.NET Core server-side Blazor apps: Server apps running on IIS, including IIS with Azure Virtual Machines (VMs) running Windows OS and Azure App Service.
- Host and deploy ASP.NET Core Blazor WebAssembly: Includes additional guidance for Blazor WebAssembly apps hosted on IIS, including static site hosting, custom `web.config` files, URL rewriting, sub-apps, compression, and Azure Storage static file hosting.
- IIS sub-application hosting
  - Follow the guidance in the App base path section for the Blazor app prior to publishing the app. The examples use an app base path of `/CoolApp` and show how to obtain the base path from app settings or other configuration providers.
  - Follow the sub-application configuration guidance in Advanced configuration. The sub-app's folder path under the root site becomes the virtual path of the sub-app. For an app base path of `/CoolApp`, the Blazor app is placed in a folder named `CoolApp` under the root site and the sub-app takes on a virtual path of `/CoolApp`.

Sharing an app pool among ASP.NET Core apps isn't supported, including for Blazor apps. Use one app pool per app when hosting with IIS, and avoid the use of IIS's virtual directories for hosting multiple apps.

App base path

The *app base path* is the app's root URL path. Successful routing in Blazor apps requires framework configuration for any root URL path that isn't at the default app base path `/`.

Consider the following ASP.NET Core app and Blazor sub-app:

- The ASP.NET Core app is named `MyApp`:
  - The app physically resides at `d:/MyApp`.
  - Requests are received at `https://www.contoso.com/{MYAPP RESOURCE}`.
- A Blazor app named `CoolApp` is a sub-app of `MyApp`:
  - The sub-app physically resides at `d:/MyApp/CoolApp`.
  - Requests are received at `https://www.contoso.com/CoolApp/{COOLAPP RESOURCE}`.

Without specifying additional configuration for `CoolApp`, the sub-app in this scenario has no knowledge of where it resides on the server. For example, the app can't construct correct relative URLs to its resources without knowing that it resides at the relative URL

path `/CoolApp/`. This scenario also applies in various hosting and reverse proxy scenarios when an app isn't hosted at a root URL path.

Background

An anchor tag's destination (href) can be composed with either of two endpoints:

- Absolute locations that include a scheme (defaults to the page's scheme if omitted), host, port, and path or just a forward slash (`/`) followed by the path.

  Examples: `https://example.com/a/b/c` or `/a/b/c`

- Relative locations that contain just a path and do not start with a forward slash (`/`). These are resolved relative to the current document URL or the `<base>` tag's value, if specified.

  Example: `a/b/c`

The presence of a trailing slash (`/`) in a configured app base path is significant to compute the base path for URLs of the app. For example, `https://example.com/a` has a base path of `https://example.com/`, while `https://example.com/a/` with a trailing slash has a base path of `https://example.com/a`.

For the sources of links that pertain to Blazor in ASP.NET Core apps:

- URLs in Razor components (`.razor`) are typically relative.
- URLs in scripts, such as the Blazor scripts (`blazor.*.js`), are relative to the document.

If you're rendering a Blazor app from different documents (for example, `/Admin/B/C/` and `/Admin/D/E/`), you must take the app base path into account, or the base path is different when the app renders in each document and the resources are fetched from the wrong URLs.

There are two approaches to deal with the challenge of resolving relative links correctly:

- Map the resources dynamically using the document they were rendered on as the root.
- Set a consistent base path for the document and map the resources under that base path.

The first option is more complicated and isn't the most typical approach, as it makes navigation different for each document. Consider the following example for rendering a

page `/Something/Else`:

- Rendered under `/Admin/B/C/`, the page is rendered with a path of `/Admin/B/C/Something/Else`.
- Rendered under `/Admin/D/E/`, the page is rendered ***at the same path*** of `/Admin/B/C/Something/Else`.

Under the first approach, routing offers IDynamicEndpointMetadata and MatcherPolicy, which in combination can be the basis for implementing a completely dynamic solution that determines at runtime about how requests are routed.

For the second option, which is the usual approach taken, the app sets the base path in the document and maps the server endpoints to paths under the base. The following guidance adopts this approach.

Server-side Blazor

Map the SignalR hub of a server-side Blazor app by passing the path to MapBlazorHub in the `Program` file:

C#

```
app.MapBlazorHub("base/path");
```

The benefit of using MapBlazorHub is that you can map patterns, such as `"{tenant}"` and not just concrete paths.

You can also map the SignalR hub when the app is in a virtual folder with a branched middleware pipeline. In the following example, requests to `/base/path/` are handled by Blazor's SignalR hub:

C#

```
app.Map("/base/path/", subapp => {
    subapp.UsePathBase("/base/path/");
    subapp.UseRouting();
    subapp.UseEndpoints(endpoints => endpoints.MapBlazorHub());
});
```

Configure the `<base>` tag, per the guidance in the Configure the app base path section.

Standalone Blazor WebAssembly

In a standalone Blazor WebAssembly app, only the `<base>` tag is configured, per the guidance in the Configure the app base path section.

Configure the app base path

To provide configuration for the Blazor app's base path of `https://www.contoso.com/CoolApp/`, set the app base path (<base>), which is also called the relative root path.

By configuring the app base path, a component that isn't in the root directory can construct URLs relative to the app's root path. Components at different levels of the directory structure can build links to other resources at locations throughout the app. The app base path is also used to intercept selected hyperlinks where the `href` target of the link is within the app base path URI space. The Router component handles the internal navigation.

Place the the `<base>` tag in `<head>` markup (location of <head> content) before any elements with attribute values that are URLs, such as the `href` attributes of `<link>` elements.

In many hosting scenarios, the relative URL path to the app is the root of the app. In these default cases, the app's relative URL base path is `/` configured as `<base href="/" />` in <head> content.

> **Note**
>
> In some hosting scenarios, such as GitHub Pages and IIS sub-apps, the app base path must be set to the server's relative URL path of the app.

- In a server-side Blazor app, use ***either*** of the following approaches:
  - Option 1: Use the `<base>` tag to set the app's base path (location of <head> content):

    ```
    HTML
    <base href="/CoolApp/">
    ```

    The trailing slash is required.
  - Option 2: Call UsePathBase ***first*** in the app's request processing pipeline (`Program.cs`) immediately after the WebApplicationBuilder is built

(`builder.Build()`) to configure the base path for any following middleware that interacts with the request path:

```
C#

app.UsePathBase("/CoolApp");
```

Calling UsePathBase is recommended when you also wish to run the Blazor Server app locally. For example, supply the launch URL in `Properties/launchSettings.json`:

```
XML

"launchUrl": "https://localhost:{PORT}/CoolApp",
```

The `{PORT}` placeholder in the preceding example is the port that matches the secure port in the `applicationUrl` configuration path. The following example shows the full launch profile for an app at port 7279:

```
XML

"BlazorSample": {
  "commandName": "Project",
  "dotnetRunMessages": true,
  "launchBrowser": true,
  "applicationUrl": "https://localhost:7279;http://localhost:5279",
  "launchUrl": "https://localhost:7279/CoolApp",
  "environmentVariables": {
    "ASPNETCORE_ENVIRONMENT": "Development"
}
```

For more information on the `launchSettings.json` file, see Use multiple environments in ASP.NET Core. For additional information on Blazor app base paths and hosting, see <base href="/" /> or base-tag alternative for Blazor MVC integration (dotnet/aspnetcore #43191).

- Standalone Blazor WebAssembly (`wwwroot/index.html`):

```
HTML

<base href="/CoolApp/">
```

The trailing slash is required.

> **Note**
>
> When using WebApplication (see Migrate from ASP.NET Core 5.0 to 6.0), app.UseRouting must be called after UsePathBase so that the Routing Middleware can observe the modified path before matching routes. Otherwise, routes are matched before the path is rewritten by UsePathBase as described in the Middleware Ordering and Routing articles.

Don't prefix links throughout the app with a forward slash. Either avoid the use of a path segment separator or use dot-slash (`./`) relative path notation:

- ❌ Incorrect: `<a href="/account">`
- ✔️ Correct: `<a href="account">`
- ✔️ Correct: `<a href="./account">`

In Blazor WebAssembly web API requests with the HttpClient service, confirm that JSON helpers (HttpClientJsonExtensions) don't prefix URLs with a forward slash (`/`):

- ❌ Incorrect: `var rsp = await client.GetFromJsonAsync("/api/Account");`
- ✔️ Correct: `var rsp = await client.GetFromJsonAsync("api/Account");`

Don't prefix Navigation Manager relative links with a forward slash. Either avoid the use of a path segment separator or use dot-slash (`./`) relative path notation (`Navigation` is an injected NavigationManager):

- ❌ Incorrect: `Navigation.NavigateTo("/other");`
- ✔️ Correct: `Navigation.NavigateTo("other");`
- ✔️ Correct: `Navigation.NavigateTo("./other");`

In typical configurations for Azure/IIS hosting, additional configuration usually isn't required. In some non-IIS hosting and reverse proxy hosting scenarios, additional Static File Middleware configuration might be required:

- To serve static files correctly (for example, `app.UseStaticFiles("/CoolApp");`).
- To serve the Blazor script (`_framework/blazor.*.js`). For more information, see ASP.NET Core Blazor static files.

For a Blazor WebAssembly app with a non-root relative URL path (for example, `<base href="/CoolApp/">`), the app fails to find its resources *when run locally*. To overcome this problem during local development and testing, you can supply a *path base* argument that matches the `href` value of the `<base>` tag at runtime. **Don't include a trailing slash.**

To pass the path base argument when running the app locally, execute the `dotnet watch` (or `dotnet run`) command from the app's directory with the `--pathbase` option:

.NET CLI

```
dotnet watch --pathbase=/{RELATIVE URL PATH (no trailing slash)}
```

For a Blazor WebAssembly app with a relative URL path of `/CoolApp/` (`<base href="/CoolApp/">`), the command is:

.NET CLI

```
dotnet watch --pathbase=/CoolApp
```

If you prefer to configure the app's launch profile to specify the `pathbase` automatically instead of manually with `dotnet watch` (or `dotnet run`), set the `commandLineArgs` property in `Properties/launchSettings.json`. The following also configures the launch URL (`launchUrl`):

JSON

```
"commandLineArgs": "--pathbase=/{RELATIVE URL PATH (no trailing slash)}",
"launchUrl": "{RELATIVE URL PATH (no trailing slash)}",
```

Using `CoolApp` as the example:

JSON

```
"commandLineArgs": "--pathbase=/CoolApp",
"launchUrl": "CoolApp",
```

Using either `dotnet watch` (or `dotnet run`) with the `--pathbase` option or a launch profile configuration that sets the base path, the Blazor WebAssembly app responds locally at `http://localhost:port/CoolApp`.

For more information on the `launchSettings.json` file, see Use multiple environments in ASP.NET Core. For additional information on Blazor app base paths and hosting, see <base href="/" /> or base-tag alternative for Blazor MVC integration (dotnet/aspnetcore #43191).

Obtain the app base path from configuration

The following guidance explains how to obtain the path for the `<base>` tag from an app settings file for different environments.

Add the app settings file to the app. The following example is for the `Staging` environment (`appsettings.Staging.json`):

JSON

```
{
  "AppBasePath": "staging/"
}
```

In a server-side Blazor app, load the base path from configuration in <head> content:

razor

```
@inject IConfiguration Config

...

<head>
    ...
    <base href="/@(Config.GetValue<string>("AppBasePath"))" />
    ...
</head>
```

Alternatively, a server-side app can obtain the value from configuration for UsePathBase. Place the following code ***first*** in the app's request processing pipeline (`Program.cs`) immediately after the WebApplicationBuilder is built (`builder.Build()`). The following example uses the configuration key `AppBasePath`:

C#

```
app.UsePathBase($"/{app.Configuration.GetValue<string>("AppBasePath")}");
```

In a client-side Blazor WebAssembly app:

- Remove the `<base>` tag from `wwwroot/index.html`:

  diff

  ```
  - <base href="..." />
  ```

- Supply the app base path via a HeadContent component in the `App` component (`App.razor`):

razor

```razor
@inject IConfiguration Config

...

<HeadContent>
    <base href="/@(Config.GetValue<string>("AppBasePath"))" />
</HeadContent>
```

If there's no configuration value to load, for example in non-staging environments, the preceding `href` resolves to the root path `/`.

The examples in this section focus on supplying the app base path from app settings, but the approach of reading the path from IConfiguration is valid for any configuration provider. For more information, see the following resources:

- ASP.NET Core Blazor configuration
- Configuration in ASP.NET Core

Blazor Server `MapFallbackToPage` configuration

This section only applies to Blazor Server apps. MapFallbackToPage isn't supported in Blazor Web Apps and Blazor WebAssembly apps.

In scenarios where an app requires a separate area with custom resources and Razor components:

- Create a folder within the app's `Pages` folder to hold the resources. For example, an administrator section of an app is created in a new folder named `Admin` (`Pages/Admin`).
- Create a root page (`_Host.cshtml`) for the area. For example, create a `Pages/Admin/_Host.cshtml` file from the app's main root page (`Pages/_Host.cshtml`). Don't provide an `@page` directive in the Admin `_Host` page.
- Add a layout to the area's folder (for example, `Pages/Admin/_Layout.razor`). In the layout for the separate area, set the `<base>` tag `href` to match the area's folder (for example, `<base href="/Admin/" />`). For demonstration purposes, add `~/` to the static resources in the page. For example:
  - `~/css/bootstrap/bootstrap.min.css`
  - `~/css/site.css`
  - `~/BlazorSample.styles.css` (the example app's namespace is `BlazorSample`)

    - `~/_framework/blazor.server.js` (Blazor script)

- If the area should have its own static asset folder, add the folder and specify its location to Static File Middleware in `Program.cs` (for example, `app.UseStaticFiles("/Admin/wwwroot")`).

- Razor components are added to the area's folder. At a minimum, add an `Index` component to the area folder with the correct `@page` directive for the area. For example, add a `Pages/Admin/Index.razor` file based on the app's default `Pages/Index.razor` file. Indicate the Admin area as the route template at the top of the file (`@page "/admin"`). Add additional components as needed. For example, `Pages/Admin/Component1.razor` with an `@page` directive and route template of `@page "/admin/component1`.

- In `Program.cs`, call MapFallbackToPage for the area's request path immediately before the fallback root page path to the `_Host` page:

```
C#

...
app.UseRouting();

app.MapBlazorHub();
app.MapFallbackToPage("~/Admin/{*clientroutes:nonfile}",
"/Admin/_Host");
app.MapFallbackToPage("/_Host");

app.Run();
```

Deployment

For deployment guidance, see the following topics:

- Host and deploy ASP.NET Core Blazor WebAssembly
- Host and deploy ASP.NET Core server-side Blazor apps

Host and deploy server-side Blazor apps

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to host and deploy server-side Blazor apps (Blazor Web Apps and Blazor Server apps) using ASP.NET Core.

Host configuration values

Server-side Blazor apps can accept Generic Host configuration values.

Deployment

Using a server-side hosting model, Blazor is executed on the server from within an ASP.NET Core app. UI updates, event handling, and JavaScript calls are handled over a SignalR connection.

A web server capable of hosting an ASP.NET Core app is required. Visual Studio includes a server-side app project template. For more information on Blazor project templates, see ASP.NET Core Blazor project structure.

Publish an app in Release configuration and deploy the contents of the `bin/Release/{TARGET FRAMEWORK}/publish` folder, where the `{TARGET FRAMEWORK}` placeholder is the target framework.

Scalability

When considering the scalability of a single server (scale up), the memory available to an app is likely the first resource that the app exhausts as user demands increase. The available memory on the server affects the:

- Number of active circuits that a server can support.
- UI latency on the client.

For guidance on building secure and scalable server-side Blazor apps, see the following resources:

- Threat mitigation guidance for ASP.NET Core Blazor static server-side rendering
- Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering

Each circuit uses approximately 250 KB of memory for a minimal *Hello World*-style app. The size of a circuit depends on the app's code and the state maintenance requirements associated with each component. We recommend that you measure resource demands during development for your app and infrastructure, but the following baseline can be a starting point in planning your deployment target: If you expect your app to support 5,000 concurrent users, consider budgeting at least 1.3 GB of server memory to the app (or ~273 KB per user).

SignalR configuration

SignalR's hosting and scaling conditions apply to Blazor apps that use SignalR.

For more information on SignalR in Blazor apps, including configuration guidance, see ASP.NET Core Blazor SignalR guidance.

Transports

Blazor works best when using WebSockets as the SignalR transport due to lower latency, better reliability, and improved security. Long Polling is used by SignalR when WebSockets isn't available or when the app is explicitly configured to use Long Polling.

A console warning appears if Long Polling is utilized:

> Failed to connect via WebSockets, using the Long Polling fallback transport. This may be due to a VPN or proxy blocking the connection.

Global deployment and connection failures

Recommendations for global deployments to geographical data centers:

- Deploy the app to the regions where most of the users reside.
- Take into consideration the increased latency for traffic across continents. To control the appearance of the reconnection UI, see ASP.NET Core Blazor SignalR guidance.
- Consider using the Azure SignalR Service.

Azure App Service

Hosting on Azure App Service requires configuration for WebSockets and session affinity, also called Application Request Routing (ARR) affinity.

> ⓘ **Note**
>
> A Blazor app on Azure App Service doesn't require Azure SignalR Service.

Enable the following for the app's registration in Azure App Service:

- WebSockets to allow the WebSockets transport to function. The default setting is **Off**.
- Session affinity to route requests from a user back to the same App Service instance. The default setting is **On**.

1. In the Azure portal, navigate to the web app in **App Services**.
2. Open **Settings** > **Configuration**.
3. Set **Web sockets** to **On**.
4. Verify that **Session affinity** is set to **On**.

Azure SignalR Service

The optional Azure SignalR Service works in conjunction with the app's SignalR hub for scaling up a server-side app to a large number of concurrent connections. In addition, the service's global reach and high-performance data centers significantly aid in reducing latency due to geography.

The service isn't required for Blazor apps hosted in Azure App Service or Azure Container Apps but can be helpful in other hosting environments:

- To facilitate connection scale out.
- Handle global distribution.

The Azure SignalR Service with SDK v1.26.1 or later supports SignalR stateful reconnect (WithStatefulReconnect).

In the event that the app uses Long Polling or falls back to Long Polling instead of WebSockets, you may need to configure the maximum poll interval (`MaxPollIntervalInSeconds`, default: 5 seconds, limit: 1-300 seconds), which defines the maximum poll interval allowed for Long Polling connections in the Azure SignalR

Service. If the next poll request doesn't arrive within the maximum poll interval, the service closes the client connection.

For guidance on how to add the service as a dependency to a production deployment, see Publish an ASP.NET Core SignalR app to Azure App Service.

For more information, see:

- Azure SignalR Service
- What is Azure SignalR Service?
- ASP.NET Core SignalR production hosting and scaling
- Publish an ASP.NET Core SignalR app to Azure App Service

Azure Container Apps

For a deeper exploration of scaling server-side Blazor apps on the Azure Container Apps service, see Scaling ASP.NET Core Apps on Azure. The tutorial explains how to create and integrate the services required to host apps on Azure Container Apps. Basic steps are also provided in this section.

1. Configure Azure Container Apps service for session affinity by following the guidance in Session Affinity in Azure Container Apps (Azure documentation).

2. The ASP.NET Core Data Protection (DP) service must be configured to persist keys in a centralized location that all container instances can access. The keys can be stored in Azure Blob Storage and protected with Azure Key Vault. The DP service uses the keys to deserialize Razor components. To configure the DP service to use Azure Blob Storage and Azure Key Vault, reference the following NuGet packages:

   - Azure.Identity: Provides classes to work with the Azure identity and access management services.
   - Microsoft.Extensions.Azure: Provides helpful extension methods to perform core Azure configurations.
   - Azure.Extensions.AspNetCore.DataProtection.Blobs: Allows storing ASP.NET Core Data Protection keys in Azure Blob Storage so that keys can be shared across several instances of a web app.
   - Azure.Extensions.AspNetCore.DataProtection.Keys: Enables protecting keys at rest using the Azure Key Vault Key Encryption/Wrapping feature.

   > **Note**
   >
   > For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet

documentation). Confirm correct package versions at NuGet.org.

3. Update `Program.cs` with the following highlighted code:

```C#
using Azure.Identity;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.Azure;
var builder = WebApplication.CreateBuilder(args);
var BlobStorageUri = builder.Configuration["AzureURIs:BlobStorage"];
var KeyVaultURI = builder.Configuration["AzureURIs:KeyVault"];

builder.Services.AddRazorPages();
builder.Services.AddHttpClient();
builder.Services.AddServerSideBlazor();

builder.Services.AddAzureClientsCore();

builder.Services.AddDataProtection()
                .PersistKeysToAzureBlobStorage(new Uri(BlobStorageUri),
                                               new
DefaultAzureCredential())
                .ProtectKeysWithAzureKeyVault(new Uri(KeyVaultURI),
                                               new
DefaultAzureCredential());
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding changes allow the app to manage the DP service using a centralized, scalable architecture. DefaultAzureCredential discovers the container app managed identity after the code is deployed to Azure and uses it to connect to blob storage and the app's key vault.

4. To create the container app managed identity and grant it access to blob storage and a key vault, complete the following steps:
   a. In the Azure Portal, navigate to the overview page of the container app.
   b. Select **Service Connector** from the left navigation.
   c. Select **+ Create** from the top navigation.
   d. In the **Create connection** flyout menu, enter the following values:

      - **Container**: Select the container app you created to host your app.
      - **Service type**: Select **Blob Storage**.
      - **Subscription**: Select the subscription that owns the container app.
      - **Connection name**: Enter a name of `scalablerazorstorage`.
      - **Client type**: Select **.NET** and then select **Next**.

   e. Select **System assigned managed identity** and select **Next**.
   f. Use the default network settings and select **Next**.
   g. After Azure validates the settings, select **Create**.

   Repeat the preceding settings for the key vault. Select the appropriate key vault service and key in the **Basics** tab.

IIS

When using IIS, enable:

- WebSockets on IIS.
- Session affinity with Application Request Routing.

For more information, see the guidance and external IIS resource cross-links in Publish an ASP.NET Core app to IIS.

Kubernetes

Create an ingress definition with the following Kubernetes annotations for session affinity :

YAML

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: <ingress-name>
  annotations:
    nginx.ingress.kubernetes.io/affinity: "cookie"
    nginx.ingress.kubernetes.io/session-cookie-name: "affinity"
```

```
    nginx.ingress.kubernetes.io/session-cookie-expires: "14400"
    nginx.ingress.kubernetes.io/session-cookie-max-age: "14400"
```

Linux with Nginx

Follow the guidance for an ASP.NET Core SignalR app with the following changes:

- Change the `location` path from `/hubroute` (`location /hubroute { ... }`) to the root path `/` (`location / { ... }`).
- Remove the configuration for proxy buffering (`proxy_buffering off;`) because the setting only applies to Server-Sent Events (SSE), which aren't relevant to Blazor app client-server interactions.

For more information and configuration guidance, consult the following resources:

- ASP.NET Core SignalR production hosting and scaling
- Host ASP.NET Core on Linux with Nginx
- Configure ASP.NET Core to work with proxy servers and load balancers
- NGINX as a WebSocket Proxy
- WebSocket proxying
- Consult developers on non-Microsoft support forums:
  - Stack Overflow (tag: blazor)
  - ASP.NET Core Slack Team
  - Blazor Gitter

Linux with Apache

To host a Blazor app behind Apache on Linux, configure `ProxyPass` for HTTP and WebSockets traffic.

In the following example:

- Kestrel server is running on the host machine.
- The app listens for traffic on port 5000.

```

ProxyPreserveHost   On
ProxyPassMatch      ^/_blazor/(.*) http://localhost:5000/_blazor/$1
ProxyPass           /_blazor ws://localhost:5000/_blazor
ProxyPass           / http://localhost:5000/
ProxyPassReverse    / http://localhost:5000/
```

Enable the following modules:

```
a2enmod   proxy
a2enmod   proxy_wstunnel
```

Check the browser console for WebSockets errors. Example errors:

- Firefox can't establish a connection to the server at ws://the-domain-name.tld/_blazor?id=XXX
- Error: Failed to start the transport 'WebSockets': Error: There was an error with the transport.
- Error: Failed to start the transport 'LongPolling': TypeError: this.transport is undefined
- Error: Unable to connect to the server with any of the available transports. WebSockets failed
- Error: Cannot send data if the connection is not in the 'Connected' State.

For more information and configuration guidance, consult the following resources:

- Configure ASP.NET Core to work with proxy servers and load balancers
- Apache documentation
- Consult developers on non-Microsoft support forums:
  - Stack Overflow (tag: blazor)
  - ASP.NET Core Slack Team
  - Blazor Gitter

Measure network latency

JS interop can be used to measure network latency, as the following example demonstrates.

`MeasureLatency.razor`:

razor

```
@inject IJSRuntime JS

<h2>Measure Latency</h2>

@if (latency is null)
{
    <span>Calculating...</span>
}
```

```
else
{
    <span>@(latency.Value.TotalMilliseconds)ms</span>
}

@code {
    private DateTime startTime;
    private TimeSpan? latency;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (firstRender)
        {
            startTime = DateTime.UtcNow;
            var _ = await JS.InvokeAsync<string>("toString");
            latency = DateTime.UtcNow - startTime;
            StateHasChanged();
        }
    }
}
```

For a reasonable UI experience, we recommend a sustained UI latency of 250 ms or less.

Memory management

On the server, a new circuit is created for each user session. Each user session corresponds to rendering a single document in the browser. For example, multiple tabs create multiple sessions.

Blazor maintains a constant connection to the browser, called a *circuit*, that initiated the session. Connections can be lost at any time for any of several reasons, such as when the user loses network connectivity or abruptly closes the browser. When a connection is lost, Blazor has a recovery mechanism that places a limited number of circuits in a "disconnected" pool, giving clients a limited amount of time to reconnect and re-establish the session (default: 3 minutes).

After that time, Blazor releases the circuit and discards the session. From that point on, the circuit is eligible for garbage collection (GC) and is claimed when a collection for the circuit's GC generation is triggered. One important aspect to understand is that circuits have a long lifetime, which means that most of the objects rooted by the circuit eventually reach Gen 2. As a result, you might not see those objects released until a Gen 2 collection happens.

Measure memory usage in general

Prerequisites:

- The app must be published in **Release** configuration. **Debug** configuration measurements aren't relevant, as the generated code isn't representative of the code used for a production deployment.
- The app must run without a debugger attached, as this might also affect the behavior of the app and spoil the results. In Visual Studio, start the app without debugging by selecting **Debug** > **Start Without Debugging** from the menu bar or <kbd>Ctrl</kbd>+<kbd>F5</kbd> using the keyboard.
- Consider the different types of memory to understand how much memory is actually used by .NET. Generally, developers inspect app memory usage in Task Manager on Windows OS, which typically offers an upper bound of the actual memory in use. For more information, consult the following articles:
  - .NET Memory Performance Analysis : In particular, see the section on Memory Fundamentals .
  - Work flow of diagnosing memory performance issues (three-part series) : Links to the three articles of the series are at the top of each article in the series.

Memory usage applied to Blazor

We compute the memory used by blazor as follows:

(**Active Circuits** × **Per-circuit Memory**) + (**Disconnected Circuits** × **Per-circuit Memory**)

The amount of memory a circuit uses and the maximum potential active circuits that an app can maintain is largely dependent on how the app is written. The maximum number of possible active circuits is roughly described by:

Maximum Available Memory / **Per-circuit Memory** = **Maximum Potential Active Circuits**

For a memory leak to occur in Blazor, the following must be true:

- The memory must be allocated by the framework, not the app. If you allocate a 1 GB array in the app, the app must manage the disposal of the array.
- The memory must not be actively used, which means the circuit isn't active and has been evicted from the disconnected circuits cache. If you have the maximum active circuits running, running out of memory is a scale issue, not a memory leak.
- A garbage collection (GC) for the circuit's GC generation has run, but the garbage collector hasn't been able to claim the circuit because another object in the framework is holding a strong reference to the circuit.

In other cases, there's no memory leak. If the circuit is active (connected or disconnected), the circuit is still in use.

If a collection for the circuit's GC generation doesn't run, the memory isn't released because the garbage collector doesn't need to free the memory at that time.

If a collection for a GC generation runs and frees the circuit, you must validate the memory against the GC stats, not the process, as .NET might decide to keep the virtual memory active.

If the memory isn't freed, you must find a circuit that isn't either active or disconnected and that's rooted by another object in the framework. In any other case, the inability to free memory is an app issue in developer code.

Reduce memory usage

Adopt any of the following strategies to reduce an app's memory usage:

- Limit the total amount of memory used by the .NET process. For more information, see Runtime configuration options for garbage collection.
- Reduce the number of disconnected circuits.
- Reduce the time a circuit is allowed to be in the disconnected state.
- Trigger a garbage collection manually to perform a collection during downtime periods.
- Configure the garbage collection in Workstation mode, which aggressively triggers garbage collection, instead of Server mode.

Heap size for some mobile device browsers

When building a Blazor app that runs on the client and targets mobile device browsers, especially Safari on iOS, decreasing the maximum memory for the app with the MSBuild property `EmccMaximumHeapSize` may be required. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

Additional actions and considerations

- Capture a memory dump of the process when memory demands are high and identify the objects are taking the most memory and where are those objects are rooted (what holds a reference to them).
- You can examine the statistics on how memory in your app is behaving using `dotnet-counters`. For more information see Investigate performance counters (dotnet-counters).
- Even when a GC is triggered, .NET holds on to the memory instead of returning it to the OS immediately, as it's likely that it will reuse the memory the near future.

This avoids committing and decommitting memory constantly, which is expensive. You'll see this reflected if you use `dotnet-counters` because you'll see the GCs happen and the amount of used memory go down to 0 (zero), but you won't see the working set counter decrease, which is the sign that .NET is holding on to the memory to reuse it. For more information on project file (`.csproj`) settings to control this behavior, see Runtime configuration options for garbage collection.

- Server GC doesn't trigger garbage collections until it determines it's absolutely necessary to do so to avoid freezing your app and considers that your app is the only thing running on the machine, so it can use all the memory in the system. If the system has 50 GB, the garbage collector seeks to use the full 50 GB of available memory before it triggers a Gen 2 collection.
- For information on disconnected circuit retention configuration, see ASP.NET Core Blazor SignalR guidance.

Measuring memory

- Publish the app in Release configuration.
- Run a published version of the app.
- Don't attach a debugger to the running app.
- Does triggering a Gen 2 forced, compacting collection (`GC.Collect(2, GCCollectionMode.Aggressive | GCCollectionMode.Forced, blocking: true, compacting: true))` free the memory?
- Consider if your app is allocating objects on the large object heap.
- Are you testing the memory growth after the app is warmed up with requests and processing? Typically, there are caches that are populated when code executes for the first time that add a constant amount of memory to the footprint of the app.

Host and deploy ASP.NET Core Blazor WebAssembly

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to host and deploy Blazor WebAssembly using ASP.NET Core, Content Delivery Networks (CDN), file servers, and GitHub Pages.

With the Blazor WebAssembly hosting model:

- The Blazor app, its dependencies, and the .NET runtime are downloaded to the browser in parallel.
- The app is executed directly on the browser UI thread.

This article pertains to the deployment scenario where the Blazor app is placed on a static hosting web server or service, .NET isn't used to serve the Blazor app. This strategy is covered in the Standalone deployment section, which includes information on hosting a Blazor WebAssembly app as an IIS sub-app.

Subdomain and IIS sub-application hosting

Subdomain hosting doesn't require special configuration of the app. You ***don't*** need to configure the app base path (the `<base>` tag in `wwwroot/index.html`) to host the app at a subdomain.

IIS sub-application hosting ***does*** require you to set the app base path. For more information and cross-links to further guidance on IIS sub-application hosting, see Host and deploy ASP.NET Core Blazor.

Decrease maximum heap size for some mobile device browsers

When building a Blazor app that runs on the client (`.Client` project of a Blazor Web App or standalone Blazor WebAssembly app) and targets mobile device browsers,

especially Safari on iOS, decreasing the maximum memory for the app with the MSBuild property `EmccMaximumHeapSize` may be required. The default value is 2,147,483,648 bytes, which may be too large and result in the app crashing if the app attempts to allocate more memory with the browser failing to grant it. The following example sets the value to 268,435,456 bytes in the `Program` file:

XML

```
<EmccMaximumHeapSize>268435456</EmccMaximumHeapSize>
```

For more information on Mono/WebAssembly MSBuild properties and targets, see WasmApp.Common.targets (dotnet/runtime GitHub repository).

Webcil packaging format for .NET assemblies

Webcil is a web-friendly packaging format for .NET assemblies designed to enable using Blazor WebAssembly in restrictive network environments. Webcil files use a standard WebAssembly wrapper, where the assemblies are deployed as WebAssembly files that use the standard `.wasm` file extension.

Webcil is the default packaging format when you publish a Blazor WebAssembly app. To disable the use of Webcil, set the following MSBuild property in the app's project file:

XML

```
<PropertyGroup>
  <WasmEnableWebcil>false</WasmEnableWebcil>
</PropertyGroup>
```

Customize how boot resources are loaded

Customize how boot resources are loaded using the `loadBootResource` API. For more information, see ASP.NET Core Blazor startup.

Compression

When a Blazor WebAssembly app is published, the output is statically compressed during publish to reduce the app's size and remove the overhead for runtime compression. The following compression algorithms are used:

- Brotli (highest level)

- Gzip

Blazor relies on the host to serve the appropriate compressed files. When hosting a Blazor WebAssembly standalone app, additional work might be required to ensure that statically-compressed files are served:

- For IIS `web.config` compression configuration, see the IIS: Brotli and Gzip compression section.
- When hosting on static hosting solutions that don't support statically-compressed file content negotiation, consider configuring the app to fetch and decode Brotli compressed files:

Obtain the JavaScript Brotli decoder from the google/brotli GitHub repository. The minified decoder file is named `decode.min.js` and found in the repository's js folder.

> **Note**
>
> If the minified version of the `decode.js` script (`decode.min.js`) fails, try using the unminified version (`decode.js`) instead.

Update the app to use the decoder.

In the `wwwroot/index.html` file, set `autostart` to `false` on Blazor's `<script>` tag:

HTML

```
<script src="_framework/blazor.webassembly.js" autostart="false"></script>
```

After Blazor's `<script>` tag and before the closing `</body>` tag, add the following JavaScript code `<script>` block. The following function calls `fetch` with cache: 'no-cache' to keep the browser's cache updated.

Blazor Web App:

HTML

```
<script type="module">
  import { BrotliDecode } from './decode.min.js';
  Blazor.start({
    webAssembly: {
      loadBootResource: function (type, name, defaultUri, integrity) {
        if (type !== 'dotnetjs' && location.hostname !== 'localhost' && type
!== 'configuration' && type !== 'manifest') {
          return (async function () {
            const response = await fetch(defaultUri + '.br', { cache: 'no-
```

```
cache' });
            if (!response.ok) {
              throw new Error(response.statusText);
            }
            const originalResponseBuffer = await response.arrayBuffer();
            const originalResponseArray = new
Int8Array(originalResponseBuffer);
            const decompressedResponseArray =
BrotliDecode(originalResponseArray);
            const contentType = type ===
              'dotnetwasm' ? 'application/wasm' : 'application/octet-
stream';
            return new Response(decompressedResponseArray,
              { headers: { 'content-type': contentType } });
          })();
        }
      }
    }
  });
</script>
```

Standalone Blazor WebAssembly:

HTML

```
<script type="module">
  import { BrotliDecode } from './decode.min.js';
  Blazor.start({
    loadBootResource: function (type, name, defaultUri, integrity) {
      if (type !== 'dotnetjs' && location.hostname !== 'localhost' && type
!== 'configuration') {
        return (async function () {
          const response = await fetch(defaultUri + '.br', { cache: 'no-
cache' });
          if (!response.ok) {
            throw new Error(response.statusText);
          }
          const originalResponseBuffer = await response.arrayBuffer();
          const originalResponseArray = new
Int8Array(originalResponseBuffer);
          const decompressedResponseArray =
BrotliDecode(originalResponseArray);
          const contentType = type ===
            'dotnetwasm' ? 'application/wasm' : 'application/octet-stream';
          return new Response(decompressedResponseArray,
            { headers: { 'content-type': contentType } });
        })();
      }
    }
  });
</script>
```

For more information on loading boot resources, see ASP.NET Core Blazor startup.

To disable compression, add the `CompressionEnabled` MSBuild property to the app's project file and set the value to `false`:

XML

```
<PropertyGroup>
  <CompressionEnabled>false</CompressionEnabled>
</PropertyGroup>
```

The `CompressionEnabled` property can be passed to the dotnet publish command with the following syntax in a command shell:

.NET CLI

```
dotnet publish -p:CompressionEnabled=false
```

Rewrite URLs for correct routing

Routing requests for page components in a Blazor WebAssembly app isn't as straightforward as routing requests in a Blazor Server app. Consider a Blazor WebAssembly app with two components:

- `Main.razor`: Loads at the root of the app and contains a link to the `About` component (`href="About"`).
- `About.razor`: `About` component.

When the app's default document is requested using the browser's address bar (for example, `https://www.contoso.com/`):

1. The browser makes a request.
2. The default page is returned, which is usually `index.html`.
3. `index.html` bootstraps the app.
4. Router component loads, and the Razor `Main` component is rendered.

In the Main page, selecting the link to the `About` component works on the client because the Blazor router stops the browser from making a request on the Internet to `www.contoso.com` for `About` and serves the rendered `About` component itself. All of the requests for internal endpoints *within the Blazor WebAssembly app* work the same way: Requests don't trigger browser-based requests to server-hosted resources on the Internet. The router handles the requests internally.

If a request is made using the browser's address bar for `www.contoso.com/About`, the request fails. No such resource exists on the app's Internet host, so a *404 - Not Found* response is returned.

Because browsers make requests to Internet-based hosts for client-side pages, web servers and hosting services must rewrite all requests for resources not physically on the server to the `index.html` page. When `index.html` is returned, the app's Blazor router takes over and responds with the correct resource.

When deploying to an IIS server, you can use the URL Rewrite Module with the app's published `web.config` file. For more information, see the IIS section.

Standalone deployment

A *standalone deployment* serves the Blazor WebAssembly app as a set of static files that are requested directly by clients. Any static file server is able to serve the Blazor app.

Standalone deployment assets are published into either the `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` or `bin\Release\{TARGET FRAMEWORK}\browser-wasm\publish\` folder (depending on the version of the .NET SDK in use), where the `{TARGET FRAMEWORK}` placeholder is the target framework.

Azure App Service

Blazor WebAssembly apps can be deployed to Azure App Services on Windows, which hosts the app on IIS.

Deploying a standalone Blazor WebAssembly app to Azure App Service for Linux isn't currently supported. We recommend hosting a standalone Blazor WebAssembly app using Azure Static Web Apps, which supports this scenario.

Azure Static Web Apps

Use one of the following approaches to deploy a Blazor WebAssembly app to Azure Static Web Apps:

- Deploy from Visual Studio
- Deploy from Visual Studio Code
- Deploy from GitHub

Deploy from Visual Studio

To deploy from Visual Studio, create a publish profile for Azure Static Web Apps:

1. Save any unsaved work in the project, as a Visual Studio restart might be required during the process.

2. In Visual Studio's **Publish** UI, select **Target** > **Azure** > **Specific Target** > **Azure Static Web Apps** to create a publish profile.

3. If the **Azure WebJobs Tools** component for Visual Studio isn't installed, a prompt appears to install the **ASP.NET and web development** component. Follow the prompts to install the tools using the Visual Studio Installer. Visual Studio closes and reopens automatically while installing the tools. After the tools are installed, start over at the first step to create the publish profile.

4. In the publish profile configuration, provide the **Subscription name**. Select an existing instance, or select **Create a new instance**. When creating a new instance in the Azure portal's **Create Static Web App** UI, set the **Deployment details** > **Source** to **Other**. Wait for the deployment to complete in the Azure portal before proceeding.

5. In the publish profile configuration, select the Azure Static Web Apps instance from the instance's resource group. Select **Finish** to create the publish profile. If Visual Studio prompts to install the Static Web Apps (SWA) CLI, install the CLI by following the prompts. The SWA CLI requires NPM/Node.js (Visual Studio documentation).

After the publish profile is created, deploy the app to the Azure Static Web Apps instance using the publish profile by selecting the **Publish** button.

Deploy from Visual Studio Code

To deploy from Visual Studio Code, see Quickstart: Build your first static site with Azure Static Web Apps.

Deploy from GitHub

To deploy from a GitHub repository, see Tutorial: Building a static web app with Blazor in Azure Static Web Apps.

IIS

IIS is a capable static file server for Blazor apps. To configure IIS to host Blazor, see Build a Static Website on IIS.

Published assets are created in the `/bin/Release/{TARGET FRAMEWORK}/publish` or `bin\Release\{TARGET FRAMEWORK}\browser-wasm\publish` folder, depending on which version of the SDK is used and where the `{TARGET FRAMEWORK}` placeholder is the target framework. Host the contents of the `publish` folder on the web server or hosting service.

web.config

When a Blazor project is published, a `web.config` file is created with the following IIS configuration:

- MIME types
- HTTP compression is enabled for the following MIME types:
  - `application/octet-stream`
  - `application/wasm`
- URL Rewrite Module rules are established:
  - Serve the sub-directory where the app's static assets reside (`wwwroot/{PATH REQUESTED}`).
  - Create SPA fallback routing so that requests for non-file assets are redirected to the app's default document in its static assets folder (`wwwroot/index.html`).

Use a custom `web.config`

To use a custom `web.config` file:

1. Place the custom `web.config` file in the project's root folder.
2. Publish the project. For more information, see Host and deploy ASP.NET Core Blazor.

If the SDK's `web.config` generation or transformation during publish either doesn't move the file to published assets in the `publish` folder or modifies the custom configuration in your custom `web.config` file, use any of the following approaches as needed to take full control of the process:

- If the SDK doesn't generate the file, for example, in a standalone Blazor WebAssembly app at `/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot` or `bin\Release\{TARGET FRAMEWORK}\browser-wasm\publish`, depending on which version of the SDK is used and where the `{TARGET FRAMEWORK}` placeholder is the target framework, set the `<PublishIISAssets>` property to `true` in the project file (`.csproj`). Usually for standalone WebAssembly apps, this is the only required

setting to move a custom `web.config` file and prevent transformation of the file by the SDK.

```xml
<PropertyGroup>
  <PublishIISAssets>true</PublishIISAssets>
</PropertyGroup>
```

- Disable the SDK's `web.config` transformation in the project file (`.csproj`):

```xml
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

- Add a custom target to the project file (`.csproj`) to move a custom `web.config` file. In the following example, the custom `web.config` file is placed by the developer at the root of the project. If the `web.config` file resides elsewhere, specify the path to the file in `SourceFiles`. The following example specifies the `publish` folder with `$(PublishDir)`, but provide a path to `DestinationFolder` for a custom output location.

```xml
<Target Name="CopyWebConfig" AfterTargets="Publish">
  <Copy SourceFiles="web.config" DestinationFolder="$(PublishDir)" />
</Target>
```

Install the URL Rewrite Module

The URL Rewrite Module is required to rewrite URLs. The module isn't installed by default, and it isn't available for install as a Web Server (IIS) role service feature. The module must be downloaded from the IIS website. Use the Web Platform Installer to install the module:

1. Locally, navigate to the URL Rewrite Module downloads page. For the English version, select **WebPI** to download the WebPI installer. For other languages, select the appropriate architecture for the server (x86/x64) to download the installer.
2. Copy the installer to the server. Run the installer. Select the **Install** button and accept the license terms. A server restart isn't required after the install completes.

Configure the website

Set the website's **Physical path** to the app's folder. The folder contains:

- The `web.config` file that IIS uses to configure the website, including the required redirect rules and file content types.
- The app's static asset folder.

Host as an IIS sub-app

If a standalone app is hosted as an IIS sub-app, perform either of the following:

- Disable the inherited ASP.NET Core Module handler.

  Remove the handler in the Blazor app's published `web.config` file by adding a `<handlers>` section to the `<system.webServer>` section of the file:

  XML

  ```xml
  <handlers>
    <remove name="aspNetCore" />
  </handlers>
  ```

- Disable inheritance of the root (parent) app's `<system.webServer>` section using a `<location>` element with `inheritInChildApplications` set to `false`:

  XML

  ```xml
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <location path="." inheritInChildApplications="false">
      <system.webServer>
        <handlers>
          <add name="aspNetCore" ... />
        </handlers>
        <aspNetCore ... />
      </system.webServer>
    </location>
  </configuration>
  ```

  > **Note**
  >
  > Disabling inheritance of the root (parent) app's `<system.webServer>` section is the default configuration for published apps using the .NET SDK.

Removing the handler or disabling inheritance is performed in addition to configuring the app's base path. Set the app base path in the app's `index.html` file to the IIS alias used when configuring the sub-app in IIS.

Configure the app's base path by following the guidance in the Host and deploy ASP.NET Core Blazor article.

Brotli and Gzip compression

This section only applies to standalone Blazor WebAssembly apps.

IIS can be configured via `web.config` to serve Brotli or Gzip compressed Blazor assets for standalone Blazor WebAssembly apps. For an example configuration file, see web.config.

Additional configuration of the example `web.config` file might be required in the following scenarios:

- The app's specification calls for either of the following:
  - Serving compressed files that aren't configured by the example `web.config` file.
  - Serving compressed files configured by the example `web.config` file in an uncompressed format.
- The server's IIS configuration (for example, `applicationHost.config`) provides server-level IIS defaults. Depending on the server-level configuration, the app might require a different IIS configuration than what the example `web.config` file contains.

For more information on custom `web.config` files, see the Use a custom web.config section.

Troubleshooting

If a *500 - Internal Server Error* is received and IIS Manager throws errors when attempting to access the website's configuration, confirm that the URL Rewrite Module is installed. When the module isn't installed, the `web.config` file can't be parsed by IIS. This prevents the IIS Manager from loading the website's configuration and the website from serving Blazor's static files.

For more information on troubleshooting deployments to IIS, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Azure Storage

Azure Storage static file hosting allows serverless Blazor app hosting. Custom domain names, the Azure Content Delivery Network (CDN), and HTTPS are supported.

When the blob service is enabled for static website hosting on a storage account:

- Set the **Index document name** to `index.html`.
- Set the **Error document path** to `index.html`. Razor components and other non-file endpoints don't reside at physical paths in the static content stored by the blob service. When a request for one of these resources is received that the Blazor router should handle, the *404 - Not Found* error generated by the blob service routes the request to the **Error document path**. The `index.html` blob is returned, and the Blazor router loads and processes the path.

If files aren't loaded at runtime due to inappropriate MIME types in the files' `Content-Type` headers, take either of the following actions:

- Configure your tooling to set the correct MIME types (`Content-Type` headers) when the files are deployed.
- Change the MIME types (`Content-Type` headers) for the files after the app is deployed.

  In Storage Explorer (Azure portal) for each file:

  1. Right-click the file and select **Properties**.
  2. Set the **ContentType** and select the **Save** button.

For more information, see Static website hosting in Azure Storage.

Nginx

The following `nginx.conf` file is simplified to show how to configure Nginx to send the `index.html` file whenever it can't find a corresponding file on disk.

```
events { }
http {
    server {
        listen 80;

        location / {
            root      /usr/share/nginx/html;
            try_files $uri $uri/ /index.html =404;
        }
```

```
    }
}
```

When setting the NGINX burst rate limit with limit_req, Blazor WebAssembly apps may require a large `burst` parameter value to accommodate the relatively large number of requests made by an app. Initially, set the value to at least 60:

```
http {
    server {
        ...

        location / {
            ...

            limit_req zone=one burst=60 nodelay;
        }
    }
}
```

Increase the value if browser developer tools or a network traffic tool indicates that requests are receiving a *503 - Service Unavailable* status code.

For more information on production Nginx web server configuration, see Creating NGINX Plus and NGINX Configuration Files.

Apache

To deploy a Blazor WebAssembly app to Apache:

1. Create the Apache configuration file. The following example is a simplified configuration file (`blazorapp.config`):

   ```
   <VirtualHost *:80>
       ServerName www.example.com
       ServerAlias *.example.com

       DocumentRoot "/var/www/blazorapp"
       ErrorDocument 404 /index.html

       AddType application/wasm .wasm

       <Directory "/var/www/blazorapp">
           Options -Indexes
           AllowOverride None
   ```

```
    </Directory>

    <IfModule mod_deflate.c>
        AddOutputFilterByType DEFLATE text/css
        AddOutputFilterByType DEFLATE application/javascript
        AddOutputFilterByType DEFLATE text/html
        AddOutputFilterByType DEFLATE application/octet-stream
        AddOutputFilterByType DEFLATE application/wasm
        <IfModule mod_setenvif.c>
            BrowserMatch ^Mozilla/4 gzip-only-text/html
            BrowserMatch ^Mozilla/4.0[678] no-gzip
            BrowserMatch bMSIE !no-gzip !gzip-only-text/html
        </IfModule>
    </IfModule>

    ErrorLog /var/log/httpd/blazorapp-error.log
    CustomLog /var/log/httpd/blazorapp-access.log common
</VirtualHost>
```

1. Place the Apache configuration file into the `/etc/httpd/conf.d/` directory.
2. Place the app's published assets (`/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot`, where the `{TARGET FRAMEWORK}` placeholder is the target framework) into the `/var/www/blazorapp` directory (the location specified to `DocumentRoot` in the configuration file).
3. Restart the Apache service.

For more information, see mod_mime and mod_deflate.

GitHub Pages

The default GitHub Action, which deploys pages, skips deployment of folders starting with underscore, for example, the `_framework` folder. To deploy folders starting with underscore, add an empty `.nojekyll` file to the Git branch.

Git treats JavaScript (JS) files, such as `blazor.webassembly.js`, as text and converts line endings from CRLF (carriage return-line feed) to LF (line feed) in the deployment pipeline. These changes to JS files produce different file hashes than Blazor sends to the client in the `blazor.boot.json` file. The mismatches result in integrity check failures on the client. One approach to solving this problem is to add a `.gitattributes` file with `*.js binary` line before adding the app's assets to the Git branch. The `*.js binary` line configures Git to treat JS files as binary files, which avoids processing the files in the deployment pipeline. The file hashes of the unprocessed files match the entries in the

`blazor.boot.json` file, and client-side integrity checks pass. For more information, see ASP.NET Core Blazor WebAssembly .NET runtime and app bundle caching.

To handle URL rewrites, add a `wwwroot/404.html` file with a script that handles redirecting the request to the `index.html` page. For an example, see the SteveSandersonMS/BlazorOnGitHubPages GitHub repository:

- wwwroot/404.html
- Live site

When using a project site instead of an organization site, update the `<base>` tag in `wwwroot/index.html`. Set the `href` attribute value to the GitHub repository name with a trailing slash (for example, `/my-repository/`). In the SteveSandersonMS/BlazorOnGitHubPages GitHub repository, the base `href` is updated at publish by the .github/workflows/main.yml configuration file.

> **Note**
>
> The SteveSandersonMS/BlazorOnGitHubPages GitHub repository isn't owned, maintained, or supported by the .NET Foundation or Microsoft.

Standalone with Docker

A standalone Blazor WebAssembly app is published as a set of static files for hosting by a static file server.

To host the app in Docker:

- Choose a Docker container with web server support, such as Ngnix or Apache.
- Copy the `publish` folder assets to a location folder defined in the web server for serving static files.
- Apply additional configuration as needed to serve the Blazor WebAssembly app.

For configuration guidance, see the following resources:

- Nginx section or Apache section of this article
- Docker Documentation

Host configuration values

Blazor WebAssembly apps can accept the following host configuration values as command-line arguments at runtime in the development environment.

Content root

The `--contentroot` argument sets the absolute path to the directory that contains the app's content files (content root). In the following examples, `/content-root-path` is the app's content root path.

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

  .NET CLI

  ```
  dotnet watch --contentroot=/content-root-path
  ```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when the app is run with the Visual Studio Debugger and from a command prompt with `dotnet watch` (or `dotnet run`).

  JSON

  ```
  "commandLineArgs": "--contentroot=/content-root-path"
  ```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

  Console

  ```
  --contentroot=/content-root-path
  ```

Path base

The `--pathbase` argument sets the app base path for an app run locally with a non-root relative URL path (the `<base>` tag `href` is set to a path other than `/` for staging and production). In the following examples, `/relative-URL-path` is the app's path base. For more information, see App base path.

> **Important**
>
> Unlike the path provided to `href` of the `<base>` tag, don't include a trailing slash (`/`) when passing the `--pathbase` argument value. If the app base path is provided

in the `<base>` tag as `<base href="/CoolApp/">` (includes a trailing slash), pass the command-line argument value as `--pathbase=/CoolApp` (no trailing slash).

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

  .NET CLI

  ```
  dotnet watch --pathbase=/relative-URL-path
  ```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when running the app with the Visual Studio Debugger and from a command prompt with `dotnet watch` (or `dotnet run`).

  JSON

  ```
  "commandLineArgs": "--pathbase=/relative-URL-path"
  ```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

  Console

  ```
  --pathbase=/relative-URL-path
  ```

URLs

The `--urls` argument sets the IP addresses or host addresses with ports and protocols to listen on for requests.

- Pass the argument when running the app locally at a command prompt. From the app's directory, execute:

  .NET CLI

  ```
  dotnet watch --urls=http://127.0.0.1:0
  ```

- Add an entry to the app's `launchSettings.json` file in the **IIS Express** profile. This setting is used when running the app with the Visual Studio Debugger and from a command prompt with `dotnet watch` (or `dotnet run`).

JSON

```
"commandLineArgs": "--urls=http://127.0.0.1:0"
```

- In Visual Studio, specify the argument in **Properties** > **Debug** > **Application arguments**. Setting the argument in the Visual Studio property page adds the argument to the `launchSettings.json` file.

Console

```
--urls=http://127.0.0.1:0
```

Configure the Trimmer

Blazor performs Intermediate Language (IL) trimming on each Release build to remove unnecessary IL from the output assemblies. For more information, see Configure the Trimmer for ASP.NET Core Blazor.

Change the file name extension of DLL files

This section applies to ASP.NET Core 6.x and 7.x. In ASP.NET Core in .NET 8 or later, .NET assemblies are deployed as WebAssembly files (`.wasm`) using the Webcil file format. In ASP.NET Core in .NET 8 or later, this section only applies if the Webcil file format has been disabled in the app's project file.

If a firewall, anti-virus program, or network security appliance is blocking the transmission of the app's dynamic-link library (DLL) files (`.dll`), you can follow the guidance in this section to change the file name extensions of the app's published DLL files.

> **Note**
>
> Changing the file name extensions of the app's DLL files might not resolve the problem because many security systems scan the content of the app's files, not merely check file extensions.
>
> For a more robust approach in environments that block the download and execution of DLL files, use ASP.NET Core in .NET 8 or later, which packages .NET assemblies as WebAssembly files (`.wasm`) using the Webcil file format. For more

information, see the *Webcil packaging format for .NET assemblies* section in an 8.0 or later version of this article.

Third-party approaches exist for dealing with this problem. For more information, see the resources at Awesome Blazor.

After publishing the app, use a shell script or DevOps build pipeline to rename `.dll` files to use a different file extension in the directory of the app's published output.

In the following examples:

- PowerShell (PS) is used to update the file extensions.
- `.dll` files are renamed to use the `.bin` file extension from the command line.
- Files listed in the published `blazor.boot.json` file with a `.dll` file extension are updated to the `.bin` file extension.
- If service worker assets are also in use, a PowerShell command updates the `.dll` files listed in the `service-worker-assets.js` file to the `.bin` file extension.

To use a different file extension than `.bin`, replace `.bin` in the following commands with the desired file extension.

On Windows:

PowerShell

```
dir {PATH} | rename-item -NewName { $_.name -replace ".dll\b",".bin" }
((Get-Content {PATH}\blazor.boot.json -Raw) -replace '.dll"','.bin"') | Set-
Content {PATH}\blazor.boot.json
```

In the preceding command, the `{PATH}` placeholder is the path to the published `_framework` folder (for example, `.\bin\Release\net6.0\browser-wasm\publish\wwwroot\_framework` from the project's root folder).

If service worker assets are also in use:

PowerShell

```
((Get-Content {PATH}\service-worker-assets.js -Raw) -replace
'.dll"','.bin"') | Set-Content {PATH}\service-worker-assets.js
```

In the preceding command, the `{PATH}` placeholder is the path to the published `service-worker-assets.js` file.

On Linux or macOS:

Console

```
for f in {PATH}/*; do mv "$f" "`echo $f | sed -e 's/\.dll/.bin/g'`"; done
sed -i 's/\.dll"/.bin"/g' {PATH}/blazor.boot.json
```

In the preceding command, the `{PATH}` placeholder is the path to the published `_framework` folder (for example, `.\bin\Release\net6.0\browser-wasm\publish\wwwroot\_framework` from the project's root folder).

If service worker assets are also in use:

Console

```
sed -i 's/\.dll"/.bin"/g' {PATH}/service-worker-assets.js
```

In the preceding command, the `{PATH}` placeholder is the path to the published `service-worker-assets.js` file.

To address the compressed `blazor.boot.json.gz` and `blazor.boot.json.br` files, adopt either of the following approaches:

- Remove the compressed `blazor.boot.json.gz` and `blazor.boot.json.br` files. **Compression is disabled with this approach.**
- Recompress the updated `blazor.boot.json` file.

The preceding guidance for the compressed `blazor.boot.json` file also applies when service worker assets are in use. Remove or recompress `service-worker-assets.js.br` and `service-worker-assets.js.gz`. Otherwise, file integrity checks fail in the browser.

The following Windows example for .NET 6 uses a PowerShell script placed at the root of the project. The following script, which disables compression, is the basis for further modification if you wish to recompress the `blazor.boot.json` file.

`ChangeDLLExtensions.ps1:`:

PowerShell

```
param([string]$filepath,[string]$tfm)
dir $filepath\bin\Release\$tfm\browser-wasm\publish\wwwroot\_framework |
rename-item -NewName { $_.name -replace ".dll\b",".bin" }
((Get-Content $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\blazor.boot.json -Raw) -replace
'.dll"','.bin"') | Set-Content $filepath\bin\Release\$tfm\browser-
```

```
wasm\publish\wwwroot\_framework\blazor.boot.json
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\blazor.boot.json.gz
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\blazor.boot.json.br
```

If service worker assets are also in use, add the following commands:

PowerShell

```
((Get-Content $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\service-worker-assets.js -Raw) -replace
'.dll"','.bin"') | Set-Content $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\wwwroot\service-worker-assets.js
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\wwwroot\service-worker-assets.js.gz
Remove-Item $filepath\bin\Release\$tfm\browser-
wasm\publish\wwwroot\_framework\wwwroot\service-worker-assets.js.br
```

In the project file, the script is executed after publishing the app for the `Release` configuration:

XML

```
<Target Name="ChangeDLLFileExtensions" AfterTargets="AfterPublish"
Condition="'$(Configuration)'=='Release'">
  <Exec Command="powershell.exe -command &quot;&amp; {
.\ChangeDLLExtensions.ps1 '$(SolutionDir)' '$(TargetFramework)'}&quot;" />
</Target>
```

> **Note**
>
> When renaming and lazy loading the same assemblies, see the guidance in Lazy load assemblies in ASP.NET Core Blazor WebAssembly.

Usually, the app's server requires static asset configuration to serve the files with the updated extension. For an app hosted by IIS, add a MIME map entry (`<mimeMap>`) for the new file extension in the static content section (`<staticContent>`) in a custom `web.config` file. The following example assumes that the file extension is changed from `.dll` to `.bin`:

XML

```
<staticContent>
  ...
```

```
  <mimeMap fileExtension=".bin" mimeType="application/octet-stream" />
  ...
</staticContent>
```

Include an update for compressed files if compression is in use:

```

<mimeMap fileExtension=".bin.br" mimeType="application/octet-stream" />
<mimeMap fileExtension=".bin.gz" mimeType="application/octet-stream" />
```

Remove the entry for the `.dll` file extension:

```
diff

- <mimeMap fileExtension=".dll" mimeType="application/octet-stream" />
```

Remove entries for compressed `.dll` files if compression is in use:

```
diff

- <mimeMap fileExtension=".dll.br" mimeType="application/octet-stream" />
- <mimeMap fileExtension=".dll.gz" mimeType="application/octet-stream" />
```

For more information on custom `web.config` files, see the Use a custom web.config section.

Prior deployment corruption

Typically on deployment:

- Only the files that have changed are replaced, which usually results in a faster deployment.
- Existing files that aren't part of the new deployment are left in place for use by the new deployment.

In rare cases, lingering files from a prior deployment can corrupt a new deployment. Completely deleting the existing deployment (or locally-published app prior to deployment) may resolve the issue with a corrupted deployment. Often, deleting the existing deployment ***once*** is sufficient to resolve the problem, including for a DevOps build and deploy pipeline.

If you determine that clearing a prior deployment is always required when a DevOps build and deploy pipeline is in use, you can temporarily add a step to the build pipeline to delete the prior deployment for each new deployment until you troubleshoot the exact cause of the corruption.

Resolve integrity check failures

When Blazor WebAssembly downloads an app's startup files, it instructs the browser to perform integrity checks on the responses. Blazor sends SHA-256 hash values for DLL (`.dll`), WebAssembly (`.wasm`), and other files in the `blazor.boot.json` file, which isn't cached on clients. The file hashes of cached files are compared to the hashes in the `blazor.boot.json` file. For cached files with a matching hash, Blazor uses the cached files. Otherwise, files are requested from the server. After a file is downloaded, its hash is checked again for integrity validation. An error is generated by the browser if any downloaded file's integrity check fails.

Blazor's algorithm for managing file integrity:

- Ensures that the app doesn't risk loading an inconsistent set of files, for example if a new deployment is applied to your web server while the user is in the process of downloading the application files. Inconsistent files can result in a malfunctioning app.
- Ensures the user's browser never caches inconsistent or invalid responses, which can prevent the app from starting even if the user manually refreshes the page.
- Makes it safe to cache the responses and not check for server-side changes until the expected SHA-256 hashes themselves change, so subsequent page loads involve fewer requests and complete faster.

If the web server returns responses that don't match the expected SHA-256 hashes, an error similar to the following example appears in the browser's developer console:

> Failed to find a valid digest in the 'integrity' attribute for resource 'https://myapp.example.com/_framework/MyBlazorApp.dll' with computed SHA-256 integrity 'IIa70iwvmEg5WiDV17OpQ5eCztNYqL186J56852RpJY='. The resource has been blocked.

In most cases, the warning doesn't indicate a problem with integrity checking. Instead, the warning usually means that some other problem exists.

For Blazor WebAssembly's boot reference source, see the Boot.WebAssembly.ts file in the dotnet/aspnetcore GitHub repository.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Diagnosing integrity problems

When an app is built, the generated `blazor.boot.json` manifest describes the SHA-256 hashes of boot resources at the time that the build output is produced. The integrity check passes as long as the SHA-256 hashes in `blazor.boot.json` match the files delivered to the browser.

Common reasons why this fails include:

- The web server's response is an error (for example, a *404 - Not Found* or a *500 - Internal Server Error*) instead of the file the browser requested. This is reported by the browser as an integrity check failure and not as a response failure.
- Something has changed the contents of the files between the build and delivery of the files to the browser. This might happen:
  - If you or build tools manually modify the build output.
  - If some aspect of the deployment process modified the files. For example if you use a Git-based deployment mechanism, bear in mind that Git transparently converts Windows-style line endings to Unix-style line endings if you commit files on Windows and check them out on Linux. Changing file line endings change the SHA-256 hashes. To avoid this problem, consider using .gitattributes to treat build artifacts as binary files.
  - The web server modifies the file contents as part of serving them. For example, some content distribution networks (CDNs) automatically attempt to minify HTML, thereby modifying it. You may need to disable such features.
- The `blazor.boot.json` file fails to load properly or is improperly cached on the client. Common causes include either of the following:
  - Misconfigured or malfunctioning custom developer code.
  - One or more misconfigured intermediate caching layers.

To diagnose which of these applies in your case:

1. Note which file is triggering the error by reading the error message.

2. Open your browser's developer tools and look in the *Network* tab. If necessary, reload the page to see the list of requests and responses. Find the file that is triggering the error in that list.
3. Check the HTTP status code in the response. If the server returns anything other than *200 - OK* (or another 2xx status code), then you have a server-side problem to diagnose. For example, status code 403 means there's an authorization problem, whereas status code 500 means the server is failing in an unspecified manner. Consult server-side logs to diagnose and fix the app.
4. If the status code is *200 - OK* for the resource, look at the response content in browser's developer tools and check that the content matches up with the data expected. For example, a common problem is to misconfigure routing so that requests return your `index.html` data even for other files. Make sure that responses to `.wasm` requests are WebAssembly binaries and that responses to `.dll` requests are .NET assembly binaries. If not, you have a server-side routing problem to diagnose.
5. Seek to validate the app's published and deployed output with the Troubleshoot integrity PowerShell script.

If you confirm that the server is returning plausibly correct data, there must be something else modifying the contents in between build and delivery of the file. To investigate this:

- Examine the build toolchain and deployment mechanism in case they're modifying files after the files are built. An example of this is when Git transforms file line endings, as described earlier.
- Examine the web server or CDN configuration in case they're set up to modify responses dynamically (for example, trying to minify HTML). It's fine for the web server to implement HTTP compression (for example, returning `content-encoding: br` or `content-encoding: gzip`), since this doesn't affect the result after decompression. However, it's *not* fine for the web server to modify the uncompressed data.

Troubleshoot integrity PowerShell script

Use the integrity.ps1 PowerShell script to validate a published and deployed Blazor app. The script is provided for PowerShell Core 7 or later as a starting point when the app has integrity issues that the Blazor framework can't identify. Customization of the script might be required for your apps, including if running on version of PowerShell later than version 7.2.0.

The script checks the files in the `publish` folder and downloaded from the deployed app to detect issues in the different manifests that contain integrity hashes. These checks should detect the most common problems:

- You modified a file in the published output without realizing it.
- The app wasn't correctly deployed to the deployment target, or something changed within the deployment target's environment.
- There are differences between the deployed app and the output from publishing the app.

Invoke the script with the following command in a PowerShell command shell:

PowerShell

```
.\integrity.ps1 {BASE URL} {PUBLISH OUTPUT FOLDER}
```

In the following example, the script is executed on a locally-running app at `https://localhost:5001/`:

PowerShell

```
.\integrity.ps1 https://localhost:5001/
C:\TestApps\BlazorSample\bin\Release\net6.0\publish\
```

Placeholders:

- `{BASE URL}`: The URL of the deployed app. A trailing slash (`/`) is required.
- `{PUBLISH OUTPUT FOLDER}`: The path to the app's `publish` folder or location where the app is published for deployment.

> **Note**
>
> When cloning the `dotnet/AspNetCore.Docs` GitHub repository, the `integrity.ps1` script might be quarantined by **Bitdefender** or another virus scanner present on the system. Usually, the file is trapped by a virus scanner's *heuristic scanning* technology, which merely looks for patterns in files that might indicate the presence of malware. To prevent the virus scanner from quarantining the file, add an exception to the virus scanner prior to cloning the repo. The following example is a typical path to the script on a Windows system. Adjust the path as needed for other systems. The `{USER}` placeholder is the user's path segment.
>
> ```
> ```

```
C:\Users\
{USER}\Documents\GitHub\AspNetCore.Docs\aspnetcore\blazor\host-and-
deploy\webassembly\_samples\integrity.ps1
```

Warning: *Creating virus scanner exceptions is dangerous and should only be performed when you're certain that the file is safe.*

Comparing the checksum of a file to a valid checksum value doesn't guarantee file safety, but modifying a file in a way that maintains a checksum value isn't trivial for malicious users. Therefore, checksums are useful as a general security approach. Compare the checksum of the local `integrity.ps1` file to one of the following values:

- SHA256: `32c24cb667d79a701135cb72f6bae490d81703323f61b8af2c7e5e5dc0f0c2bb`
- MD5: `9cee7d7ec86ee809a329b5406fbf21a8`

Obtain the file's checksum on Windows OS with the following command. Provide the path and file name for the `{PATH AND FILE NAME}` placeholder and indicate the type of checksum to produce for the `{SHA512|MD5}` placeholder, either `SHA256` or `MD5`:

Console

```
CertUtil -hashfile {PATH AND FILE NAME} {SHA256|MD5}
```

If you have any cause for concern that checksum validation isn't secure enough in your environment, consult your organization's security leadership for guidance.

For more information, see Overview of threat protection by Microsoft Defender Antivirus.

Disable integrity checking for non-PWA apps

In most cases, don't disable integrity checking. Disabling integrity checking doesn't solve the underlying problem that has caused the unexpected responses and results in losing the benefits listed earlier.

There may be cases where the web server can't be relied upon to return consistent responses, and you have no choice but to temporarily disable integrity checks until the underlying problem is resolved.

To disable integrity checks, add the following to a property group in the Blazor WebAssembly app's project file (`.csproj`):

```xml
<BlazorCacheBootResources>false</BlazorCacheBootResources>
```

`BlazorCacheBootResources` also disables Blazor's default behavior of caching the `.dll`, `.wasm`, and other files based on their SHA-256 hashes because the property indicates that the SHA-256 hashes can't be relied upon for correctness. Even with this setting, the browser's normal HTTP cache may still cache those files, but whether or not this happens depends on your web server configuration and the `cache-control` headers that it serves.

> **Note**
>
> The `BlazorCacheBootResources` property doesn't disable integrity checks for Progressive Web Applications (PWAs). For guidance pertaining to PWAs, see the Disable integrity checking for PWAs section.

We can't provide an exhaustive list of scenarios where disabling integrity checking is required. Servers can answer a request in arbitrary ways outside of the scope of the Blazor framework. The framework provides the `BlazorCacheBootResources` setting to make the app runnable at the cost of *losing a guarantee of integrity that the app can provide*. Again, we don't recommend disabling integrity checking, especially for production deployments. Developers should seek to solve the underlying integrity problem that's causing integrity checking to fail.

A few general cases that can cause integrity issues are:

- Running on HTTP where integrity can't be checked.
- If your deployment process modifies the files after publish in any way.
- If your host modifies the files in any way.

Disable integrity checking for PWAs

Blazor's Progressive Web Application (PWA) template contains a suggested `service-worker.published.js` file that's responsible for fetching and storing application files for offline use. This is a separate process from the normal app startup mechanism and has its own separate integrity checking logic.

Inside the `service-worker.published.js` file, following line is present:

```JavaScript
.map(asset => new Request(asset.url, { integrity: asset.hash }));
```

To disable integrity checking, remove the `integrity` parameter by changing the line to the following:

```JavaScript
.map(asset => new Request(asset.url));
```

Again, disabling integrity checking means that you lose the safety guarantees offered by integrity checking. For example, there is a risk that if the user's browser is caching the app at the exact moment that you deploy a new version, it could cache some files from the old deployment and some from the new deployment. If that happens, the app becomes stuck in a broken state until you deploy a further update.

ASP.NET Core Blazor WebAssembly .NET runtime and app bundle caching

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

When a Blazor WebAssembly app loads in the browser, the app downloads boot resources from the server:

- JavaScript code to bootstrap the app
- .NET runtime and assemblies
- Locale specific data

Except for Blazor's boot resources file (`blazor.boot.json`), WebAssembly .NET runtime and app bundle files are cached on clients. The `blazor.boot.json` file contains a manifest of the files that make up the app that must be downloaded along with a hash of the file's content that's used to detect whether any of the boot resources have changed. Blazor caches downloaded files using the browser Cache API.

When Blazor WebAssembly downloads an app's startup files, it instructs the browser to perform integrity checks on the responses. Blazor sends SHA-256 hash values for DLL (`.dll`), WebAssembly (`.wasm`), and other files in the `blazor.boot.json` file. The file hashes of cached files are compared to the hashes in the `blazor.boot.json` file. For cached files with a matching hash, Blazor uses the cached files. Otherwise, files are requested from the server. After a file is downloaded, its hash is checked again for integrity validation. An error is generated by the browser if any downloaded file's integrity check fails.

Blazor's algorithm for managing file integrity:

- Ensures that the app doesn't risk loading an inconsistent set of files, for example if a new deployment is applied to your web server while the user is in the process of downloading the application files. Inconsistent files can result in a malfunctioning app.
- Ensures the user's browser never caches inconsistent or invalid responses, which can prevent the app from starting even if the user manually refreshes the page.

- Makes it safe to cache the responses and not check for server-side changes until the expected SHA-256 hashes themselves change, so subsequent page loads involve fewer requests and complete faster.

If the web server returns responses that don't match the expected SHA-256 hashes, an error similar to the following example appears in the browser's developer console:

> Failed to find a valid digest in the 'integrity' attribute for resource 'https://myapp.example.com/_framework/MyBlazorApp.dll' with computed SHA-256 integrity 'IIa70iwvmEg5WiDV17OpQ5eCztNYqL186J56852RpJY='. The resource has been blocked.

In most cases, the warning doesn't indicate a problem with integrity checking. Instead, the warning usually means that some other problem exists.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Diagnosing integrity problems

When an app is built, the generated `blazor.boot.json` manifest describes the SHA-256 hashes of boot resources at the time that the build output is produced. The integrity check passes as long as the SHA-256 hashes in `blazor.boot.json` match the files delivered to the browser.

Common reasons why this fails include:

- The web server's response is an error (for example, a *404 - Not Found* or a *500 - Internal Server Error*) instead of the file the browser requested. This is reported by the browser as an integrity check failure and not as a response failure.
- Something has changed the contents of the files between the build and delivery of the files to the browser. This might happen:
  - If you or build tools manually modify the build output.
  - If some aspect of the deployment process modified the files. For example if you use a Git-based deployment mechanism, bear in mind that Git transparently converts Windows-style line endings to Unix-style line endings if you commit

files on Windows and check them out on Linux. Changing file line endings change the SHA-256 hashes. To avoid this problem, consider using .gitattributes to treat build artifacts as binary files.
    - The web server modifies the file contents as part of serving them. For example, some content distribution networks (CDNs) automatically attempt to minify HTML, thereby modifying it. You may need to disable such features.
- The `blazor.boot.json` file fails to load properly or is improperly cached on the client. Common causes include either of the following:
    - Misconfigured or malfunctioning custom developer code.
    - One or more misconfigured intermediate caching layers.

To diagnose which of these applies in your case:

1. Note which file is triggering the error by reading the error message.
2. Open your browser's developer tools and look in the *Network* tab. If necessary, reload the page to see the list of requests and responses. Find the file that is triggering the error in that list.
3. Check the HTTP status code in the response. If the server returns anything other than *200 - OK* (or another 2xx status code), then you have a server-side problem to diagnose. For example, status code 403 means there's an authorization problem, whereas status code 500 means the server is failing in an unspecified manner. Consult server-side logs to diagnose and fix the app.
4. If the status code is *200 - OK* for the resource, look at the response content in browser's developer tools and check that the content matches up with the data expected. For example, a common problem is to misconfigure routing so that requests return your `index.html` data even for other files. Make sure that responses to `.wasm` requests are WebAssembly binaries and that responses to `.dll` requests are .NET assembly binaries. If not, you have a server-side routing problem to diagnose.
5. Seek to validate the app's published and deployed output with the Troubleshoot integrity PowerShell script.

If you confirm that the server is returning plausibly correct data, there must be something else modifying the contents in between build and delivery of the file. To investigate this:

- Examine the build toolchain and deployment mechanism in case they're modifying files after the files are built. An example of this is when Git transforms file line endings, as described earlier.
- Examine the web server or CDN configuration in case they're set up to modify responses dynamically (for example, trying to minify HTML). It's fine for the web server to implement HTTP compression (for example, returning `content-encoding:`

`br` or `content-encoding: gzip`), since this doesn't affect the result after decompression. However, it's *not* fine for the web server to modify the uncompressed data.

Troubleshoot integrity PowerShell script

Use the integrity.ps1 PowerShell script to validate a published and deployed Blazor app. The script is provided for PowerShell Core 7 or later as a starting point when the app has integrity issues that the Blazor framework can't identify. Customization of the script might be required for your apps, including if running on version of PowerShell later than version 7.2.0.

The script checks the files in the `publish` folder and downloaded from the deployed app to detect issues in the different manifests that contain integrity hashes. These checks should detect the most common problems:

- You modified a file in the published output without realizing it.
- The app wasn't correctly deployed to the deployment target, or something changed within the deployment target's environment.
- There are differences between the deployed app and the output from publishing the app.

Invoke the script with the following command in a PowerShell command shell:

PowerShell

```
.\integrity.ps1 {BASE URL} {PUBLISH OUTPUT FOLDER}
```

In the following example, the script is executed on a locally-running app at `https://localhost:5001/`:

PowerShell

```
.\integrity.ps1 https://localhost:5001/
C:\TestApps\BlazorSample\bin\Release\net6.0\publish\
```

Placeholders:

- `{BASE URL}`: The URL of the deployed app. A trailing slash (`/`) is required.
- `{PUBLISH OUTPUT FOLDER}`: The path to the app's `publish` folder or location where the app is published for deployment.

Note

When cloning the `dotnet/AspNetCore.Docs` GitHub repository, the `integrity.ps1` script might be quarantined by **Bitdefender** or another virus scanner present on the system. Usually, the file is trapped by a virus scanner's *heuristic scanning* technology, which merely looks for patterns in files that might indicate the presence of malware. To prevent the virus scanner from quarantining the file, add an exception to the virus scanner prior to cloning the repo. The following example is a typical path to the script on a Windows system. Adjust the path as needed for other systems. The `{USER}` placeholder is the user's path segment.

```
C:\Users\
{USER}\Documents\GitHub\AspNetCore.Docs\aspnetcore\blazor\host-and-
deploy\webassembly\_samples\integrity.ps1
```

Warning: *Creating virus scanner exceptions is dangerous and should only be performed when you're certain that the file is safe.*

Comparing the checksum of a file to a valid checksum value doesn't guarantee file safety, but modifying a file in a way that maintains a checksum value isn't trivial for malicious users. Therefore, checksums are useful as a general security approach. Compare the checksum of the local `integrity.ps1` file to one of the following values:

- SHA256: `32c24cb667d79a701135cb72f6bae490d81703323f61b8af2c7e5e5dc0f0c2bb`
- MD5: `9cee7d7ec86ee809a329b5406fbf21a8`

Obtain the file's checksum on Windows OS with the following command. Provide the path and file name for the `{PATH AND FILE NAME}` placeholder and indicate the type of checksum to produce for the `{SHA512|MD5}` placeholder, either `SHA256` or `MD5`:

Console

```
CertUtil -hashfile {PATH AND FILE NAME} {SHA256|MD5}
```

If you have any cause for concern that checksum validation isn't secure enough in your environment, consult your organization's security leadership for guidance.

For more information, see Overview of threat protection by Microsoft Defender Antivirus.

Disable resource caching and integrity checks for non-PWA apps

In most cases, don't disable integrity checking. Disabling integrity checking doesn't solve the underlying problem that has caused the unexpected responses and results in losing the benefits listed earlier.

There may be cases where the web server can't be relied upon to return consistent responses, and you have no choice but to temporarily disable integrity checks until the underlying problem is resolved.

To disable integrity checks, add the following to a property group in the Blazor WebAssembly app's project file (`.csproj`):

XML

```
<BlazorCacheBootResources>false</BlazorCacheBootResources>
```

`BlazorCacheBootResources` also disables Blazor's default behavior of caching the `.dll`, `.wasm`, and other files based on their SHA-256 hashes because the property indicates that the SHA-256 hashes can't be relied upon for correctness. Even with this setting, the browser's normal HTTP cache may still cache those files, but whether or not this happens depends on your web server configuration and the `cache-control` headers that it serves.

> ⓘ Note
>
> The `BlazorCacheBootResources` property doesn't disable integrity checks for Progressive Web Applications (PWAs). For guidance pertaining to PWAs, see the Disable integrity checking for PWAs section.

We can't provide an exhaustive list of scenarios where disabling integrity checking is required. Servers can answer a request in arbitrary ways outside of the scope of the Blazor framework. The framework provides the `BlazorCacheBootResources` setting to make the app runnable at the cost of *losing a guarantee of integrity that the app can provide*. Again, we don't recommend disabling integrity checking, especially for

production deployments. Developers should seek to solve the underlying integrity problem that's causing integrity checking to fail.

A few general cases that can cause integrity issues are:

- Running on HTTP where integrity can't be checked.
- If your deployment process modifies the files after publish in any way.
- If your host modifies the files in any way.

Disable resource caching and integrity checks for PWAs

Blazor's Progressive Web Application (PWA) template contains a suggested `service-worker.published.js` file that's responsible for fetching and storing application files for offline use. This is a separate process from the normal app startup mechanism and has its own separate integrity checking logic.

Inside the `service-worker.published.js` file, following line is present:

JavaScript

```
.map(asset => new Request(asset.url, { integrity: asset.hash }));
```

To disable integrity checking, remove the `integrity` parameter by changing the line to the following:

JavaScript

```
.map(asset => new Request(asset.url));
```

Again, disabling integrity checking means that you lose the safety guarantees offered by integrity checking. For example, there is a risk that if the user's browser is caching the app at the exact moment that you deploy a new version, it could cache some files from the old deployment and some from the new deployment. If that happens, the app becomes stuck in a broken state until you deploy a further update.

Additional resources

Boot resource loading

Avoid HTTP caching issues when upgrading ASP.NET Core Blazor apps

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

When Blazor apps are incorrectly upgraded or configured, it can result in non-seamless upgrades for existing users. This article discusses some of the common HTTP caching issues that can occur when upgrading Blazor apps across major versions. It also provides some recommended actions to ensure a smooth transition for your users.

While future Blazor releases might provide better solutions for dealing with HTTP caching issues, it's ultimately up to the app to correctly configure caching. Proper caching configuration ensures that the app's users always have the most up-to-date version of the app, improving their experience and reducing the likelihood of encountering errors.

Common problems that negatively impact the user upgrade experience include:

- **Incorrect handling of project and package updates**: This happens if you don't update all of the app's deployed projects to use the same major framework version or if you use packages from a previous version when a newer version is available as part of the major upgrade.
- **Incorrect configuration of caching headers**: HTTP caching headers control how, where, and for how long the app's responses are cached. If headers aren't configured correctly, users might receive stale content.
- **Incorrect configuration of other layers**: Content Delivery Networks (CDNs) and other layers of the deployed app can cause issues if incorrectly configured. For example, CDNs are designed to cache and deliver content to improve performance and reduce latency. If a CDN is incorrectly serving cached versions of assets, it can lead to stale content delivery to the user.

Detect and diagnose upgrade issues

Upgrade issues typically appear as a failure to start the app in the browser. Normally, a warning indicates the presence of a stale asset or an asset that's missing or inconsistent

with the app.

- First, check if the app loads successfully within a clean browser instance. Use a private browser mode to load the app, such as Microsoft Edge InPrivate mode or Google Chrome Incognito mode. If the app fails to load, it likely means that one or more packages or the framework wasn't correctly updated.
- If the app loads correctly in a clean browser instance, then it's likely that the app is being served from a stale cache. In most cases, a hard browser refresh with `Ctrl` + `F5` flushes the cache, which permits the app to load and run with the latest assets.
- If the app continues to fail, then it's likely that a stale CDN cache is serving the app. Try to flush the DNS cache via whatever mechanism your CDN provider offers.

Recommended actions before an upgrade

The prior process for serving the app might make the update process more challenging. For example, avoiding or incorrectly using caching headers in the past can lead to current caching problems for users. You can take the actions in the following sections to mitigate the issue and improve the upgrade process for users.

Align framework packages with the framework version

Ensure that framework packages line up with the framework version. Using packages from a previous version when a newer version is available can lead to compatibility issues. It's also important to ensure that all of the app's deployed projects use the same major framework version. This consistency helps to avoid unexpected behavior and errors.

Verify the presence of correct caching headers

The correct caching headers should be present on responses to resource requests. This includes `ETag`, `Cache-Control`, and other caching headers. The configuration of these headers is dependent on the hosting service or hosting server platform. They are particularly important for assets such as the Blazor script (`blazor.webassembly.js`) and anything the script downloads.

Incorrect HTTP caching headers may also impact service workers. Service workers rely on caching headers to manage cached resources effectively. Therefore, incorrect or missing headers can disrupt the service worker's functionality.

Use `Clear-Site-Data` to delete state in the browser

Consider using the Clear-Site-Data header to delete state in the browser.

Usually the source of cache state problems is limited to the HTTP browser cache, so use of the `cache` directive should be sufficient. This action can help to ensure that the browser fetches the latest resources from the server, rather than serving stale content from the cache.

You can optionally include the `storage` directive to clear local storage caches at the same time that you're clearing the HTTP browser cache. However, apps that use client storage might experience a loss of important information if the `storage` directive is used.

Append a query string to the Blazor script tag

If none of the previous recommended actions are effective, possible to use for your deployment, or apply to your app, consider temporarily appending a query string to the Blazor script's `<script>` tag source. This action should be enough in most situations to force the browser to bypass the local HTTP cache and download a new version of the app. There's no need to read or use the query string in the app.

In the following example, the query string `temporaryQueryString=1` is temporarily applied to the `<script>` tag's relative external source URI:

HTML

```
<script src="_framework/blazor.webassembly.js?temporaryQueryString=1">
</script>
```

After all of the app's users have reloaded the app, the query string can be removed.

Alternatively, you can apply a persistent query string with relevant versioning. The following example assumes that the version of the app matches the .NET release version (`8` for .NET 8):

HTML

```
<script src="_framework/blazor.webassembly.js?version=8"></script>
```

For the location of the Blazor script `<script>` tag, see ASP.NET Core Blazor project structure.

Configure the Trimmer for ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to control the Intermediate Language (IL) Trimmer when building a Blazor app.

Blazor WebAssembly performs Intermediate Language (IL) trimming to reduce the size of the published output. Trimming occurs when publishing an app.

Trimming may have detrimental effects for the published app. In apps that use reflection, the IL Trimmer often can't determine the required types for runtime reflection and trim them away. For example, complex framework types for JS interop, such as KeyValuePair, might be trimmed and not available at runtime for JS interop calls. In these cases, we recommend creating your own custom types instead. The IL Trimmer is also unable to react to an app's dynamic behavior at runtime. To ensure the trimmed app works correctly once deployed, test published output frequently while developing.

Configuration

To configure the IL Trimmer, see the Trimming options article in the .NET Fundamentals documentation, which includes guidance on the following subjects:

- Disable trimming for the entire app with the `<PublishTrimmed>` property in the project file.
- Control how aggressively unused IL is discarded by the IL Trimmer.
- Stop the IL Trimmer from trimming specific assemblies.
- "Root" assemblies for trimming.
- Surface warnings for reflected types by setting the `<SuppressTrimAnalysisWarnings>` property to `false` in the project file.
- Control symbol trimming and debugger support.
- Set IL Trimmer features for trimming framework library features.

Default trimmer granularity

The default trimmer granularity for Blazor apps is `partial`. To trim all assemblies, change the granularity to `full` in the app's project file:

XML

```xml
<ItemGroup>
  <TrimMode>full</TrimMode>
</ItemGroup>
```

For more information, see Trimming options (.NET documentation).

Additional resources

- Trim self-contained deployments and executables
- ASP.NET Core Blazor performance best practices

Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps

Article • 11/12/2024

This article explains how to enable hosted Blazor WebAssembly deployments in environments that block the download and execution of dynamic-link library (DLL) files.

> **Note**
>
> This guidance addresses environments that block clients from downloading and executing DLLs. In .NET 8 or later, Blazor uses the Webcil file format to address this problem. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly. Multipart bundling using the experimental NuGet package described by this article isn't supported for Blazor apps in .NET 8 or later. You can use the guidance in this article to create your own multipart bundling NuGet package for .NET 8 or later.

Blazor WebAssembly apps require dynamic-link libraries (DLLs) to function, but some environments block clients from downloading and executing DLLs. In a subset of these environments, changing the file name extension of DLL files (.dll) is sufficient to bypass security restrictions, but security products are often able to scan the content of files traversing the network and block or quarantine DLL files. This article describes one approach for enabling Blazor WebAssembly apps in these environments, where a multipart bundle file is created from the app's DLLs so that the DLLs can be downloaded together bypassing security restrictions.

A hosted Blazor WebAssembly app can customize its published files and packaging of app DLLs using the following features:

- JavaScript initializers that allow customizing the Blazor boot process.
- MSBuild extensibility to transform the list of published files and define *Blazor Publish Extensions*. Blazor Publish Extensions are files defined during the publish process that provide an alternative representation for the set of files required to run a published Blazor WebAssembly app. In this article, a Blazor Publish Extension is created that produces a multipart bundle with all of the app's DLLs packed into a single file so that the DLLs can be downloaded together.

The approach demonstrated in this article serves as a starting point for developers to devise their own strategies and custom loading processes.

> ⚠ **Warning**
>
> Any approach taken to circumvent a security restriction must be carefully considered for its security implications. We recommend exploring the subject further with your organization's network security professionals before adopting the approach in this article. Alternatives to consider include:
>
> - Enable security appliances and security software to permit network clients to download and use the exact files required by a Blazor WebAssembly app.
> - Switch from the Blazor WebAssembly hosting model to the **Blazor Server hosting model**, which maintains all of the app's C# code on the server and doesn't require downloading DLLs to clients. Blazor Server also offers the advantage of keeping C# code private without requiring the use of web API apps for C# code privacy with Blazor WebAssembly apps.

Experimental NuGet package and sample app

The approach described in this article is used by the *experimental* Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle package (NuGet.org) for apps targeting .NET 6 or later. The package contains MSBuild targets to customize the Blazor publish output and a JavaScript initializer to use a custom boot resource loader, each of which are described in detail later in this article.

Experimental code (includes the NuGet package reference source and CustomPackagedApp sample app)

> ⚠ **Warning**
>
> Experimental and preview features are provided for the purpose of collecting feedback and aren't supported for production use.

Later in this article, the Customize the Blazor WebAssembly loading process via a NuGet package section with its three subsections provide detailed explanations on the configuration and code in the `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle` package. The detailed explanations are important to understand when you create your own strategy and custom loading process for Blazor WebAssembly apps. To use the published, experimental, unsupported NuGet package without customization as a **local demonstration**, perform the following steps:

1. Use an existing hosted Blazor WebAssembly solution or create a new solution from the Blazor WebAssembly project template using Visual Studio or by passing the -ho|--hosted option to the dotnet new command (`dotnet new blazorwasm -ho`). For more information, see Tooling for ASP.NET Core Blazor.

2. In the **Client** project, add the experimental `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle` package.

   > ⓘ **Note**
   >
   > For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

3. In the **Server** project, add an endpoint for serving the bundle file (`app.bundle`). Example code can be found in the Serve the bundle from the host server app section of this article.

4. Publish the app in Release configuration.

Customize the Blazor WebAssembly loading process via a NuGet package

> ⚠ **Warning**
>
> The guidance in this section with its three subsections pertains to building a NuGet package from scratch to implement your own strategy and custom loading process. The *experimental* Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle package (NuGet.org) for .NET 6 and 7 is based on the guidance in this section. When using the provided package in a **local demonstration** of the multipart bundle download approach, you don't need to follow the guidance in this section. For guidance on how to use the provided package, see the Experimental NuGet package and sample app section.

Blazor app resources are packed into a multipart bundle file and loaded by the browser via a custom JavaScript (JS) initializer. For an app consuming the package with the JS initializer, the app only requires that the bundle file is served when requested. All of the other aspects of this approach are handled transparently.

Four customizations are required to how a default published Blazor app loads:

- An MSBuild task to transform the publish files.
- A NuGet package with MSBuild targets that hooks into the Blazor publishing process, transforms the output, and defines one or more Blazor Publish Extension files (in this case, a single bundle).
- A JS initializer to update the Blazor WebAssembly resource loader callback so that it loads the bundle and provides the app with the individual files.
- A helper on the host **Server** app to ensure that the bundle is served to clients on request.

Create an MSBuild task to customize the list of published files and define new extensions

Create an MSBuild task as a public C# class that can be imported as part of an MSBuild compilation and that can interact with the build.

The following are required for the C# class:

- A new class library project.
- A project target framework of `netstandard2.0`.
- References to MSBuild packages:
  - Microsoft.Build.Framework
  - Microsoft.Build.Utilities.Core

> **Note**
>
> The NuGet package for the examples in this article are named after the package provided by Microsoft, `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle`. For guidance on naming and producing your own NuGet package, see the following NuGet articles:
>
> - Package authoring best practices
> - Package ID prefix reservation

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks.csproj`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
    <LangVersion>8.0</LangVersion>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Build.Framework" Version="
{VERSION}" />
    <PackageReference Include="Microsoft.Build.Utilities.Core" Version="
{VERSION}" />
  </ItemGroup>

</Project>
```

Determine the latest package versions for the `{VERSION}` placeholders at NuGet.org:

- Microsoft.Build.Framework
- Microsoft.Build.Utilities.Core

To create the MSBuild task, create a public C# class extending Microsoft.Build.Utilities.Task (not System.Threading.Tasks.Task) and declare three properties:

- `PublishBlazorBootStaticWebAsset`: The list of files to publish for the Blazor app.
- `BundlePath`: The path where the bundle is written.
- `Extension`: The new Publish Extensions to include in the build.

The following example `BundleBlazorAssets` class is a starting point for further customization:

- In the `Execute` method, the bundle is created from the following three file types:
  - JavaScript files (`dotnet.js`)
  - WASM files (`dotnet.wasm`)
  - App DLLs (`.dll`)
- A `multipart/form-data` bundle is created. Each file is added to the bundle with its respective descriptions via the Content-Disposition header and the Content-Type header.
- After the bundle is created, the bundle is written to a file.
- The build is configured for the extension. The following code creates an extension item and adds it to the `Extension` property. Each extension item contains three pieces of data:
  - The path to the extension file.
  - The URL path relative to the root of the Blazor WebAssembly app.
  - The name of the extension, which groups the files produced by a given extension.

After accomplishing the preceding goals, the MSBuild task is created for customizing the Blazor publish output. Blazor takes care of gathering the extensions and making sure that the extensions are copied to the correct location in the publish output folder (for example, `bin\Release\net6.0\publish`). The same optimizations (for example, compression) are applied to the JavaScript, WASM, and DLL files as Blazor applies to other files.

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks/BundleBlazorAssets.cs`:

C#

```
using System.IO;
using System.Net.Http;
using System.Net.Http.Headers;
using Microsoft.Build.Framework;
using Microsoft.Build.Utilities;

namespace Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks
{
    public class BundleBlazorAssets : Task
    {
        [Required]
        public ITaskItem[]? PublishBlazorBootStaticWebAsset { get; set; }

        [Required]
        public string? BundlePath { get; set; }

        [Output]
        public ITaskItem[]? Extension { get; set; }

        public override bool Execute()
        {
            var bundle = new MultipartFormDataContent(
                "--0a7e8441d64b4bf89086b85e59523b7d");

            foreach (var asset in PublishBlazorBootStaticWebAsset)
            {
                var name =
Path.GetFileName(asset.GetMetadata("RelativePath"));
                var fileContents = File.OpenRead(asset.ItemSpec);
                var content = new StreamContent(fileContents);
                var disposition = new ContentDispositionHeaderValue("form-
data");
                disposition.Name = name;
                disposition.FileName = name;
                content.Headers.ContentDisposition = disposition;
                var contentType = Path.GetExtension(name) switch
                {
                    ".js" => "text/javascript",
                    ".wasm" => "application/wasm",
```

```
                    _ => "application/octet-stream"
                };
                content.Headers.ContentType =
                    MediaTypeHeaderValue.Parse(contentType);
                bundle.Add(content);
            }

            using (var output = File.Open(BundlePath,
FileMode.OpenOrCreate))
            {
                output.SetLength(0);

bundle.CopyToAsync(output).ConfigureAwait(false).GetAwaiter()
                    .GetResult();
                output.Flush(true);
            }

            var bundleItem = new TaskItem(BundlePath);
            bundleItem.SetMetadata("RelativePath", "app.bundle");
            bundleItem.SetMetadata("ExtensionName", "multipart");

            Extension = new ITaskItem[] { bundleItem };

            return true;
        }
    }
}
```

Author a NuGet package to automatically transform the publish output

Generate a NuGet package with MSBuild targets that are automatically included when the package is referenced:

- Create a new Razor class library (RCL) project.
- Create a targets file following NuGet conventions to automatically import the package in consuming projects. For example, create `build\net6.0\{PACKAGE ID}.targets`, where `{PACKAGE ID}` is the package identifier of the package.
- Collect the output from the class library containing the MSBuild task and confirm the output is packed in the right location.
- Add the necessary MSBuild code to attach to the Blazor pipeline and invoke the MSBuild task to generate the bundle.

The approach described in this section only uses the package to deliver targets and content, which is different from most packages where the package includes a library DLL.

> ⚠ **Warning**
>
> The sample package described in this section demonstrates how to customize the Blazor publish process. **The sample NuGet package is for use as a local demonstration only. Using this package in production is not supported.**

> ⓘ **Note**
>
> The NuGet package for the examples in this article are named after the package provided by Microsoft, `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle`. For guidance on naming and producing your own NuGet package, see the following NuGet articles:
>
> - Package authoring best practices
> - Package ID prefix reservation

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.csproj`:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <NoWarn>NU5100</NoWarn>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <Description>
      Sample demonstration package showing how to customize the Blazor
publish
      process. Using this package in production is not supported!
    </Description>
    <IsPackable>true</IsPackable>
    <IsShipping>true</IsShipping>
    <IncludeBuildOutput>false</IncludeBuildOutput>
  </PropertyGroup>

  <ItemGroup>
    <None Update="build\**"
          Pack="true"
          PackagePath="%(Identity)" />
    <Content Include="_._"
             Pack="true"
             PackagePath="lib\net6.0\_._" />
  </ItemGroup>
```

```
  <Target Name="GetTasksOutputDlls"
          BeforeTargets="CoreCompile">
    <MSBuild
Projects="..\Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tas
ks\Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks.csproj"
             Targets="Publish;PublishItemsOutputGroup"
             Properties="Configuration=Release">
      <Output TaskParameter="TargetOutputs"
              ItemName="_TasksProjectOutputs" />
    </MSBuild>
    <ItemGroup>
      <Content Include="@(_TasksProjectOutputs)"
               Condition="'%(_TasksProjectOutputs.Extension)' == '.dll'"
               Pack="true"
               PackagePath="tasks\%(_TasksProjectOutputs.TargetPath)"
               KeepMetadata="Pack;PackagePath" />
    </ItemGroup>
  </Target>

</Project>
```

> **Note**
>
> The `<NoWarn>NU5100</NoWarn>` property in the preceding example suppresses the warning about the assemblies placed in the `tasks` folder. For more information, see NuGet Warning NU5100.

Add a `.targets` file to wire up the MSBuild task to the build pipeline. In this file, the following goals are accomplished:

- Import the task into the build process. Note that the path to the DLL is relative to the ultimate location of the file in the package.
- The `ComputeBlazorExtensionsDependsOn` property attaches the custom target to the Blazor WebAssembly pipeline.
- Capture the `Extension` property on the task output and add it to `BlazorPublishExtension` to tell Blazor about the extension. Invoking the task in the target produces the bundle. The list of published files is provided by the Blazor WebAssembly pipeline in the `PublishBlazorBootStaticWebAsset` item group. The bundle path is defined using the `IntermediateOutputPath` (typically inside the `obj` folder). Ultimately, the bundle is copied automatically to the correct location in the publish output folder (for example, `bin\Release\net6.0\publish`).

When the package is referenced, it generates a bundle of the Blazor files during publish.

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle/build/net6.0/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.targets`:

XML

```
<Project>
  <UsingTask

TaskName="Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks.BundleBlazorAssets"

AssemblyFile="$(MSBuildThisProjectFileDirectory)..\..\tasks\Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.Tasks.dll" />

  <PropertyGroup>
    <ComputeBlazorExtensionsDependsOn>
      $(ComputeBlazorExtensionsDependsOn);_BundleBlazorDlls
    </ComputeBlazorExtensionsDependsOn>
  </PropertyGroup>

  <Target Name="_BundleBlazorDlls">
    <BundleBlazorAssets
      PublishBlazorBootStaticWebAsset="@(PublishBlazorBootStaticWebAsset)"
      BundlePath="$(IntermediateOutputPath)bundle.multipart">
      <Output TaskParameter="Extension"
              ItemName="BlazorPublishExtension"/>
    </BundleBlazorAssets>
  </Target>

</Project>
```

Automatically bootstrap Blazor from the bundle

The NuGet package leverages JavaScript (JS) initializers to automatically bootstrap a Blazor WebAssembly app from the bundle instead of using individual DLL files. JS initializers are used to change the Blazor boot resource loader and use the bundle.

To create a JS initializer, add a JS file with the name `{NAME}.lib.module.js` to the `wwwroot` folder of the package project, where the `{NAME}` placeholder is the package identifier. For example, the file for the Microsoft package is named `Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.lib.module.js`. The exported functions `beforeWebAssemblyStart` and `afterWebAssemblyStarted` handle loading.

The JS initializers:

- Detect if the Publish Extension is available by checking for `extensions.multipart`, which is the extension name (`ExtensionName`) provided in the Create an MSBuild task to customize the list of published files and define new extensions section.
- Download the bundle and parse the contents into a resources map using generated object URLs.
- Update the boot resource loader (options.loadBootResource) with a custom function that resolves the resources using the object URLs.
- After the app has started, revoke the object URLs to release memory in the `afterWebAssemblyStarted` function.

`Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle/wwwroot/Microsoft.AspNetCore.Components.WebAssembly.MultipartBundle.lib.module.js`:

JavaScript

```
const resources = new Map();

export async function beforeWebAssemblyStart(options, extensions) {
  if (!extensions || !extensions.multipart) {
    return;
  }

  try {
    const integrity = extensions.multipart['app.bundle'];
    const bundleResponse =
      await fetch('app.bundle', { integrity: integrity, cache: 'no-cache'
});
    const bundleFromData = await bundleResponse.formData();
    for (let value of bundleFromData.values()) {
      resources.set(value, URL.createObjectURL(value));
    }
    options.loadBootResource = function (type, name, defaultUri, integrity)
{
      return resources.get(name) ?? null;
    }
  } catch (error) {
    console.log(error);
  }
}

export async function afterWebAssemblyStarted(blazor) {
  for (const [_, url] of resources) {
    URL.revokeObjectURL(url);
  }
}
```

Serve the bundle from the host server app

Due to security restrictions, ASP.NET Core doesn't serve the `app.bundle` file. A request processing helper is required to serve the file when it's requested by clients.

> **Note**
>
> Since the same optimizations are transparently applied to the Publish Extensions that are applied to the app's files, the `app.bundle.gz` and `app.bundle.br` compressed asset files are produced automatically on publish.

Place C# code in `Program.cs` of the **Server** project immediately before the line that sets the fallback file to `index.html` (`app.MapFallbackToFile("index.html");`) to respond to a request for the bundle file (for example, `app.bundle`):

C#

```
app.MapGet("app.bundle", (HttpContext context) =>
{
    string? contentEncoding = null;
    var contentType =
        "multipart/form-data; boundary=\"-
-0a7e8441d64b4bf89086b85e59523b7d\"";
    var fileName = "app.bundle";

    var acceptEncodings = context.Request.Headers.AcceptEncoding;

    if (Microsoft.Net.Http.Headers.StringWithQualityHeaderValue
        .StringWithQualityHeaderValue
        .TryParseList(acceptEncodings, out var encodings))
    {
        if (encodings.Any(e => e.Value == "br"))
        {
            contentEncoding = "br";
            fileName += ".br";
        }
        else if (encodings.Any(e => e.Value == "gzip"))
        {
            contentEncoding = "gzip";
            fileName += ".gz";
        }
    }

    if (contentEncoding != null)
    {
        context.Response.Headers.ContentEncoding = contentEncoding;
    }

    return Results.File(
        app.Environment.WebRootFileProvider.GetFileInfo(fileName)
```

```
            .CreateReadStream(), contentType);
});
```

The content type matches the type defined earlier in the build task. The endpoint checks for the content encodings accepted by the browser and serves the optimal file, Brotli (`.br`) or Gzip (`.gz`).

ASP.NET Core Blazor with Entity Framework Core (EF Core)

Article • 01/16/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article explains how to use Entity Framework Core (EF Core) in server-side Blazor apps.

Server-side Blazor is a stateful app framework. The app maintains an ongoing connection to the server, and the user's state is held in the server's memory in a *circuit*. One example of user state is data held in dependency injection (DI) service instances that are scoped to the circuit. The unique application model that Blazor provides requires a special approach to use Entity Framework Core.

> **Note**
>
> This article addresses EF Core in server-side Blazor apps. Blazor WebAssembly apps run in a WebAssembly sandbox that prevents most direct database connections. Running EF Core in Blazor WebAssembly is beyond the scope of this article.

This guidance applies to components that adopt interactive server-side rendering (interactive SSR) in a Blazor Web App.

Secure authentication flow required for production apps

This article pertains to the use of a local database that doesn't require user authentication. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production Blazor apps, see the articles in the Blazor *Security and Identity* node.

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Build a Blazor movie database app tutorial

For a tutorial experience building an app that uses EF Core for database operations, see Build a Blazor movie database app (Overview). The tutorial shows you how to create a Blazor Web App that can display and manage movies in a movie database.

Database access

EF Core relies on a DbContext as the means to configure database access and act as a *unit of work*. EF Core provides the AddDbContext extension for ASP.NET Core apps that registers the context as a *scoped* service. In server-side Blazor apps, scoped service registrations can be problematic because the instance is shared across components within the user's circuit. DbContext isn't thread safe and isn't designed for concurrent use. The existing lifetimes are inappropriate for these reasons:

- **Singleton** shares state across all users of the app and leads to inappropriate concurrent use.
- **Scoped** (the default) poses a similar issue between components for the same user.
- **Transient** results in a new instance per request; but as components can be long-lived, this results in a longer-lived context than may be intended.

The following recommendations are designed to provide a consistent approach to using EF Core in server-side Blazor apps.

- Consider using one context per operation. The context is designed for fast, low overhead instantiation:

  ```
  C#
  ```

  ```
  using var context = new ProductsDatabaseContext();

  return await context.Products.ToListAsync();
  ```

- Use a flag to prevent multiple concurrent operations:

  ```
  C#
  ```

```
if (Loading)
{
    return;
}

try
{
    Loading = true;

    ...
}
finally
{
    Loading = false;
}
```

Place database operations after the `Loading = true;` line in the `try` block.

Thread safety isn't a concern, so loading logic doesn't require locking database records. The loading logic is used to disable UI controls so that users don't inadvertently select buttons or update fields while data is fetched.

- If there's any chance that multiple threads may access the same code block, inject a factory and make a new instance per operation. Otherwise, injecting and using the context is usually sufficient.
- For longer-lived operations that take advantage of EF Core's change tracking or concurrency control, scope the context to the lifetime of the component.

New `DbContext` instances

The fastest way to create a new DbContext instance is by using `new` to create a new instance. However, there are scenarios that require resolving additional dependencies:

- Using DbContextOptions to configure the context.
- Using a connection string per DbContext, such as when you use ASP.NET Core's Identity model. For more information, see Multi-tenancy (EF Core documentation).

> ⚠ **Warning**
>
> Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure

authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the Secret Manager tool is recommended for securing sensitive data. For more information, see Securely maintain sensitive data and credentials.

The recommended approach to create a new DbContext with dependencies is to use a factory. EF Core 5.0 or later provides a built-in factory for creating new contexts.

The following example configures SQLite and enables data logging in an app that manages contacts. The code uses an extension method (AddDbContextFactory) to configure the database factory for DI and provide default options:

C#

```
builder.Services.AddDbContextFactory<ContactContext>(opt =>
    opt.UseSqlite($"Data Source={nameof(ContactContext.ContactsDb)}.db"));
```

The factory is injected into components to create new DbContext instances.

Scope a database context to a component method

The factory is injected into the component:

razor

```
@inject IDbContextFactory<ContactContext> DbFactory
```

Create a DbContext for a method using the factory (`DbFactory`):

C#

```
private async Task DeleteContactAsync()
{
    using var context = DbFactory.CreateDbContext();

    if (context.Contacts is not null)
    {
        var contact = await context.Contacts.FirstAsync(...);

        if (contact is not null)
        {
```

```
            context.Contacts?.Remove(contact);
            await context.SaveChangesAsync();
        }
    }
}
```

Scope a database context to the lifetime of the component

You may wish to create a DbContext that exists for the lifetime of a component. This allows you to use it as a unit of work and take advantage of built-in features, such as change tracking and concurrency resolution.

Implement IDisposable and inject the factory into the component:

razor

```
@implements IDisposable
@inject IDbContextFactory<ContactContext> DbFactory
```

Establish a property for the DbContext:

C#

```
private ContactContext? Context { get; set; }
```

OnInitializedAsync is overridden to create the DbContext:

C#

```
protected override async Task OnInitializedAsync()
{
    Context = DbFactory.CreateDbContext();
}
```

The DbContext is disposed when the component is disposed:

C#

```
public void Dispose() => Context?.Dispose();
```

Enable sensitive data logging

EnableSensitiveDataLogging includes application data in exception messages and framework logging. The logged data can include the values assigned to properties of entity instances and parameter values for commands sent to the database. Logging data with EnableSensitiveDataLogging is a **security risk**, as it may expose passwords and other Personally Identifiable Information (PII) when it logs SQL statements executed against the database.

We recommend only enabling EnableSensitiveDataLogging for development and testing:

C#

```
#if DEBUG
    services.AddDbContextFactory<ContactContext>(opt =>
        opt.UseSqlite($"Data Source={nameof(ContactContext.ContactsDb)}.db")
        .EnableSensitiveDataLogging());
#else
    services.AddDbContextFactory<ContactContext>(opt =>
        opt.UseSqlite($"Data Source=
{nameof(ContactContext.ContactsDb)}.db"));
#endif
```

Additional resources

- EF Core documentation
- Blazor samples GitHub repository (dotnet/blazor-samples)
- Blazor *Security and Identity* node articles

ASP.NET Core Blazor advanced scenarios (render tree construction)

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article describes the advanced scenario for building Blazor render trees manually with RenderTreeBuilder.

> **Warning**
>
> Use of RenderTreeBuilder to create components is an *advanced scenario*. A malformed component (for example, an unclosed markup tag) can result in undefined behavior. Undefined behavior includes broken content rendering, loss of app features, and ***compromised security***.

Manually build a render tree (`RenderTreeBuilder`)

RenderTreeBuilder provides methods for manipulating components and elements, including building components manually in C# code.

Consider the following `PetDetails` component, which can be manually rendered in another component.

`PetDetails.razor`:

```razor
<h2>Pet Details</h2>

<p>@PetDetailsQuote</p>

@code
{
    [Parameter]
```

```
    public string? PetDetailsQuote { get; set; }
}
```

In the following `BuiltContent` component, the loop in the `CreateComponent` method generates three `PetDetails` components.

In RenderTreeBuilder methods with a sequence number, sequence numbers are source code line numbers. The Blazor difference algorithm relies on the sequence numbers corresponding to distinct lines of code, not distinct call invocations. When creating a component with RenderTreeBuilder methods, hardcode the arguments for sequence numbers. **Using a calculation or counter to generate the sequence number can lead to poor performance.** For more information, see the Sequence numbers relate to code line numbers and not execution order section.

`BuiltContent.razor`:

razor

```
@page "/built-content"

<PageTitle>Built Content</PageTitle>

<h1>Built Content Example</h1>

<div>
    @CustomRender
</div>

<button @onclick="RenderComponent">
    Create three Pet Details components
</button>

@code {
    private RenderFragment? CustomRender { get; set; }

    private RenderFragment CreateComponent() => builder =>
    {
        for (var i = 0; i < 3; i++)
        {
            builder.OpenComponent(0, typeof(PetDetails));
            builder.AddAttribute(1, "PetDetailsQuote", "Someone's best
friend!");
            builder.CloseComponent();
        }
    };

    private void RenderComponent() => CustomRender = CreateComponent();
}
```

> ⚠ **Warning**
>
> The types in Microsoft.AspNetCore.Components.RenderTree allow processing of the *results* of rendering operations. These are internal details of the Blazor framework implementation. These types should be considered *unstable* and subject to change in future releases.

Sequence numbers relate to code line numbers and not execution order

Razor component files (`.razor`) are always compiled. Executing compiled code has a potential advantage over interpreting code because the compile step that yields the compiled code can be used to inject information that improves app performance at runtime.

A key example of these improvements involves *sequence numbers*. Sequence numbers indicate to the runtime which outputs came from which distinct and ordered lines of code. The runtime uses this information to generate efficient tree diffs in linear time, which is far faster than is normally possible for a general tree diff algorithm.

Consider the following Razor component file (`.razor`):

razor

```
@if (someFlag)
{
    <text>First</text>
}

Second
```

The preceding Razor markup and text content compiles into C# code similar to the following:

C#

```
if (someFlag)
{
    builder.AddContent(0, "First");
}

builder.AddContent(1, "Second");
```

When the code executes for the first time and `someFlag` is `true`, the builder receives the sequence in the following table.

Expand table

Sequence	Type	Data
0	Text node	First
1	Text node	Second

Imagine that `someFlag` becomes `false` and the markup is rendered again. This time, the builder receives the sequence in the following table.

Expand table

Sequence	Type	Data
1	Text node	Second

When the runtime performs a diff, it sees that the item at sequence `0` was removed, so it generates the following trivial *edit script* with a single step:

- Remove the first text node.

The problem with generating sequence numbers programmatically

Imagine instead that you wrote the following render tree builder logic:

C#

```
var seq = 0;

if (someFlag)
{
    builder.AddContent(seq++, "First");
}

builder.AddContent(seq++, "Second");
```

The first output is reflected in the following table.

Expand table

Sequence	Type	Data
0	Text node	First
1	Text node	Second

This outcome is identical to the prior case, so no negative issues exist. `someFlag` is `false` on the second rendering, and the output is seen in the following table.

Expand table

Sequence	Type	Data
0	Text node	Second

This time, the diff algorithm sees that *two* changes have occurred. The algorithm generates the following edit script:

- Change the value of the first text node to `Second`.
- Remove the second text node.

Generating the sequence numbers has lost all the useful information about where the `if/else` branches and loops were present in the original code. This results in a diff **twice as long** as before.

This is a trivial example. In more realistic cases with complex and deeply nested structures, and especially with loops, the performance cost is usually higher. Instead of immediately identifying which loop blocks or branches have been inserted or removed, the diff algorithm must recurse deeply into the render trees. This usually results in building longer edit scripts because the diff algorithm is misinformed about how the old and new structures relate to each other.

Guidance and conclusions

- App performance suffers if sequence numbers are generated dynamically.
- The necessary information doesn't exist to permit the framework to generate sequence numbers automatically at runtime unless the information is captured at compile time.
- Don't write long blocks of manually-implemented RenderTreeBuilder logic. Prefer `.razor` files and allow the compiler to deal with the sequence numbers. If you're unable to avoid manual RenderTreeBuilder logic, split long blocks of code into smaller pieces wrapped in OpenRegion/CloseRegion calls. Each region has its own

separate space of sequence numbers, so you can restart from zero (or any other arbitrary number) inside each region.

- If sequence numbers are hardcoded, the diff algorithm only requires that sequence numbers increase in value. The initial value and gaps are irrelevant. One legitimate option is to use the code line number as the sequence number, or start from zero and increase by ones or hundreds (or any preferred interval).
- For loops, the sequence numbers should increase in your source code, not in terms of runtime behavior. The fact that, at runtime, the numbers repeat is how the diffing system realises you're in a loop.
- Blazor uses sequence numbers, while other tree-diffing UI frameworks don't use them. Diffing is far faster when sequence numbers are used, and Blazor has the advantage of a compile step that deals with sequence numbers automatically for developers authoring `.razor` files.

Tutorial: Create an ASP.NET Core app with Angular in Visual Studio

Article • 11/08/2024

In this article, you learn how to build an ASP.NET Core project to act as an API backend and an Angular project to act as the UI.

Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular and React. The templates provide a built-in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

> **Note**
>
> This article describes the project creation process using the updated template in Visual Studio 2022 version 17.8.

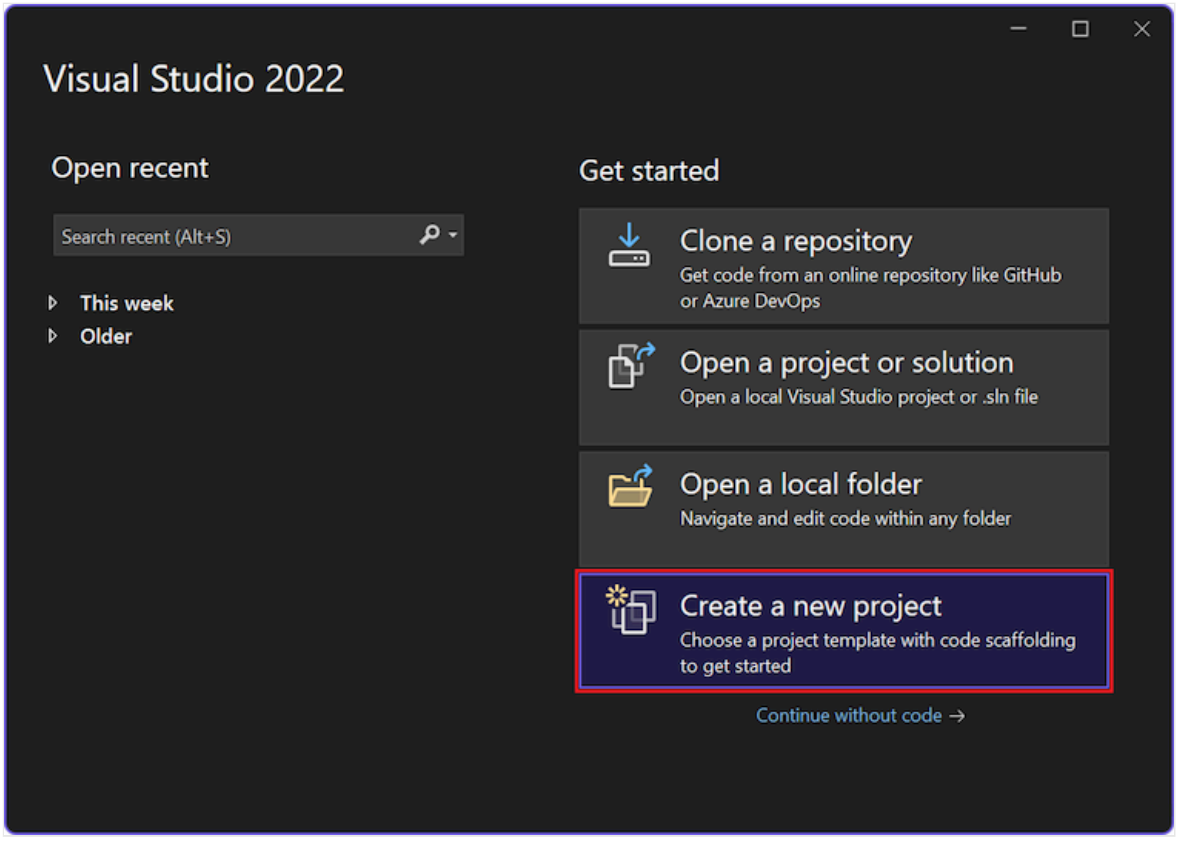
Prerequisites

Make sure to install the following:

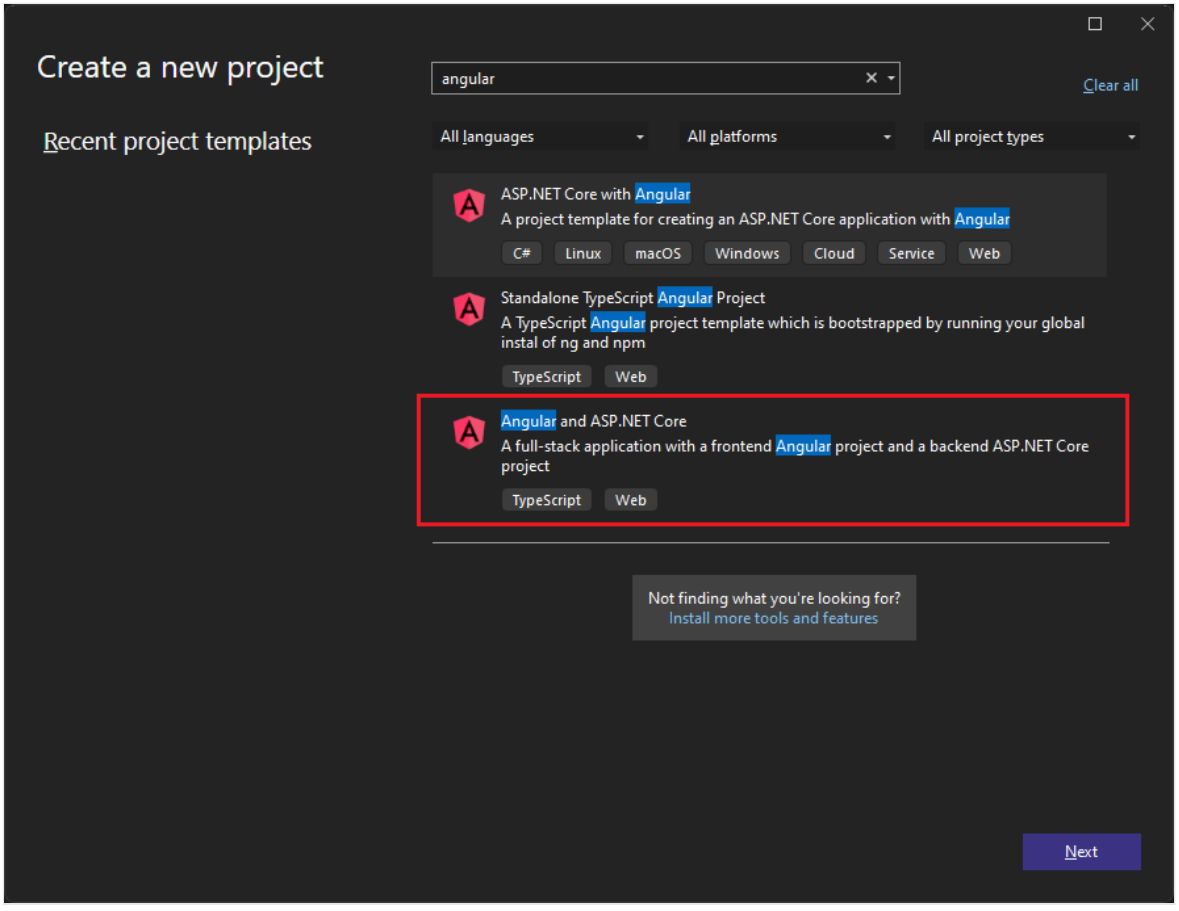
- Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- Angular CLI (https://angular.dev/tools/cli), which can be the version of your choice. The front-end project is created using the framework CLI tools you have installed on your local machine, so this determines the Angular version used in the template.

Create the frontend app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.



2. Search for Angular in the search bar at the top and then select **Angular and ASP.NET Core**.

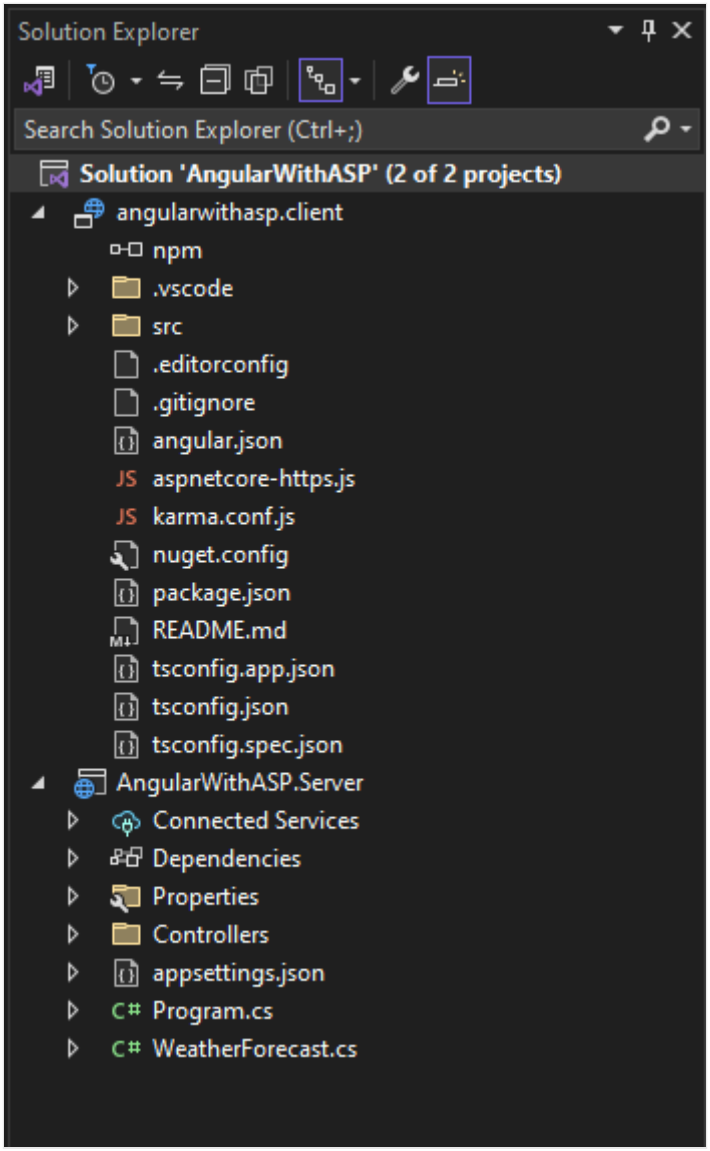


3. Name the project **AngularWithASP** and then select **Next**.

   In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following:



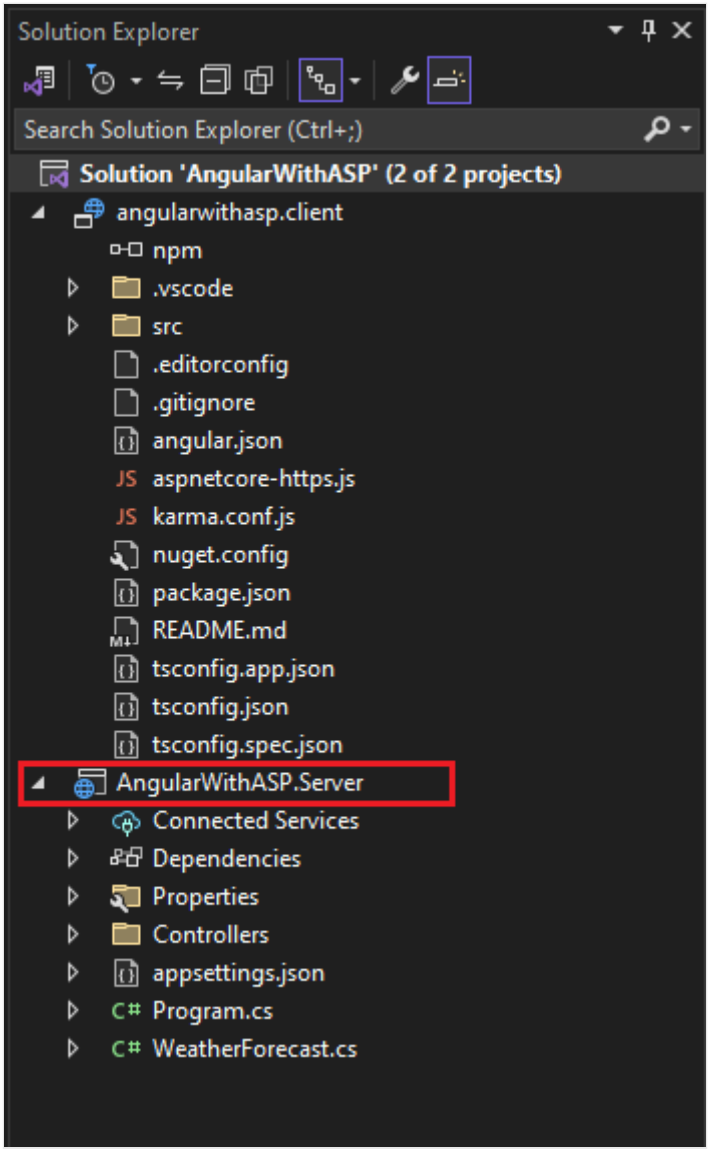
   Compared to the standalone Angular template, you see some new and modified files for integration with ASP.NET Core:

   - aspnetcore-https.js
   - proxy.conf.js
   - package.json(modified)
   - angular.json(modified)
   - app.components.ts
   - app.module.ts

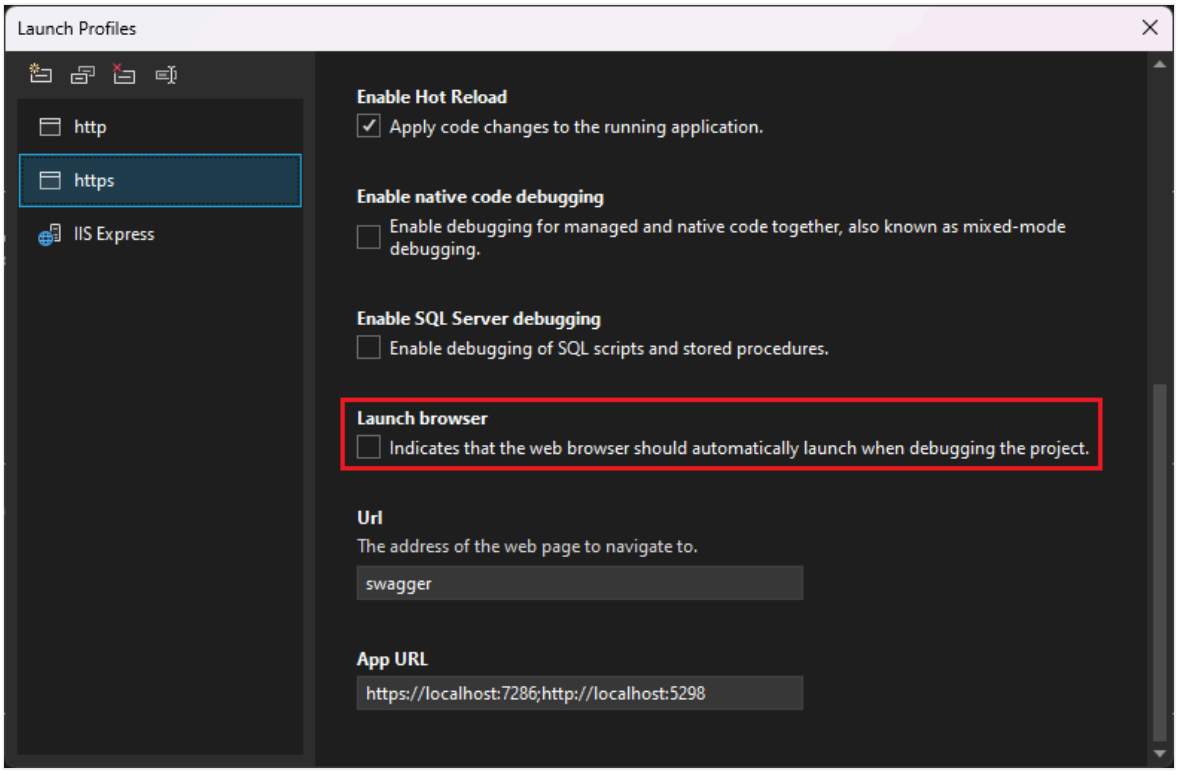
   For more information on some of these project files, see Next steps.

Set the project properties

1. In Solution Explorer, right-click the **AngularWithASP.Server** project and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.



This value prevents opening the web page with the source weather data.

> ⓘ Note
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

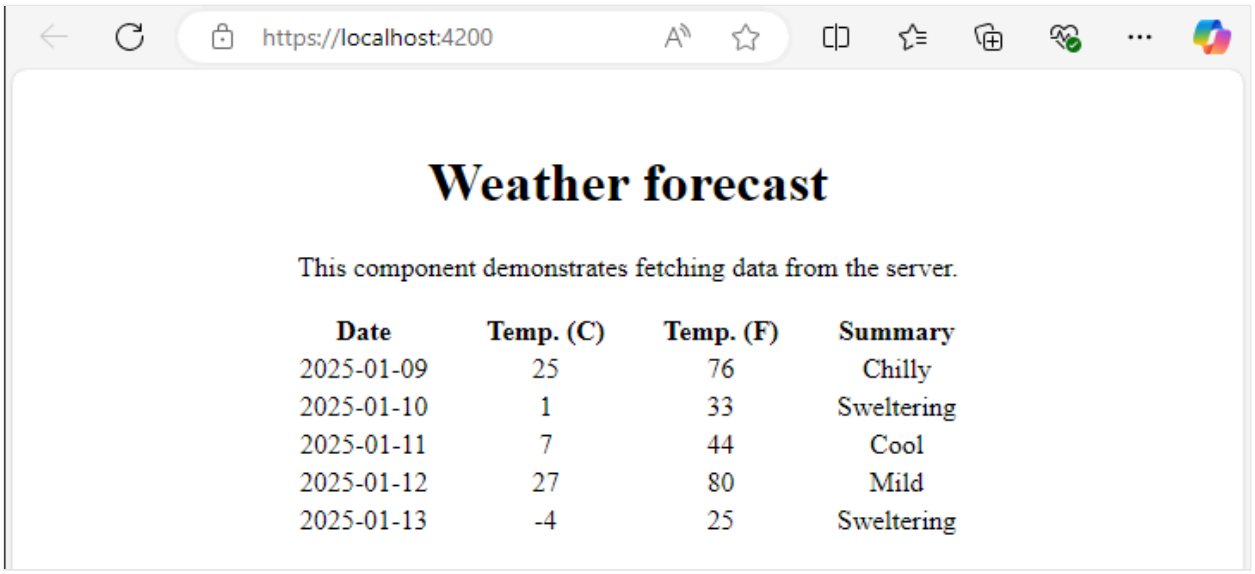
Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Angular CLI running the ng start command

> ⓘ Note
>
> Check console output for messages. For example there might be a message to update Node.js.

The Angular app appears and is populated via the API (the localhost port may vary from the screenshot).



If you don't see the app, see Troubleshooting.

Publish the project

Starting in Visual Studio 2022 version 17.3, you can publish the integrated solution using the Visual Studio Publish tool.

> ⓘ **Note**
>
> To use publish, create your JavaScript project using Visual Studio 2022 version 17.3 or later.

1. In Solution Explorer, right-click the **AngularWithASP.Server** project and select **Add** > **Project Reference**.

   Make sure the **angularwithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following:

XML

```
 <ProjectReference
Include="..\angularwithasp.client\angularwithasp.client.esproj">
   <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
 </ProjectReference>
```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

C#

```
app.UseDefaultFiles();
app.UseStaticFiles();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
   app.UseSwagger();
   app.UseSwaggerUI();
}
```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

   If you publish to a folder, see ASP.NET Core directory structure for more information on the files added to the *publish* folder.

Troubleshooting

Proxy error

You may see the following error:

Windows Command Prompt

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:5001 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend.

- Once you see the backend command prompt up and running, just refresh the Angular app in the browser.
- Also, verify that the backend is configured to start before the front end. To verify, select the solution in Solution Explorer, choose **Properties** from the **Project menu**. Next, select **Configure Startup Projects** and make sure that the backend ASP.NET Core project is first in the list. If it's not first, select the project and use the Up arrow button to make it the first project in the launch list.

Verify port

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

1. Go to the `launchSettings.json` file in your ASP.NET Core project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:7049`.

2. Then, go to the `proxy.conf.js` file for your Angular project (look in the *src* folder). Update the target property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

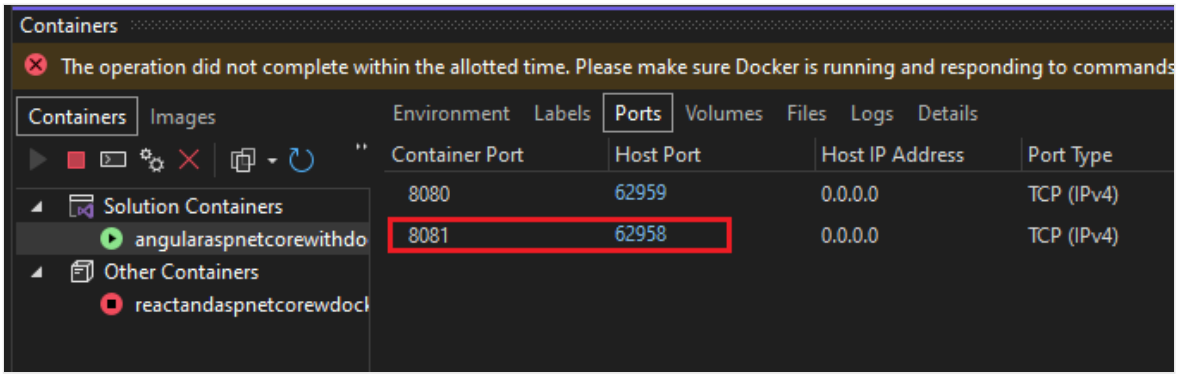
   JavaScript

   ```
   target: 'https://localhost:7049',
   ```

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



2. Open the `proxy.conf.js` file for the Angular project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

   JavaScript

   ```
   const target = env.ASPNETCORE_HTTPS_PORT ?
   `https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
      env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
   'https://localhost:7209';
   ```

   Change `https://localhost:7209` to the matching HTTPS port (in this example, `https://localhost:62958`).

3. Restart the app.

Next steps

For more information about SPA applications in ASP.NET Core, see the Angular section under Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js* and *proxy.conf.js*, although details of the implementation are different due to project template differences. For example, instead of a ClientApp folder, the Angular files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Tutorial: Create an ASP.NET Core app with React in Visual Studio

Article • 01/09/2025

In this article, you learn how to build an ASP.NET Core project to act as an API backend and a React project to act as the UI.

Currently, Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular and React. The templates provide a built-in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

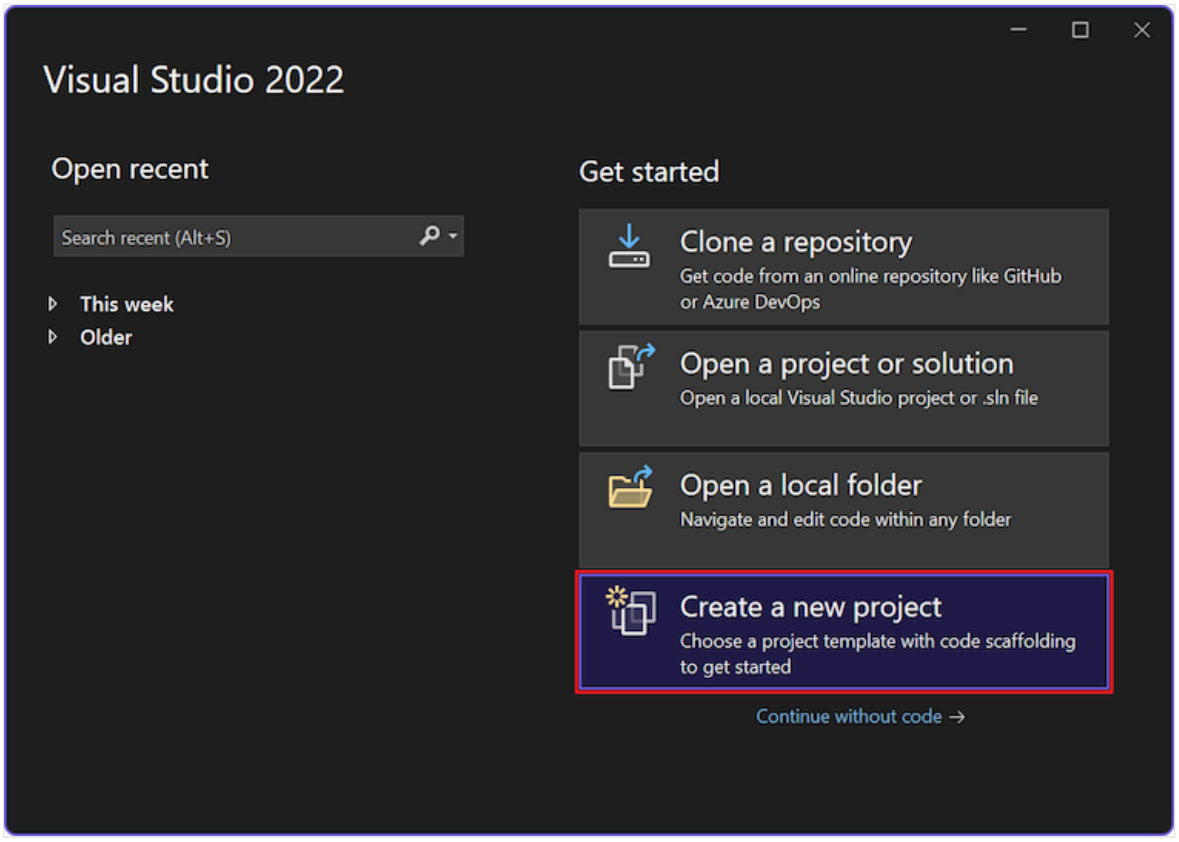
> **Note**
>
> This article describes the project creation process using the updated template in Visual Studio 2022 version 17.11, which uses the Vite CLI. Vite determines the version of React using project dependencies, such as those configured in *package.json*.

Prerequisites

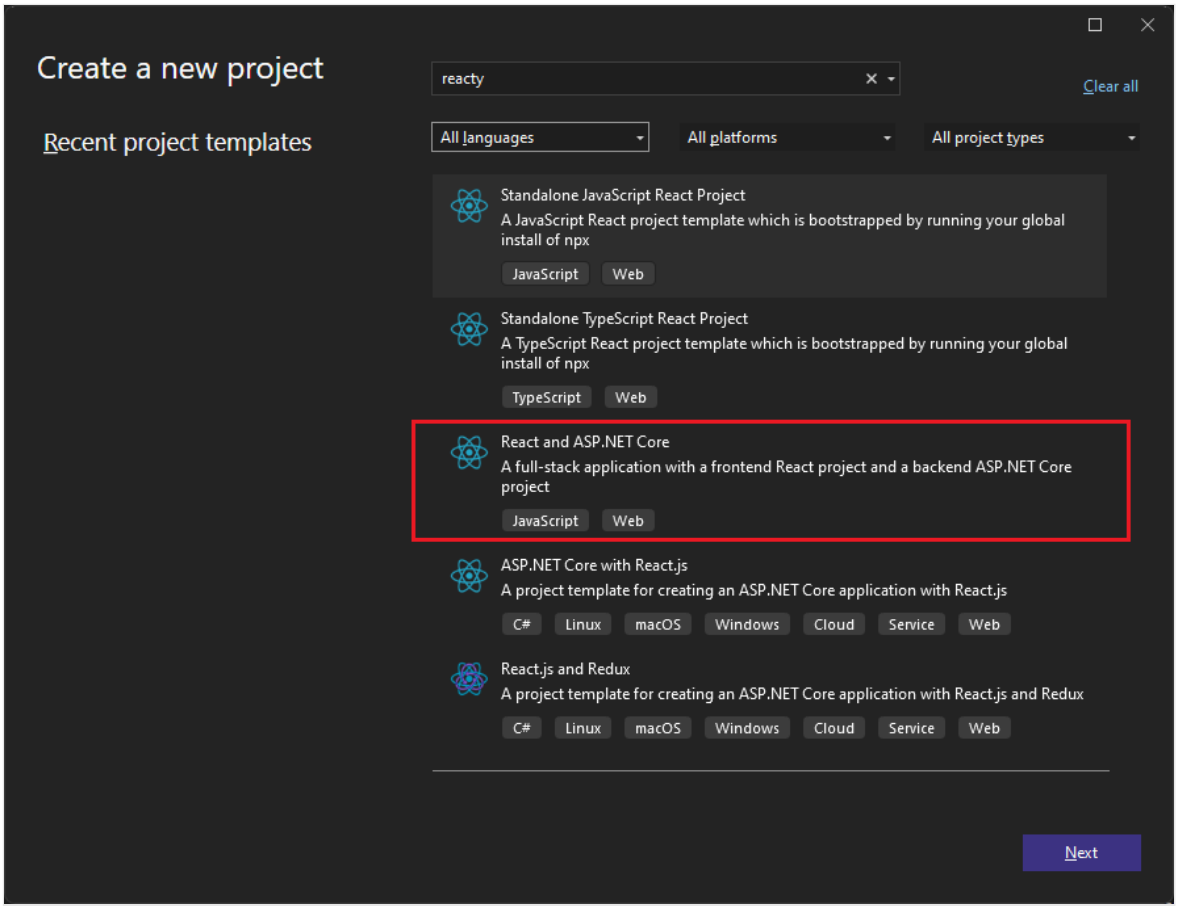
- Visual Studio 2022 version 17.11 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js

Create the frontend app

1. In the Start window, select **Create a new project**.



2. Search for React in the search bar at the top and then select **React and ASP.NET Core**. This template is a JavaScript template.

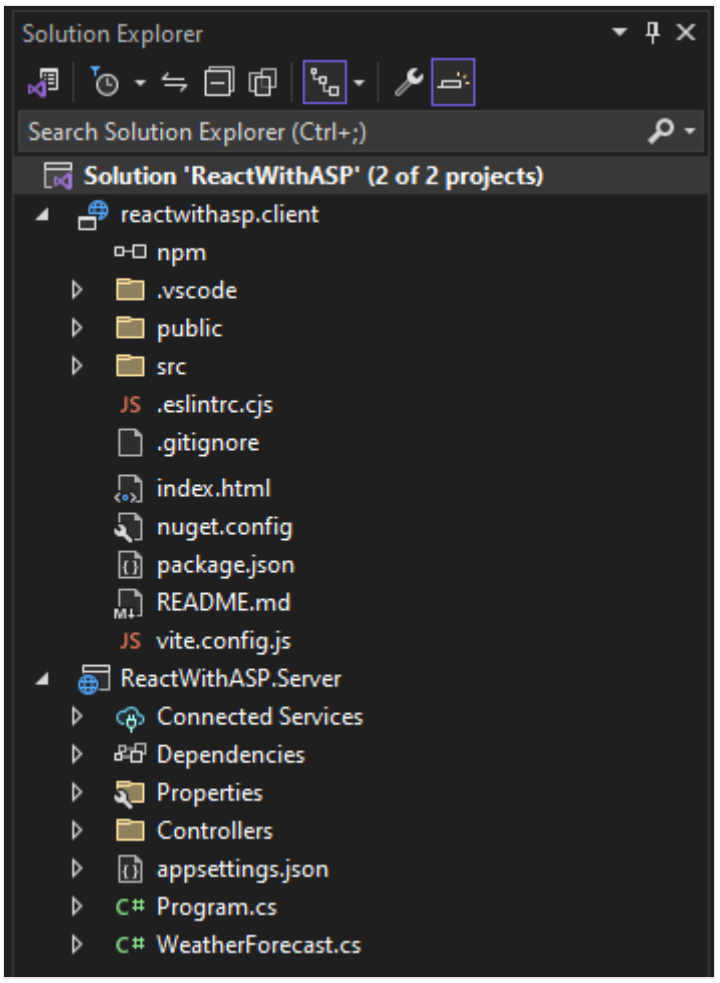


3. Name the project **ReactWithASP** and then select **Next**.

In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following project information:



   Compared to the standalone React template, you see some new and modified files for integration with ASP.NET Core:

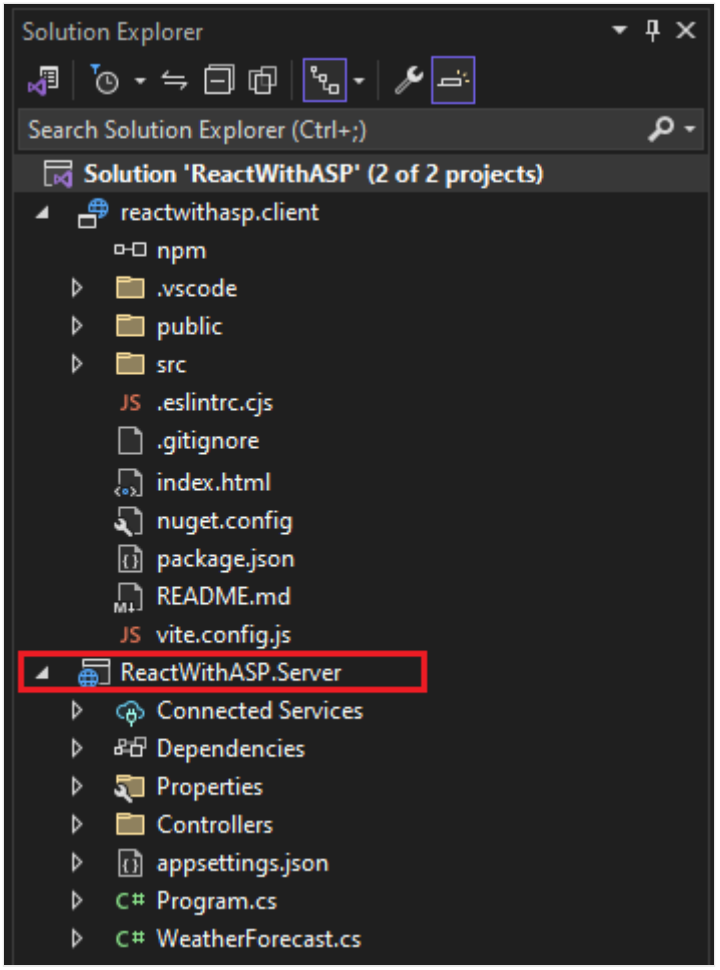
   - vite.config.js
   - App.js (modified)
   - App.test.js (modified)

5. Select an installed browser from the Debug toolbar, such as Chrome or Microsoft Edge.

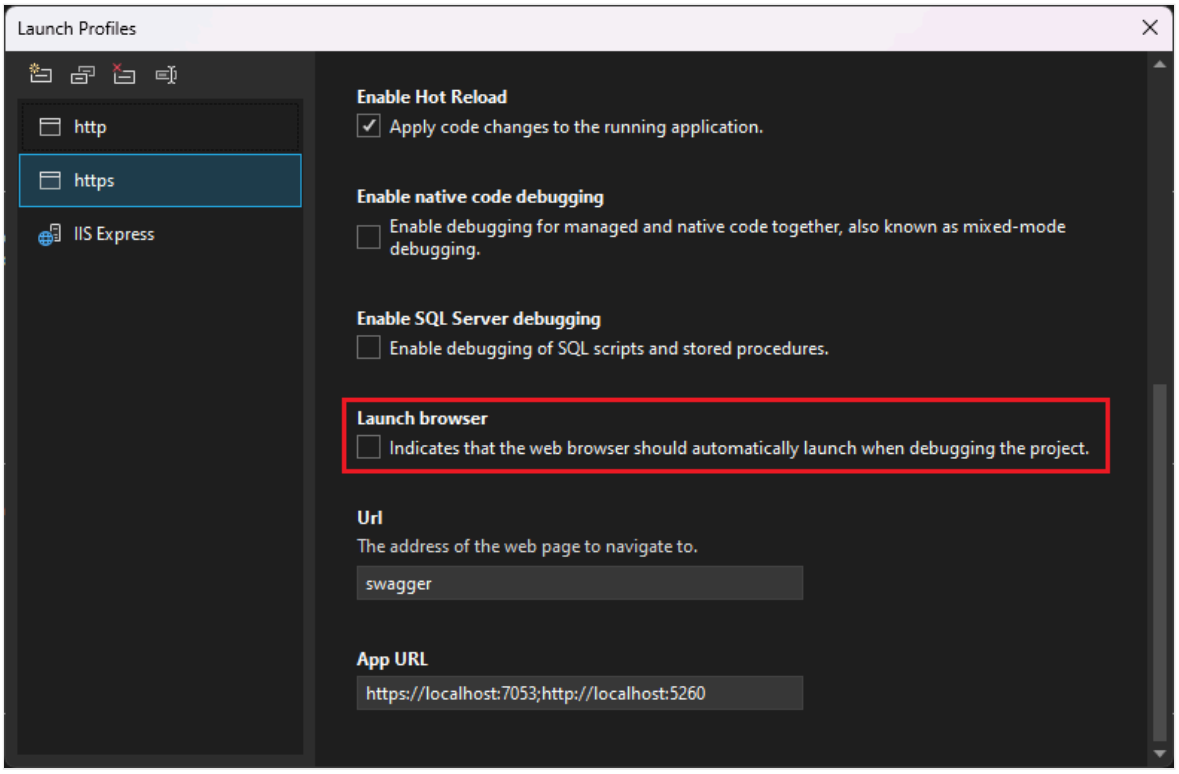
   If the browser you want is not yet installed, install the browser first, and then select it.

Set the project properties

1. In Solution Explorer, right-click the **ReactWithASP.Server** project and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.



This value prevents opening the web page with the source weather data.

> ⓘ Note
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

Start the project

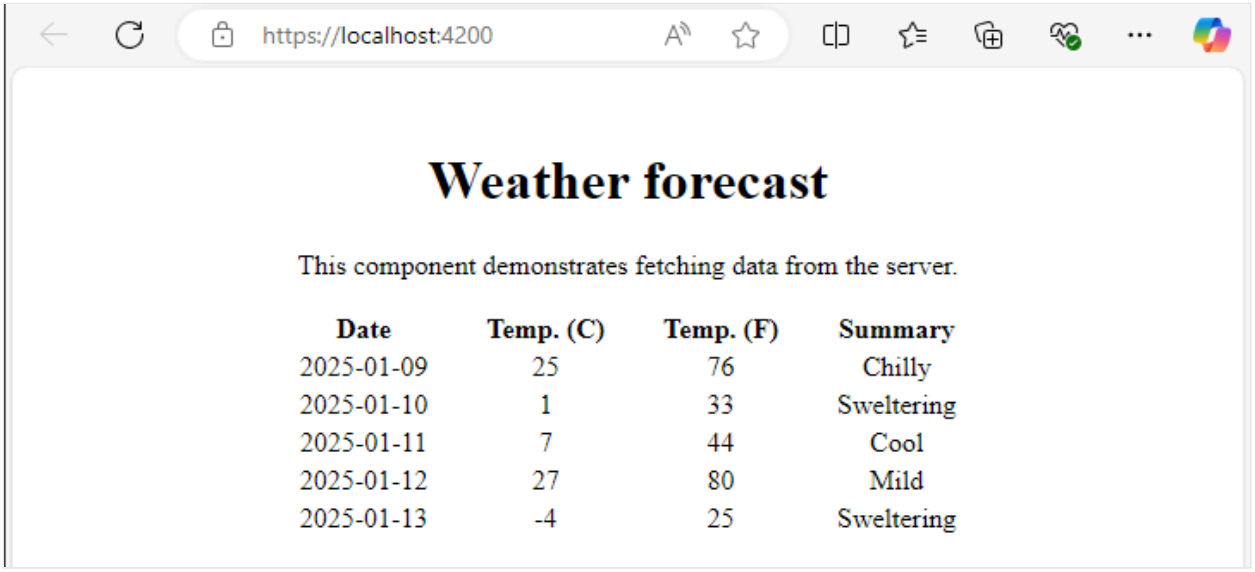
Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Vite CLI showing a message such as `VITE v4.4.9 ready in 780 ms`

> ⓘ Note

> Check console output for messages. For example there might be a message to update Node.js.

The React app appears and is populated via the API (the localhost port may vary from the screenshot).



If you don't see the app, see Troubleshooting.

Publish the project

1. In Solution Explorer, right-click the **ReactWithASP.Server** project and select **Add** > **Project Reference**.

   Make sure the **reactwithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML

   ```
    <ProjectReference
   Include="..\reactwithasp.client\reactwithasp.client.esproj">
   ```

```
    <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
  </ProjectReference>
```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#

   ```
   app.UseDefaultFiles();
   app.UseStaticFiles();

   // Configure the HTTP request pipeline.
   if (app.Environment.IsDevelopment())
   {
      app.UseSwagger();
      app.UseSwaggerUI();
   }
   ```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

   If you publish to a folder, see ASP.NET Core directory structure for more information on the files added to the *publish* folder.

Troubleshooting

Proxy error

You may see the following error:

Windows Command Prompt

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:7183 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend.

- Once you see the backend command prompt up and running, just refresh the React app in the browser.
- Also, verify that the backend is configured to start before the front end. To verify, select the solution in Solution Explorer, choose **Properties** from the **Project menu**. Next, select **Configure Startup Projects** and make sure that the backend ASP.NET Core project is first in the list. If it's not first, select the project and use the Up arrow button to make it the first project in the launch list.

Verify ports

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

1. Make sure that the port numbers match. Go to the `launchSettings.json` file in the ASP.NET Core **ReactWithASP.Server** project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It looks similar to `https://localhost:7183`.

2. Open the `vite.config.js` file for the React project. Update the `target` property to match the `applicationUrl` property in *launchSettings.json*. The updated value looks similar to the following:

   JavaScript

   ```
   target: 'https://localhost:7183/',
   ```

Privacy error

You may see the following certificate error:

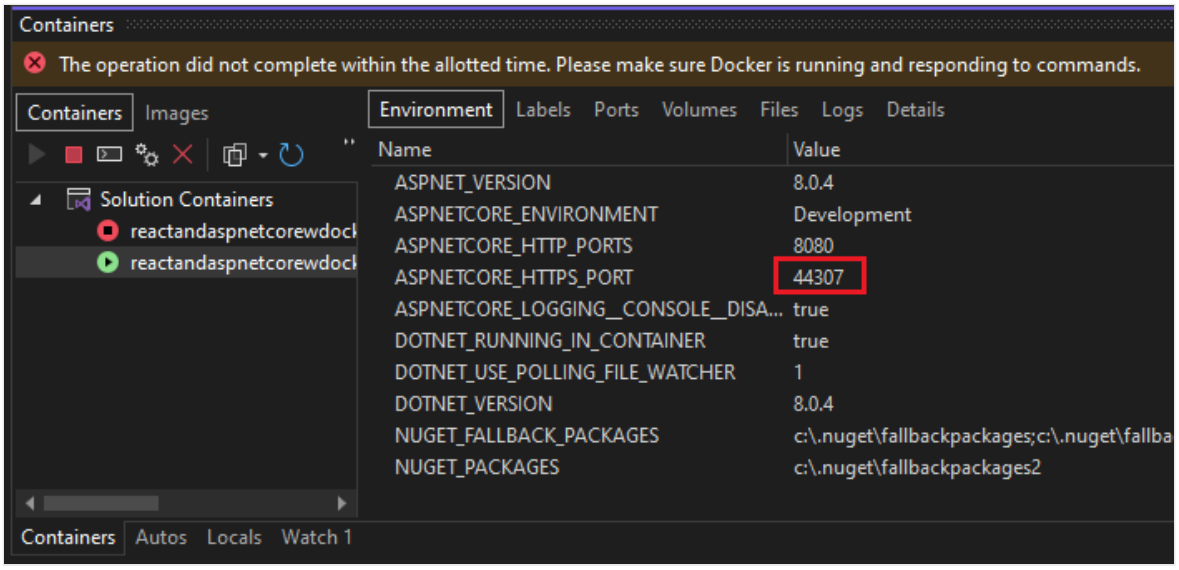
```
Your connection isn't private
```

Try deleting the React certificates from *%appdata%\local\asp.net\https* or *%appdata%\roaming\asp.net\https*, and then retry.

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



2. Open the `vite.config.js` file for the React project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

```
JavaScript
```

```javascript
const target = env.ASPNETCORE_HTTPS_PORT ?
`https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
   env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
'https://localhost:7143';
```

   change `https://localhost:7143` to the matching HTTPS port (in this example, `https://localhost:44307`).

3. Restart the app.

Next steps

For more information about SPA applications in ASP.NET Core, see the React section under Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js*, although details of the implementation are different based on the template differences. For example, instead of a ClientApp folder, the React files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Tutorial: Create an ASP.NET Core app with Vue in Visual Studio

Article • 11/06/2024

In this article, you learn how to build an ASP.NET Core project to act as an API backend and a Vue project to act as the UI.

Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular, React, and Vue. The templates provide a built-in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

> **Note**
>
> This article describes the project creation process using the updated template in Visual Studio 2022 version 17.11, which uses the Vite CLI. Vite determines the version of Vue using project dependencies, such as those configured in *package.json*.

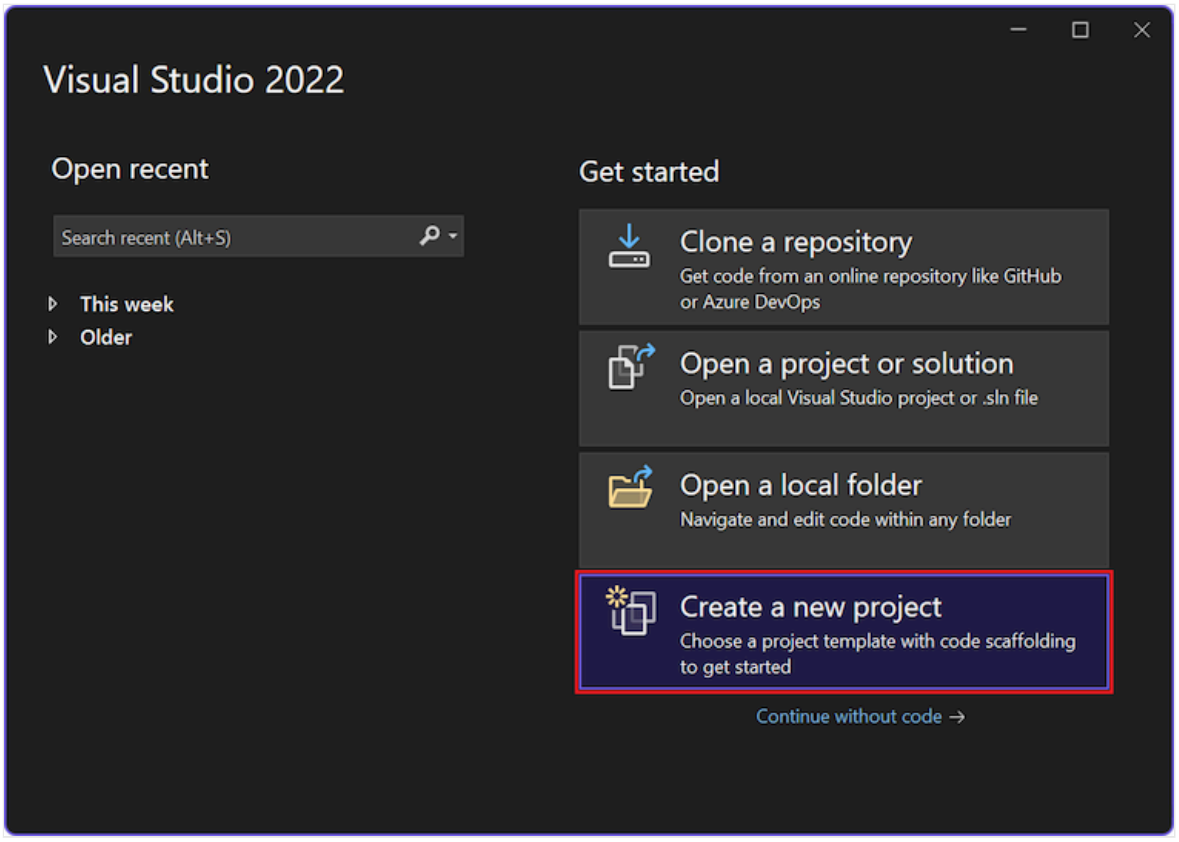
Prerequisites

Make sure to install the following:

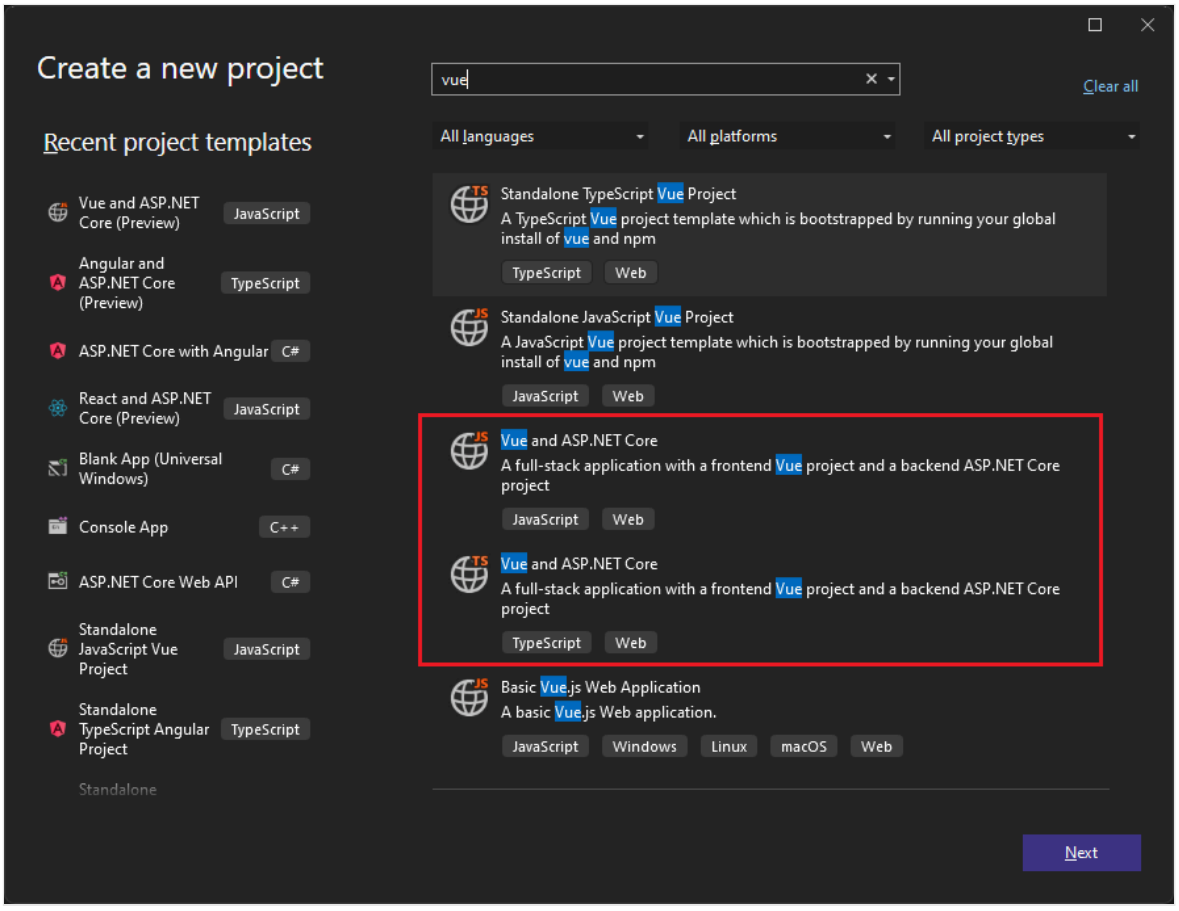
- Visual Studio 2022 version 17.11 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js.

Create the frontend app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.



2. Search for Vue in the search bar at the top and then select **Vue and ASP.NET Core** with either JavaScript or TypeScript as the selected language.

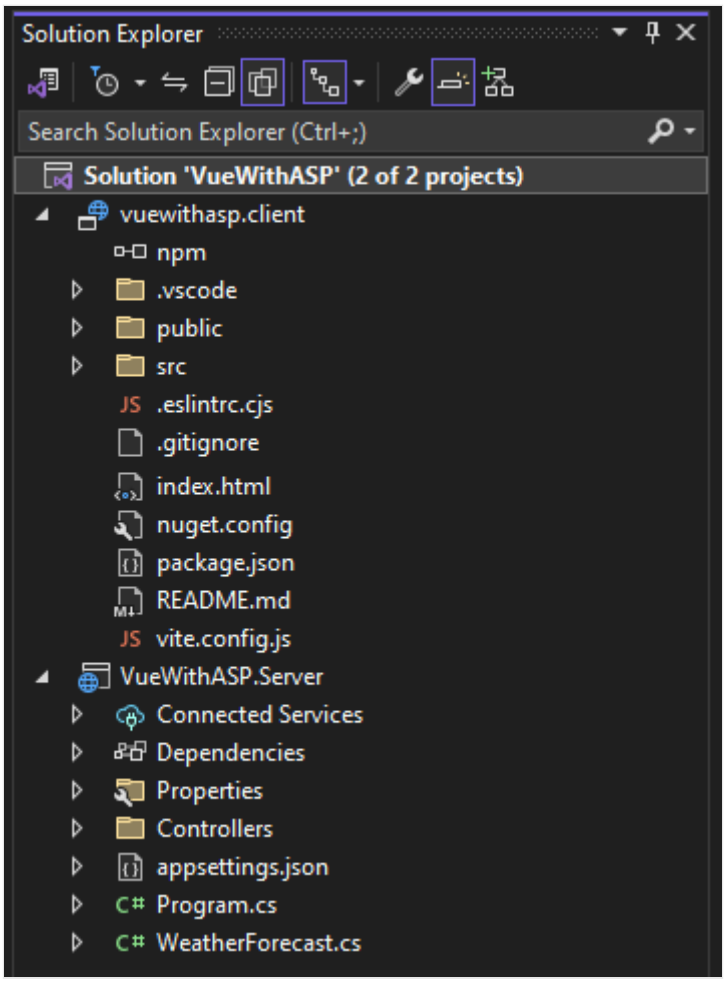


3. Name the project **VueWithASP** and then select **Next**.

In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following project information:

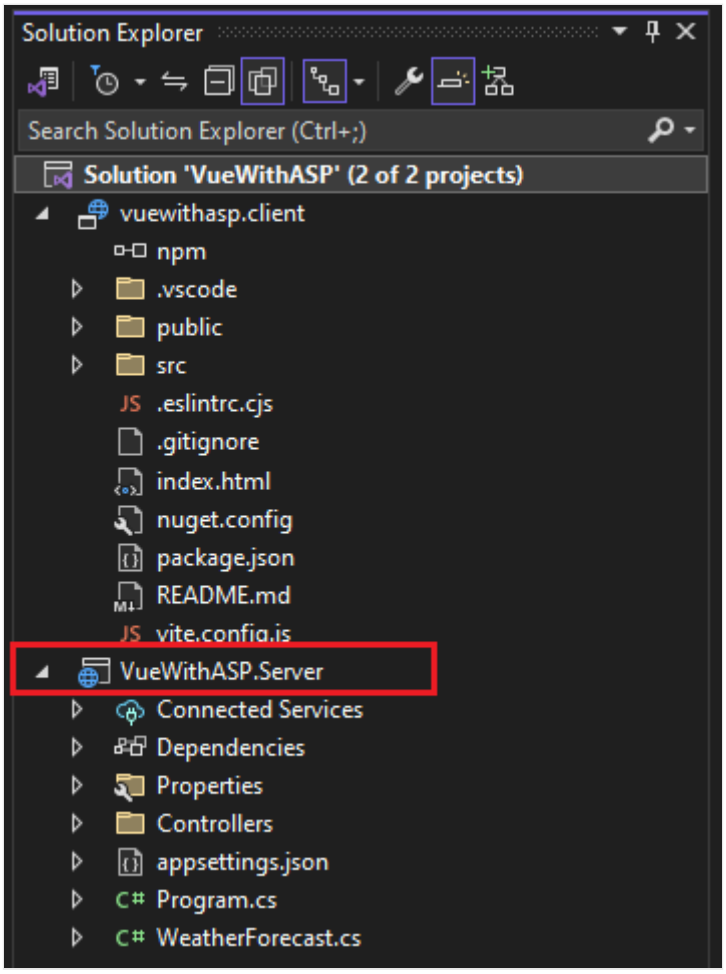


   Compared to the standalone Vue template, you see some new and modified files for integration with ASP.NET Core:

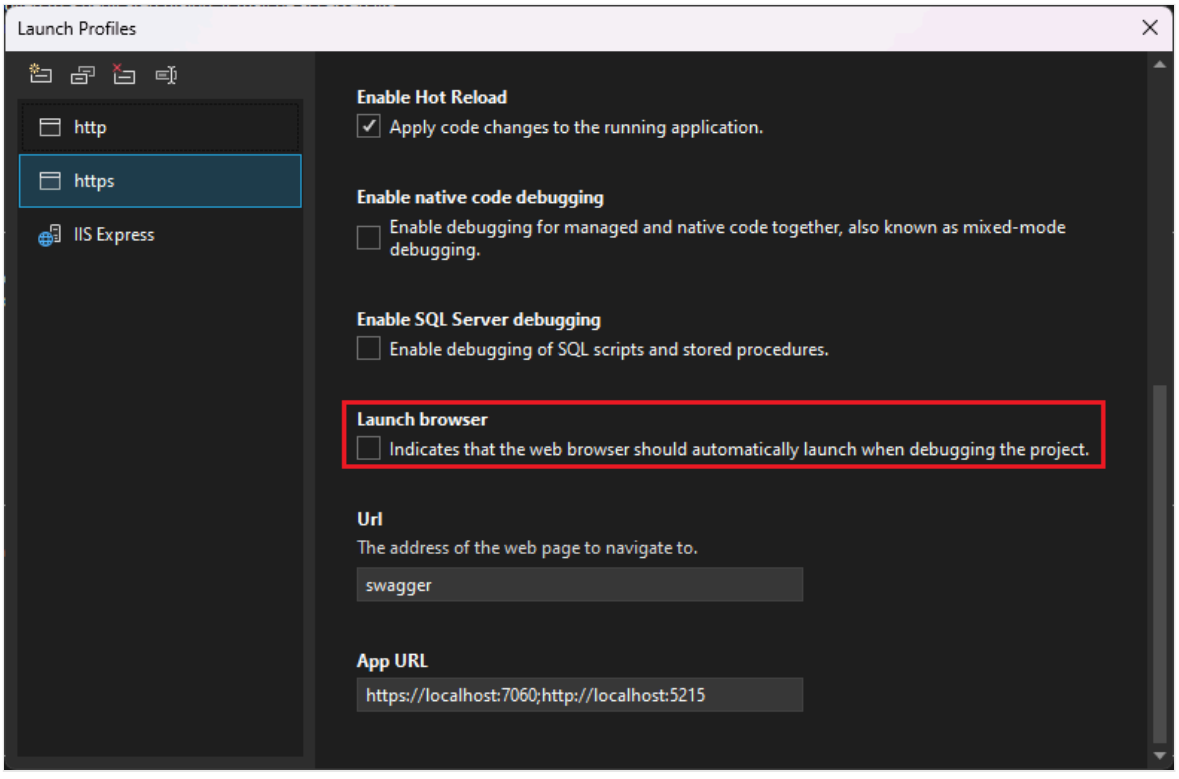
   - vite.config.json (modified)
   - HelloWorld.vue (modified)
   - package.json (modified)

Set the project properties

1. In Solution Explorer, right-click the **VueWithASP.Server** and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.



This value prevents opening the web page with the source weather data.

> ⓘ Note
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

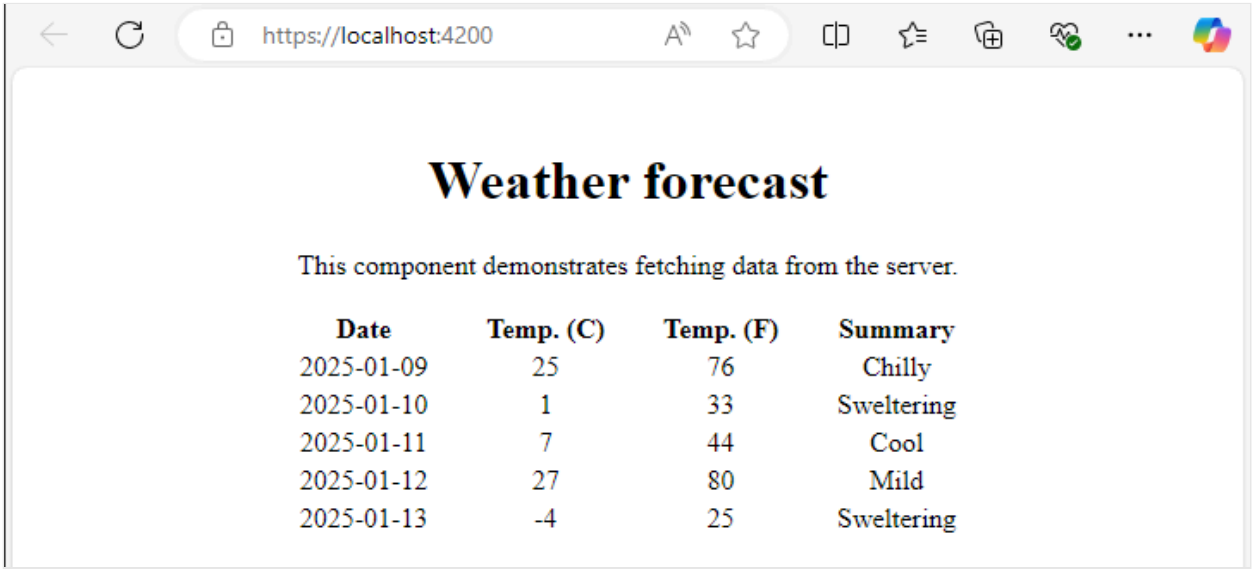
Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Vite CLI showing a message such as `VITE v4.4.9 ready in 780 ms`

> ⓘ Note
>
> Check console output for messages. For example there might be a message to update Node.js.

The Vue app appears and is populated via the API (the localhost port may vary from the screenshot).



If you don't see the app, see Troubleshooting.

Publish the project

Starting in Visual Studio 2022 version 17.3, you can publish the integrated solution using the Visual Studio Publish tool.

> **Note**
>
> To use publish, create your JavaScript project using Visual Studio 2022 version 17.3 or later.

1. In Solution Explorer, right-click the **VueWithASP.Server** project and select **Add** > **Project Reference**.

   Make sure the **vuewithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML

   ```
    <ProjectReference
   Include="..\vuewithasp.client\vuewithasp.client.esproj">
      <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
    </ProjectReference>
   ```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#

   ```
   app.UseDefaultFiles();
   app.UseStaticFiles();

   // Configure the HTTP request pipeline.
   if (app.Environment.IsDevelopment())
   {
   ```

```
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

   If you publish to a folder, see ASP.NET Core directory structure for more information on the files added to the *publish* folder.

Troubleshooting

Proxy error

You may see the following error:

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:5173 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend.

- Once you see the backend command prompt up and running, just refresh the Vue app in the browser.
- Also, verify that the backend is configured to start before the front end. To verify, select the solution in Solution Explorer, choose **Properties** from the **Project menu**. Next, select **Configure Startup Projects** and make sure that the backend ASP.NET Core project is first in the list. If it's not first, select the project and use the Up arrow button to make it the first project in the launch list.

Otherwise, if the port is in use, try incrementing the port number by **1** in `launchSettings.json` and *vite.config.js*.

Privacy error

You may see the following certificate error:

```
Your connection isn't private
```

Try deleting the Vue certificates from *%appdata%\local\asp.net\https* or *%appdata%\roaming\asp.net\https*, and then retry.

Verify ports

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

1. Make sure that the port numbers match. Go to the `launchSettings.json` file in your ASP.NET Core project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:7142`.

2. Then, go to the `vite.config.js` file for your Vue project. Update the `target` property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

   JavaScript

   ```
   target: 'https://localhost:7142/',
   ```

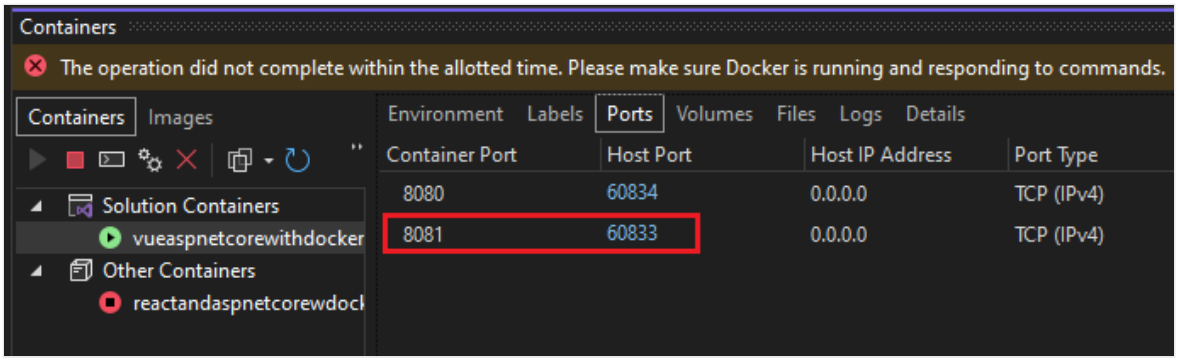
Outdated version of Vue

If you see the console message **Could not find the file 'C:\Users\Me\source\repos\vueprojectname\package.json'** when you create the project, you may need to update your version of the Vite CLI. After you update the Vite CLI, you may also need to delete the `.vuerc` file in *C:\Users\[yourprofilename]*.

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



2. Open the `vite.config.js` file for the Vue project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

   JavaScript

   ```
   const target = env.ASPNETCORE_HTTPS_PORT ?
   `https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
      env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
   'https://localhost:7163';
   ```

   change `https://localhost:7163` to the matching HTTPS port (in this example, `https://localhost:60833`).

3. Restart the app.

If you are using a Docker configuration created in older versions of Visual Studio, the backend may start up using the Docker profile and not listen on the configured port 5173. To resolve:

Edit the Docker profile in the `launchSettings.json` by adding the following properties:

JSON

```
"httpPort": 5175,
"sslPort": 5173
```

Next steps

For more information about SPA applications in ASP.NET Core, see Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js*, although details of the implementation are different due to differences between the project templates and the Vue.js framework vs. other frameworks. For example, instead of a ClientApp folder, the Vue files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

JavaScript and TypeScript in Visual Studio

Article • 06/27/2024

Visual Studio 2022 provides rich support for JavaScript development, both using JavaScript directly, and also using the TypeScript programming language, which was developed to provide a more productive and enjoyable JavaScript development experience, especially when developing projects at scale. You can write JavaScript or TypeScript code in Visual Studio for many application types and services.

JavaScript language service

The JavaScript experience in Visual Studio 2022 is powered by the same engine that provides TypeScript support. This engine gives you better feature support, richness, and integration immediately out-of-the-box.

The option to restore to the legacy JavaScript language service is no longer available. Users have the new JavaScript language service out-of-the-box. The new language service is solely based on the TypeScript language service, which is powered by static analysis. This service enables us to provide you with better tooling, so your JavaScript code can benefit from richer IntelliSense based on type definitions. The new service is lightweight and consumes less memory than the legacy service, providing you with better performance as your code scales. We also improved performance of the language service to handle larger projects.

TypeScript support

By default, Visual Studio 2022 provides language support for JavaScript and TypeScript files to power IntelliSense without any specific project configuration.

For compiling TypeScript, Visual Studio gives you the flexibility to choose which version of TypeScript to use on a per-project basis.

In MSBuild compilation scenarios such as ASP.NET Core, the TypeScript NuGet package is the recommended method of adding TypeScript compilation support to your project. Visual Studio will give you the option to add this package the first time you add a TypeScript file to your project. This package is also available at any time through the NuGet package manager. When the NuGet package is used, the corresponding

language service version will be used for language support in your project. Note: The minimum supported version of this package is 3.6.

Projects configured for npm, such as Node.js projects, can specify their own version of the TypeScript language service by adding the TypeScript npm package. You can specify the version using the npm manager in supported projects. Note: The minimum supported version of this package is 2.1.

The TypeScript SDK has been deprecated in Visual Studio 2022. Existing projects that rely on the SDK should be upgraded to use the NuGet package. For projects that cannot be upgraded immediately, the SDK is still available on the Visual Studio Marketplace and as an optional component in the Visual Studio installer.

> **Tip**
>
> For projects developed in Visual Studio 2022, we encourage you to use the TypeScript NuGet or the TypeScript npm package for greater portability across different platforms and environments. For more information, see **Compile TypeScript code using NuGet** and **Compile TypeScript code using tsc**.

Project templates

Starting in Visual Studio 2022, there is a new JavaScript/TypeScript project type (.esproj), called the JavaScript Project System (JSPS), which allows you to create standalone Angular, React, and Vue projects in Visual Studio. These front-end projects are created using the framework CLI tools you have installed on your local machine, so the version of the template is up to you. To migrate from existing Node.js projects to the new project system, see Migrate Node.js projects. For MSBuild information for the new project type, see MSBuild properties for JSPS

Within these new projects, you can run JavaScript and TypeScript unit tests, easily add and connect ASP.NET Core API projects and download your npm modules using the npm manager. Check out some of the quickstarts and tutorials to get started. For more information, see Visual Studio tutorials | JavaScript and TypeScript.

> **Note**
>
> A simplified, updated template is available starting in Visual Studio 2022 version 17.5. Compared to the ASP.NET SPA templates available in Visual Studio, the *.esproj*

SPA templates for ASP.NET Core provide better npm dependency management, and better build and publish support.

Feedback

Was this page helpful? Yes No

Overview of Single Page Apps (SPAs) in ASP.NET Core

Article • 10/10/2023

Visual Studio provides project templates for creating single-page apps (SPAs) based on JavaScript frameworks such as Angular, React, and Vue that have an ASP.NET Core backend. These templates:

- Create a Visual Studio solution with a frontend project and a backend project.
- Use the Visual Studio project type for JavaScript and TypeScript (*.esproj*) for the frontend.
- Use an ASP.NET Core project for the backend.

Projects created by using the Visual Studio templates can be run from the command line on Windows, Linux, and macOS. To run the app, use `dotnet run --launch-profile https` to run the server project. Running the server project automatically starts the frontend JavaScript development server. The `https` launch profile is currently required.

Visual Studio tutorials

To get started, follow one of the tutorials in the Visual Studio documentation:

- Create an ASP.NET Core app with Angular
- Create an ASP.NET Core app with React
- Create an ASP.NET Core app with Vue

For more information, see JavaScript and TypeScript in Visual Studio

ASP.NET Core SPA templates

Visual Studio includes templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend. These templates are available in Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

The Visual Studio templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend offer the following benefits:

- Clean project separation for the frontend and backend.
- Stay up-to-date with the latest frontend framework versions.
- Integrate with the latest frontend framework command-line tooling, such as Vite.

- Templates for both JavaScript & TypeScript (only TypeScript for Angular).
- Rich JavaScript and TypeScript code editing experience.
- Integrate JavaScript build tools with the .NET build.
- npm dependency management UI.
- Compatible with Visual Studio Code debugging and launch configuration.
- Run frontend unit tests in Test Explorer using JavaScript test frameworks.

Legacy ASP.NET Core SPA templates

Earlier versions of the .NET SDK included what are now legacy templates for building SPA apps with ASP.NET Core. For documentation on these older templates, see the ASP.NET Core 7.0 version of the SPA overview and the Angular and React articles.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use the Angular project template with ASP.NET Core

Article • 09/29/2023

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the **.NET 7 version of this article**.

Visual Studio provides project templates for creating single-page apps (SPAs) based on JavaScript frameworks such as Angular, React, and Vue that have an ASP.NET Core backend. These templates:

- Create a Visual Studio solution with a frontend project and a backend project.
- Use the Visual Studio project type for JavaScript and TypeScript (*.esproj*) for the frontend.
- Use an ASP.NET Core project for the backend.

Visual Studio tutorial

To get started with an Angular project, follow the Create an ASP.NET Core app with Angular tutorial in the Visual Studio documentation.

For more information, see JavaScript and TypeScript in Visual Studio

ASP.NET Core SPA templates

Visual Studio includes templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend. These templates are available in Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

The Visual Studio templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend offer the following benefits:

- Clean project separation for the frontend and backend.
- Stay up-to-date with the latest frontend framework versions.

- Integrate with the latest frontend framework command-line tooling, such as Vite.
- Templates for both JavaScript & TypeScript (only TypeScript for Angular).
- Rich JavaScript and TypeScript code editing experience.
- Integrate JavaScript build tools with the .NET build.
- npm dependency management UI.
- Compatible with Visual Studio Code debugging and launch configuration.
- Run frontend unit tests in Test Explorer using JavaScript test frameworks.

Legacy ASP.NET Core SPA templates

Earlier versions of the .NET SDK included what are now legacy templates for building SPA apps with ASP.NET Core. For documentation on these older templates, see the ASP.NET Core 7.0 version of the SPA overview and the Angular and React articles.

Use React with ASP.NET Core

Article • 09/29/2023

Visual Studio provides project templates for creating single-page apps (SPAs) based on JavaScript frameworks such as Angular, React, and Vue that have an ASP.NET Core backend. These templates:

- Create a Visual Studio solution with a frontend project and a backend project.
- Use the Visual Studio project type for JavaScript and TypeScript (*.esproj*) for the frontend.
- Use an ASP.NET Core project for the backend.

Projects created by using the Visual Studio templates can be run from the command line on Windows, Linux, and macOS. To run the app, use `dotnet run --launch-profile https` to run the server project. Running the server project automatically starts the frontend JavaScript development server. The `https` launch profile is currently required.

Visual Studio tutorial

To get started, follow the Create an ASP.NET Core app with React tutorial in the Visual Studio documentation.

For more information, see JavaScript and TypeScript in Visual Studio

ASP.NET Core SPA templates

Visual Studio includes templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend. These templates are available in Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

The Visual Studio templates for building ASP.NET Core apps with a JavaScript or TypeScript frontend offer the following benefits:

- Clean project separation for the frontend and backend.
- Stay up-to-date with the latest frontend framework versions.
- Integrate with the latest frontend framework command-line tooling, such as Vite.
- Templates for both JavaScript & TypeScript (only TypeScript for Angular).
- Rich JavaScript and TypeScript code editing experience.
- Integrate JavaScript build tools with the .NET build.
- npm dependency management UI.
- Compatible with Visual Studio Code debugging and launch configuration.

- Run frontend unit tests in Test Explorer using JavaScript test frameworks.

Legacy ASP.NET Core SPA templates

Earlier versions of the .NET SDK included what are now legacy templates for building SPA apps with ASP.NET Core. For documentation on these older templates, see the ASP.NET Core 7.0 version of the SPA overview and the Angular and React articles.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Client-side library acquisition in ASP.NET Core with LibMan

Article • 06/03/2022

By Scott Addie

Library Manager (LibMan) is a lightweight, client-side library acquisition tool. LibMan downloads popular libraries and frameworks from the file system or from a content delivery network (CDN). The supported CDNs include CDNJS, jsDelivr, and unpkg. The selected library files are fetched and placed in the appropriate location within the ASP.NET Core project.

LibMan use cases

LibMan offers the following benefits:

- Only the library files you need are downloaded.
- Additional tooling, such as Node.js, npm, and WebPack, isn't necessary to acquire a subset of files in a library.
- Files can be placed in a specific location without resorting to build tasks or manual file copying.

LibMan isn't a package management system. If you're already using a package manager, such as npm or yarn, continue doing so. LibMan wasn't developed to replace those tools.

Additional resources

- Use LibMan with ASP.NET Core in Visual Studio
- Use the LibMan CLI with ASP.NET Core
- LibMan GitHub repository

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Use the LibMan CLI with ASP.NET Core

Article • 01/11/2024

Library Manager (LibMan) is a lightweight, client-side library acquisition tool. LibMan downloads popular libraries and frameworks from the file system or from a content delivery network (CDN). The supported CDNs include CDNJS, jsDelivr, and unpkg. The selected library files are fetched and placed in the appropriate location within the ASP.NET Core project.

Prerequisites

- latest .NET SDK

Installation

The following command installs LibMan:

.NET CLI

```
dotnet tool install -g Microsoft.Web.LibraryManager.Cli
```

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

A .NET Core Global Tool is installed from the Microsoft.Web.LibraryManager.Cli NuGet package.

Usage

Console

```
libman
```

To view the installed LibMan version:

Console

```
libman --version
```

To view the available CLI commands:

Console

```
libman --help
```

The preceding command displays output similar to the following:

Console

```
 1.0.163+g45474d37ed

Usage: libman [options] [command]

Options:
  --help|-h  Show help information
  --version  Show version information

Commands:
  cache      List or clean libman cache contents
  clean      Deletes all library files defined in libman.json from the
project
  init       Create a new libman.json
  install    Add a library definition to the libman.json file, and download
the
             library to the specified location
  restore    Downloads all files from provider and saves them to specified
             destination
  uninstall  Deletes all files for the specified library from their
specified
             destination, then removes the specified library definition from
             libman.json
  update     Updates the specified library

Use "libman [command] --help" for more information about a command.
```

The following sections outline the available CLI commands.

Initialize LibMan in the project

The `libman init` command creates a `libman.json` file if one doesn't exist. The file is created with the default item template content.

Synopsis

Console

```
libman init [-d|--default-destination] [-p|--default-provider] [--verbosity]
libman init [-h|--help]
```

Options

The following options are available for the `libman init` command:

- `-d|--default-destination <PATH>`

  A path relative to the current folder. Library files are installed in this location if no `destination` property is defined for a library in `libman.json`. The `<PATH>` value is written to the `defaultDestination` property of `libman.json`.

- `-p|--default-provider <PROVIDER>`

  The provider to use if no provider is defined for a given library. The `<PROVIDER>` value is written to the `defaultProvider` property of `libman.json`. Replace `<PROVIDER>` with one of the following values:
  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

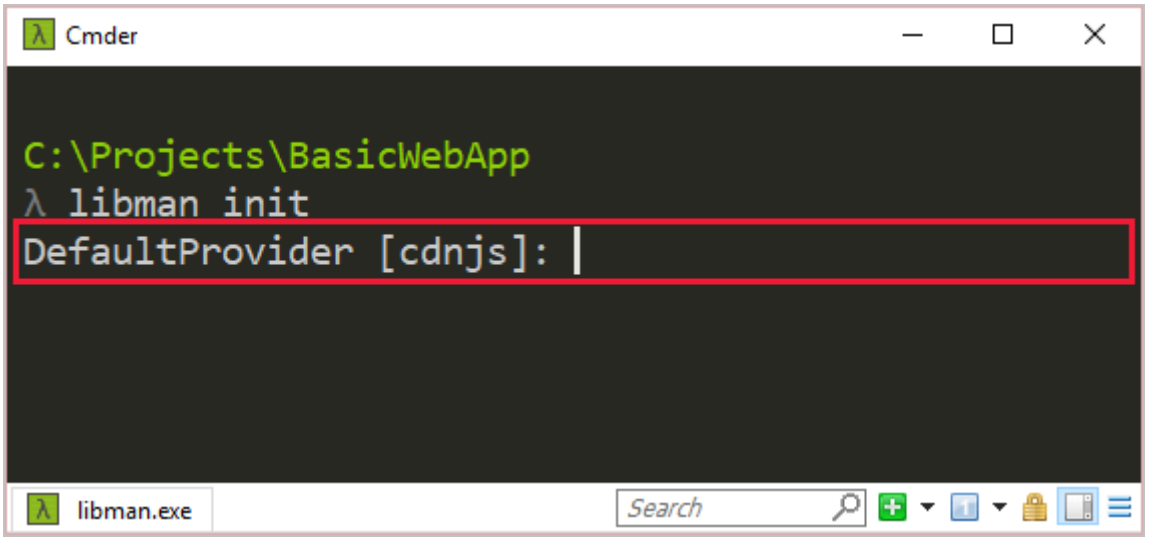
Examples

To create a `libman.json` file in an ASP.NET Core project:

- Navigate to the project root.
- Run the following command:

```
Console

libman init
```

- Type the name of the default provider, or press `Enter` to use the default CDNJS provider. Valid values include:
  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`



A `libman.json` file is added to the project root with the following content:

```
JSON

{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

Add library files

The `libman install` command downloads and installs library files into the project. A `libman.json` file is added if one doesn't exist. The `libman.json` file is modified to store configuration details for the library files.

Synopsis

Console

```
libman install <LIBRARY> [-d|--destination] [--files] [-p|--provider] [--
verbosity]
libman install [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to install. This name may include version number notation (for example, `@1.2.0`).

Options

The following options are available for the `libman install` command:

- `-d|--destination <PATH>`

  The location to install the library. If not specified, the default location is used. If no `defaultDestination` property is specified in `libman.json`, this option is required.

  Note: There are limitations to the destination path. For example, when the package source has a full project structure and not just the distribution folder, you can't specify moving a folder. For more information, see Issue #407 and Issue #702

- `--files <FILE>`

  Specify the name of the file to install from the library. If not specified, all files from the library are installed. Provide one `--files` option per file to be installed. Relative paths are supported too. For example: `--files dist/browser/signalr.js`.

- `-p|--provider <PROVIDER>`

  The name of the provider to use for the library acquisition. Replace `<PROVIDER>` with one of the following values:
  - `cdnjs`
  - `filesystem`
  - `jsdelivr`
  - `unpkg`

If not specified, the `defaultProvider` property in `libman.json` is used. If no `defaultProvider` property is specified in `libman.json`, this option is required.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

Consider the following `libman.json` file:

JSON

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
```

To install the jQuery version 3.2.1 `jquery.min.js` file to the *wwwroot/scripts/jquery* folder using the CDNJS provider:

Console

```
libman install jquery@3.2.1 --provider cdnjs --destination
wwwroot/scripts/jquery --files jquery.min.js
```

The `libman.json` file resembles the following:

JSON

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
```

```
        "jquery.min.js"
      ]
    }
  ]
}
```

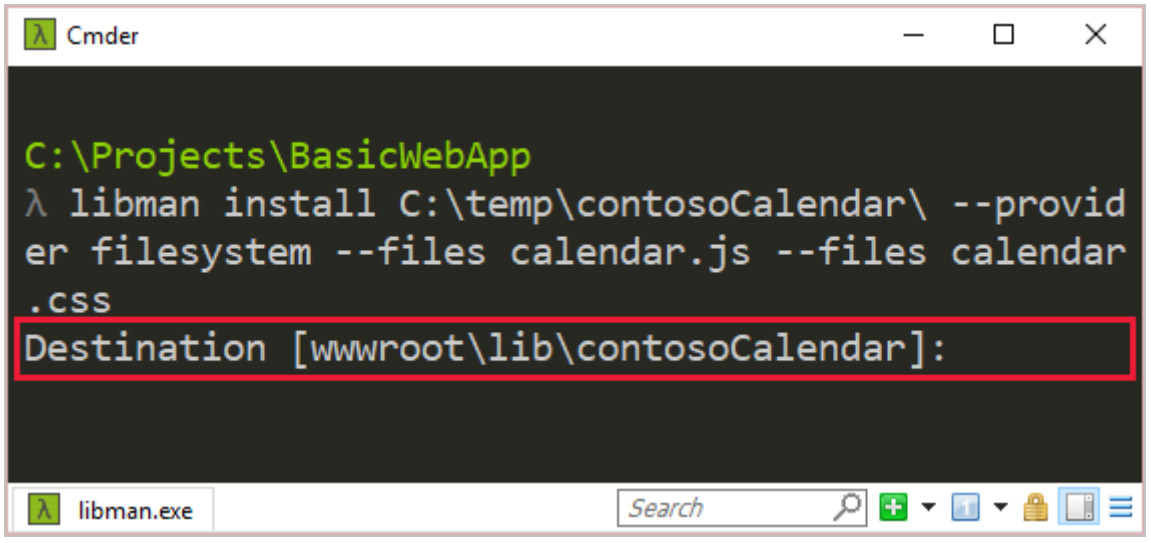
To install the `calendar.js` and `calendar.css` files from *C:\temp\contosoCalendar* using the file system provider:

Console

```
libman install C:\temp\contosoCalendar\ --provider filesystem --files
calendar.js --files calendar.css
```

The following prompt appears for two reasons:

- The `libman.json` file doesn't contain a `defaultDestination` property.
- The `libman install` command doesn't contain the `-d|--destination` option.



After accepting the default destination, the `libman.json` file resembles the following:

JSON

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.2.1",
      "destination": "wwwroot/scripts/jquery",
      "files": [
        "jquery.min.js"
      ]
    },
    {
```

```
      "library": "C:\\temp\\contosoCalendar\\",
      "provider": "filesystem",
      "destination": "wwwroot/lib/contosoCalendar",
      "files": [
        "calendar.js",
        "calendar.css"
      ]
    }
  ]
}
```

Restore library files

The `libman restore` command installs library files defined in `libman.json`. The following rules apply:

- If no `libman.json` file exists in the project root, an error is returned.
- If a library specifies a provider, the `defaultProvider` property in `libman.json` is ignored.
- If a library specifies a destination, the `defaultDestination` property in `libman.json` is ignored.

Synopsis

Console

```
libman restore [--verbosity]
libman restore [-h|--help]
```

Options

The following options are available for the `libman restore` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

To restore the library files defined in `libman.json`:

```
Console

libman restore
```

Delete library files

The `libman clean` command deletes library files previously restored via LibMan. Folders that become empty after this operation are deleted. The library files' associated configurations in the `libraries` property of `libman.json` aren't removed.

Synopsis

```
Console

libman clean [--verbosity]
libman clean [-h|--help]
```

Options

The following options are available for the `libman clean` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

To delete library files installed via LibMan:

```
Console
```

```
libman clean
```

Uninstall library files

The `libman uninstall` command:

- Deletes all files associated with the specified library from the destination in `libman.json`.
- Removes the associated library configuration from `libman.json`.

An error occurs when:

- No `libman.json` file exists in the project root.
- The specified library doesn't exist.

If more than one library with the same name is installed, you're prompted to choose one.

Synopsis

Console

```
libman uninstall <LIBRARY> [--verbosity]
libman uninstall [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to uninstall. This name may include version number notation (for example, `@1.2.0`).

Options

The following options are available for the `libman uninstall` command:

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:

- `quiet`
- `normal`
- `detailed`

Examples

Consider the following `libman.json` file:

JSON

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

- To uninstall jQuery, either of the following commands succeed:

  Console

  ```
  libman uninstall jquery
  ```

  Console

```
libman uninstall jquery@3.3.1
```

- To uninstall the Lodash files installed via the `filesystem` provider:

```
Console

libman uninstall C:\temp\lodash\
```

Update library version

The `libman update` command updates a library installed via LibMan to the specified version.

An error occurs when:

- No `libman.json` file exists in the project root.
- The specified library doesn't exist.

If more than one library with the same name is installed, you're prompted to choose one.

Synopsis

```
Console

libman update <LIBRARY> [-pre] [--to] [--verbosity]
libman update [-h|--help]
```

Arguments

`LIBRARY`

The name of the library to update.

Options

The following options are available for the `libman update` command:

- `-pre`

  Obtain the latest prerelease version of the library.

- `--to <VERSION>`

  Obtain a specific version of the library.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

- To update jQuery to the latest version:

  ```
  Console
  libman update jquery
  ```

- To update jQuery to version 3.3.1:

  ```
  Console
  libman update jquery --to 3.3.1
  ```

- To update jQuery to the latest prerelease version:

  ```
  Console
  libman update jquery -pre
  ```

Manage library cache

The `libman cache` command manages the LibMan library cache. The `filesystem` provider doesn't use the library cache.

Synopsis

```
```

Console

```
libman cache clean [<PROVIDER>] [--verbosity]
libman cache list [--files] [--libraries] [--verbosity]
libman cache [-h|--help]
```

Arguments

`PROVIDER`

Only used with the `clean` command. Specifies the provider cache to clean. Valid values include:

- `cdnjs`
- `filesystem`
- `jsdelivr`
- `unpkg`

Options

The following options are available for the `libman cache` command:

- `--files`

  List the names of files that are cached.

- `--libraries`

  List the names of libraries that are cached.

- `-h|--help`

  Show help information.

- `--verbosity <LEVEL>`

  Set the verbosity of the output. Replace `<LEVEL>` with one of the following values:
  - `quiet`
  - `normal`
  - `detailed`

Examples

- To view the names of cached libraries per provider, use one of the following commands:

  Console

  ```
  libman cache list
  ```

  Console

  ```
  libman cache list --libraries
  ```

  Output similar to the following is displayed:

  Console

  ```
  Cache contents:
  ---------------
  unpkg:
      knockout
      react
      vue
  cdnjs:
      font-awesome
      jquery
      knockout
      lodash.js
      react
  ```

- To view the names of cached library files per provider:

  Console

  ```
  libman cache list --files
  ```

  Output similar to the following is displayed:

  Console

  ```
  Cache contents:
  ---------------
  unpkg:
      knockout:
          <list omitted for brevity>
      react:
          <list omitted for brevity>
      vue:
          <list omitted for brevity>
  cdnjs:
  ```

```
    font-awesome
        metadata.json
    jquery
        metadata.json
        3.2.1\core.js
        3.2.1\jquery.js
        3.2.1\jquery.min.js
        3.2.1\jquery.min.map
        3.2.1\jquery.slim.js
        3.2.1\jquery.slim.min.js
        3.2.1\jquery.slim.min.map
        3.3.1\core.js
        3.3.1\jquery.js
        3.3.1\jquery.min.js
        3.3.1\jquery.min.map
        3.3.1\jquery.slim.js
        3.3.1\jquery.slim.min.js
        3.3.1\jquery.slim.min.map
    knockout
        metadata.json
        3.4.2\knockout-debug.js
        3.4.2\knockout-min.js
    lodash.js
        metadata.json
        4.17.10\lodash.js
        4.17.10\lodash.min.js
    react
        metadata.json
```

Notice the preceding output shows that jQuery versions 3.2.1 and 3.3.1 are cached under the CDNJS provider.

- To empty the library cache for the CDNJS provider:

Console

```
libman cache clean cdnjs
```

After emptying the CDNJS provider cache, the `libman cache list` command displays the following:

Console

```
Cache contents:
---------------
unpkg:
    knockout
    react
    vue
```

```
cdnjs:
    (empty)
```

- To empty the cache for all supported providers:

  Console

  ```
  libman cache clean
  ```

  After emptying all provider caches, the `libman cache list` command displays the following:

  Console

  ```
  Cache contents:
  ---------------
  unpkg:
      (empty)
  cdnjs:
      (empty)
  ```

Additional resources

- Install a Global Tool
- Use LibMan with ASP.NET Core in Visual Studio
- LibMan GitHub repository

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use LibMan with ASP.NET Core in Visual Studio

Article • 09/21/2022

By Scott Addie

Visual Studio has built-in support for LibMan in ASP.NET Core projects, including:

- Support for configuring and running LibMan restore operations on build.
- Menu items for triggering LibMan restore and clean operations.
- Search dialog for finding libraries and adding the files to a project.
- Editing support for `libman.json`—the LibMan manifest file.

View or download sample code (how to download)

Prerequisites

- Visual Studio 2019 with the **ASP.NET and web development** workload

Add library files

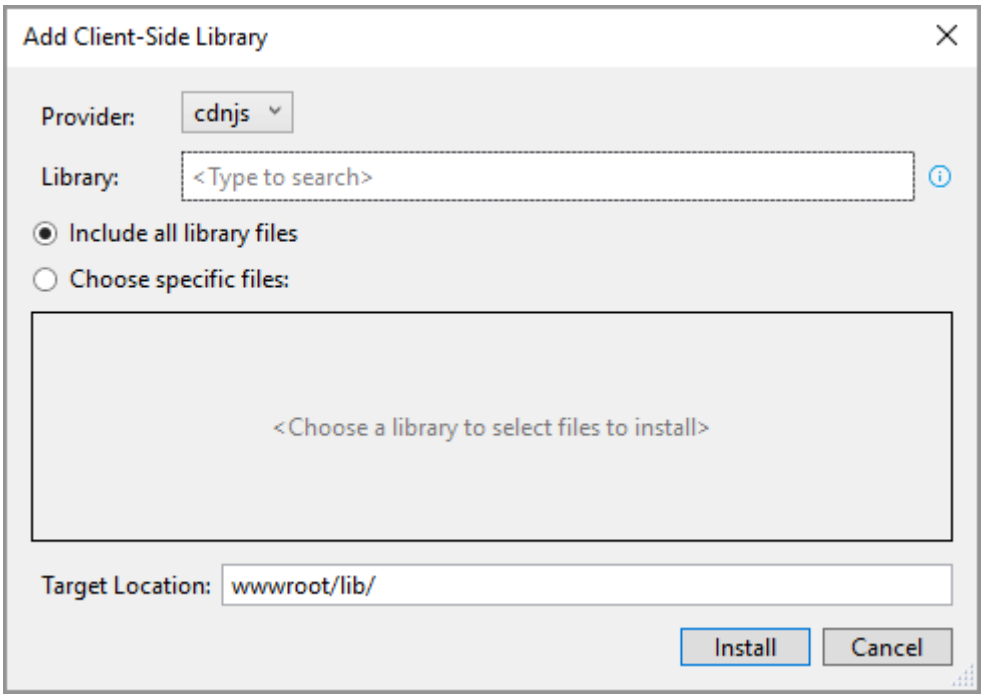
Library files can be added to an ASP.NET Core project in two different ways:

1. Use the Add Client-Side Library dialog
2. Manually configure LibMan manifest file entries

Use the Add Client-Side Library dialog

Follow these steps to install a client-side library:

- In **Solution Explorer**, right-click the project folder in which the files should be added. Choose **Add** > **Client-Side Library**. The **Add Client-Side Library** dialog appears:



- Select the library provider from the **Provider** drop down. CDNJS is the default provider.
- Type the library name to fetch in the **Library** text box. IntelliSense provides a list of libraries beginning with the provided text.
- Select the library from the IntelliSense list. Notice the library name is suffixed with the `@` symbol and the latest stable version known to the selected provider.
- Decide which files to include:
  - Select the **Include all library files** radio button to include all of the library's files.
  - Select the **Choose specific files** radio button to include a subset of the library's files. When the radio button is selected, the file selector tree is enabled. Check the boxes to the left of the file names to download.
- Specify the project folder for storing the files in the **Target Location** text box. As a recommendation, store each library in a separate folder.

  The suggested **Target Location** folder is based on the location from which the dialog launched:
  - If launched from the project root:
    - *wwwroot/lib* is used if *wwwroot* exists.
    - *lib* is used if *wwwroot* doesn't exist.
  - If launched from a project folder, the corresponding folder name is used.

  The folder suggestion is suffixed with the library name. The following table illustrates folder suggestions when installing jQuery in a Razor Pages project.

Expand table

Launch location	Suggested folder
project root (if *wwwroot* exists)	*wwwroot/lib/jquery/*
project root (if *wwwroot* doesn't exist)	*lib/jquery/*
Pages folder in project	*Pages/jquery/*

- Click the **Install** button to download the files, per the configuration in `libman.json`.
- Review the **Library Manager** feed of the **Output** window for installation details. For example:

Console

```
Restore operation started...
Restoring libraries for project LibManSample
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

Manually configure LibMan manifest file entries

All LibMan operations in Visual Studio are based on the content of the project root's LibMan manifest (`libman.json`). You can manually edit `libman.json` to configure library files for the project. Visual Studio restores all library files once `libman.json` is saved.

To open `libman.json` for editing, the following options exist:

- Double-click the `libman.json` file in **Solution Explorer**.
- Right-click the project in **Solution Explorer** and select **Manage Client-Side Libraries**. †
- Select **Manage Client-Side Libraries** from the Visual Studio **Project** menu. †

† If the `libman.json` file doesn't already exist in the project root, it will be created with the default item template content.

Visual Studio offers rich JSON editing support such as colorization, formatting, IntelliSense, and schema validation. The LibMan manifest's JSON schema is found at https://json.schemastore.org/libman.

With the following manifest file, LibMan retrieves files per the configuration defined in the `libraries` property. An explanation of the object literals defined within `libraries` follows:

- A subset of jQuery version 3.3.1 is retrieved from the CDNJS provider. The subset is defined in the `files` property—`jquery.min.js`, `jquery.js`, and *jquery.min.map*. The files are placed in the project's *wwwroot/lib/jquery* folder.
- The entirety of Bootstrap version 4.1.3 is retrieved and placed in a *wwwroot/lib/bootstrap* folder. The object literal's `provider` property overrides the `defaultProvider` property value. LibMan retrieves the Bootstrap files from the unpkg provider.
- A subset of Lodash was approved by a governing body within the organization. The `lodash.js` and `lodash.min.js` files are retrieved from the local file system at *C:\temp\lodash*. The files are copied to the project's *wwwroot/lib/lodash* folder.

JSON

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.3.1",
      "files": [
        "jquery.min.js",
        "jquery.js",
        "jquery.min.map"
      ],
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "provider": "unpkg",
      "library": "bootstrap@4.1.3",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "provider": "filesystem",
      "library": "C:\\temp\\lodash\\",
      "files": [
        "lodash.js",
        "lodash.min.js"
      ],
      "destination": "wwwroot/lib/lodash/"
    }
  ]
}
```

> **Note**
>
> LibMan only supports one version of each library from each provider. The `libman.json` file fails schema validation if it contains two libraries with the same library name for a given provider.

Restore library files

To restore library files from within Visual Studio, there must be a valid `libman.json` file in the project root. Restored files are placed in the project at the location specified for each library.

Library files can be restored in an ASP.NET Core project in two ways:

1. Restore files during build
2. Restore files manually

Restore files during build

LibMan can restore the defined library files as part of the build process. By default, the *restore-on-build* behavior is disabled.

To enable and test the restore-on-build behavior:

- Right-click `libman.json` in **Solution Explorer** and select **Enable Restore Client-Side Libraries on Build** from the context menu.
- Click the **Yes** button when prompted to install a NuGet package. The Microsoft.Web.LibraryManager.Build NuGet package is added to the project:

  XML

  ```
  <PackageReference Include="Microsoft.Web.LibraryManager.Build"
  Version="1.0.113" />
  ```

- Build the project to confirm LibMan file restoration occurs. The `Microsoft.Web.LibraryManager.Build` package injects an MSBuild target that runs LibMan during the project's build operation.
- Review the **Build** feed of the **Output** window for a LibMan activity log:

  Console

```
1>------ Build started: Project: LibManSample, Configuration: Debug Any
CPU ------
1>
1>Restore operation started...
1>Restoring library jquery@3.3.1...
1>Restoring library bootstrap@4.1.3...
1>
1>2 libraries restored in 10.66 seconds
1>LibManSample ->
C:\LibManSample\bin\Debug\netcoreapp2.1\LibManSample.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
==========
```

When the restore-on-build behavior is enabled, the `libman.json` context menu displays a **Disable Restore Client-Side Libraries on Build** option. Selecting this option removes the `Microsoft.Web.LibraryManager.Build` package reference from the project file. Consequently, the client-side libraries are no longer restored on each build.

Regardless of the restore-on-build setting, you can manually restore at any time from the `libman.json` context menu. For more information, see Restore files manually.

Restore files manually

To manually restore library files:

- For all projects in the solution:
  - Right-click the solution name in **Solution Explorer**.
  - Select the **Restore Client-Side Libraries** option.
- For a specific project:
  - Right-click the `libman.json` file in **Solution Explorer**.
  - Select the **Restore Client-Side Libraries** option.

While the restore operation is running:

- The Task Status Center (TSC) icon on the Visual Studio status bar will be animated and will read *Restore operation started*. Clicking the icon opens a tooltip listing the known background tasks.
- Messages will be sent to the status bar and the **Library Manager** feed of the **Output** window. For example:

Console

```
Restore operation started...
Restoring libraries for project LibManSample
```

```
Restoring library jquery@3.3.1... (LibManSample)
wwwroot/lib/jquery/jquery.min.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.js written to destination (LibManSample)
wwwroot/lib/jquery/jquery.min.map written to destination (LibManSample)
Restore operation completed
1 libraries restored in 2.32 seconds
```

Delete library files

To perform the *clean* operation, which deletes library files previously restored in Visual Studio:

- Right-click the `libman.json` file in **Solution Explorer**.
- Select the **Clean Client-Side Libraries** option.

To prevent unintentional removal of non-library files, the clean operation doesn't delete whole directories. It only removes files that were included in the previous restore.

While the clean operation is running:

- The TSC icon on the Visual Studio status bar will be animated and will read *Client libraries operation started*. Clicking the icon opens a tooltip listing the known background tasks.
- Messages are sent to the status bar and the **Library Manager** feed of the **Output** window. For example:

Console

```
Clean libraries operation started...
Clean libraries operation completed
2 libraries were successfully deleted in 1.91 secs
```

The clean operation only deletes files from the project. Library files stay in the cache for faster retrieval on future restore operations. To manage library files stored in the local machine's cache, use the LibMan CLI.

Uninstall library files

To uninstall library files:

- Open `libman.json`.
- Position the caret inside the corresponding `libraries` object literal.

- Click the light bulb icon that appears in the left margin, and select **Uninstall <library_name>@<library_version>**:



Alternatively, you can manually edit and save the LibMan manifest (`libman.json`). The restore operation runs when the file is saved. Library files that are no longer defined in `libman.json` are removed from the project.

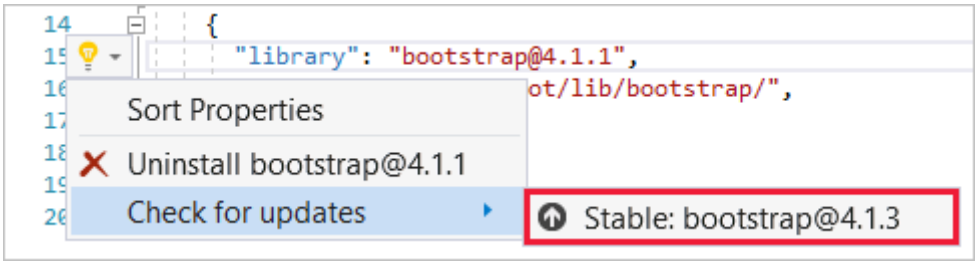
Update library version

To check for an updated library version:

- Open `libman.json`.
- Position the caret inside the corresponding `libraries` object literal.
- Click the light bulb icon that appears in the left margin. Hover over **Check for updates**.

LibMan checks for a library version newer than the version installed. The following outcomes can occur:

- A **No updates found** message is displayed if the latest version is already installed.
- The latest stable version is displayed if not already installed.



- If a pre-release newer than the installed version is available, the pre-release is displayed.

To downgrade to an older library version, manually edit the `libman.json` file. When the file is saved, the LibMan restore operation:

- Removes redundant files from the previous version.
- Adds new and updated files from the new version.

Additional resources

- Use the LibMan CLI with ASP.NET Core
- LibMan GitHub repository

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly

Article • 08/22/2024

By Aaron Shumaker

This article explains how to interact with JavaScript (JS) in client-side WebAssembly using JS `[JSImport]`/`[JSExport]` interop (System.Runtime.InteropServices.JavaScript API).

`[JSImport]`/`[JSExport]` interop is applicable when running a .NET WebAssembly module in a JS host in the following scenarios:

- JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project.
- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.
- Other .NET WebAssembly platforms that support `[JSImport]`/`[JSExport]` interop.

Prerequisites

.NET SDK (latest version)

Any of the following project types:

- A WebAssembly Browser App project created according to JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project.
- A Blazor client-side project created according to JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.
- A project created for a commercial or open-source platform that supports `[JSImport]`/`[JSExport]` interop (System.Runtime.InteropServices.JavaScript API).

Sample app

View or download sample code (how to download): Select an 8.0 or later version folder that matches the version of .NET that you're adopting. Within the version folder, access the sample named `WASMBrowserAppImportExportInterop`.

JS interop using `[JSImport]`/`[JSExport]` attributes

The `[JSImport]` attribute is applied to a .NET method to indicate that a corresponding JS method should be called when the .NET method is called. This allows .NET developers to define "imports" that enable .NET code to call into JS. Additionally, an Action can be passed as a parameter, and JS can invoke the action to support a callback or event subscription pattern.

The `[JSExport]` attribute is applied to a .NET method to expose it to JS code. This allows JS code to initiate calls to the .NET method.

Importing JS methods

The following example imports a standard built-in JS method (`console.log`) into C#. `[JSImport]` is limited to importing methods of globally-accessible objects. For example, `log` is a method defined on the `console` object, which is defined on the globally-accessible object `globalThis`. The `console.log` method is mapped to a C# proxy method, `ConsoleLog`, which accepts a string for the log message:

C#

```
public partial class GlobalInterop
{
    [JSImport("globalThis.console.log")]
    public static partial void ConsoleLog(string text);
}
```

In `Program.Main`, `ConsoleLog` is called with the message to log:

C#

```
GlobalInterop.ConsoleLog("Hello World!");
```

The output appears in the browser's console.

The following demonstrates importing a method declared in JS.

The following custom JS method (`globalThis.callAlert`) spawns an alert dialog (window.alert) with the message passed in `text`:

JavaScript

```
globalThis.callAlert = function (text) {
  globalThis.window.alert(text);
}
```

The `globalThis.callAlert` method is mapped to a C# proxy method (`CallAlert`), which accepts a string for the message:

C#

```
using System.Runtime.InteropServices.JavaScript;

public partial class GlobalInterop
{
    [JSImport("globalThis.callAlert")]
    public static partial void CallAlert(string text);
}
```

In `Program.Main`, `CallAlert` is called, passing the text for the alert dialog message:

C#

```
GlobalInterop.CallAlert("Hello World");
```

The C# class declaring the `[JSImport]` method doesn't have an implementation. At compile time, a source-generated partial class contains the .NET code that implements the marshalling of the call and types to invoke the corresponding JS method. In Visual Studio, using the **Go To Definition** or **Go To Implementation** options respectively navigates to either the source-generated partial class or the developer-defined partial class.

In the preceding example, the intermediate `globalThis.callAlert` JS declaration is used to wrap existing JS code. This article informally refers to the intermediate JS declaration as a *JS shim*. JS shims fill the gap between the .NET implementation and existing JS capabilities/libraries. In many cases, such as the preceding trivial example, the JS shim isn't necessary, and methods could be imported directly, as demonstrated in the earlier `ConsoleLog` example. As this article demonstrates in the upcoming sections, a JS shim can:

- Encapsulate additional logic.
- Manually map types.
- Reduce the number of objects or calls crossing the interop boundary.
- Manually map static calls to instance methods.

Loading JavaScript declarations

JS declarations which are intended to be imported with `[JSImport]` are typically loaded in the context of the same page or JS host that loaded .NET WebAssembly. This can be accomplished with:

- A `<script>...</script>` block declaring inline JS.
- A script source (`src`) declaration (`<script src="./some.js"></script>`) that loads an external JS file (`.js`).
- A JS ES6 module (`<script type='module' src="./moduleName.js"></script>`).
- A JS ES6 module loaded using JSHost.ImportAsync from .NET WebAssembly.

Examples in this article use JSHost.ImportAsync. When calling ImportAsync, client-side .NET WebAssembly requests the file using the `moduleUrl` parameter, and thus it expects the file to be accessible as a static web asset, much the same way as a `<script>` tag retrieves a file with a `src` URL. For example, the following C# code within a WebAssembly Browser App project maintains the JS file (`.js`) at the path `/wwwroot/scripts/ExampleShim.js`:

C#

```
await JSHost.ImportAsync("ExampleShim", "/scripts/ExampleShim.js");
```

Depending on the platform that's loading WebAssembly, a dot-prefixed URL, such as `./scripts/`, might refer to an incorrect subdirectory, such as `/_framework/scripts/`, because the WebAssembly package is initialized by framework scripts under `/_framework/`. In that case, prefixing the URL with `../scripts/` refers to the correct path. Prefixing with `/scripts/` works if the site is hosted at the root of the domain. A typical approach involves configuring the correct base path for the given environment with an HTML `<base>` tag and using the `/scripts/` prefix to refer to the path relative to the base path. Tilde notation `~/` prefixes aren't supported by JSHost.ImportAsync.

> **ⓘ Important**
>
> If JS is loaded from a JavaScript module, then `[JSImport]` attributes must include the module name as the second parameter. For example, `[JSImport("globalThis.callAlert", "ExampleShim")]` indicates the imported method was declared in a JavaScript module named "`ExampleShim`."

Type mappings

Parameters and return types in the .NET method signature are automatically converted to or from appropriate JS types at runtime if a unique mapping is supported. This may result in values converted by value or references wrapped in a proxy type. This process is known as *type marshalling*. Use JSMarshalAsAttribute<T> to control how the imported method parameters and return types are marshalled.

Some types don't have a default type mapping. For example, a `long` can be marshalled as System.Runtime.InteropServices.JavaScript.JSType.Number or System.Runtime.InteropServices.JavaScript.JSType.BigInt, so the JSMarshalAsAttribute<T> is required to avoid a compile-time error.

The following type mapping scenarios are supported:

- Passing Action or Func<TResult> as parameters, which are are marshalled as callable JS methods. This allows .NET code to invoke listeners in response to JS callbacks or events.
- Passing JS references and .NET managed object references in either direction, which as marshaled as proxy objects and kept alive across the interop boundary until the proxy is garbage collected.
- Marshalling asynchronous JS methods or a JS Promise with a Task result, and vice versa.

Most of the marshalled types work in both directions, as parameters and as return values, on both imported and exported methods.

The following table indicates the supported type mappings.

Expand table

.NET	JavaScript	`Nullable`	`Task` ➔ `Promise`	`JSMarshalAs` optional	`Array of`
`Boolean`	`Boolean`	✅	✅	✅	
`Byte`	`Number`	✅	✅	✅	✅
`Char`	`String`	✅	✅	✅	
`Int16`	`Number`	✅	✅	✅	
`Int32`	`Number`	✅	✅	✅	✅
`Int64`	`Number`	✅	✅		

.NET	JavaScript	`Nullable`	`Task` ➔ `Promise`	`JSMarshalAs` optional	`Array of`
`Int64`	`BigInt`	✅	✅		
`Single`	`Number`	✅	✅	✅	
`Double`	`Number`	✅	✅	✅	✅
`IntPtr`	`Number`	✅	✅	✅	
`DateTime`	`Date`	✅	✅		
`DateTimeOffset`	`Date`	✅	✅		
`Exception`	`Error`		✅	✅	
`JSObject`	`Object`		✅	✅	✅
`String`	`String`		✅	✅	✅
`Object`	`Any`		✅		✅
`Span<Byte>`	`MemoryView`				
`Span<Int32>`	`MemoryView`				
`Span<Double>`	`MemoryView`				
`ArraySegment<Byte>`	`MemoryView`				
`ArraySegment<Int32>`	`MemoryView`				
`ArraySegment<Double>`	`MemoryView`				
`Task`	`Promise`			✅	
`Action`	`Function`				
`Action<T1>`	`Function`				
`Action<T1, T2>`	`Function`				
`Action<T1, T2, T3>`	`Function`				
`Func<TResult>`	`Function`				
`Func<T1, TResult>`	`Function`				
`Func<T1, T2, TResult>`	`Function`				
`Func<T1, T2, T3,`	`Function`				

.NET	JavaScript	Nullable	Task ➔ Promise	JSMarshalAs optional	Array of
TResult>					

The following conditions apply to type mapping and marshalled values:

- The `Array of` column indicates if the .NET type can be marshalled as a JS Array. Example: C# `int[]` (`Int32`) mapped to JS `Array` of `Number`s.
- When passing a JS value to C# with a value of the wrong type, the framework throws an exception in most cases. The framework doesn't perform compile-time type checking in JS.
- `JSObject`, `Exception`, `Task` and `ArraySegment` create `GCHandle` and a proxy. You can trigger disposal in developer code or allow .NET garbage collection (GC) to dispose of the objects later. These types carry significant performance overhead.
- `Array`: Marshaling an array creates a copy of the array in JS or .NET.
- `MemoryView`
  - `MemoryView` is a JS class for the .NET WebAssembly runtime to marshal `Span` and `ArraySegment`.
  - Unlike marshaling an array, marshaling a `Span` or `ArraySegment` doesn't create a copy of the underlying memory.
  - `MemoryView` can only be properly instantiated by the .NET WebAssembly runtime. Therefore, it isn't possible to import a JS method as a .NET method that has a parameter of `Span` or `ArraySegment`.
  - `MemoryView` created for a `Span` is only valid for the duration of the interop call. As `Span` is allocated on the call stack, which doesn't persist after the interop call, it isn't possible to export a .NET method that returns a `Span`.
  - `MemoryView` created for an `ArraySegment` survives after the interop call and is useful for sharing a buffer. Calling `dispose()` on a `MemoryView` created for an `ArraySegment` disposes the proxy and unpins the underlying .NET array. We recommend calling `dispose()` in a `try-finally` block for `MemoryView`.

Some combinations of type mappings that require nested generic types in JSMarshalAs aren't currently supported. For example, attempting to materialize an array from a Promise such as `[return: JSMarshalAs<JSType.Promise<JSType.Array<JSType.Number>>>()]` generates a compile-time error. An appropriate workaround varies depending on the scenario, but this specific scenario is further explored in the Type mapping limitations section.

JS primitives

The following example demonstrates `[JSImport]` leveraging type mappings of several primitive JS types and the use of JSMarshalAs, where explicit mappings are required at compile time.

`PrimitivesShim.js`:

JavaScript

```
globalThis.counter = 0;

// Takes no parameters and returns nothing.
export function incrementCounter() {
  globalThis.counter += 1;
};

// Returns an int.
export function getCounter() { return globalThis.counter; };

// Takes a parameter and returns nothing. JS doesn't restrict the parameter
type,
// but we can restrict it in the .NET proxy, if desired.
export function logValue(value) { console.log(value); };

// Called for various .NET types to demonstrate mapping to JS primitive
types.
export function logValueAndType(value) { console.log(typeof value, value);
};
```

`PrimitivesInterop.cs`:

C#

```
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class PrimitivesInterop
{
    // Importing an existing JS method.
    [JSImport("globalThis.console.log")]
    public static partial void ConsoleLog([JSMarshalAs<JSType.Any>] object
value);

    // Importing static methods from a JS module.
    [JSImport("incrementCounter", "PrimitivesShim")]
    public static partial void IncrementCounter();

    [JSImport("getCounter", "PrimitivesShim")]
    public static partial int GetCounter();

    // The JS shim method name isn't required to match the C# method name.
```

```
    [JSImport("logValue", "PrimitivesShim")]
    public static partial void LogInt(int value);

    // A second mapping to the same JS method with compatible type.
    [JSImport("logValue", "PrimitivesShim")]
    public static partial void LogString(string value);

    // Accept any type as parameter. .NET types are mapped to JS types where
    // possible. Otherwise, they're marshalled as an untyped object
reference
    // to the .NET object proxy. The JS implementation logs to browser
console
    // the JS type and value to demonstrate results of marshalling.
    [JSImport("logValueAndType", "PrimitivesShim")]
    public static partial void LogValueAndType(
        [JSMarshalAs<JSType.Any>] object value);

    // Some types have multiple mappings and require explicit marshalling to
the
    // desired JS type. A long/Int64 can be mapped as either a Number or
BigInt.
    // Passing a long value to the above method generates an error at
runtime:
    // "ToJS for System.Int64 is not implemented." ("ToJS" means "to
JavaScript")
    // If the parameter declaration `Method(JSMarshalAs<JSType.Any>] long
value)`
    // is used, a compile-time error is generated:
    // "Type long is not supported by source-generated JS interop...."
    // Instead, explicitly map the long parameter to either a JSType.Number
or
    // JSType.BigInt. Note that runtime overflow errors are possible in JS
if the
    // C# value is too large.
    [JSImport("logValueAndType", "PrimitivesShim")]
    public static partial void LogValueAndTypeForNumber(
        [JSMarshalAs<JSType.Number>] long value);

    [JSImport("logValueAndType", "PrimitivesShim")]
    public static partial void LogValueAndTypeForBigInt(
        [JSMarshalAs<JSType.BigInt>] long value);
}

public static class PrimitivesUsage
{
    public static async Task Run()
    {
        // Ensure JS module loaded.
        await JSHost.ImportAsync("PrimitivesShim", "/PrimitivesShim.js");

        // Call a proxy to a static JS method, console.log().
        PrimitivesInterop.ConsoleLog("Printed from JSImport of
console.log()");

        // Basic examples of JS interop with an integer.
```

```
        PrimitivesInterop.IncrementCounter();
        int counterValue = PrimitivesInterop.GetCounter();
        PrimitivesInterop.LogInt(counterValue);
        PrimitivesInterop.LogString("I'm a string from .NET in your
browser!");

        // Mapping some other .NET types to JS primitives.
        PrimitivesInterop.LogValueAndType(true);
        PrimitivesInterop.LogValueAndType(0x3A); // Byte literal
        PrimitivesInterop.LogValueAndType('C');
        PrimitivesInterop.LogValueAndType((Int16)12);
        // JS Number has a lower max value and can generate overflow errors.
        PrimitivesInterop.LogValueAndTypeForNumber(9007199254740990L); //
Int64/Long
        // Next line: Int64/Long, JS BigInt supports larger numbers.
        PrimitivesInterop.LogValueAndTypeForBigInt(1234567890123456789L);//
        PrimitivesInterop.LogValueAndType(3.14f); // Single floating point
literal
        PrimitivesInterop.LogValueAndType(3.14d); // Double floating point
literal
        PrimitivesInterop.LogValueAndType("A string");
    }
}
```

In `Program.Main`:

C#

```
await PrimitivesUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> Printed from JSImport of console.log()
> 1
> I'm a string from .NET in your browser!
> boolean true
> number 58
> number 67
> number 12
> number 9007199254740990
> bigint 1234567890123456789n
> number 3.140000104904175
> number 3.14
> string A string

JS `Date` objects

The example in this section demonstrates importing methods which have a JS Date object as its return or parameter. Dates are marshalled across interop by-value, meaning they are copied in much the same way as JS primitives.

A `Date` object is timezone agnostic. A .NET DateTime is adjusted relative to its DateTimeKind when marshalled to a `Date`, but timezone information isn't preserved. Consider initializing a DateTime with a DateTimeKind.Utc or DateTimeKind.Local consistent with the value it represents.

`DateShim.js`:

JavaScript

```
export function incrementDay(date) {
  date.setDate(date.getDate() + 1);
  return date;
}

export function logValueAndType(value) {
  console.log("Date:", value)
}
```

`DateInterop.cs`:

C#

```
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class DateInterop
{
    [JSImport("incrementDay", "DateShim")]
    [return: JSMarshalAs<JSType.Date>] // Explicit JSMarshalAs for a return
type
    public static partial DateTime IncrementDay(
        [JSMarshalAs<JSType.Date>] DateTime date);

    [JSImport("logValueAndType", "DateShim")]
    public static partial void LogValueAndType(
        [JSMarshalAs<JSType.Date>] DateTime value);
}

public static class DateUsage
{
    public static async Task Run()
    {
        // Ensure JS module loaded.
        await JSHost.ImportAsync("DateShim", "/DateShim.js");
```

```
        // Basic examples of interop with a C# DateTime and JS Date.
        DateTime date = new DateTime(1968, 12, 21, 12, 51, 0,
DateTimeKind.Utc);
        DateInterop.LogValueAndType(date);
        date = DateInterop.IncrementDay(date);
        DateInterop.LogValueAndType(date);
    }
}
```

In `Program.Main`:

C#

```
await DateUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> Date: Sat Dec 21 1968 07:51:00 GMT-0500 (Eastern Standard Time)
> Date: Sun Dec 22 1968 07:51:00 GMT-0500 (Eastern Standard Time)

The preceding timezone information (`GMT-0500 (Eastern Standard Time)`) depends on local timezone of your computer/browser.

JS object references

Whenever a JS method returns an object reference, it's represented in .NET as a JSObject. The original JS object continues its lifetime within the JS boundary, while .NET code can access and modify it by reference through the JSObject. While the type itself exposes a limited API, the ability to hold a JS object reference and return or pass it across the interop boundary enables support for several interop scenarios.

The JSObject provides methods to access properties, but it doesn't provide direct access to instance methods. As the following `Summarize` method demonstrates, instance methods can be accessed indirectly by implementing a static method that takes the instance as a parameter.

`JSObjectShim.js`:

JavaScript

```
export function createObject() {
  return {
    name: "Example JS Object",
    answer: 41,
    question: null,
```

```
    summarize: function () {
      return `Question: "${this.question}" Answer: ${this.answer}`;
    }
  };
}

export function incrementAnswer(object) {
  object.answer += 1;
  // Don't return the modified object, since the reference is modified.
}

// Proxy an instance method call.
export function summarize(object) {
  return object.summarize();
}
```

`JSObjectInterop.cs`:

C#

```
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class JSObjectInterop
{
    [JSImport("createObject", "JSObjectShim")]
    public static partial JSObject CreateObject();

    [JSImport("incrementAnswer", "JSObjectShim")]
    public static partial void IncrementAnswer(JSObject jsObject);

    [JSImport("summarize", "JSObjectShim")]
    public static partial string Summarize(JSObject jsObject);

    [JSImport("globalThis.console.log")]
    public static partial void ConsoleLog([JSMarshalAs<JSType.Any>] object
value);
}

public static class JSObjectUsage
{
    public static async Task Run()
    {
        await JSHost.ImportAsync("JSObjectShim", "/JSObjectShim.js");

        JSObject jsObject = JSObjectInterop.CreateObject();
        JSObjectInterop.ConsoleLog(jsObject);
        JSObjectInterop.IncrementAnswer(jsObject);
        // An updated object isn't retrieved. The change is reflected in the
        // existing instance.
        JSObjectInterop.ConsoleLog(jsObject);
```

```
        // JSObject exposes several methods for interacting with properties.
        jsObject.SetProperty("question", "What is the answer?");
        JSObjectInterop.ConsoleLog(jsObject);

        // We can't directly JSImport an instance method on the jsObject,
but we
        // can pass the object reference and have the JS shim call the
instance
        // method.
        string summary = JSObjectInterop.Summarize(jsObject);
        Console.WriteLine("Summary: " + summary);
    }
}
```

In `Program.Main`:

C#

```
await JSObjectUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> {name: 'Example JS Object', answer: 41, question: null, Symbol(wasm cs_owned_js_handle): 5, summarize: *ƒ*}
> {name: 'Example JS Object', answer: 42, question: null, Symbol(wasm cs_owned_js_handle): 5, summarize: *ƒ*}
> {name: 'Example JS Object', answer: 42, question: 'What is the answer?', Symbol(wasm cs_owned_js_handle): 5, summarize: *ƒ*}
> Summary: Question: "What is the answer?" Answer: 42

Asynchronous interop

Many JS APIs are asynchronous and signal completion through either a callback, a Promise, or an async method. Ignoring asynchronous capabilities is often not an option, as subsequent code may depend upon the completion of the asynchronous operation and must be awaited.

JS methods using the `async` keyword or returning a Promise can be awaited in C# by a method returning a Task. As demonstrated below, the `async` keyword isn't used on the C# method with the `[JSImport]` attribute because it doesn't use the `await` keyword within it. However, consuming code calling the method would typically use the `await` keyword and be marked as `async`, as demonstrated in the `PromisesUsage` example.

JS with a callback, such as a `setTimeout`, can be wrapped in a Promise before returning from JS. Wrapping a callback in a Promise, as demonstrated in the function assigned to `Wait2Seconds`, is only appropriate when the callback is called exactly once. Otherwise, a C# Action can be passed to listen for a callback that may be called zero or many times, which is demonstrated in the Subscribing to JS events section.

`PromisesShim.js`:

JavaScript

```
export function wait2Seconds() {
  // This also demonstrates wrapping a callback-based API in a promise to
  // make it awaitable.
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(); // Resolve promise after 2 seconds
    }, 2000);
  });
}

// Return a value via resolve in a promise.
export function waitGetString() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("String From Resolve"); // Return a string via promise
    }, 500);
  });
}

export function waitGetDate() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(new Date('1988-11-24')); // Return a date via promise
    }, 500);
  });
}

// Demonstrates an awaitable fetch.
export function fetchCurrentUrl() {
  // This method returns the promise returned by .then(*.text())
  // and .NET awaits the returned promise.
  return fetch(globalThis.window.location, { method: 'GET' })
    .then(response => response.text());
}

// .NET can await JS methods using the async/await JS syntax.
export async function asyncFunction() {
  await wait2Seconds();
}

// A Promise.reject can be used to signal failure and is bubbled to .NET
code
```

```
// as a JSException.
export function conditionalSuccess(shouldSucceed) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldSucceed)
        resolve(); // Success
      else
        reject("Reject: ShouldSucceed == false"); // Failure
    }, 500);
  });
}
```

Don't use the `async` keyword in the C# method signature. Returning Task or Task<TResult> is sufficient.

When calling asynchronous JS methods, we often want to wait until the JS method completes execution. If loading a resource or making a request, we likely want the following code to assume the action is completed.

If the JS shim returns a Promise, then C# can treat it as an awaitable Task/Task<TResult>.

`PromisesInterop.cs`:

C#

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class PromisesInterop
{
    // For a promise with void return type, declare a Task return type:
    [JSImport("wait2Seconds", "PromisesShim")]
    public static partial Task Wait2Seconds();

    [JSImport("waitGetString", "PromisesShim")]
    public static partial Task<string> WaitGetString();

    // Some return types require a [return: JSMarshalAs...] declaring the
    // Promise's return type corresponding to Task<T>.
    [JSImport("waitGetDate", "PromisesShim")]
    [return: JSMarshalAs<JSType.Promise<JSType.Date>>()]
    public static partial Task<DateTime> WaitGetDate();

    [JSImport("fetchCurrentUrl", "PromisesShim")]
    public static partial Task<string> FetchCurrentUrl();

    [JSImport("asyncFunction", "PromisesShim")]
    public static partial Task AsyncFunction();
```

```
    [JSImport("conditionalSuccess", "PromisesShim")]
    public static partial Task ConditionalSuccess(bool shouldSucceed);
}

public static class PromisesUsage
{
    public static async Task Run()
    {
        await JSHost.ImportAsync("PromisesShim", "/PromisesShim.js");

        Stopwatch sw = new();
        sw.Start();

        await PromisesInterop.Wait2Seconds(); // Await Promise
        Console.WriteLine($"Waited {sw.Elapsed.TotalSeconds:#.0}s.");

        sw.Restart();
        string str =
            await PromisesInterop.WaitGetString(); // Await promise (string
return)
        Console.WriteLine(
            $"Waited {sw.Elapsed.TotalSeconds:#.0}s for WaitGetString:
'{str}'");

        sw.Restart();
        // Await promise with string return.
        DateTime date = await PromisesInterop.WaitGetDate();
        Console.WriteLine(
            $"Waited {sw.Elapsed.TotalSeconds:#.0}s for WaitGetDate:
'{date}'");

        // Await a JS fetch.
        string responseText = await PromisesInterop.FetchCurrentUrl();
        Console.WriteLine($"responseText.Length: {responseText.Length}");

        sw.Restart();

        await PromisesInterop.AsyncFunction(); // Await an async JS method
        Console.WriteLine(
            $"Waited {sw.Elapsed.TotalSeconds:#.0}s for AsyncFunction.");

        try
        {
            // Handle a promise rejection. Await an async JS method.
            await PromisesInterop.ConditionalSuccess(shouldSucceed: false);
        }
        catch (JSException ex) // Catch JS exception
        {
            Console.WriteLine($"JS Exception Caught: '{ex.Message}'");
        }
    }
}
```

In `Program.Main`:

C#

```
await PromisesUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> Waited 2.0s.
> Waited .5s for WaitGetString: 'String From Resolve'
> Waited .5s for WaitGetDate: '11/24/1988 12:00:00 AM'
> responseText.Length: 582
> Waited 2.0s for AsyncFunction.
> JS Exception Caught: 'Reject: ShouldSucceed == false'

Type mapping limitations

Some type mappings requiring nested generic types in the JSMarshalAs definition aren't currently supported. For example, returning a Promise for an array such as `[return: JSMarshalAs<JSType.Promise<JSType.Array<JSType.Number>>>()]` generates a compile-time error. An appropriate workaround varies depending on the scenario, but one option is to represent the array as a JSObject reference. This may be sufficient if accessing individual elements within .NET isn't necessary and the reference can be passed to other JS methods that act on the array. Alternatively, a dedicated method can take the JSObject reference as a parameter and return the materialized array, as demonstrated by the following `UnwrapJSObjectAsIntArray` example. In this case, the JS method has no type checking, and the developer has the responsibility to ensure a JSObject wrapping the appropriate array type is passed.

JavaScript

```
export function waitGetIntArrayAsObject() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve([1, 2, 3, 4, 5]); // Return an array from the Promise
    }, 500);
  });
}

export function unwrapJSObjectAsIntArray(jsObject) {
  return jsObject;
}
```

C#

```
// Not supported, generates compile-time error.
// [JSImport("waitGetArray", "PromisesShim")]
// [return: JSMarshalAs<JSType.Promise<JSType.Array<JSType.Number>>>()]
// public static partial Task<int[]> WaitGetIntArray();

// Workaround, take the return the call and pass it to
UnwrapJSObjectAsIntArray.
// Return a JSObject reference to a JS number array.
[JSImport("waitGetIntArrayAsObject", "PromisesShim")]
[return: JSMarshalAs<JSType.Promise<JSType.Object>>()]
public static partial Task<JSObject> WaitGetIntArrayAsObject();

// Takes a JSObject reference to a JS number array, and returns the array as
a C#
// int array.
[JSImport("unwrapJSObjectAsIntArray", "PromisesShim")]
[return: JSMarshalAs<JSType.Array<JSType.Number>>()]
public static partial int[] UnwrapJSObjectAsIntArray(JSObject intArray);
//...
```

In `Program.Main`:

C#

```
JSObject arrayAsJSObject = await PromisesInterop.WaitGetIntArrayAsObject();
int[] intArray = PromisesInterop.UnwrapJSObjectAsIntArray(arrayAsJSObject);
```

Performance considerations

Marshalling of calls and the overhead of tracking objects across the interop boundary is more expensive than native .NET operations but should still demonstrate acceptable performance for a typical web app with moderate demand.

Object proxies, such as JSObject, which maintain references across the interop boundary, have additional memory overhead and impact how garbage collection affects these objects. Additionally, available memory might be exhausted without triggering garbage collection in some scenarios because memory pressure from JS and .NET isn't shared. This risk is significant when an excessive number of large objects are referenced across the interop boundary by relatively small JS objects, or vice versa where large .NET objects are referenced by JS proxies. In such cases, we recommend following deterministic disposal patterns with `using` scopes leveraging the IDisposable interface on JS objects.

The following benchmarks, which leverage earlier example code, demonstrate that interop operations are roughly an order of magnitude slower than those that remain within the .NET boundary, but the interop operations remain relatively fast. Additionally, consider that a user's device capabilities impact performance.

`JSObjectBenchmark.cs`:

C#

```
using System;
using System.Diagnostics;

public static class JSObjectBenchmark
{
    public static void Run()
    {
        Stopwatch sw = new();
        var jsObject = JSObjectInterop.CreateObject();

        sw.Start();

        for (int i = 0; i < 1000000; i++)
        {
            JSObjectInterop.IncrementAnswer(jsObject);
        }

        sw.Stop();

        Console.WriteLine(
            $"JS interop elapsed time: {sw.Elapsed.TotalSeconds:#.0000}
seconds " +
            $"at {sw.Elapsed.TotalMilliseconds / 1000000d:#.000000} ms per "
+
            "operation");

        var pocoObject =
            new PocoObject { Question = "What is the answer?", Answer = 41
};
        sw.Restart();

        for (int i = 0; i < 1000000; i++)
        {
            pocoObject.IncrementAnswer();
        }

        sw.Stop();

        Console.WriteLine($".NET elapsed time:
{sw.Elapsed.TotalSeconds:#.0000} " +
            $"seconds at {sw.Elapsed.TotalMilliseconds / 1000000d:#.000000}
ms " +
            "per operation");
```

```
        Console.WriteLine($"Begin Object Creation");

        sw.Restart();

        for (int i = 0; i < 1000000; i++)
        {
            var jsObject2 = JSObjectInterop.CreateObject();
            JSObjectInterop.IncrementAnswer(jsObject2);
        }

        sw.Stop();

        Console.WriteLine(
            $"JS interop elapsed time: {sw.Elapsed.TotalSeconds:#.0000}
seconds " +
            $"at {sw.Elapsed.TotalMilliseconds / 1000000d:#.000000} ms per "
+
            "operation");

        sw.Restart();

        for (int i = 0; i < 1000000; i++)
        {
            var pocoObject2 =
                new PocoObject { Question = "What is the answer?", Answer =
0 };
            pocoObject2.IncrementAnswer();
        }

        sw.Stop();
        Console.WriteLine(
            $".NET elapsed time: {sw.Elapsed.TotalSeconds:#.0000} seconds at
" +
            $"{sw.Elapsed.TotalMilliseconds / 1000000d:#.000000} ms per
operation");
    }

    public class PocoObject // Plain old CLR object
    {
        public string Question { get; set; }
        public int Answer { get; set; }

        public void IncrementAnswer() => Answer += 1;
    }
}
```

In `Program.Main`:

C#

```
JSObjectBenchmark.Run();
```

The preceding example displays the following output in the browser's debug console:

> JS interop elapsed time: .2536 seconds at .000254 ms per operation
> .NET elapsed time: .0210 seconds at .000021 ms per operation
> Begin Object Creation
> JS interop elapsed time: 2.1686 seconds at .002169 ms per operation
> .NET elapsed time: .1089 seconds at .000109 ms per operation

Subscribing to JS events

.NET code can subscribe to JS events and handle JS events by passing a C# Action to a JS function to act as a handler. The JS shim code handles subscribing to the event.

> ⚠ **Warning**
>
> Interacting with individual properties of the DOM via JS interop, as the guidance in this section demonstrates, is relatively slow and may lead to the creation of many proxies that create high garbage collection pressure. The following pattern isn't generally recommended. Use the following pattern for no more than a few elements. For more information, see the Performance considerations section.

A nuance of `removeEventListener` is that it requires a reference to the function previously passed to `addEventListener`. When a C# Action is passed across the interop boundary, it's wrapped in a JS proxy object. Therefore, passing the same C# Action to both `addEventListener` and `removeEventListener` results in generating two different JS proxy objects wrapping the Action. These references are different, thus `removeEventListener` isn't able to find the event listener to remove. To address this problem, the following examples wrap the C# Action in a JS function and return the reference as a JSObject from the subscribe call to pass later to the unsubscribe call. Because the C# Action is returned and passed as a JSObject, the same reference is used for both calls, and the event listener can be removed.

`EventsShim.js`:

JavaScript

```
export function subscribeEventById(elementId, eventName, listenerFunc) {
  const elementObj = document.getElementById(elementId);

  // Need to wrap the Managed C# action in JS func (only because it is being
  // returned).
  let handler = function (event) {
```

```
    listenerFunc(event.type, event.target.id); // Decompose object to
primitives
  }.bind(elementObj);

  elementObj.addEventListener(eventName, handler, false);
  // Return JSObject reference so it can be used for removeEventListener
later.
  return handler;
}

// Param listenerHandler must be the JSObject reference returned from the
prior
// SubscribeEvent call.
export function unsubscribeEventById(elementId, eventName, listenerHandler)
{
  const elementObj = document.getElementById(elementId);
  elementObj.removeEventListener(eventName, listenerHandler, false);
}

export function triggerClick(elementId) {
  const elementObj = document.getElementById(elementId);
  elementObj.click();
}

export function getElementById(elementId) {
  return document.getElementById(elementId);
}

export function subscribeEvent(elementObj, eventName, listenerFunc) {
  let handler = function (e) {
    listenerFunc(e);
  }.bind(elementObj);

  elementObj.addEventListener(eventName, handler, false);
  return handler;
}

export function unsubscribeEvent(elementObj, eventName, listenerHandler) {
  return elementObj.removeEventListener(eventName, listenerHandler, false);
}

export function subscribeEventFailure(elementObj, eventName, listenerFunc) {
  // It's not strictly required to wrap the C# action listenerFunc in a JS
  // function.
  elementObj.addEventListener(eventName, listenerFunc, false);
  // If you need to return the wrapped proxy object, you will receive an
error
  // when it tries to wrap the existing proxy in an additional proxy:
  // Error: "JSObject proxy of ManagedObject proxy is not supported."
  return listenerFunc;
}
```

`EventsInterop.cs`:

C#

```
using System;
using System.Runtime.InteropServices.JavaScript;
using System.Threading.Tasks;

public partial class EventsInterop
{
    [JSImport("subscribeEventById", "EventsShim")]
    public static partial JSObject SubscribeEventById(string elementId,
        string eventName,
        [JSMarshalAs<JSType.Function<JSType.String, JSType.String>>]
        Action<string, string> listenerFunc);

    [JSImport("unsubscribeEventById", "EventsShim")]
    public static partial void UnsubscribeEventById(string elementId,
        string eventName, JSObject listenerHandler);

    [JSImport("triggerClick", "EventsShim")]
    public static partial void TriggerClick(string elementId);

    [JSImport("getElementById", "EventsShim")]
    public static partial JSObject GetElementById(string elementId);

    [JSImport("subscribeEvent", "EventsShim")]
    public static partial JSObject SubscribeEvent(JSObject htmlElement,
        string eventName,
        [JSMarshalAs<JSType.Function<JSType.Object>>]
        Action<JSObject> listenerFunc);

    [JSImport("unsubscribeEvent", "EventsShim")]
    public static partial void UnsubscribeEvent(JSObject htmlElement,
        string eventName, JSObject listenerHandler);
}

public static class EventsUsage
{
    public static async Task Run()
    {
        await JSHost.ImportAsync("EventsShim", "/EventsShim.js");

        Action<string, string> listenerFunc = (eventName, elementId) =>
            Console.WriteLine(
                $"In C# event listener: Event {eventName} from ID
{elementId}");

        // Assumes two buttons exist on the page with ids of "btn1" and
"btn2"
        JSObject listenerHandler1 =
            EventsInterop.SubscribeEventById("btn1", "click", listenerFunc);
        JSObject listenerHandler2 =
            EventsInterop.SubscribeEventById("btn2", "click", listenerFunc);
        Console.WriteLine("Subscribed to btn1 & 2.");
        EventsInterop.TriggerClick("btn1");
```

```
        EventsInterop.TriggerClick("btn2");

        EventsInterop.UnsubscribeEventById("btn2", "click",
listenerHandler2);
        Console.WriteLine("Unsubscribed btn2.");
        EventsInterop.TriggerClick("btn1");
        EventsInterop.TriggerClick("btn2"); // Doesn't trigger because
unsubscribed
        EventsInterop.UnsubscribeEventById("btn1", "click",
listenerHandler1);
        // Pitfall: Using a different handler for unsubscribe silently
fails.
        // EventsInterop.UnsubscribeEventById("btn1", "click",
listenerHandler2);

        // With JSObject as event target and event object.
        Action<JSObject> listenerFuncForElement = (eventObj) =>
        {
            string eventType = eventObj.GetPropertyAsString("type");
            JSObject target = eventObj.GetPropertyAsJSObject("target");
            Console.WriteLine(
                $"In C# event listener: Event {eventType} from " +
                $"ID {target.GetPropertyAsString("id")}");
        };

        JSObject htmlElement = EventsInterop.GetElementById("btn1");
        JSObject listenerHandler3 = EventsInterop.SubscribeEvent(
            htmlElement, "click", listenerFuncForElement);
        Console.WriteLine("Subscribed to btn1.");
        EventsInterop.TriggerClick("btn1");
        EventsInterop.UnsubscribeEvent(htmlElement, "click",
listenerHandler3);
        Console.WriteLine("Unsubscribed btn1.");
        EventsInterop.TriggerClick("btn1");
    }
}
```

In `Program.Main`:

C#

```
await EventsUsage.Run();
```

The preceding example displays the following output in the browser's debug console:

> Subscribed to btn1 & 2.
> In C# event listener: Event click from ID btn1
> In C# event listener: Event click from ID btn2
> Unsubscribed btn2.
> In C# event listener: Event click from ID btn1

> Subscribed to btn1.
> In C# event listener: Event click from ID btn1
> Unsubscribed btn1.

JS `[JSImport]`/`[JSExport]` interop scenarios

The following articles focus on running a .NET WebAssembly module in a JS host, such as a browser:

- JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project
- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor

JavaScript `[JSImport]`/`[JSExport]` interop with a WebAssembly Browser App project

Article • 08/22/2024

This article explains how to setup a WebAssembly Browser App project to run .NET from JavaScript (JS) using JS `[JSImport]`/`[JSExport]` interop. For additional information and examples, see JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly.

For additional guidance, see the Configuring and hosting .NET WebAssembly applications guidance in the .NET Runtime (`dotnet/runtime`) GitHub repository.

Existing JS apps can use the expanded client-side WebAssembly support to reuse .NET libraries from JS or to build novel .NET-based apps and frameworks.

> **Note**
>
> This article focuses on running .NET from JS apps without any dependency on Blazor. For guidance on using `[JSImport]`/`[JSExport]` interop in Blazor WebAssembly apps, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

These approaches are appropriate when you only expect the Blazor app to run on WebAssembly (WASM). Libraries can make a runtime check to determine if the app is running on WASM by calling OperatingSystem.IsBrowser.

Prerequisites

.NET SDK (latest version)

Install the `wasm-tools` workload in an administrative command shell, which brings in the related MSBuild targets:

.NET CLI

```
dotnet workload install wasm-tools
```

The tools can also be installed via Visual Studio's installer under the **ASP.NET and web development** workload in the Visual Studio installer. Select the **.NET WebAssembly**

build tools option from the list of optional components.

Optionally, install the `wasm-experimental` workload, which adds the following experimental project templates:

- *WebAssembly Browser App* for getting started with .NET on WebAssembly in a browser app.
- *WebAssembly Console App* for getting started in a Node.js-based console app.

After installing the workload, these new templates can be selected when creating a new project. This workload isn't required if you plan to integrate JS `[JSImport]`/`[JSExport]` interop into an existing JS app.

.NET CLI

```
dotnet workload install wasm-experimental
```

The templates can also be installed from the Microsoft.NET.Runtime.WebAssembly.Templates NuGet package with the following command:

.NET CLI

```
dotnet new install Microsoft.NET.Runtime.WebAssembly.Templates
```

For more information, see the Experimental workload and project templates section.

Namespace

The JS interop API described in this article is controlled by attributes in the System.Runtime.InteropServices.JavaScript namespace.

Project configuration

To configure a project (`.csproj`) to enable JS interop:

- Set the target framework moniker (`{TARGET FRAMEWORK}` placeholder):

  XML

  ```
  <TargetFramework>{TARGET FRAMEWORK}</TargetFramework>
  ```

.NET 7 (`net7.0`) or later is supported.

- Enable the AllowUnsafeBlocks property, which permits the code generator in the Roslyn compiler to use pointers for JS interop:

  XML

  ```xml
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  ```

  > **⚠ Warning**
  >
  > The JS interop API requires enabling AllowUnsafeBlocks. Be careful when implementing your own unsafe code in .NET apps, which can introduce security and stability risks. For more information, see Unsafe code, pointer types, and function pointers.

The following is an example project file (`.csproj`) after configuration. The `{TARGET FRAMEWORK}` placeholder is the target framework:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.WebAssembly">

  <PropertyGroup>
    <TargetFramework>{TARGET FRAMEWORK}</TargetFramework>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  </PropertyGroup>

</Project>
```

JavaScript interop on WASM

APIs in the following example are imported from `dotnet.js`. These APIs enable you to set up named modules that can be imported into your C# code and call into methods exposed by your .NET code, including `Program.Main`.

> **ⓘ Important**
>
> "Import" and "export" throughout this article are defined from the perspective of .NET:
>
> - An app imports JS methods so that they can be called from .NET.

- The app exports .NET methods so that they can be called from JS.

In the following example:

- The `dotnet.js` file is used to create and start the .NET WebAssembly runtime. `dotnet.js` is generated as part of the app's build output.

  > **ⓘ Important**
  >
  > To integrate with an existing app, copy the contents of the publish output folder† to the existing app's deployment assets so that it can be served along with the rest of the app. For production deployments, publish the app with the `dotnet publish -c Release` command in a command shell and deploy the output folder's contents with the app.
  >
  > †The publish output folder is the target location of your publish profile. The default for a **Release** profile in .NET 8 or later is `bin/Release/{TARGET FRAMEWORK}/publish`, where the `{TARGET FRAMEWORK}` placeholder is the target framework (for example, `net8.0`).

- `dotnet.create()` sets up the .NET WebAssembly runtime.

- `setModuleImports` associates a name with a module of JS functions for import into .NET. The JS module contains a `window.location.href` function, which returns the current page address (URL). The name of the module can be any string (it doesn't need to be a file name), but it must match the name used with the `JSImportAttribute` (explained later in this article). The `window.location.href` function is imported into C# and called by the C# method `GetHRef`. The `GetHRef` method is shown later in this section.

- `exports.MyClass.Greeting()` calls into .NET (`MyClass.Greeting`) from JS. The `Greeting` C# method returns a string that includes the result of calling the `window.location.href` function. The `Greeting` method is shown later in this section.

- `dotnet.run()` runs `Program.Main`.

JS module:

```
JavaScript
```

```
import { dotnet } from './_framework/dotnet.js'

const { setModuleImports, getAssemblyExports, getConfig } = await dotnet
  .withDiagnosticTracing(false)
  .withApplicationArgumentsFromQuery()
  .create();

setModuleImports('main.js', {
  window: {
    location: {
      href: () => globalThis.window.location.href
    }
  }
});

const config = getConfig();
const exports = await getAssemblyExports(config.mainAssemblyName);
const text = exports.MyClass.Greeting();
console.log(text);

document.getElementById('out').innerHTML = text;
await dotnet.run();
```

To import a JS function so it can be called from C#, use the new JSImportAttribute on a matching method signature. The first parameter to the JSImportAttribute is the name of the JS function to import and the second parameter is the name of the module.

In the following example, the `window.location.href` function is called from the `main.js` module when `GetHRef` method is called:

C#

```
[JSImport("window.location.href", "main.js")]
internal static partial string GetHRef();
```

In the imported method signature, you can use .NET types for parameters and return values, which are marshalled automatically by the runtime. Use JSMarshalAsAttribute<T> to control how the imported method parameters are marshalled. For example, you might choose to marshal a `long` as System.Runtime.InteropServices.JavaScript.JSType.Number or System.Runtime.InteropServices.JavaScript.JSType.BigInt. You can pass Action/Func<TResult> callbacks as parameters, which are marshalled as callable JS functions. You can pass both JS and managed object references, and they are marshaled as proxy objects, keeping the object alive across the boundary until the proxy is garbage collected. You can also import and export asynchronous methods with a Task result, which are marshaled as JS promises. Most of the marshalled types work in both

directions, as parameters and as return values, on both imported and exported methods.

For additional type mapping information and examples, see JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly.

Functions accessible on the global namespace can be imported by using the globalThis prefix in the function name and by using the `[JSImport]` attribute without providing a module name. In the following example, console.log is prefixed with `globalThis`. The imported function is called by the C# `Log` method, which accepts a C# string message (`message`) and marshalls the C# string to a JS String for `console.log`:

C#

```
[JSImport("globalThis.console.log")]
internal static partial void Log([JSMarshalAs<JSType.String>] string
message);
```

To export a .NET method so it can be called from JS, use the JSExportAttribute.

In the following example, the `Greeting` method returns a string that includes the result of calling the `GetHRef` method. As shown earlier, the `GetHref` C# method calls into JS for the `window.location.href` function from the `main.js` module. `window.location.href` returns the current page address (URL):

C#

```
[JSExport]
internal static string Greeting()
{
    var text = $"Hello, World! Greetings from {GetHRef()}";
    Console.WriteLine(text);
    return text;
}
```

Experimental workload and project templates

To demonstrate the JS interop functionality and obtain JS interop project templates, install the `wasm-experimental` workload:

.NET CLI

```
dotnet workload install wasm-experimental
```

The `wasm-experimental` workload contains two project templates: `wasmbrowser` and `wasmconsole`. These templates are experimental at this time, which means the developer workflow for the templates is evolving. However, the .NET and JS APIs used in the templates are supported in .NET 8 and provide a foundation for using .NET on WASM from JS.

The templates can also be installed from the Microsoft.NET.Runtime.WebAssembly.Templates NuGet package with the following command:

.NET CLI

```
dotnet new install Microsoft.NET.Runtime.WebAssembly.Templates
```

Browser app

You can create a browser app with the `wasmbrowser` template from the command line, which creates a web app that demonstrates using .NET and JS together in a browser:

.NET CLI

```
dotnet new wasmbrowser
```

Alternatively in Visual Studio, you can create the app using the **WebAssembly Browser App** project template.

Build the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet build
```

Build and run the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet run
```

Alternatively, install and use the dotnet serve command:

.NET CLI

```
dotnet serve -d:bin/$(Configuration)/{TARGET FRAMEWORK}/publish
```

In the preceding example, the `{TARGET FRAMEWORK}` placeholder is the target framework moniker.

Node.js console app

You can create a console app with the `wasmconsole` template, which creates an app that runs under WASM as a Node.js or V8 console app:

.NET CLI

```
dotnet new wasmconsole
```

Alternatively in Visual Studio, you can create the app using the **WebAssembly Console App** project template.

Build the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet build
```

Build and run the app from Visual Studio or by using the .NET CLI:

.NET CLI

```
dotnet run
```

Alternatively, start any static file server from the publish output directory that contains the `main.mjs` file:

```
node bin/$(Configuration)/{TARGET FRAMEWORK}/{PATH}/main.mjs
```

In the preceding example, the `{TARGET FRAMEWORK}` placeholder is the target framework moniker, and the `{PATH}` placeholder is the path to the `main.mjs` file.

Additional resources

- JavaScript `[JSImport]`/`[JSExport]` interop in .NET WebAssembly
- JavaScript JSImport/JSExport interop with ASP.NET Core Blazor
- API documentation
  - [JSImport] attribute
  - [JSExport] attribute
- In the `dotnet/runtime` GitHub repository:
  - Configuring and hosting .NET WebAssembly applications
  - .NET WebAssembly runtime
  - dotnet.d.ts file (.NET WebAssembly runtime configuration)
- Use .NET from any JavaScript app in .NET 7

Use Grunt in ASP.NET Core

Article • 06/03/2022

Grunt is a JavaScript task runner that automates script minification, TypeScript compilation, code quality "lint" tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is fully supported in Visual Studio.

This example uses an empty ASP.NET Core project as its starting point, to show how to automate the client build process from scratch.

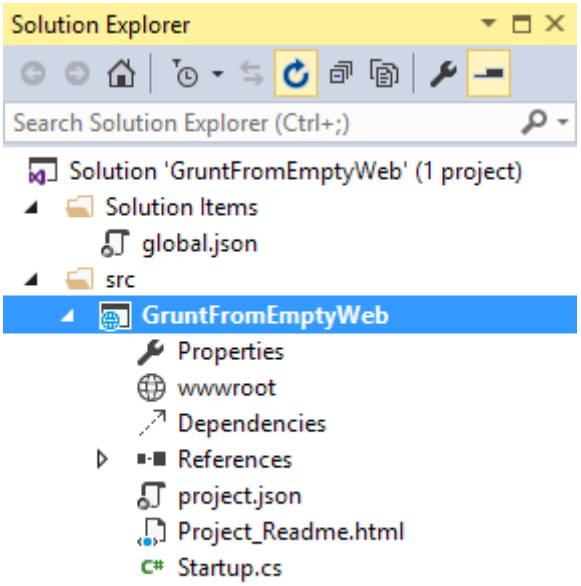
The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

- **grunt**: The Grunt task runner package.
- **grunt-contrib-clean**: A plugin that removes files or directories.
- **grunt-contrib-jshint**: A plugin that reviews JavaScript code quality.
- **grunt-contrib-concat**: A plugin that joins files into a single file.
- **grunt-contrib-uglify**: A plugin that minifies JavaScript to reduce size.
- **grunt-contrib-watch**: A plugin that watches file activity.

Preparing the application

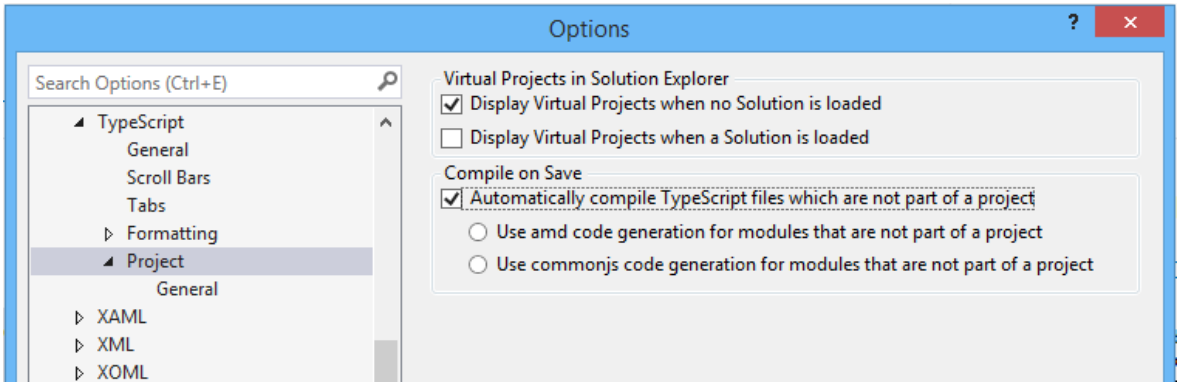
To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio settings and will be our raw material to process using Grunt.

1. In Visual Studio, create a new `ASP.NET Web Application`.
2. In the **New ASP.NET Project** dialog, select the ASP.NET Core **Empty** template and click the OK button.
3. In the Solution Explorer, review the project structure. The `\src` folder includes empty `wwwroot` and `Dependencies` nodes.



4. Add a new folder named `TypeScript` to your project directory.

5. Before adding any files, make sure that Visual Studio has the option 'compile on save' for TypeScript files checked. Navigate to **Tools** > **Options** > **Text Editor** > **Typescript** > **Project**:



6. Right-click the `TypeScript` directory and select **Add** > **New Item** from the context menu. Select the **JavaScript file** item and name the file `Tastes.ts` (note the *.ts extension). Copy the line of TypeScript code below into the file (when you save, a new `Tastes.js` file will appear with the JavaScript source).

```TypeScript
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. Add a second file to the **TypeScript** directory and name it `Food.ts`. Copy the code below into the file.

```TypeScript
```

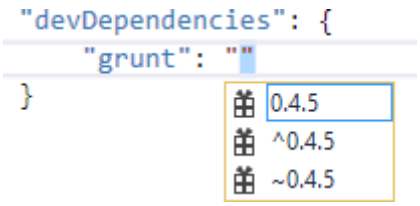
```
class Food {
  constructor(name: string, calories: number) {
    this._name = name;
    this._calories = calories;
  }

  private _name: string;
  get Name() {
    return this._name;
  }

  private _calories: number;
  get Calories() {
    return this._calories;
  }

  private _taste: Tastes;
  get Taste(): Tastes { return this._taste }
  set Taste(value: Tastes) {
    this._taste = value;
  }
}
```

Configuring NPM

Next, configure NPM to download grunt and grunt-tasks.

1. In the Solution Explorer, right-click the project and select **Add** > **New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, `package.json`, and click the **Add** button.

2. In the `package.json` file, inside the `devDependencies` object braces, enter "grunt". Select `grunt` from the Intellisense list and press the Enter key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press `Ctrl-Space` if Intellisense doesn't appear).



> ⓘ **Note**

> NPM uses semantic versioning to organize dependencies. Semantic versioning, also known as SemVer, identifies packages with the numbering scheme <major>.<minor>.<patch>. Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (0.4.5 in the example above) is considered the latest stable version of the package. The caret (^) symbol matches the most recent major version and the tilde (~) matches the most recent minor version. See the NPM semver version parser reference as a guide to the full expressivity that SemVer provides.

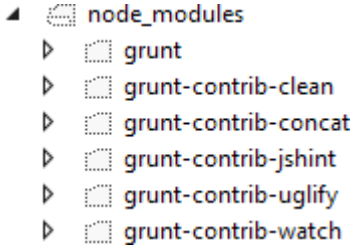
3. Add more dependencies to load grunt-contrib-* packages for *clean*, *jshint*, *concat*, *uglify*, and *watch* as shown in the example below. The versions don't need to match the example.

   JSON

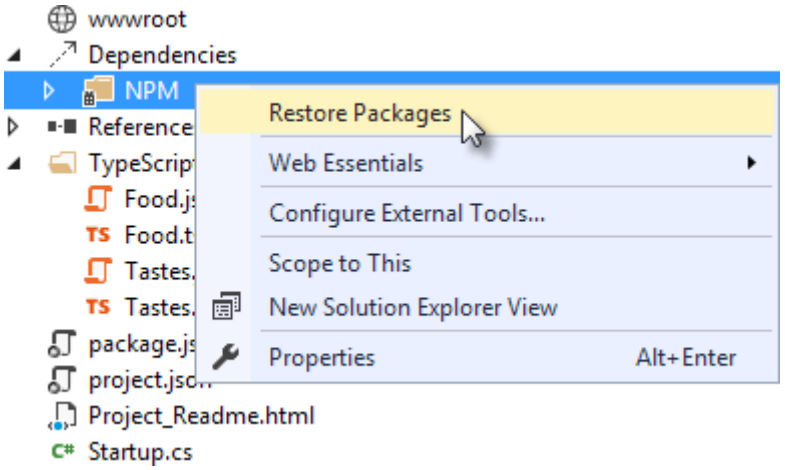
   ```
   "devDependencies": {
     "grunt": "0.4.5",
     "grunt-contrib-clean": "0.6.0",
     "grunt-contrib-jshint": "0.11.0",
     "grunt-contrib-concat": "0.5.1",
     "grunt-contrib-uglify": "0.8.0",
     "grunt-contrib-watch": "0.6.1"
   }
   ```

4. Save the `package.json` file.

The packages for each `devDependencies` item will download, along with any files that each package requires. You can find the package files in the *node_modules* directory by enabling the **Show All Files** button in **Solution Explorer**.



> **Note**
>
> If you need to, you can manually restore dependencies in **Solution Explorer** by right-clicking on `Dependencies\NPM` and selecting the **Restore Packages** menu option.



Configuring Grunt

Grunt is configured using a manifest named `Gruntfile.js` that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add** > **New Item**. Select the **JavaScript File** item template, change the name to `Gruntfile.js`, and click the **Add** button.

2. Add the following code to `Gruntfile.js`. The `initConfig` function sets options for each package, and the remainder of the module loads and register tasks.

   JavaScript

   ```
   module.exports = function (grunt) {
     grunt.initConfig({
     });
   };
   ```

3. Inside the `initConfig` function, add options for the `clean` task as shown in the example `Gruntfile.js` below. The `clean` task accepts an array of directory strings. This task removes files from *wwwroot/lib* and removes the entire */temp* directory.

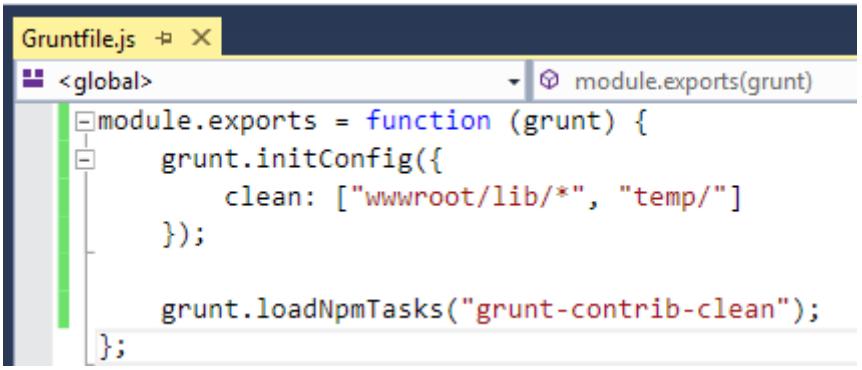
   JavaScript

   ```
   module.exports = function (grunt) {
     grunt.initConfig({
       clean: ["wwwroot/lib/*", "temp/"],
     });
   };
   ```

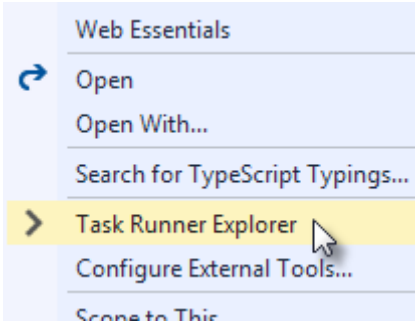
4. Below the `initConfig` function, add a call to `grunt.loadNpmTasks`. This will make the task runnable from Visual Studio.

```
JavaScript

grunt.loadNpmTasks("grunt-contrib-clean");
```

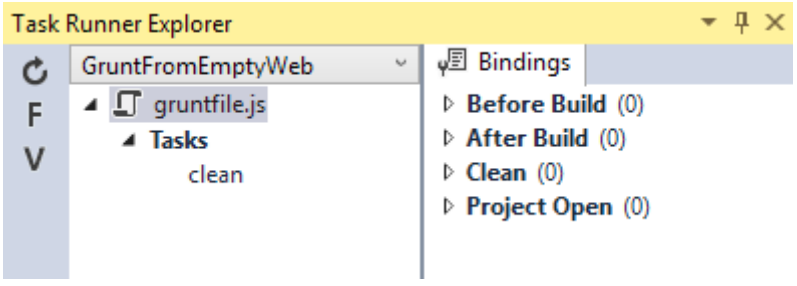
5. Save `Gruntfile.js`. The file should look something like the screenshot below.



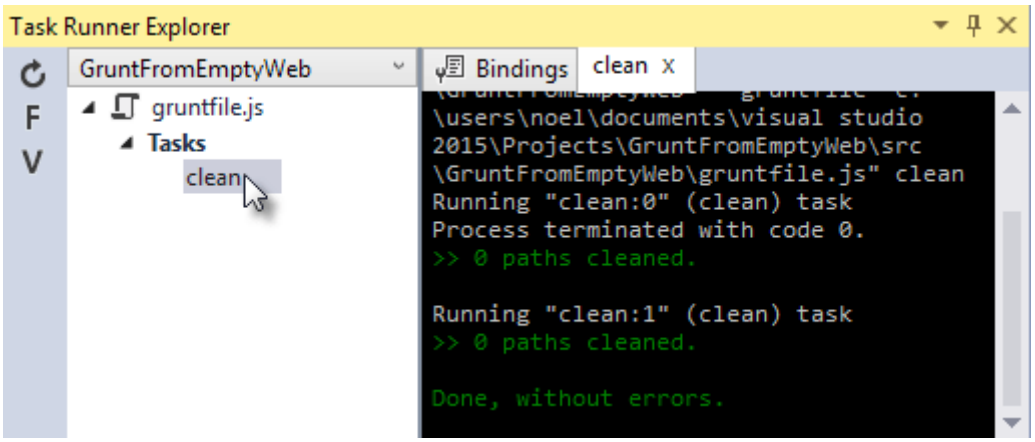
6. Right-click `Gruntfile.js` and select **Task Runner Explorer** from the context menu. The **Task Runner Explorer** window will open.



7. Verify that `clean` shows under **Tasks** in the **Task Runner Explorer**.



8. Right-click the clean task and select **Run** from the context menu. A command window displays progress of the task.



> **Note**
>
> There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

9. In the `initConfig` function, add an entry for `concat` using the code below.

   The `src` property array lists files to combine, in the order that they should be combined. The `dest` property assigns the path to the combined file that's produced.

   JavaScript

```
concat: {
  all: {
    src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
    dest: 'temp/combined.js'
  }
},
```

> **Note**
>
> The `all` property in the code above is the name of a target. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using IntelliSense or assign your own.

10. Add the `jshint` task using the code below.

    The jshint `code-quality` utility is run against every JavaScript file found in the *temp* directory.

    JavaScript

```
jshint: {
  files: ['temp/*.js'],
  options: {
    '-W069': false,
  }
},
```

> **Note**
>
> The option "-W069" is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, `Tastes["Sweet"]` instead of `Tastes.Sweet`. The option turns off the warning to allow the rest of the process to continue.

11. Add the `uglify` task using the code below.

    The task minifies the `combined.js` file found in the temp directory and creates the result file in wwwroot/lib following the standard naming convention *<file name>.min.js*.

    JavaScript

    ```
    uglify: {
     all: {
       src: ['temp/combined.js'],
       dest: 'wwwroot/lib/combined.min.js'
     }
    },
    ```

12. Under the call to `grunt.loadNpmTasks` that loads `grunt-contrib-clean`, include the same call for jshint, concat, and uglify using the code below.

    JavaScript

    ```
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    ```

13. Save `Gruntfile.js`. The file should look something like the example below.

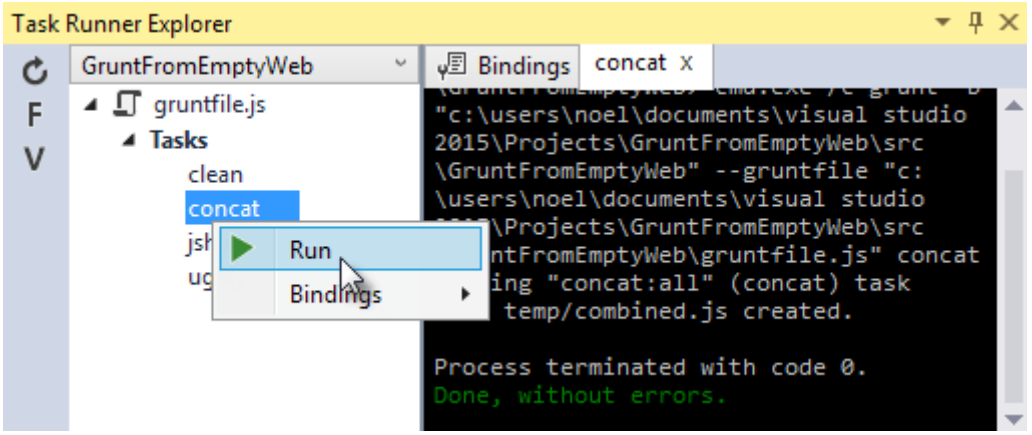
Gruntfile.js

module | exports(grunt)

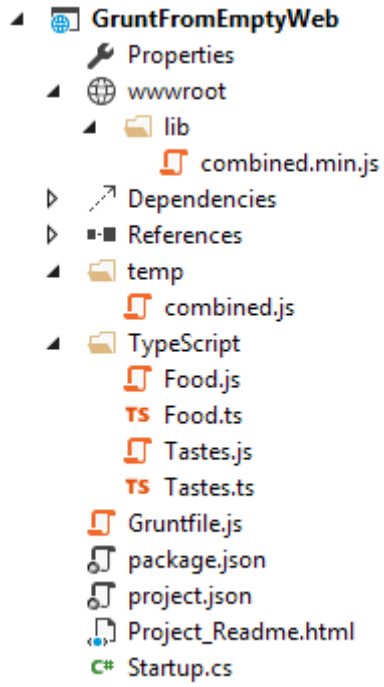
```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"],
        concat: {
            all: {
                src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
                dest: 'temp/combined.js'
            }
        },
        jshint: { files: ['temp/*.js'], options: { '-W069': false, } },
        uglify: {
            all: {
                src: ['temp/combined.js'],
                dest: 'wwwroot/lib/combined.min.js'
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-clean');
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
};
```


110 %

14. Notice that the **Task Runner Explorer** Tasks list includes `clean`, `concat`, `jshint` and `uglify` tasks. Run each task in order and observe the results in **Solution Explorer**. Each task should run without errors.



The concat task creates a new `combined.js` file and places it into the temp directory. The `jshint` task simply runs and doesn't produce output. The `uglify` task creates a new `combined.min.js` file and places it into *wwwroot/lib*. On completion, the solution should look something like the screenshot below:



> **Note**
>
> For more information on the options for each package, visit https://www.npmjs.com/ and lookup the package name in the search box on the main page. For example, you can look up the grunt-contrib-clean package to get a documentation link that explains all of its parameters.

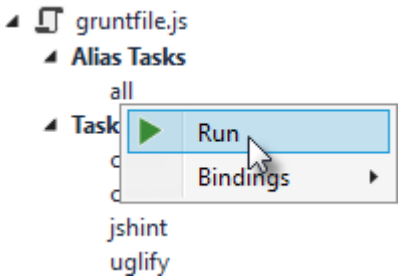
All together now

Use the Grunt `registerTask()` method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean -> concat -> jshint -> uglify, add the code below to the module. The code should be added to the same level as the loadNpmTasks() calls, outside initConfig.

JavaScript

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The `all` task will run `clean`, `concat`, `jshint` and `uglify`, in order.



Watching for changes

A `watch` task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to initConfig to watch for changes to *.js files in the TypeScript directory. If a JavaScript file is changed, `watch` will run the `all` task.

JavaScript

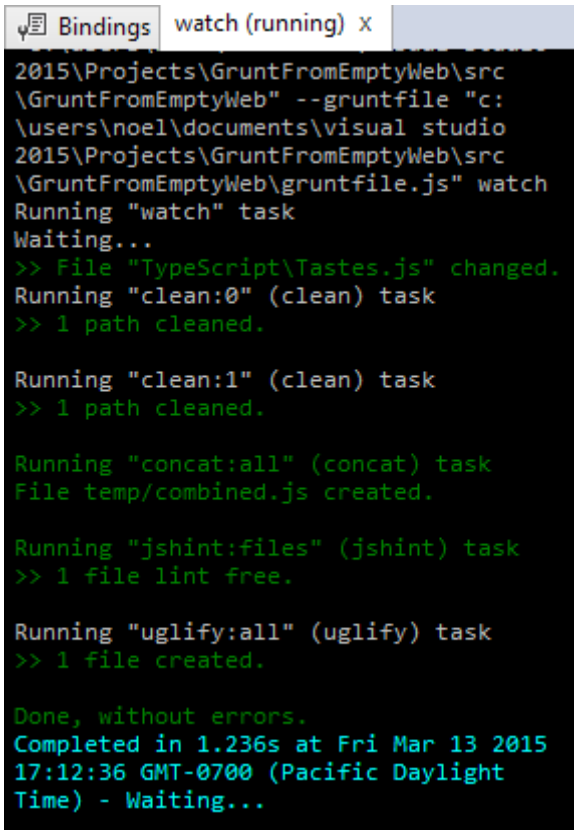
```
watch: {
  files: ["TypeScript/*.js"],
  tasks: ["all"]
}
```

Add a call to `loadNpmTasks()` to show the `watch` task in Task Runner Explorer.

JavaScript

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

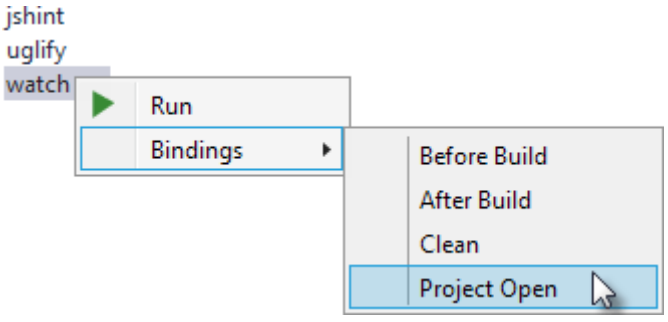
Right-click the watch task in Task Runner Explorer and select Run from the context menu. The command window that shows the watch task running will display a "Waiting..." message. Open one of the TypeScript files, add a space, and then save the file. This will trigger the watch task and trigger the other tasks to run in order. The screenshot below shows a sample run.



Binding to Visual Studio events

Unless you want to manually start your tasks every time you work in Visual Studio, bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Bind `watch` so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the watch task and select **Bindings** > **Project Open** from the context menu.



Unload and reload the project. When the project loads again, the watch task starts running automatically.

Summary

Grunt is a powerful task runner that can be used to automate most client-build tasks. Grunt leverages NPM to deliver its packages, and features tooling integration with Visual Studio. Visual Studio's Task Runner Explorer detects changes to configuration files

and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

Bundle and minify static assets in ASP.NET Core

Article • 11/17/2022

By Scott Addie and David Pine

This article explains the benefits of applying bundling and minification, including how these features can be used with ASP.NET Core web apps.

What is bundling and minification

Bundling and minification are two distinct performance optimizations you can apply in a web app. Used together, bundling and minification improve performance by reducing the number of server requests and reducing the size of the requested static assets.

Bundling and minification primarily improve the first page request load time. Once a web page has been requested, the browser caches the static assets (JavaScript, CSS, and images). So, bundling and minification don't improve performance when requesting the same page, or pages, on the same site requesting the same assets. If the expires header isn't set correctly on the assets and if bundling and minification aren't used, the browser's freshness heuristics mark the assets stale after a few days. Additionally, the browser requires a validation request for each asset. In this case, bundling and minification provide a performance improvement even after the first page request.

Bundling

Bundling combines multiple files into a single file. Bundling reduces the number of server requests that are necessary to render a web asset, such as a web page. You can create any number of individual bundles specifically for CSS, JavaScript, etc. Fewer files mean fewer HTTP requests from the browser to the server or from the service providing your application. This results in improved first page load performance.

Minification

Minification removes unnecessary characters from code without altering functionality. The result is a significant size reduction in requested assets (such as CSS, images, and JavaScript files). Common side effects of minification include shortening variable names to one character and removing comments and unnecessary whitespace.

Consider the following JavaScript function:

```JavaScript
AddAltToImg = function (imageTagAndImageID, imageContext) {
  ///<signature>
  ///<summary> Adds an alt tab to the image
  // </summary>
  //<param name="imgElement" type="String">The image selector.</param>
  //<param name="ContextForImage" type="String">The image context.</param>
  ///</signature>
  var imageElement = $(imageTagAndImageID, imageContext);
  imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

Minification reduces the function to the following:

```JavaScript
AddAltToImg=function(t,a){var
r=$(t,a);r.attr("alt",r.attr("id").replace(/ID/,""))};
```

In addition to removing the comments and unnecessary whitespace, the following parameter and variable names were renamed as follows:

Expand table

Original	Renamed
`imageTagAndImageID`	`t`
`imageContext`	`a`
`imageElement`	`r`

Impact of bundling and minification

The following table outlines differences between individually loading assets and using bundling and minification for a typical web app.

Expand table

Action	Without B/M	With B/M	Reduction
File Requests	18	7	61%

Action	Without B/M	With B/M	Reduction
Bytes Transferred (KB)	265	156	41%
Load Time (ms)	2360	885	63%

The load time improved, but this example ran locally. Greater performance gains are realized when using bundling and minification with assets transferred over a network.

The test app used to generate the figures in the preceding table demonstrates typical improvements that might not apply to a given app. We recommend testing an app to determine if bundling and minification yields an improved load time.

Choose a bundling and minification strategy

ASP.NET Core is compatible with WebOptimizer, an open-source bundling and minification solution. For set up instructions and sample projects, see WebOptimizer. ASP.NET Core doesn't provide a native bundling and minification solution.

Third-party tools, such as Gulp and Webpack, provide workflow automation for bundling and minification, as well as linting and image optimization. By using bundling and minification, the minified files are created prior to the app's deployment. Bundling and minifying before deployment provides the advantage of reduced server load. However, it's important to recognize that bundling and minification increases build complexity and only works with static files.

Environment-based bundling and minification

As a best practice, the bundled and minified files of your app should be used in a production environment. During development, the original files make for easier debugging of the app.

Specify which files to include in your pages by using the Environment Tag Helper in your views. The Environment Tag Helper only renders its contents when running in specific environments.

The following `environment` tag renders the unprocessed CSS files when running in the `Development` environment:

CSHTML

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
```

```
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
```

The following `environment` tag renders the bundled and minified CSS files when running in an environment other than `Development`. For example, running in `Production` or `Staging` triggers the rendering of these stylesheets:

CSHTML

```
<environment exclude="Development">
    <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only" asp-fallback-test-
property="position" asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css" asp-append-
version="true" />
</environment>
```

Additional resources

- Use multiple environments
- Tag Helpers

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Browser link in ASP.NET Core

Article • 06/07/2024

By Nicolò Carandini and Tom Dykstra

Browser link is a Visual Studio feature. It creates a communication channel between the development environment and one or more web browsers. Use browser link to:

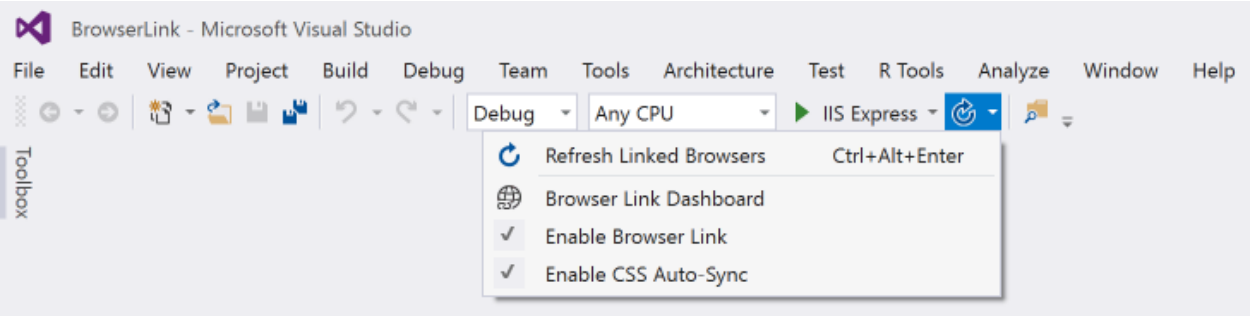
- Refresh your web app in several browsers at once.
- Test across multiple browsers with specific settings such as screen sizes.
- Select UI elements in browsers in real-time, see what markup and source it's correlated to in Visual Studio.
- Conduct real-time browser test automation.

Runtime compilation vs. hot reload

Use browser Link with runtime compilation or hot reload to see the effect of run-time changes in Razor (`.cshtml`) files. We recommend hot reload.

How to use browser link

When you have an ASP.NET Core project open, Visual Studio shows the browser link toolbar control next to the **Debug Type** toolbar control:

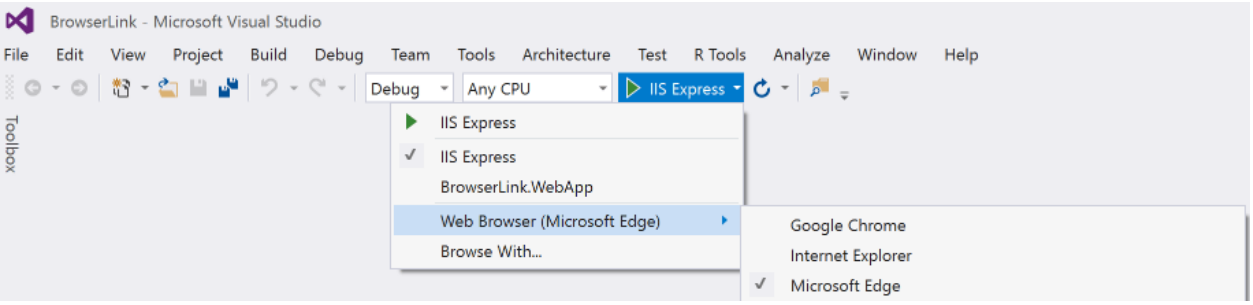


From the browser link toolbar control, you can:

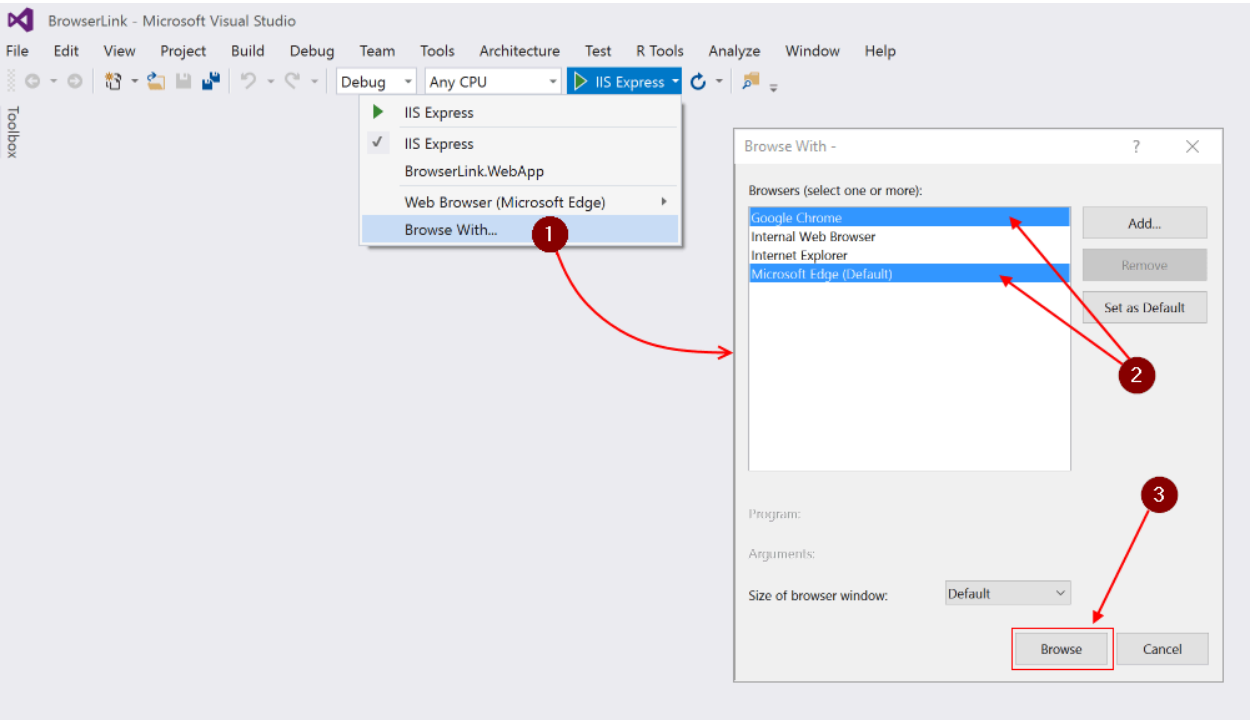
- Refresh the web app in several browsers at once.
- Open the **Browser Link Dashboard**.
- Enable or disable **Browser Link**.
- Enable or disable **CSS Hot Reload**.

Refresh the web app in several browsers at once

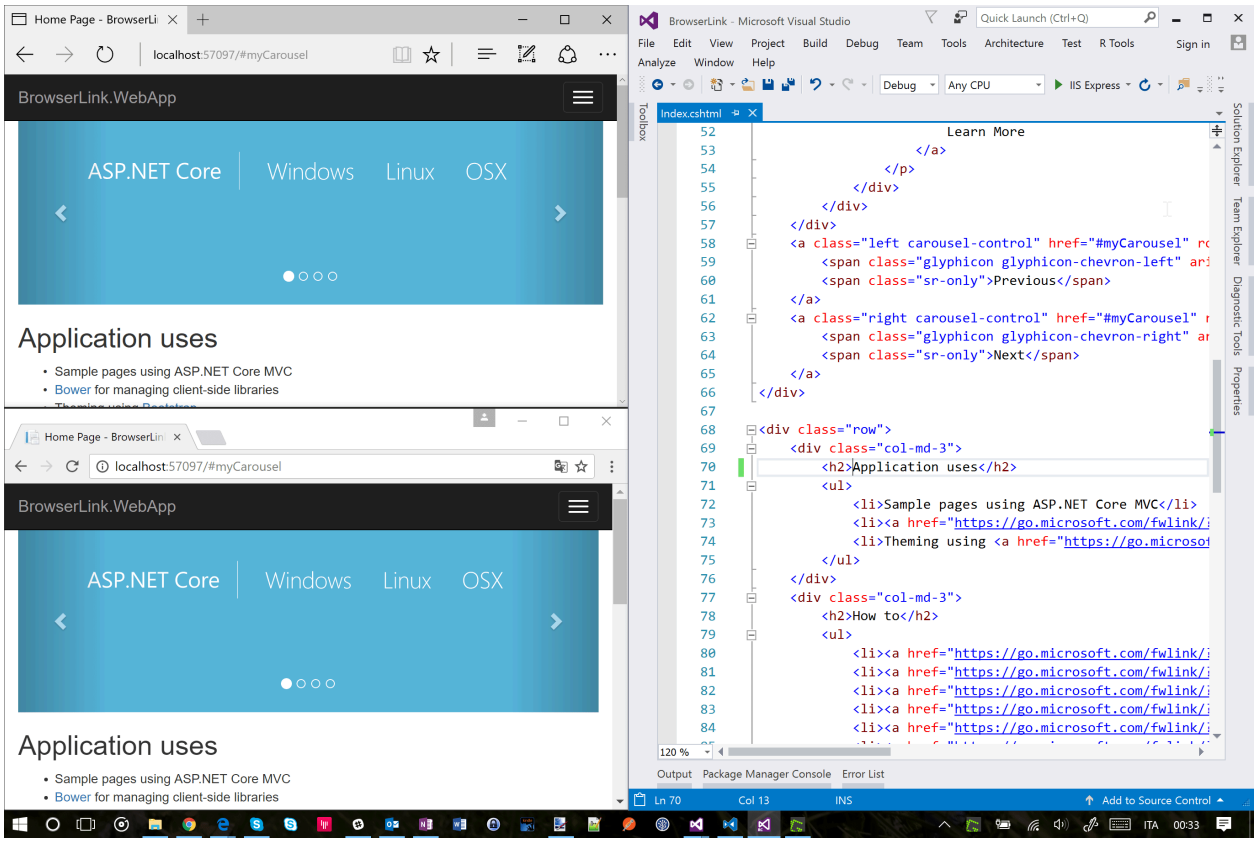
To choose a single web browser to launch when starting the project, use the drop-down menu in the **Debug Target** toolbar control:



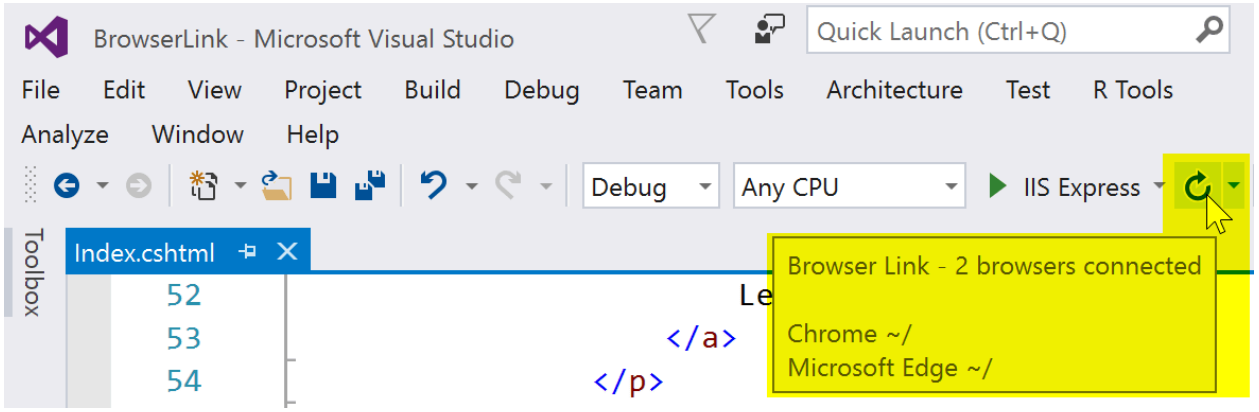
To open multiple browsers at once, choose **Browse with...** from the same drop-down. Hold down the `Ctrl` key to select the browsers you want, and then click **Browse**:



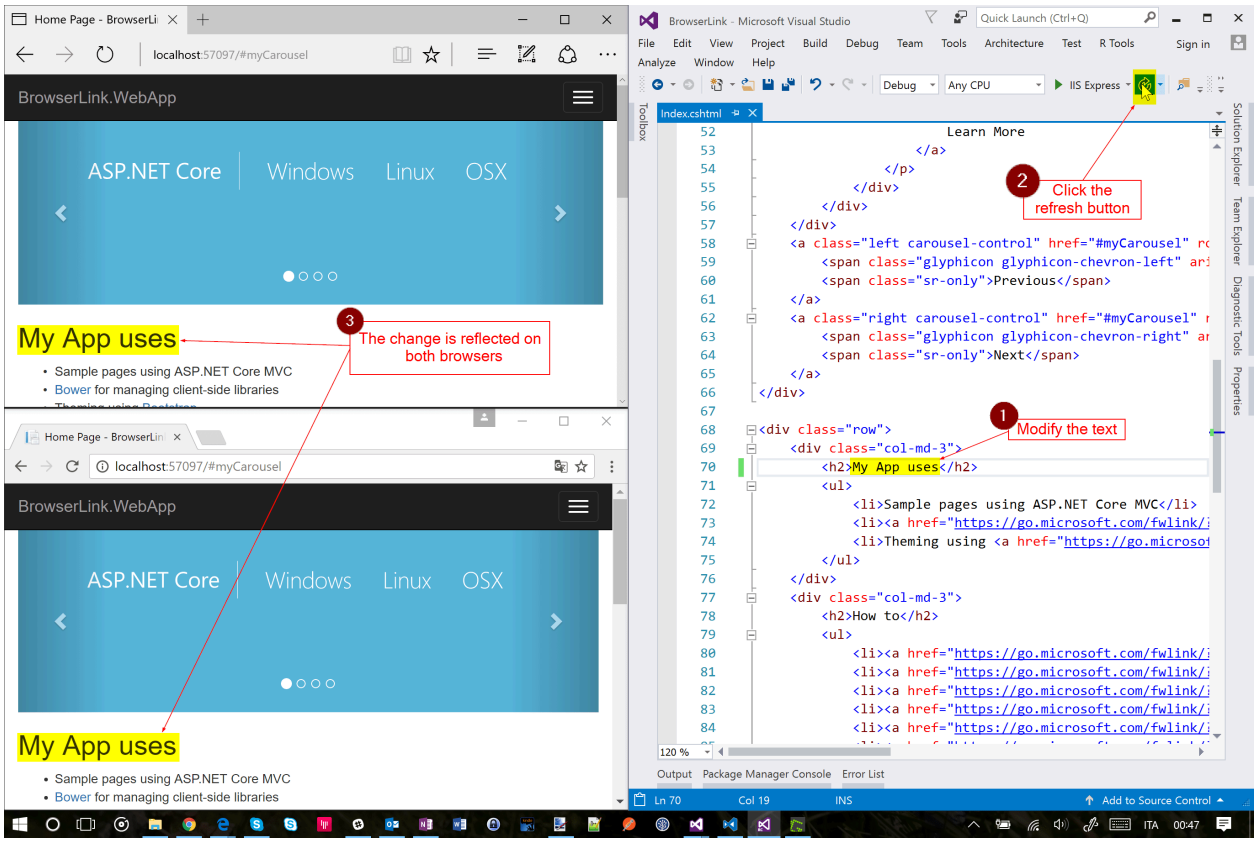
The following screenshot shows Visual Studio with the Index view open and two open browsers:



Hover over the browser link toolbar control to see the browsers that are connected to the project:



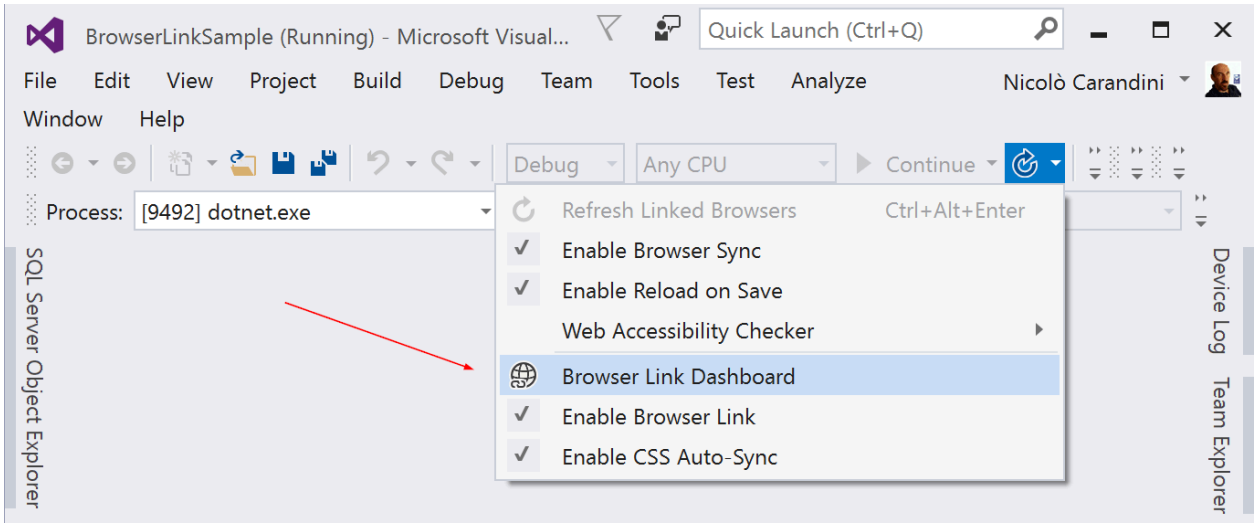
Change the Index view, and all connected browsers are updated when you click the browser link refresh button:



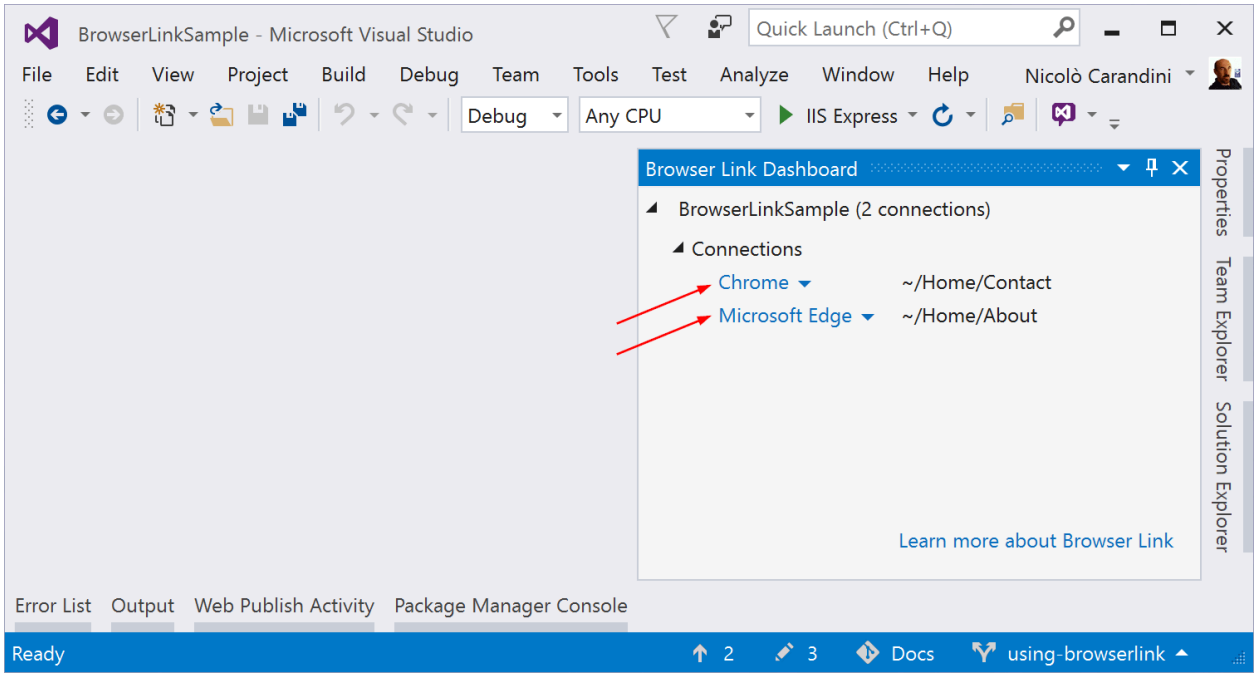
browser link also works with browsers that you launch from outside Visual Studio and navigate to the app URL.

The browser link dashboard

Open the **browser link dashboard** window from the browser link drop down menu to manage the connection with open browsers:



The connected browsers are shown with the path to the page that each browser is showing:



You can also click on an individual browser name to refresh only that browser.

Enable or disable browser link

When you re-enable browser link after disabling it, you must refresh the browsers to reconnect them.

Enable or disable CSS hot reload

When CSS hot reload is enabled, connected browsers are automatically refreshed when you make any change to CSS files.

How it works

browser link uses SignalR to create a communication channel between Visual Studio and the browser. When browser link is enabled, Visual Studio acts as a SignalR server that multiple clients (browsers) can connect to. browser link also registers a middleware component in the ASP.NET Core request pipeline. This component injects special `<script>` references into every page request from the server. You can see the script references by selecting **View source** in the browser and scrolling to the end of the `<body>` tag content:

HTML

```
    <!-- Visual Studio browser link -->
    <script type="application/json" id="__browserLink_initializationData">
```

```
{"requestId":"a717d5a07c1741949a7cefd6fa2bad08","requestMappingFromServer":f
alse}
    </script>
    <script type="text/javascript"
src="http://localhost:54139/b6e36e429d034f578ebccd6a79bf19bf/browserLink"
async="async"></script>
    <!-- End browser link -->
</body>
```

Your source files aren't modified. The middleware component injects the script references dynamically.

Because the browser-side code is all JavaScript, it works on all browsers that SignalR supports without requiring a browser plug-in.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Session and state management in ASP.NET Core

Article • 05/29/2024

By Rick Anderson, Kirk Larkin, and Diana LaRose

HTTP is a stateless protocol. By default, HTTP requests are independent messages that don't retain user values. This article describes several approaches to preserve user data between requests.

For Blazor state management guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor state management.

State management

State can be stored using several approaches. Each approach is described later in this article.

Expand table

Storage approach	Storage mechanism
Cookies	HTTP cookies. May include data stored using server-side app code.
Session state	HTTP cookies and server-side app code
TempData	HTTP cookies or session state
Query strings	HTTP query strings
Hidden fields	HTTP form fields
HttpContext.Items	Server-side app code
Cache	Server-side app code

SignalR/Blazor Server and HTTP context-based state management

SignalR apps shouldn't use session state and other state management approaches that rely upon a stable HTTP context to store information. SignalR apps can store per-connection state in Context.Items in the hub. For more information and alternative state

management approaches for Blazor Server apps, see ASP.NET Core Blazor state management.

Cookies

Cookies store data across requests. Because cookies are sent with every request, their size should be kept to a minimum. Ideally, only an identifier should be stored in a cookie with the data stored by the app. Most browsers restrict cookie size to 4096 bytes. Only a limited number of cookies are available for each domain.

Because cookies are subject to tampering, they must be validated by the app. Cookies can be deleted by users and expire on clients. However, cookies are generally the most durable form of data persistence on the client.

Cookies are often used for personalization, where content is customized for a known user. The user is only identified and not authenticated in most cases. The cookie can store the user's name, account name, or unique user ID such as a GUID. The cookie can be used to access the user's personalized settings, such as their preferred website background color.

See the European Union General Data Protection Regulations (GDPR) when issuing cookies and dealing with privacy concerns. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

Session state

Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app. Session state uses a store maintained by the app to persist data across requests from a client. The session data is backed by a cache and considered ephemeral data. The site should continue to function without the session data. Critical application data should be stored in the user database and cached in session only as a performance optimization.

Session isn't supported in SignalR apps because a SignalR Hub may execute independent of an HTTP context. For example, this can occur when a long polling request is held open by a hub beyond the lifetime of the request's HTTP context.

ASP.NET Core maintains session state by providing a cookie to the client that contains a session ID. The cookie session ID:

- Is sent to the app with each request.
- Is used by the app to fetch the session data.

Session state exhibits the following behaviors:

- The session cookie is specific to the browser. Sessions aren't shared across browsers.
- Session cookies are deleted when the browser session ends.
- If a cookie is received for an expired session, a new session is created that uses the same session cookie.
- Empty sessions aren't retained. The session must have at least one value set to persist the session across requests. When a session isn't retained, a new session ID is generated for each new request.
- The app retains a session for a limited time after the last request. The app either sets the session timeout or uses the default value of 20 minutes. Session state is ideal for storing user data:
  - That's specific to a particular session.
  - Where the data doesn't require permanent storage across sessions.
- Session data is deleted either when the ISession.Clear implementation is called or when the session expires.
- There's no default mechanism to inform app code that a client browser has been closed or when the session cookie is deleted or expired on the client.
- Session state cookies aren't marked essential by default. Session state isn't functional unless tracking is permitted by the site visitor. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.
- **Note**: There is no replacement for the cookieless session feature from the ASP.NET Framework because it's considered insecure and can lead to session fixation attacks.

> ⚠ **Warning**
>
> Don't store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows. A session might not be restricted to a single user. The next user might continue to browse the app with the same session cookie.

The in-memory cache provider stores session data in the memory of the server where the app resides. In a server farm scenario:

- Use *sticky sessions* to tie each session to a specific app instance on an individual server. Azure App Service uses Application Request Routing (ARR) to enforce sticky sessions by default. However, sticky sessions can affect scalability and complicate web app updates. A better approach is to use a Redis or SQL Server

distributed cache, which doesn't require sticky sessions. For more information, see Distributed caching in ASP.NET Core.

- The session cookie is encrypted via IDataProtector. Data Protection must be properly configured to read session cookies on each machine. For more information, see ASP.NET Core Data Protection Overview and Key storage providers.

Configure session state

Middleware for managing session state is included in the framework. To enable the session middleware, `Program.cs` must contain:

- Any of the IDistributedCache memory caches. The `IDistributedCache` implementation is used as a backing store for session. For more information, see Distributed caching in ASP.NET Core.
- A call to AddSession
- A call to UseSession

The following code shows how to set up the in-memory session provider with a default in-memory implementation of `IDistributedCache`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseRouting();

app.UseAuthorization();

app.UseSession();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

The preceding code sets a short timeout to simplify testing.

The order of middleware is important. Call `UseSession` after `UseRouting` and before `MapRazorPages` and `MapDefaultControllerRoute` . See Middleware Ordering.

HttpContext.Session is available after session state is configured.

`HttpContext.Session` can't be accessed before `UseSession` has been called.

A new session with a new session cookie can't be created after the app has begun writing to the response stream. The exception is recorded in the web server log and not displayed in the browser.

Load session state asynchronously

The default session provider in ASP.NET Core loads session records from the underlying IDistributedCache backing store asynchronously only if the ISession.LoadAsync method is explicitly called before the TryGetValue, Set, or Remove methods. If `LoadAsync` isn't called first, the underlying session record is loaded synchronously, which can incur a performance penalty at scale.

To have apps enforce this pattern, wrap the DistributedSessionStore and DistributedSession implementations with versions that throw an exception if the `LoadAsync` method isn't called before `TryGetValue`, `Set`, or `Remove`. Register the wrapped versions in the services container.

Session options

To override session defaults, use SessionOptions.

Expand table

Option	Description
Cookie	Determines the settings used to create the cookie. Name defaults to SessionDefaults.CookieName (`.AspNetCore.Session`). Path defaults to SessionDefaults.CookiePath (`/`). SameSite defaults to SameSiteMode.Lax (`1`). HttpOnly defaults to `true`. IsEssential defaults to `false`.
IdleTimeout	The `IdleTimeout` indicates how long the session can be idle before its contents are abandoned. Each session access resets the timeout. This setting only applies to the content of the session, not the cookie. The default is 20 minutes.
IOTimeout	The maximum amount of time allowed to load a session from the store or to commit it back to the store. This setting may only apply to asynchronous operations. This timeout can be disabled using InfiniteTimeSpan. The default is 1 minute.

Session uses a cookie to track and identify requests from a single browser. By default, this cookie is named `.AspNetCore.Session`, and it uses a path of `/`. Because the cookie default doesn't specify a domain, it isn't made available to the client-side script on the page (because HttpOnly defaults to `true`).

To override cookie session defaults, use SessionOptions:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>
{
    options.Cookie.Name = ".AdventureWorks.Session";
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.IsEssential = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

```
app.UseAuthorization();

app.UseSession();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

The app uses the IdleTimeout property to determine how long a session can be idle before its contents in the server's cache are abandoned. This property is independent of the cookie expiration. Each request that passes through the Session Middleware resets the timeout.

Session state is *non-locking*. If two requests simultaneously attempt to modify the contents of a session, the last request overrides the first. `Session` is implemented as a *coherent session*, which means that all the contents are stored together. When two requests seek to modify different session values, the last request may override session changes made by the first.

Set and get Session values

Session state is accessed from a Razor Pages PageModel class or MVC Controller class with HttpContext.Session. This property is an ISession implementation.

The `ISession` implementation provides several extension methods to set and retrieve integer and string values. The extension methods are in the Microsoft.AspNetCore.Http namespace.

`ISession` extension methods:

- Get(ISession, String)
- GetInt32(ISession, String)
- GetString(ISession, String)
- SetInt32(ISession, String, Int32)
- SetString(ISession, String, String)

The following example retrieves the session value for the `IndexModel.SessionKeyName` key (`_Name` in the sample app) in a Razor Pages page:

C#

```
@page
@using Microsoft.AspNetCore.Http
```

```
@model IndexModel

...

Name: @HttpContext.Session.GetString(IndexModel.SessionKeyName)
```

The following example shows how to set and get an integer and a string:

C#

```
public class IndexModel : PageModel
{
    public const string SessionKeyName = "_Name";
    public const string SessionKeyAge = "_Age";

    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        if
(string.IsNullOrEmpty(HttpContext.Session.GetString(SessionKeyName)))
        {
            HttpContext.Session.SetString(SessionKeyName, "The Doctor");
            HttpContext.Session.SetInt32(SessionKeyAge, 73);
        }
        var name = HttpContext.Session.GetString(SessionKeyName);
        var age = HttpContext.Session.GetInt32(SessionKeyAge).ToString();

        _logger.LogInformation("Session Name: {Name}", name);
        _logger.LogInformation("Session Age: {Age}", age);
    }
}
```

The following markup displays the session values on a Razor Page:

CSHTML

```
@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<div class="text-center">
<p><b>Name:</b> @HttpContext.Session.GetString("_Name");<b>Age:
```

```
</b> @HttpContext.Session.GetInt32("_Age").ToString()</p>
</div>
```

All session data must be serialized to enable a distributed cache scenario, even when using the in-memory cache. String and integer serializers are provided by the extension methods of ISession. Complex types must be serialized by the user using another mechanism, such as JSON.

Use the following sample code to serialize objects:

C#

```
public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T value)
    {
        session.SetString(key, JsonSerializer.Serialize(value));
    }

    public static T? Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return value == null ? default : JsonSerializer.Deserialize<T>
(value);
    }
}
```

The following example shows how to set and get a serializable object with the `SessionExtensions` class:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Web.Extensions;    // SessionExtensions

namespace SessionSample.Pages
{
    public class Index6Model : PageModel
    {
        const string SessionKeyTime = "_Time";
        public string? SessionInfo_SessionTime { get; private set; }
        private readonly ILogger<Index6Model> _logger;

        public Index6Model(ILogger<Index6Model> logger)
        {
            _logger = logger;
        }
```

```
        public void OnGet()
        {
            var currentTime = DateTime.Now;

            // Requires SessionExtensions from sample.
            if (HttpContext.Session.Get<DateTime>(SessionKeyTime) ==
default)
            {
                HttpContext.Session.Set<DateTime>(SessionKeyTime,
currentTime);
            }
            _logger.LogInformation("Current Time: {Time}", currentTime);
            _logger.LogInformation("Session Time: {Time}",
                                   HttpContext.Session.Get<DateTime>
(SessionKeyTime));

        }
    }
}
```

> ⚠ **Warning**
>
> Storing a live object in the session should be used with caution, as there are many problems that can occur with serialized objects. For more information, see Sessions should be allowed to store objects (dotnet/aspnetcore #18159).

TempData

ASP.NET Core exposes the Razor Pages TempData or Controller TempData. This property stores data until it's read in another request. The Keep(String) and Peek(string) methods can be used to examine the data without deletion at the end of the request. Keep marks all items in the dictionary for retention. `TempData` is:

- Useful for redirection when data is required for more than a single request.
- Implemented by `TempData` providers using either cookies or session state.

TempData samples

Consider the following page that creates a customer:

C#

```
public class CreateModel : PageModel
{
    private readonly RazorPagesContactsContext _context;
```

```
    public CreateModel(RazorPagesContactsContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";

        return RedirectToPage("./IndexPeek");
    }
}
```

The following page displays `TempData["Message"]`:

CSHTML

```
@page
@model IndexModel

<h1>Peek Contacts</h1>

@{
    if (TempData.Peek("Message") != null)
    {
        <h3>Message: @TempData.Peek("Message")</h3>
    }
}

@*Content removed for brevity.*@
```

In the preceding markup, at the end of the request, `TempData["Message"]` is **not** deleted because `Peek` is used. Refreshing the page displays the contents of `TempData["Message"]`.

The following markup is similar to the preceding code, but uses `Keep` to preserve the data at the end of the request:

CSHTML

```
@page
@model IndexModel

<h1>Contacts Keep</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
    TempData.Keep("Message");
}

@*Content removed for brevity.*@
```

Navigating between the *IndexPeek* and *IndexKeep* pages won't delete `TempData["Message"]`.

The following code displays `TempData["Message"]`, but at the end of the request, `TempData["Message"]` is deleted:

CSHTML

```
@page
@model IndexModel

<h1>Index no Keep or Peek</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
}

@*Content removed for brevity.*@
```

TempData providers

The cookie-based TempData provider is used by default to store TempData in cookies.

The cookie data is encrypted using IDataProtector, encoded with Base64UrlTextEncoder, then chunked. The maximum cookie size is less than 4096 bytes due to encryption and chunking. The cookie data isn't compressed because compressing encrypted data can lead to security problems such as the CRIME and BREACH attacks. For more information on the cookie-based TempData provider, see CookieTempDataProvider.

Choose a TempData provider

Choosing a TempData provider involves several considerations, such as:

- Does the app already use session state? If so, using the session state TempData provider has no additional cost to the app beyond the size of the data.
- Does the app use TempData only sparingly for relatively small amounts of data, up to 500 bytes? If so, the cookie TempData provider adds a small cost to each request that carries TempData. If not, the session state TempData provider can be beneficial to avoid round-tripping a large amount of data in each request until the TempData is consumed.
- Does the app run in a server farm on multiple servers? If so, there's no additional configuration required to use the cookie TempData provider outside of Data Protection. For more information, see ASP.NET Core Data Protection Overview and Key storage providers.

Most web clients such as web browsers enforce limits on the maximum size of each cookie and the total number of cookies. When using the cookie TempData provider, verify the app won't exceed these limits. Consider the total size of the data. Account for increases in cookie size due to encryption and chunking.

Configure the TempData provider

The cookie-based TempData provider is enabled by default.

To enable the session-based TempData provider, use the AddSessionStateTempDataProvider extension method. Only one call to `AddSessionStateTempDataProvider` is required:

```
C#

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddRazorPages()
                    .AddSessionStateTempDataProvider();
builder.Services.AddControllersWithViews()
                    .AddSessionStateTempDataProvider();

builder.Services.AddSession();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseSession();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Query strings

A limited amount of data can be passed from one request to another by adding it to the new request's query string. This is useful for capturing state in a persistent manner that allows links with embedded state to be shared through email or social networks. Because URL query strings are public, never use query strings for sensitive data.

In addition to unintended sharing, including data in query strings can expose the app to Cross-Site Request Forgery (CSRF) attacks. Any preserved session state must protect against CSRF attacks. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Hidden fields

Data can be saved in hidden form fields and posted back on the next request. This is common in multi-page forms. Because the client can potentially tamper with the data, the app must always revalidate the data stored in hidden fields.

`HttpContext.Items`

The HttpContext.Items collection is used to store data while processing a single request. The collection's contents are discarded after a request is processed. The `Items` collection is often used to allow components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters.

In the following example, middleware adds `isVerified` to the `Items` collection:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

ILogger logger = app.Logger;

app.Use(async (context, next) =>
{
    // context.Items["isVerified"] is null
    logger.LogInformation($"Before setting: Verified:
{context.Items["isVerified"]}");
    context.Items["isVerified"] = true;
    await next.Invoke();
});

app.Use(async (context, next) =>
{
    // context.Items["isVerified"] is true
    logger.LogInformation($"Next: Verified: {context.Items["isVerified"]}");
    await next.Invoke();
});

app.MapGet("/", async context =>
{
    await context.Response.WriteAsync($"Verified:
{context.Items["isVerified"]}");
});

app.Run();
```

For middleware that's only used in a single app, it's unlikely that using a fixed `string` key would cause a key collision. However, to avoid the possibility of a key collision altogether, an `object` can be used as an item key. This approach is particularly useful for middleware that's shared between apps and also has the advantage of eliminating the use of key strings in the code. The following example shows how to use an `object` key defined in a middleware class:

C#

```
public class HttpContextItemsMiddleware
{
    private readonly RequestDelegate _next;
    public static readonly object HttpContextItemsMiddlewareKey = new();

    public HttpContextItemsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";

        await _next(httpContext);
    }
}

public static class HttpContextItemsMiddlewareExtensions
{
    public static IApplicationBuilder
        UseHttpContextItemsMiddleware(this IApplicationBuilder app)
    {
        return app.UseMiddleware<HttpContextItemsMiddleware>();
    }
}
```

Other code can access the value stored in `HttpContext.Items` using the key exposed by the middleware class:

C#

```
public class Index2Model : PageModel
{
    private readonly ILogger<Index2Model> _logger;

    public Index2Model(ILogger<Index2Model> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        HttpContext.Items

.TryGetValue(HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey,
                out var middlewareSetValue);

        _logger.LogInformation("Middleware value {MV}",
            middlewareSetValue?.ToString() ?? "Middleware value not set!");
```

```
    }
}
```

Cache

Caching is an efficient way to store and retrieve data. The app can control the lifetime of cached items. For more information, see Response caching in ASP.NET Core.

Cached data isn't associated with a specific request, user, or session. **Do not cache user-specific data that may be retrieved by other user requests.**

To cache application wide data, see Cache in-memory in ASP.NET Core.

Checking session state

ISession.IsAvailable is intended to check for transient failures. Calling `IsAvailable` before the session middleware runs throws an `InvalidOperationException`.

Libraries that need to test session availability can use `HttpContext.Features.Get<ISessionFeature>()?.Session != null`.

Common errors

- "Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

  This is typically caused by failing to configure at least one `IDistributedCache` implementation. For more information, see Distributed caching in ASP.NET Core and Cache in-memory in ASP.NET Core.

If the session middleware fails to persist a session:

- The middleware logs the exception and the request continues normally.
- This leads to unpredictable behavior.

The session middleware can fail to persist a session if the backing store isn't available. For example, a user stores a shopping cart in session. The user adds an item to the cart but the commit fails. The app doesn't know about the failure so it reports to the user that the item was added to their cart, which isn't true.

The recommended approach to check for errors is to call `await feature.Session.CommitAsync` when the app is done writing to the session. CommitAsync throws an exception if the backing store is unavailable. If `CommitAsync` fails, the app can process the exception. LoadAsync throws under the same conditions when the data store is unavailable.

Additional resources

View or download sample code (how to download)

Host ASP.NET Core in a web farm

Layout in ASP.NET Core

Article • 06/03/2022

By Steve Smith and Dave Brock

Pages and views frequently share visual and programmatic elements. This article demonstrates how to:

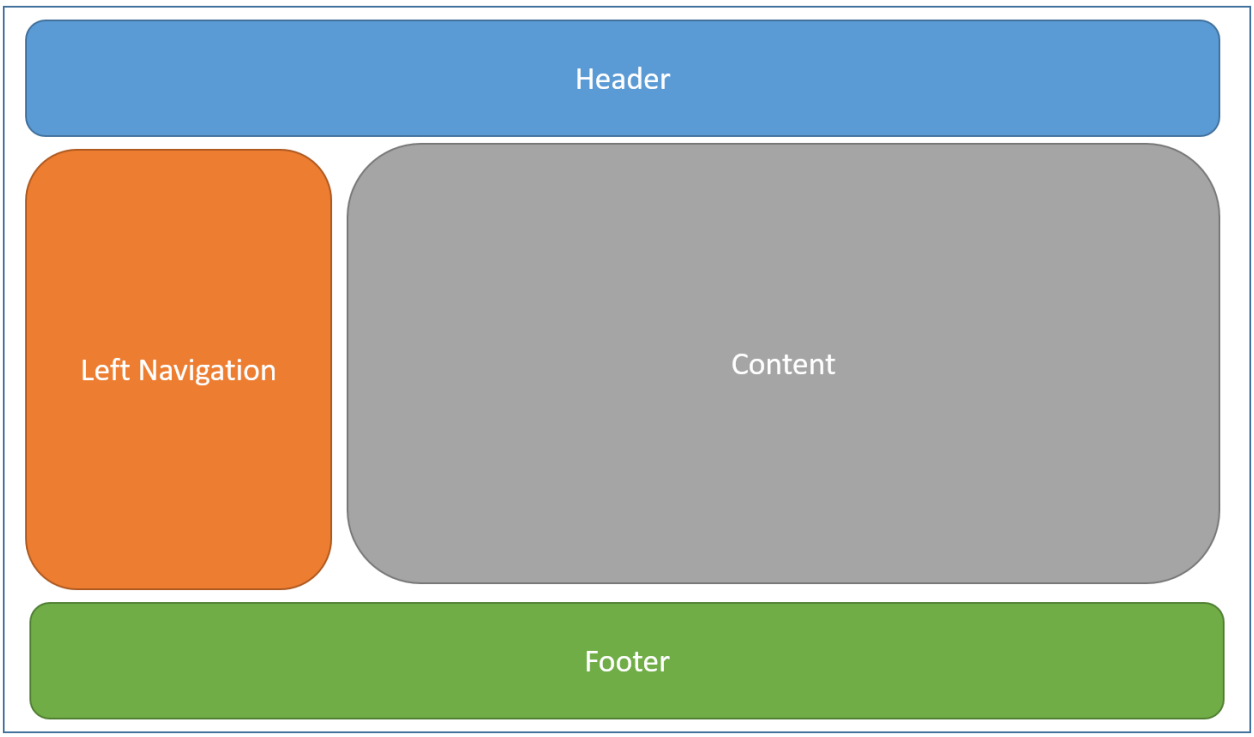
- Use common layouts.
- Share directives.
- Run common code before rendering pages or views.

This document discusses layouts for the two different approaches to ASP.NET Core MVC: Razor Pages and controllers with views. For this topic, the differences are minimal:

- Razor Pages are in the *Pages* folder.
- Controllers with views uses a *Views* folder for views.

What is a Layout

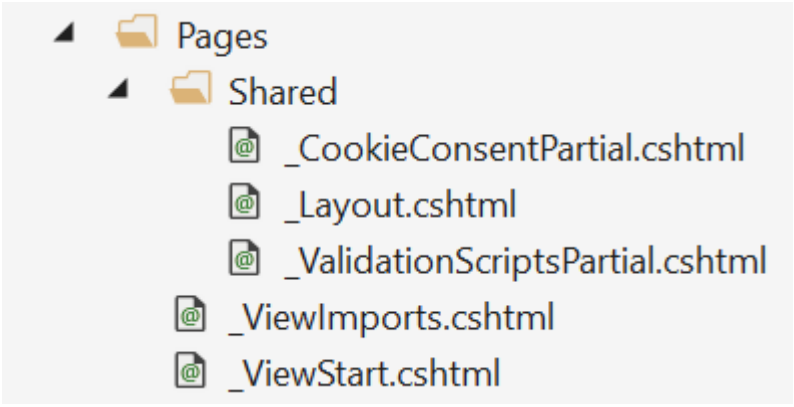
Most web apps have a common layout that provides the user with a consistent experience as they navigate from page to page. The layout typically includes common user interface elements such as the app header, navigation or menu elements, and footer.



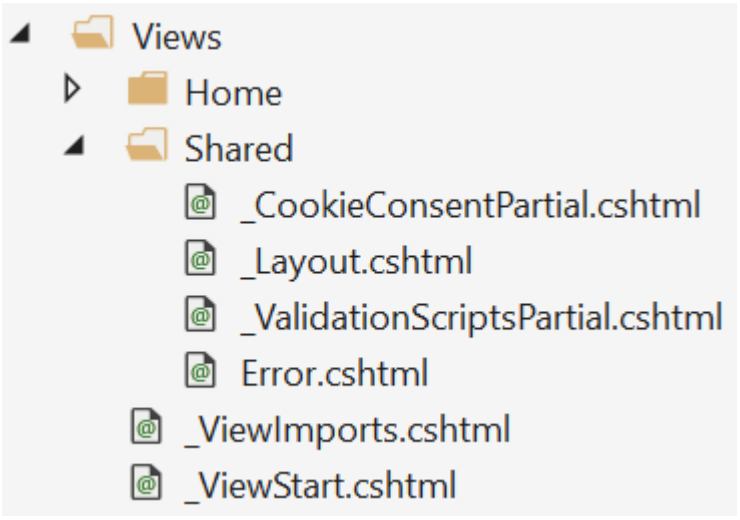
Common HTML structures such as scripts and stylesheets are also frequently used by many pages within an app. All of these shared elements may be defined in a *layout* file, which can then be referenced by any view used within the app. Layouts reduce duplicate code in views.

By convention, the default layout for an ASP.NET Core app is named `_Layout.cshtml`. The layout files for new ASP.NET Core projects created with the templates are:

- Razor Pages: `Pages/Shared/_Layout.cshtml`



- Controller with views: `Views/Shared/_Layout.cshtml`



The layout defines a top level template for views in the app. Apps don't require a layout. Apps can define more than one layout, with different views specifying different layouts.

The following code shows the layout file for a template created project with a controller and views:

CSHTML

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebApplication1</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
/>
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-
property="position" asp-fallback-test-value="absolute" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-
version="true" />
    </environment>
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-page="/Index" class="navbar-
brand">WebApplication1</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-page="/Index">Home</a></li>
                    <li><a asp-page="/About">About</a></li>
                    <li><a asp-page="/Contact">Contact</a></li>
                </ul>
            </div>
        </div>
    </nav>

    <partial name="_CookieConsentPartial" />

    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; 2018 - WebApplication1</p>
        </footer>
    </div>

    <environment include="Development">
        <script src="~/lib/jquery/dist/jquery.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
```

```
        <script src="~/js/site.js" asp-append-version="true"></script>
    </environment>
    <environment exclude="Development">
        <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-
3.3.1.min.js"
                asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
                asp-fallback-test="window.jQuery"
                crossorigin="anonymous"
                integrity="sha384-
tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">
        </script>
        <script
src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
                asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
                asp-fallback-test="window.jQuery && window.jQuery.fn &&
window.jQuery.fn.modal"
                crossorigin="anonymous"
                integrity="sha384-
Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa">
        </script>
        <script src="~/js/site.min.js" asp-append-version="true"></script>
    </environment>

    @RenderSection("Scripts", required: false)
</body>
</html>
```

Specifying a Layout

Razor views have a `Layout` property. Individual views specify a layout by setting this property:

CSHTML

```
@{
    Layout = "_Layout";
}
```

The layout specified can use a full path (for example, `/Pages/Shared/_Layout.cshtml` or `/Views/Shared/_Layout.cshtml`) or a partial name (example: `_Layout`). When a partial name is provided, the Razor view engine searches for the layout file using its standard discovery process. The folder where the handler method (or controller) exists is searched first, followed by the *Shared* folder. This discovery process is identical to the process used to discover partial views.

By default, every layout must call `RenderBody`. Wherever the call to `RenderBody` is placed, the contents of the view will be rendered.

Sections

A layout can optionally reference one or more *sections*, by calling `RenderSection`. Sections provide a way to organize where certain page elements should be placed. Each call to `RenderSection` can specify whether that section is required or optional:

HTML

```
<script type="text/javascript" src="~/scripts/global.js"></script>

@RenderSection("Scripts", required: false)
```

If a required section isn't found, an exception is thrown. Individual views specify the content to be rendered within a section using the `@section` Razor syntax. If a page or view defines a section, it must be rendered (or an error will occur).

An example `@section` definition in Razor Pages view:

HTML

```
@section Scripts {
     <script type="text/javascript" src="~/scripts/main.js"></script>
}
```

In the preceding code, `scripts/main.js` is added to the `scripts` section on a page or view. Other pages or views in the same app might not require this script and wouldn't define a scripts section.

The following markup uses the Partial Tag Helper to render `_ValidationScriptsPartial.cshtml`:

HTML

```
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

The preceding markup was generated by scaffolding Identity.

Sections defined in a page or view are available only in its immediate layout page. They cannot be referenced from partials, view components, or other parts of the view system.

Ignoring sections

By default, the body and all sections in a content page must all be rendered by the layout page. The Razor view engine enforces this by tracking whether the body and each section have been rendered.

To instruct the view engine to ignore the body or sections, call the `IgnoreBody` and `IgnoreSection` methods.

The body and every section in a Razor page must be either rendered or ignored.

Importing Shared Directives

Views and pages can use Razor directives to import namespaces and use dependency injection. Directives shared by many views may be specified in a common `_ViewImports.cshtml` file. The `_ViewImports` file supports the following directives:

- `@addTagHelper`
- `@removeTagHelper`
- `@tagHelperPrefix`
- `@using`
- `@model`
- `@inherits`
- `@inject`
- `@namespace`

The file doesn't support other Razor features, such as functions and section definitions.

A sample `_ViewImports.cshtml` file:

CSHTML

```
@using WebApplication1
@using WebApplication1.Models
@using WebApplication1.Models.AccountViewModels
@using WebApplication1.Models.ManageViewModels
@using Microsoft.AspNetCore.Identity
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The `_ViewImports.cshtml` file for an ASP.NET Core MVC app is typically placed in the *Pages* (or *Views*) folder. A `_ViewImports.cshtml` file can be placed within any folder, in which case it will only be applied to pages or views within that folder and its subfolders. `_ViewImports` files are processed starting at the root level and then for each folder

leading up to the location of the page or view itself. `_ViewImports` settings specified at the root level may be overridden at the folder level.

For example, suppose:

- The root level `_ViewImports.cshtml` file includes `@model MyModel1` and `@addTagHelper *, MyTagHelper1`.
- A subfolder `_ViewImports.cshtml` file includes `@model MyModel2` and `@addTagHelper *, MyTagHelper2`.

Pages and views in the subfolder will have access to both Tag Helpers and the `MyModel2` model.

If multiple `_ViewImports.cshtml` files are found in the file hierarchy, the combined behavior of the directives are:

- `@addTagHelper`, `@removeTagHelper`: all run, in order
- `@tagHelperPrefix`: the closest one to the view overrides any others
- `@model`: the closest one to the view overrides any others
- `@inherits`: the closest one to the view overrides any others
- `@using`: all are included; duplicates are ignored
- `@inject`: for each property, the closest one to the view overrides any others with the same property name

Running Code Before Each View

Code that needs to run before each view or page should be placed in the `_ViewStart.cshtml` file. By convention, the `_ViewStart.cshtml` file is located in the *Pages* (or *Views*) folder. The statements listed in `_ViewStart.cshtml` are run before every full view (not layouts, and not partial views). Like ViewImports.cshtml, `_ViewStart.cshtml` is hierarchical. If a `_ViewStart.cshtml` file is defined in the view or pages folder, it will be run after the one defined in the root of the *Pages* (or *Views*) folder (if any).

A sample `_ViewStart.cshtml` file:

CSHTML

```
@{
    Layout = "_Layout";
}
```

The file above specifies that all views will use the `_Layout.cshtml` layout.

`_ViewStart.cshtml` and `_ViewImports.cshtml` are **not** typically placed in the */Pages/Shared* (or */Views/Shared*) folder. The app-level versions of these files should be placed directly in the */Pages* (or */Views*) folder.

Razor syntax reference for ASP.NET Core

Article • 11/14/2023

By Rick Anderson, Taylor Mullen, and Dan Vicarel

Razor is a markup syntax for embedding .NET based code into webpages. The Razor syntax consists of Razor markup, C#, and HTML. Files containing Razor generally have a `.cshtml` file extension. Razor is also found in Razor component files (`.razor`). Razor syntax is similar to the templating engines of various JavaScript single-page application (SPA) frameworks, such as Angular, React, VueJs, and Svelte. For more information see, The features described in this article are obsolete as of ASP.NET Core 3.0.

Introduction to ASP.NET Web Programming Using the Razor Syntax provides many samples of programming with Razor syntax. Although the topic was written for ASP.NET rather than ASP.NET Core, most of the samples apply to ASP.NET Core.

Rendering HTML

The default Razor language is HTML. Rendering HTML from Razor markup is no different than rendering HTML from an HTML file. HTML markup in `.cshtml` Razor files is rendered by the server unchanged.

Razor syntax

Razor supports C# and uses the `@` symbol to transition from HTML to C#. Razor evaluates C# expressions and renders them in the HTML output.

When an `@` symbol is followed by a Razor reserved keyword, it transitions into Razor-specific markup. Otherwise, it transitions into plain HTML.

To escape an `@` symbol in Razor markup, use a second `@` symbol:

CSHTML

```
<p>@@Username</p>
```

The code is rendered in HTML with a single `@` symbol:

HTML

```
<p>@Username</p>
```

HTML attributes and content containing email addresses don't treat the `@` symbol as a transition character. The email addresses in the following example are untouched by Razor parsing:

CSHTML

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

Scalable Vector Graphics (SVG)

SVG foreignObject elements are supported:

HTML

```
@{
    string message = "foreignObject example with Scalable Vector Graphics
(SVG)";
}

<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
    <rect x="0" y="0" rx="10" ry="10" width="200" height="200"
stroke="black"
        fill="none" />
    <foreignObject x="20" y="20" width="160" height="160">
        <p>@message</p>
    </foreignObject>
</svg>
```

Implicit Razor expressions

Implicit Razor expressions start with `@` followed by C# code:

CSHTML

```
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2016)</p>
```

With the exception of the C# `await` keyword, implicit expressions must not contain spaces. If the C# statement has a clear ending, spaces can be intermingled:

CSHTML

```
<p>@await DoSomething("hello", "world")</p>
```

Implicit expressions **cannot** contain C# generics, as the characters inside the brackets (`<>`) are interpreted as an HTML tag. The following code is **not** valid:

CSHTML

```
<p>@GenericMethod<int>()</p>
```

The preceding code generates a compiler error similar to one of the following:

- The "int" element wasn't closed. All elements must be either self-closing or have a matching end tag.
- Cannot convert method group 'GenericMethod' to non-delegate type 'object'. Did you intend to invoke the method?`

Generic method calls must be wrapped in an explicit Razor expression or a Razor code block.

Explicit Razor expressions

Explicit Razor expressions consist of an `@` symbol with balanced parenthesis. To render last week's time, the following Razor markup is used:

CSHTML

```
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

Any content within the `@()` parenthesis is evaluated and rendered to the output.

Implicit expressions, described in the previous section, generally can't contain spaces. In the following code, one week isn't subtracted from the current time:

CSHTML

```
<p>Last week: @DateTime.Now - TimeSpan.FromDays(7)</p>
```

The code renders the following HTML:

HTML

```
<p>Last week: 7/7/2016 4:39:52 PM - TimeSpan.FromDays(7)</p>
```

Explicit expressions can be used to concatenate text with an expression result:

CSHTML

```
@{
    var joe = new Person("Joe", 33);
}

<p>Age@(joe.Age)</p>
```

Without the explicit expression, `<p>Age@joe.Age</p>` is treated as an email address, and `<p>Age@joe.Age</p>` is rendered. When written as an explicit expression, `<p>Age33</p>` is rendered.

Explicit expressions can be used to render output from generic methods in `.cshtml` files. The following markup shows how to correct the error shown earlier caused by the brackets of a C# generic. The code is written as an explicit expression:

CSHTML

```
<p>@(GenericMethod<int>())</p>
```

Expression encoding

C# expressions that evaluate to a string are HTML encoded. C# expressions that evaluate to `IHtmlContent` are rendered directly through `IHtmlContent.WriteTo`. C# expressions that don't evaluate to `IHtmlContent` are converted to a string by `ToString` and encoded before they're rendered.

CSHTML

```
@("<span>Hello World</span>")
```

The preceding code renders the following HTML:

HTML

```
&lt;span&gt;Hello World&lt;/span&gt;
```

The HTML is shown in the browser as plain text:

Hello World

`HtmlHelper.Raw` output isn't encoded but rendered as HTML markup.

> ⚠ **Warning**
>
> Using `HtmlHelper.Raw` on unsanitized user input is a security risk. User input might contain malicious JavaScript or other exploits. Sanitizing user input is difficult. Avoid using `HtmlHelper.Raw` with user input.

CSHTML

```
@Html.Raw("<span>Hello World</span>")
```

The code renders the following HTML:

HTML

```
<span>Hello World</span>
```

Razor code blocks

Razor code blocks start with `@` and are enclosed by `{}`. Unlike expressions, C# code inside code blocks isn't rendered. Code blocks and expressions in a view share the same scope and are defined in order:

CSHTML

```
@{
    var quote = "The future depends on what you do today. - Mahatma Gandhi";
}

<p>@quote</p>

@{
    quote = "Hate cannot drive out hate, only love can do that. - Martin
Luther King, Jr.";
}

<p>@quote</p>
```

The code renders the following HTML:

HTML

```
<p>The future depends on what you do today. - Mahatma Gandhi</p>
<p>Hate cannot drive out hate, only love can do that. - Martin Luther King,
Jr.</p>
```

In code blocks, declare local functions with markup to serve as templating methods:

CSHTML

```
@{
    void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }

    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}
```

The code renders the following HTML:

HTML

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

Implicit transitions

The default language in a code block is C#, but the Razor Page can transition back to HTML:

CSHTML

```
@{
    var inCSharp = true;
    <p>Now in HTML, was in C# @inCSharp</p>
}
```

Explicit delimited transition

To define a subsection of a code block that should render HTML, surround the characters for rendering with the Razor `<text>` tag:

CSHTML

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <text>Name: @person.Name</text>
}
```

Use this approach to render HTML that isn't surrounded by an HTML tag. Without an HTML or Razor tag, a Razor runtime error occurs.

The `<text>` tag is useful to control whitespace when rendering content:

- Only the content between the `<text>` tag is rendered.
- No whitespace before or after the `<text>` tag appears in the HTML output.

Explicit line transition

To render the rest of an entire line as HTML inside a code block, use `@:` syntax:

CSHTML

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    @:Name: @person.Name
}
```

Without the `@:` in the code, a Razor runtime error is generated.

Extra `@` characters in a Razor file can cause compiler errors at statements later in the block. These extra `@` compiler errors:

- Can be difficult to understand because the actual error occurs before the reported error.
- Is common after combining multiple implicit and explicit expressions into a single code block.

Conditional attribute rendering

Razor automatically omits attributes that aren't needed. If the value passed in is `null` or `false`, the attribute isn't rendered.

For example, consider the following razor:

```
```

CSHTML

```
<div class="@false">False</div>
<div class="@null">Null</div>
<div class="@("")">Empty</div>
<div class="@("false")">False String</div>
<div class="@("active")">String</div>
<input type="checkbox" checked="@true" name="true" />
<input type="checkbox" checked="@false" name="false" />
<input type="checkbox" checked="@null" name="null" />
```

The preceding Razor markup generates the following HTML:

HTML

```
<div>False</div>
<div>Null</div>
<div class="">Empty</div>
<div class="false">False String</div>
<div class="active">String</div>
<input type="checkbox" checked="checked" name="true">
<input type="checkbox" name="false">
<input type="checkbox" name="null">
```

Control structures

Control structures are an extension of code blocks. All aspects of code blocks (transitioning to markup, inline C#) also apply to the following structures:

Conditionals `@if, else if, else, and @switch`

`@if` controls when code runs:

CSHTML

```
@if (value % 2 == 0)
{
    <p>The value was even.</p>
}
```

`else` and `else if` don't require the `@` symbol:

CSHTML

```
@if (value % 2 == 0)
{
```

```
    <p>The value was even.</p>
}
else if (value >= 1337)
{
    <p>The value is large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```

The following markup shows how to use a switch statement:

CSHTML

```
@switch (value)
{
    case 1:
        <p>The value is 1!</p>
        break;
    case 1337:
        <p>Your number is 1337!</p>
        break;
    default:
        <p>Your number wasn't 1 or 1337.</p>
        break;
}
```

Looping `@for, @foreach, @while, and @do while`

Templated HTML can be rendered with looping control statements. To render a list of people:

CSHTML

```
@{
    var people = new Person[]
    {
          new Person("Weston", 33),
          new Person("Johnathon", 41),
          ...
    };
}
```

The following looping statements are supported:

`@for`

```
```

CSHTML

```
@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@foreach`

CSHTML

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

`@while`

CSHTML

```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
}
```

`@do while`

CSHTML

```
@{ var i = 0; }
@do
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>

    i++;
} while (i < people.Length);
```

Compound `@using`

In C#, a `using` statement is used to ensure an object is disposed. In Razor, the same mechanism is used to create HTML Helpers that contain additional content. In the following code, HTML Helpers render a `<form>` tag with the `@using` statement:

CSHTML

```
@using (Html.BeginForm())
{
    <div>
        <label>Email: <input type="email" id="Email" value=""></label>
        <button>Register</button>
    </div>
}
```

`@try, catch, finally`

Exception handling is similar to C#:

CSHTML

```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```

`@lock`

Razor has the capability to protect critical sections with lock statements:

CSHTML

```
@lock (SomeLock)
{
    // Do critical section work
}
```

Comments

Razor supports C# and HTML comments:

```cshtml
@{
    /* C# comment */
    // Another C# comment
}
<!-- HTML comment -->
```

The code renders the following HTML:

```html
<!-- HTML comment -->
```

Razor comments are removed by the server before the webpage is rendered. Razor uses `@* *@` to delimit comments. The following code is commented out, so the server doesn't render any markup:

```cshtml
@*
    @{
        /* C# comment */
        // Another C# comment
    }
    <!-- HTML comment -->
*@
```

Directives

Razor directives are represented by implicit expressions with reserved keywords following the `@` symbol. A directive typically changes the way a view is compiled or functions.

Understanding how Razor generates code for a view makes it easier to understand how directives work.

```cshtml
@{
    var quote = "Getting old ain't for wimps! - Anonymous";
```

```
}

<div>Quote of the Day: @quote</div>
```

The code generates a class similar to the following:

C#

```
public class _Views_Something_cshtml : RazorPage<dynamic>
{
    public override async Task ExecuteAsync()
    {
        var output = "Getting old ain't for wimps! - Anonymous";

        WriteLiteral("/r/n<div>Quote of the Day: ");
        Write(output);
        WriteLiteral("</div>");
    }
}
```

Later in this article, the section Inspect the Razor C# class generated for a view explains how to view this generated class.

`@attribute`

The `@attribute` directive adds the given attribute to the class of the generated page or view. The following example adds the `[Authorize]` attribute:

CSHTML

```
@attribute [Authorize]
```

The `@attribute` directive can also be used to supply a constant-based route template in a Razor component. In the following example, the `@page` directive in a component is replaced with the `@attribute` directive and the constant-based route template in `Constants.CounterRoute`, which is set elsewhere in the app to "`/counter`":

diff

```
- @page "/counter"
+ @attribute [Route(Constants.CounterRoute)]
```

`@code`

This scenario only applies to Razor components (`.razor`).

The `@code` block enables a Razor component to add C# members (fields, properties, and methods) to a component:

razor

```
@code {
    // C# members (fields, properties, and methods)
}
```

For Razor components, `@code` is an alias of @functions and recommended over `@functions`. More than one `@code` block is permissible.

`@functions`

The `@functions` directive enables adding C# members (fields, properties, and methods) to the generated class:

CSHTML

```
@functions {
    // C# members (fields, properties, and methods)
}
```

In Razor components, use `@code` over `@functions` to add C# members.

For example:

CSHTML

```
@functions {
    public string GetHello()
    {
        return "Hello";
    }
}

<div>From method: @GetHello()</div>
```

The code generates the following HTML markup:

HTML

```
<div>From method: Hello</div>
```

The following code is the generated Razor C# class:

C#

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Razor;

public class _Views_Home_Test_cshtml : RazorPage<dynamic>
{
    // Functions placed between here
    public string GetHello()
    {
        return "Hello";
    }
    // And here.
#pragma warning disable 1998
    public override async Task ExecuteAsync()
    {
        WriteLiteral("\r\n<div>From method: ");
        Write(GetHello());
        WriteLiteral("</div>\r\n");
    }
#pragma warning restore 1998
```

`@functions` methods serve as templating methods when they have markup:

CSHTML

```
@{
    RenderName("Mahatma Gandhi");
    RenderName("Martin Luther King, Jr.");
}

@functions {
    private void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }
}
```

The code renders the following HTML:

HTML

```
<p>Name: <strong>Mahatma Gandhi</strong></p>
<p>Name: <strong>Martin Luther King, Jr.</strong></p>
```

`@implements`

The `@implements` directive implements an interface for the generated class.

The following example implements System.IDisposable so that the Dispose method can be called:

CSHTML

```
@implements IDisposable

<h1>Example</h1>

@functions {
    private bool _isDisposed;

    ...

    public void Dispose() => _isDisposed = true;
}
```

`@inherits`

The `@inherits` directive provides full control of the class the view inherits:

CSHTML

```
@inherits TypeNameOfClassToInheritFrom
```

The following code is a custom Razor page type:

C#

```
using Microsoft.AspNetCore.Mvc.Razor;

public abstract class CustomRazorPage<TModel> : RazorPage<TModel>
{
    public string CustomText { get; } =
        "Gardyloo! - A Scottish warning yelled from a window before dumping"
+
        "a slop bucket on the street below.";
}
```

The `CustomText` is displayed in a view:

CSHTML

```
@inherits CustomRazorPage<TModel>
```

```
<div>Custom text: @CustomText</div>
```

The code renders the following HTML:

HTML

```
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before
dumping
    a slop bucket on the street below.
</div>
```

`@model` and `@inherits` can be used in the same view. `@inherits` can be in a `_ViewImports.cshtml` file that the view imports:

CSHTML

```
@inherits CustomRazorPage<TModel>
```

The following code is an example of a strongly-typed view:

CSHTML

```
@inherits CustomRazorPage<TModel>

<div>The Login Email: @Model.Email</div>
<div>Custom text: @CustomText</div>
```

If "rick@contoso.com" is passed in the model, the view generates the following HTML markup:

HTML

```
<div>The Login Email: rick@contoso.com</div>
<div>
    Custom text: Gardyloo! - A Scottish warning yelled from a window before
dumping
    a slop bucket on the street below.
</div>
```

`@inject`

The `@inject` directive enables the Razor Page to inject a service from the service container into a view. For more information, see Dependency injection into views.

`@layout`

This scenario only applies to Razor components (`.razor`).

The `@layout` directive specifies a layout for routable Razor components that have an `@page` directive. Layout components are used to avoid code duplication and inconsistency. For more information, see ASP.NET Core Blazor layouts.

`@model`

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

The `@model` directive specifies the type of the model passed to a view or page:

```
CSHTML

@model TypeNameOfModel
```

In an ASP.NET Core MVC or Razor Pages app created with individual user accounts, `Views/Account/Login.cshtml` contains the following model declaration:

```
CSHTML

@model LoginViewModel
```

The class generated inherits from `RazorPage<LoginViewModel>`:

```
C#

public class _Views_Account_Login_cshtml : RazorPage<LoginViewModel>
```

Razor exposes a `Model` property for accessing the model passed to the view:

```
CSHTML

<div>The Login Email: @Model.Email</div>
```

The `@model` directive specifies the type of the `Model` property. The directive specifies the `T` in `RazorPage<T>` that the generated class that the view derives from. If the `@model` directive isn't specified, the `Model` property is of type `dynamic`. For more information, see Strongly typed models and the @model keyword.

`@namespace`

The `@namespace` directive:

- Sets the namespace of the class of the generated Razor page, MVC view, or Razor component.
- Sets the root derived namespaces of a pages, views, or components classes from the closest imports file in the directory tree, `_ViewImports.cshtml` (views or pages) or `_Imports.razor` (Razor components).

CSHTML

```
@namespace Your.Namespace.Here
```

For the Razor Pages example shown in the following table:

- Each page imports `Pages/_ViewImports.cshtml`.
- `Pages/_ViewImports.cshtml` contains `@namespace Hello.World`.
- Each page has `Hello.World` as the root of it's namespace.

Expand table

Page	Namespace
`Pages/Index.cshtml`	`Hello.World`
`Pages/MorePages/Page.cshtml`	`Hello.World.MorePages`
`Pages/MorePages/EvenMorePages/Page.cshtml`	`Hello.World.MorePages.EvenMorePages`

The preceding relationships apply to import files used with MVC views and Razor components.

When multiple import files have a `@namespace` directive, the file closest to the page, view, or component in the directory tree is used to set the root namespace.

If the `EvenMorePages` folder in the preceding example has an imports file with `@namespace Another.Planet` (or the `Pages/MorePages/EvenMorePages/Page.cshtml` file contains `@namespace Another.Planet`), the result is shown in the following table.

Expand table

Page	Namespace
`Pages/Index.cshtml`	`Hello.World`
`Pages/MorePages/Page.cshtml`	`Hello.World.MorePages`
`Pages/MorePages/EvenMorePages/Page.cshtml`	`Another.Planet`

`@page`

The `@page` directive has different effects depending on the type of the file where it appears. The directive:

- In a `.cshtml` file indicates that the file is a Razor Page. For more information, see Custom routes and Introduction to Razor Pages in ASP.NET Core.
- Specifies that a Razor component should handle requests directly. For more information, see ASP.NET Core Blazor routing and navigation.

`@preservewhitespace`

This scenario only applies to Razor components (`.razor`).

When set to `false` (default), whitespace in the rendered markup from Razor components (`.razor`) is removed if:

- Leading or trailing within an element.
- Leading or trailing within a `RenderFragment` parameter. For example, child content passed to another component.
- It precedes or follows a C# code block, such as `@if` or `@foreach`.

`@rendermode`

This scenario only applies to Razor components (`.razor`).

Sets the render mode of a Razor component:

- `InteractiveServer`: Applies interactive server rendering using Blazor Server.
- `InteractiveWebAssembly`: Applies interactive WebAssembly rendering using Blazor WebAssembly.
- `InteractiveAuto`: Initially applies interactive WebAssembly rendering using Blazor Server, and then applies interactive WebAssembly rendering using WebAssembly on subsequent visits after the Blazor bundle is downloaded.

For a component instance:

```razor
<... @rendermode="InteractiveServer" />
```

In the component definition:

```razor
@rendermode InteractiveServer
```

> **Note**
>
> Blazor templates include a static `using` directive for RenderMode in the app's `_Imports` file (`Components/_Imports.razor`) for shorter `@rendermode` syntax:
>
> ```razor
> @using static Microsoft.AspNetCore.Components.Web.RenderMode
> ```
>
> Without the preceding directive, components must specify the static RenderMode class in `@rendermode` syntax explicitly:
>
> ```razor
> <Dialog @rendermode="RenderMode.InteractiveServer" />
> ```

For more information, including guidance on disabling prerendering with the directive/directive attribute, see ASP.NET Core Blazor render modes.

`@section`

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

The `@section` directive is used in conjunction with MVC and Razor Pages layouts to enable views or pages to render content in different parts of the HTML page. For more information, see Layout in ASP.NET Core.

`@typeparam`

This scenario only applies to Razor components (`.razor`).

The `@typeparam` directive declares a generic type parameter for the generated component class:

```razor
@typeparam TEntity
```

Generic types with where type constraints are supported:

```razor
@typeparam TEntity where TEntity : IEntity
```

For more information, see the following articles:

- ASP.NET Core Razor component generic type support
- ASP.NET Core Blazor templated components

`@using`

The `@using` directive adds the C# `using` directive to the generated view:

```CSHTML
@using System.IO
@{
    var dir = Directory.GetCurrentDirectory();
}
<p>@dir</p>
```

In Razor components, `@using` also controls which components are in scope.

Directive attributes

Razor directive attributes are represented by implicit expressions with reserved keywords following the `@` symbol. A directive attribute typically changes the way an element is compiled or functions.

`@attributes`

This scenario only applies to Razor components (`.razor`).

`@attributes` allows a component to render non-declared attributes. For more information, see ASP.NET Core Blazor attribute splatting and arbitrary parameters.

`@bind`

This scenario only applies to Razor components (`.razor`).

Data binding in components is accomplished with the `@bind` attribute. For more information, see ASP.NET Core Blazor data binding.

`@bind:culture`

This scenario only applies to Razor components (`.razor`).

Use the `@bind:culture` attribute with the @bind attribute to provide a System.Globalization.CultureInfo for parsing and formatting a value. For more information, see ASP.NET Core Blazor globalization and localization.

`@formname`

This scenario only applies to Razor components (`.razor`).

`@formname` assigns a form name to a Razor component's plain HTML form or a form based on EditForm (Editform documentation). The value of `@formname` should be unique, which prevents form collisions in the following situations:

- A form is placed in a component with multiple forms.
- A form is sourced from an external class library, commonly a NuGet package, for a component with multiple forms, and the app author doesn't control the source code of the library to set a different external form name than a name used by another form in the component.

For more information and examples, see ASP.NET Core Blazor forms overview.

`@on{EVENT}`

This scenario only applies to Razor components (`.razor`).

Razor provides event handling features for components. For more information, see ASP.NET Core Blazor event handling.

`@on{EVENT}:preventDefault`

This scenario only applies to Razor components (`.razor`).

Prevents the default action for the event.

`@on{EVENT}:stopPropagation`

This scenario only applies to Razor components (`.razor`).

Stops event propagation for the event.

`@key`

This scenario only applies to Razor components (`.razor`).

The `@key` directive attribute causes the components diffing algorithm to guarantee preservation of elements or components based on the key's value. For more information, see Retain element, component, and model relationships in ASP.NET Core Blazor.

`@ref`

This scenario only applies to Razor components (`.razor`).

Component references (`@ref`) provide a way to reference a component instance so that you can issue commands to that instance. For more information, see ASP.NET Core Razor components.

Templated Razor delegates

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

Razor templates allow you to define a UI snippet with the following format:

```
CSHTML

@<tag>...</tag>
```

The following example illustrates how to specify a templated Razor delegate as a Func<T,TResult>. The dynamic type is specified for the parameter of the method that

the delegate encapsulates. An object type is specified as the return value of the delegate. The template is used with a List<T> of `Pet` that has a `Name` property.

C#

```
public class Pet
{
    public string Name { get; set; }
}
```

CSHTML

```
@{
    Func<dynamic, object> petTemplate = @<p>You have a pet named
<strong>@item.Name</strong>.</p>;

    var pets = new List<Pet>
    {
        new Pet { Name = "Rin Tin Tin" },
        new Pet { Name = "Mr. Bigglesworth" },
        new Pet { Name = "K-9" }
    };
}
```

The template is rendered with `pets` supplied by a `foreach` statement:

CSHTML

```
@foreach (var pet in pets)
{
    @petTemplate(pet)
}
```

Rendered output:

HTML

```
<p>You have a pet named <strong>Rin Tin Tin</strong>.</p>
<p>You have a pet named <strong>Mr. Bigglesworth</strong>.</p>
<p>You have a pet named <strong>K-9</strong>.</p>
```

You can also supply an inline Razor template as an argument to a method. In the following example, the `Repeat` method receives a Razor template. The method uses the template to produce HTML content with repeats of items supplied from a list:

CSHTML

```
@using Microsoft.AspNetCore.Html

@functions {
    public static IHtmlContent Repeat(IEnumerable<dynamic> items, int times,
        Func<dynamic, IHtmlContent> template)
    {
        var html = new HtmlContentBuilder();

        foreach (var item in items)
        {
            for (var i = 0; i < times; i++)
            {
                html.AppendHtml(template(item));
            }
        }

        return html;
    }
}
```

Using the list of pets from the prior example, the `Repeat` method is called with:

- List<T> of `Pet`.
- Number of times to repeat each pet.
- Inline template to use for the list items of an unordered list.

CSHTML

```
<ul>
    @Repeat(pets, 3, @<li>@item.Name</li>)
</ul>
```

Rendered output:

HTML

```
<ul>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Rin Tin Tin</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>Mr. Bigglesworth</li>
    <li>K-9</li>
    <li>K-9</li>
    <li>K-9</li>
</ul>
```

Tag Helpers

This scenario only applies to MVC views and Razor Pages (`.cshtml`).

There are three directives that pertain to Tag Helpers.

Expand table

Directive	Function
@addTagHelper	Makes Tag Helpers available to a view.
@removeTagHelper	Removes Tag Helpers previously added from a view.
@tagHelperPrefix	Specifies a tag prefix to enable Tag Helper support and to make Tag Helper usage explicit.

Razor reserved keywords

Razor keywords

- `page`
- `namespace`
- `functions`
- `inherits`
- `model`
- `section`
- `helper` (Not currently supported by ASP.NET Core)

Razor keywords are escaped with `@(Razor Keyword)` (for example, `@(functions)`).

C# Razor keywords

- `case`
- `do`
- `default`
- `for`
- `foreach`
- `if`
- `else`
- `lock`

- `switch`
- `try`
- `catch`
- `finally`
- `using`
- `while`

C# Razor keywords must be double-escaped with `@(@C# Razor Keyword)` (for example, `@(@case)`). The first `@` escapes the Razor parser. The second `@` escapes the C# parser.

Reserved keywords not used by Razor

- `class`

Inspect the Razor C# class generated for a view

The Razor SDK handles compilation of Razor files. By default, the generated code files aren't emitted. To enable emitting the code files, set the `EmitCompilerGeneratedFiles` directive in the project file (`.csproj`) to `true`:

XML

```
<PropertyGroup>
  <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
</PropertyGroup>
```

When building a 6.0 project (`net6.0`) in the `Debug` build configuration, the Razor SDK generates an `obj/Debug/net6.0/generated/` directory in the project root. Its subdirectory contains the emitted Razor page code files.

View lookups and case sensitivity

The Razor view engine performs case-sensitive lookups for views. However, the actual lookup is determined by the underlying file system:

- File based source:
  - On operating systems with case insensitive file systems (for example, Windows), physical file provider lookups are case insensitive. For example, `return View("Test")` results in matches for `/Views/Home/Test.cshtml`, `/Views/home/test.cshtml`, and any other casing variant.

    - On case-sensitive file systems (for example, Linux, OSX, and with `EmbeddedFileProvider`), lookups are case-sensitive. For example, `return View("Test")` specifically matches `/Views/Home/Test.cshtml`.
- Precompiled views: With ASP.NET Core 2.0 and later, looking up precompiled views is case insensitive on all operating systems. The behavior is identical to physical file provider's behavior on Windows. If two precompiled views differ only in case, the result of lookup is non-deterministic.

Developers are encouraged to match the casing of file and directory names to the casing of:

- Area, controller, and action names.
- Razor Pages.

Matching case ensures the deployments find their views regardless of the underlying file system.

Imports used by Razor

The following imports are generated by the ASP.NET Core web templates to support Razor Files:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
```

Additional resources

Introduction to ASP.NET Web Programming Using the Razor Syntax provides many samples of programming with Razor syntax.

Create reusable UI using the Razor class library project in ASP.NET Core

Article • 11/29/2023

By Rick Anderson

Razor views, pages, controllers, page models, Razor components, View components, and data models can be built into a Razor class library (RCL). The RCL can be packaged and reused. Applications can include the RCL and override the views and pages it contains. When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (`.cshtml` file) in the web app takes precedence.

For information on how to integrate npm and webpack into the build process for a RCL, see Build client web assets for your Razor Class Library.

Create a class library containing Razor UI

Visual Studio

- From Visual Studio select **Create new a new project**.
- Select **Razor Class Library** > **Next**.
- Name the library (for example, "RazorClassLib"), > **Create**. To avoid a file name collision with the generated view library, ensure the library name doesn't end in `.Views`.
- Select **Support pages and views** if you need the library to contain pages and/or views. By default, only Razor components are supported. Select **Create**.

The Razor Class Library template defaults to Razor component development by default. The **Support pages and views** option supports pages and views. For more information on RCL support in Blazor, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Add Razor files to the RCL.

The ASP.NET Core templates assume the RCL content is in the `Areas` folder. See RCL Pages layout below to create an RCL that exposes content in `~/Pages` rather than `~/Areas/Pages`.

Reference RCL content

The RCL can be referenced by:

- NuGet package. See Creating NuGet packages and dotnet add package and Create and publish a NuGet package.
- `{ProjectName}.csproj`. See dotnet-add reference.

Override views, partial views, and pages

When a view, partial view, or Razor Page is found in both the web app and the RCL, the Razor markup (`.cshtml` file) in the web app takes precedence. For example, add `WebApp1/Areas/MyFeature/Pages/Page1.cshtml` to WebApp1, and Page1 in the WebApp1 will take precedence over Page1 in the RCL.

In the sample download, rename `WebApp1/Areas/MyFeature2` to `WebApp1/Areas/MyFeature` to test precedence.

Copy the `RazorUIClassLib/Areas/MyFeature/Pages/Shared/_Message.cshtml` partial view to `WebApp1/Areas/MyFeature/Pages/Shared/_Message.cshtml`. Update the markup to indicate the new location. Build and run the app to verify the app's version of the partial is being used.

If the RCL uses Razor Pages, enable the Razor Pages services and endpoints in the hosting app:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

```
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.MapRazorPages();
app.Run();
```

RCL Pages layout

To reference RCL content as though it is part of the web app's `Pages` folder, create the RCL project with the following file structure:

- `RazorUIClassLib/Pages`
- `RazorUIClassLib/Pages/Shared`

Suppose `RazorUIClassLib/Pages/Shared` contains two partial files: `_Header.cshtml` and `_Footer.cshtml`. The `<partial>` tags could be added to `_Layout.cshtml` file:

CSHTML

```
<body>
  <partial name="_Header">
  @RenderBody()
  <partial name="_Footer">
</body>
```

Add the `_ViewStart.cshtml` file to the RCL project's `Pages` folder to use the `_Layout.cshtml` file from the host web app:

CSHTML

```
@{
    Layout = "_Layout";
}
```

Create an RCL with static assets

An RCL may require companion static assets that can be referenced by either the RCL or the consuming app of the RCL. ASP.NET Core allows creating RCLs that include static assets that are available to a consuming app.

To include companion assets as part of an RCL, create a `wwwroot` folder in the class library and include any required files in that folder.

When packing an RCL, all companion assets in the `wwwroot` folder are automatically included in the package.

Use the `dotnet pack` command rather than the NuGet.exe version `nuget pack`.

Add client web assets to the build process

Integrating client web assets into the build pipeline is nontrivial. See Build client web assets for your Razor Class Library for more information.

Exclude static assets

To exclude static assets, add the desired exclusion path to the `$(DefaultItemExcludes)` property group in the project file. Separate entries with a semicolon (`;`).

In the following example, the `lib.css` stylesheet in the `wwwroot` folder isn't considered a static asset and isn't included in the published RCL:

XML

```
<PropertyGroup>

<DefaultItemExcludes>$(DefaultItemExcludes);wwwroot\lib.css</DefaultItemExcl
udes>
</PropertyGroup>
```

Typescript integration

To include TypeScript files in an RCL:

1. Reference the Microsoft.TypeScript.MSBuild NuGet package in the project.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

2. Place the TypeScript files (`.ts`) outside of the `wwwroot` folder. For example, place the files in a `Client` folder.

3. Configure the TypeScript build output for the `wwwroot` folder. Set the `TypescriptOutDir` property inside of a `PropertyGroup` in the project file:

   XML

   ```
   <TypescriptOutDir>wwwroot</TypescriptOutDir>
   ```

4. Include the TypeScript target as a dependency of the `PrepareForBuildDependsOn` target by adding the following target inside of a `PropertyGroup` in the project file:

   XML

   ```
   <PrepareForBuildDependsOn>
     CompileTypeScript;
     GetTypeScriptOutputForPublishing;$(PrepareForBuildDependsOn)
   </PrepareForBuildDependsOn>
   ```

Consume content from a referenced RCL

The files included in the `wwwroot` folder of the RCL are exposed to either the RCL or the consuming app under the prefix `_content/{PACKAGE ID}/`. For example, a library with an assembly name of `Razor.Class.Lib` and without a `<PackageId>` specified in its project file results in a path to static content at `_content/Razor.Class.Lib/`. When producing a NuGet package and the assembly name isn't the same as the package ID (<PackageId> in the library's project file), use the package ID as specified in the project file for `{PACKAGE ID}`.

The consuming app references static assets provided by the library with `<script>`, `<style>`, `<img>`, and other HTML tags. The consuming app must have static file support enabled in:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
```

```
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.MapRazorPages();
app.Run();
```

When running the consuming app from build output (`dotnet run`), static web assets are enabled by default in the Development environment. To support assets in other environments when running from build output, call UseStaticWebAssets on the host builder in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseWebRoot("wwwroot");
builder.WebHost.UseStaticWebAssets();

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Calling `UseStaticWebAssets` isn't required when running an app from published output (`dotnet publish`).

Multi-project development flow

When the consuming app runs:

- The assets in the RCL stay in their original folders. The assets aren't moved to the consuming app.
- Any change within the RCL's `wwwroot` folder is reflected in the consuming app after the RCL is rebuilt and without rebuilding the consuming app.

When the RCL is built, a manifest is produced that describes the static web asset locations. The consuming app reads the manifest at runtime to consume the assets from referenced projects and packages. When a new asset is added to an RCL, the RCL must be rebuilt to update its manifest before a consuming app can access the new asset.

Publish

When the app is published, the companion assets from all referenced projects and packages are copied into the `wwwroot` folder of the published app under `_content/{PACKAGE ID}/`. When producing a NuGet package and the assembly name isn't the same as the package ID (<PackageId> in the library's project file), use the package ID as specified in the project file for `{PACKAGE ID}` when examining the `wwwroot` folder for the published assets.

Additional resources

- View or download sample code (how to download)
- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- ASP.NET Core Blazor CSS isolation

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

ASP.NET Core built-in Tag Helpers

Article • 06/03/2022

By Peter Kellner

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

There are built-in Tag Helpers which aren't listed in this document. The unlisted Tag Helpers are used internally by the Razor view engine. The Tag Helper for the `~` (tilde) character is unlisted. The tilde Tag Helper expands to the root path of the website.

Built-in ASP.NET Cores

Anchor

Cache

Component

Distributed Cache

Environment

Form

Form Action

Image

Input

Label

Link

Partial

Persist Component State

Script

Select

Textarea

Validation Message

Validation Summary

Additional resources

- Tag Helpers in ASP.NET Core
- Tag Helper Components in ASP.NET Core

Tag Helpers in ASP.NET Core

Article • 07/24/2023

By Rick Anderson

What are Tag Helpers

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. For example, the built-in `ImageTagHelper` can append a version number to the image name. Whenever the image changes, the server generates a new unique version for the image, so clients are guaranteed to get the current image (instead of a stale cached image). There are many built-in Tag Helpers for common tasks - such as creating forms, links, loading assets and more - and even more available in public GitHub repositories and as NuGet packages. Tag Helpers are authored in C#, and they target HTML elements based on element name, attribute name, or parent tag. For example, the built-in `LabelTagHelper` can target the HTML `<label>` element when the `LabelTagHelper` attributes are applied. If you're familiar with HTML Helpers, Tag Helpers reduce the explicit transitions between HTML and C# in Razor views. In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. Tag Helpers compared to HTML Helpers explains the differences in more detail.

Tag Helpers aren't supported in Razor components. For more information, see ASP.NET Core Razor components.

What Tag Helpers provide

An HTML-friendly development experience

For the most part, Razor markup using Tag Helpers looks like standard HTML. Front-end designers conversant with HTML/CSS/JavaScript can edit Razor without learning C# Razor syntax.

A rich IntelliSense environment for creating HTML and Razor markup

This is in sharp contrast to HTML Helpers, the previous approach to server-side creation of markup in Razor views. Tag Helpers compared to HTML Helpers explains the differences in more detail. IntelliSense support for Tag Helpers explains the IntelliSense

environment. Even developers experienced with Razor C# syntax are more productive using Tag Helpers than writing C# Razor markup.

A way to make you more productive and able to produce more robust, reliable, and maintainable code using information only available on the server

For example, historically the mantra on updating images was to change the name of the image when you change the image. Images should be aggressively cached for performance reasons, and unless you change the name of an image, you risk clients getting a stale copy. Historically, after an image was edited, the name had to be changed and each reference to the image in the web app needed to be updated. Not only is this very labor intensive, it's also error prone (you could miss a reference, accidentally enter the wrong string, etc.) The built-in `ImageTagHelper` can do this for you automatically. The `ImageTagHelper` can append a version number to the image name, so whenever the image changes, the server automatically generates a new unique version for the image. Clients are guaranteed to get the current image. This robustness and labor savings comes essentially free by using the `ImageTagHelper`.

Most built-in Tag Helpers target standard HTML elements and provide server-side attributes for the element. For example, the `<input>` element used in many views in the *Views/Account* folder contains the `asp-for` attribute. This attribute extracts the name of the specified model property into the rendered HTML. Consider a Razor view with the following model:

C#

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
}
```

The following Razor markup:

CSHTML

```
<label asp-for="Movie.Title"></label>
```

Generates the following HTML:

HTML

```
<label for="Movie_Title">Title</label>
```

The `asp-for` attribute is made available by the `For` property in the LabelTagHelper. See Author Tag Helpers for more information.

Managing Tag Helper scope

Tag Helpers scope is controlled by a combination of `@addTagHelper`, `@removeTagHelper`, and the "!" opt-out character.

`@addTagHelper` makes Tag Helpers available

If you create a new ASP.NET Core web app named *AuthoringTagHelpers*, the following `Views/_ViewImports.cshtml` file will be added to your project:

CSHTML

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

The `@addTagHelper` directive makes Tag Helpers available to the view. In this case, the view file is `Pages/_ViewImports.cshtml`, which by default is inherited by all files in the *Pages* folder and subfolders; making Tag Helpers available. The code above uses the wildcard syntax ("*") to specify that all Tag Helpers in the specified assembly (*Microsoft.AspNetCore.Mvc.TagHelpers*) will be available to every view file in the *Views* directory or subdirectory. The first parameter after `@addTagHelper` specifies the Tag Helpers to load (we are using "*" for all Tag Helpers), and the second parameter "Microsoft.AspNetCore.Mvc.TagHelpers" specifies the assembly containing the Tag Helpers. *Microsoft.AspNetCore.Mvc.TagHelpers* is the assembly for the built-in ASP.NET Core Tag Helpers.

To expose all of the Tag Helpers in this project (which creates an assembly named *AuthoringTagHelpers*), you would use the following:

CSHTML

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

If your project contains an `EmailTagHelper` with the default namespace (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), you can provide the fully qualified name (FQN) of the Tag Helper:

CSHTML

```
@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper,
AuthoringTagHelpers
```

To add a Tag Helper to a view using an FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the assembly name (*AuthoringTagHelpers*). Most developers prefer to use the "*" wildcard syntax. The wildcard syntax allows you to insert the wildcard character "*" as the suffix in an FQN. For example, any of the following directives will bring in the `EmailTagHelper`:

CSHTML

```
@addTagHelper AuthoringTagHelpers.TagHelpers.E*, AuthoringTagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.Email*, AuthoringTagHelpers
```

As mentioned previously, adding the `@addTagHelper` directive to the `Views/_ViewImports.cshtml` file makes the Tag Helper available to all view files in the *Views* directory and subdirectories. You can use the `@addTagHelper` directive in specific view files if you want to opt-in to exposing the Tag Helper to only those views.

`@removeTagHelper` removes Tag Helpers

The `@removeTagHelper` has the same two parameters as `@addTagHelper`, and it removes a Tag Helper that was previously added. For example, `@removeTagHelper` applied to a specific view removes the specified Tag Helper from the view. Using `@removeTagHelper` in a `Views/Folder/_ViewImports.cshtml` file removes the specified Tag Helper from all of the views in *Folder*.

Controlling Tag Helper scope with the `_ViewImports.cshtml` file

You can add a `_ViewImports.cshtml` to any view folder, and the view engine applies the directives from both that file and the `Views/_ViewImports.cshtml` file. If you added an empty `Views/Home/_ViewImports.cshtml` file for the *Home* views, there would be no

change because the `_ViewImports.cshtml` file is additive. Any `@addTagHelper` directives you add to the `Views/Home/_ViewImports.cshtml` file (that are not in the default `Views/_ViewImports.cshtml` file) would expose those Tag Helpers to views only in the *Home* folder.

Opting out of individual elements

You can disable a Tag Helper at the element level with the Tag Helper opt-out character ("!"). For example, `Email` validation is disabled in the `<span>` with the Tag Helper opt-out character:

CSHTML

```
<!span asp-validation-for="Email" class="text-danger"></!span>
```

You must apply the Tag Helper opt-out character to the opening and closing tag. (The Visual Studio editor automatically adds the opt-out character to the closing tag when you add one to the opening tag). After you add the opt-out character, the element and Tag Helper attributes are no longer displayed in a distinctive font.

Using `@tagHelperPrefix` to make Tag Helper usage explicit

The `@tagHelperPrefix` directive allows you to specify a tag prefix string to enable Tag Helper support and to make Tag Helper usage explicit. For example, you could add the following markup to the `Views/_ViewImports.cshtml` file:

CSHTML

```
@tagHelperPrefix th:
```

In the code image below, the Tag Helper prefix is set to `th:`, so only those elements using the prefix `th:` support Tag Helpers (Tag Helper-enabled elements have a distinctive font). The `<label>` and `<input>` elements have the Tag Helper prefix and are Tag Helper-enabled, while the `<span>` element doesn't.

```
<div class="form-group">
    <th:label asp-for="Password" class="col-md-2"></th:label>
    <div class="col-md-10">
        <th:input asp-for="Password" class="form-control" />
        <span asp-validation-for="Password" class="text-danger"></span>
    </div>
</div>
```

The same hierarchy rules that apply to `@addTagHelper` also apply to `@tagHelperPrefix`.

Self-closing Tag Helpers

Many Tag Helpers can't be used as self-closing tags. Some Tag Helpers are designed to be self-closing tags. Using a Tag Helper that was not designed to be self-closing suppresses the rendered output. Self-closing a Tag Helper results in a self-closing tag in the rendered output. For more information, see this note in Authoring Tag Helpers.

C# in Tag Helpers attribute/declaration

Tag Helpers do not allow C# in the element's attribute or tag declaration area. For example, the following code is not valid:

CSHTML

```
<input asp-for="LastName"
       @(Model?.LicenseId == null ? "disabled" : string.Empty) />
```

The preceding code can be written as:

CSHTML

```
<input asp-for="LastName"
       disabled="@(Model?.LicenseId == null)" />
```

Normally, the `@` operator inserts a textual representation of an expression into the rendered HTML markup. However, when an expression evaluates to logical `false`, the framework removes the attribute instead. In the preceding example, the `disabled` attribute is set to `true` if either `Model` or `LicenseId` is `null`.

Tag helper initializers

While attributes can be used to configure individual instances of tag helpers, ITagHelperInitializer<TTagHelper> can be used to configure all tag helper instances of a specific kind. Consider the following example of a tag helper initializer that configures the `asp-append-version` attribute or `AppendVersion` property for all instances of `ScriptTagHelper` in the app:

C#

```
public class AppendVersionTagHelperInitializer :
ITagHelperInitializer<ScriptTagHelper>
{
    public void Initialize(ScriptTagHelper helper, ViewContext context)
    {
        helper.AppendVersion = true;
    }
}
```

To use the initializer, configure it by registering it as part of the application's startup:

C#

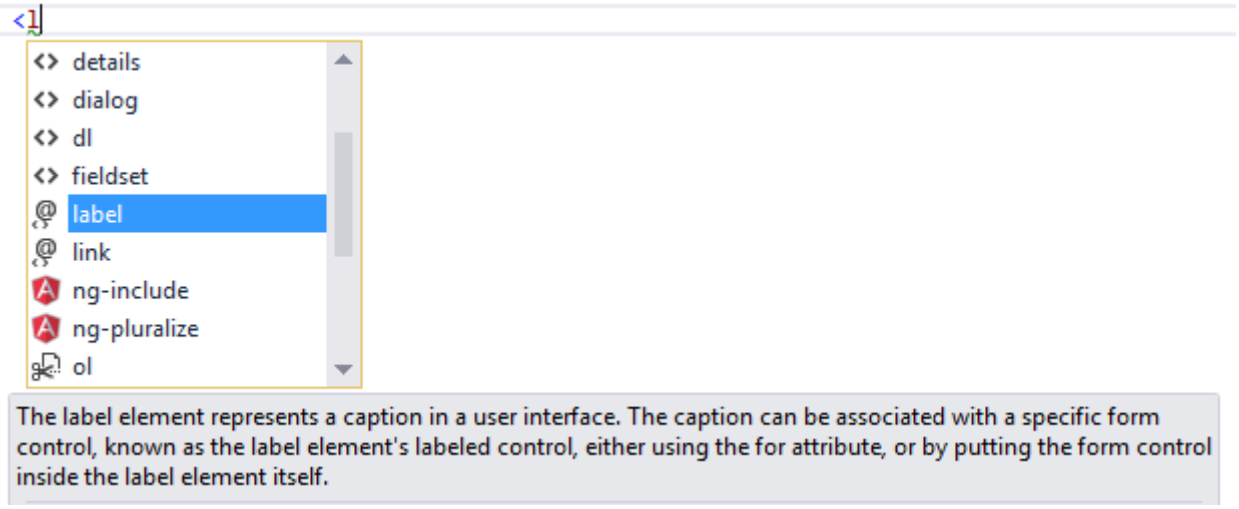
```
builder.Services.AddSingleton
    <ITagHelperInitializer<ScriptTagHelper>,
AppendVersionTagHelperInitializer>();
```

Tag Helper automatic version generation outside of wwwroot

For a Tag Helper to generate a version for a static file outside `wwwroot`, see Serve files from multiple locations

IntelliSense support for Tag Helpers

Consider writing an HTML `<label>` element. As soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



Not only do you get HTML help, but also the icon (the "@" symbol with "<>" under it).

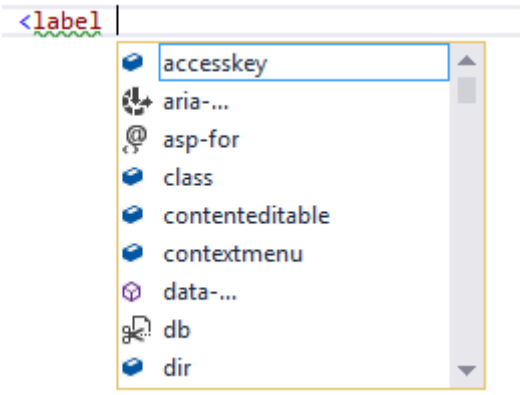


The icon identifies the element as targeted by Tag Helpers. Pure HTML elements (such as the `fieldset`) display the "<>" icon.

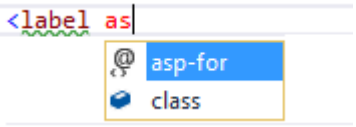
A pure HTML `<label>` tag displays the HTML tag (with the default Visual Studio color theme) in a brown font, the attributes in red, and the attribute values in blue.

```
<label class="col-md-2">Email</label>
```

After you enter `<label`, IntelliSense lists the available HTML/CSS attributes and the Tag Helper-targeted attributes:



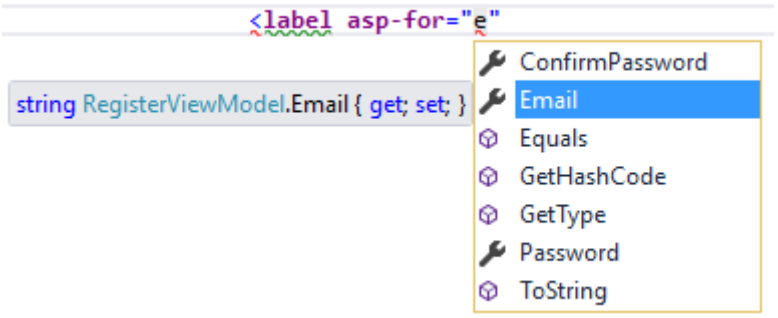
IntelliSense statement completion allows you to enter the tab key to complete the statement with the selected value:



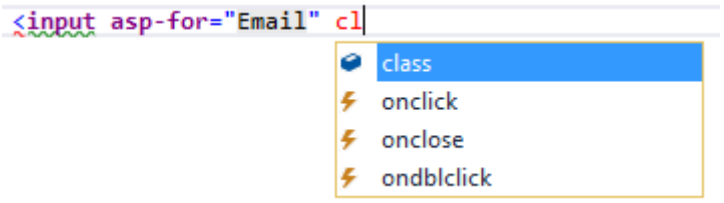
As soon as a Tag Helper attribute is entered, the tag and attribute fonts change. Using the default Visual Studio "Blue" or "Light" color theme, the font is bold purple. If you're using the "Dark" theme the font is bold teal. The images in this document were taken using the default theme.

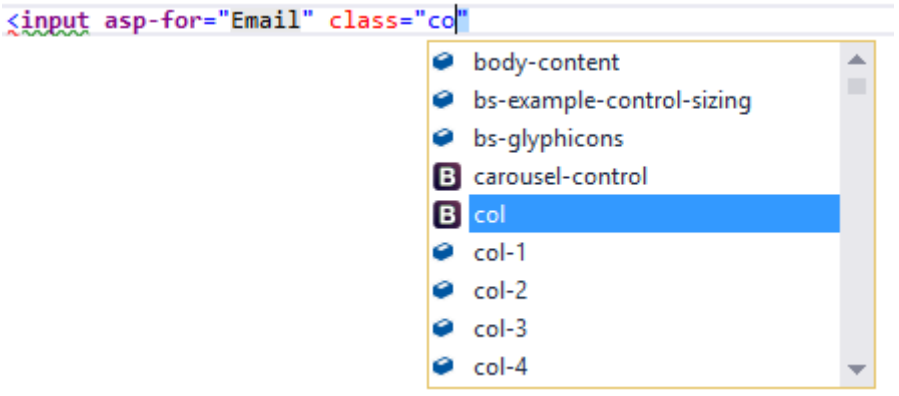
```
<label asp-for
```

You can enter the Visual Studio *CompleteWord* shortcut (Ctrl +spacebar is the default) inside the double quotes (""), and you are now in C#, just like you would be in a C# class. IntelliSense displays all the methods and properties on the page model. The methods and properties are available because the property type is `ModelExpression`. In the image below, I'm editing the `Register` view, so the `RegisterViewModel` is available.



IntelliSense lists the properties and methods available to the model on the page. The rich IntelliSense environment helps you select the CSS class:





Tag Helpers compared to HTML Helpers

Tag Helpers attach to HTML elements in Razor views, while HTML Helpers are invoked as methods interspersed with HTML in Razor views. Consider the following Razor markup, which creates an HTML label with the CSS class "caption":

CSHTML

```
@Html.Label("FirstName", "First Name:", new {@class="caption"})
```

The at (`@`) symbol tells Razor this is the start of code. The next two parameters ("FirstName" and "First Name:") are strings, so IntelliSense can't help. The last argument:

CSHTML

```
new {@class="caption"}
```

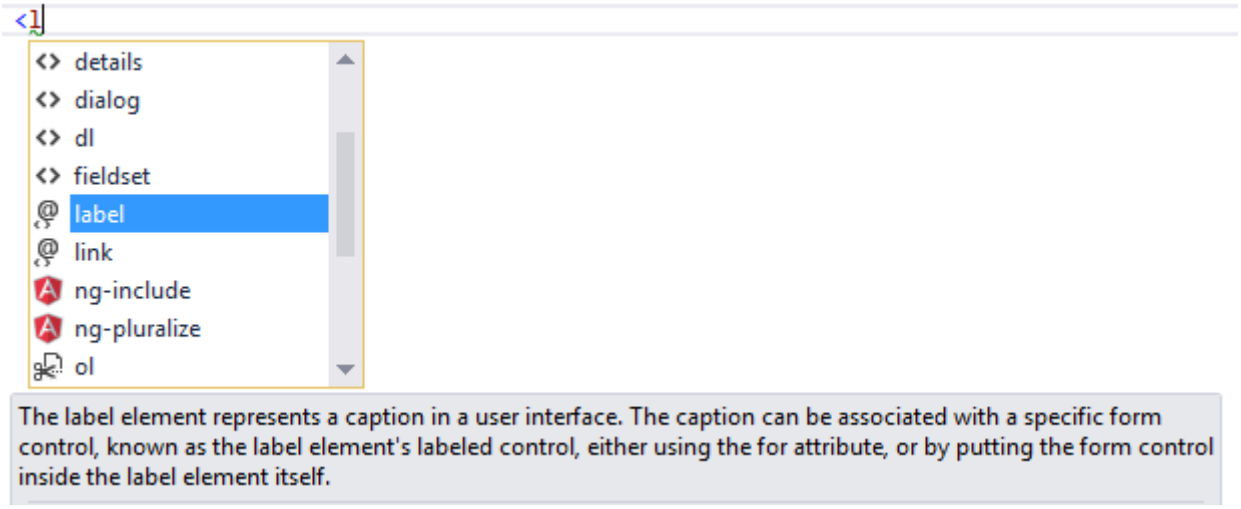
Is an anonymous object used to represent attributes. Because `class` is a reserved keyword in C#, you use the `@` symbol to force C# to interpret `@class=` as a symbol (property name). To a front-end designer (someone familiar with HTML/CSS/JavaScript and other client technologies but not familiar with C# and Razor), most of the line is foreign. The entire line must be authored with no help from IntelliSense.

Using the `LabelTagHelper`, the same markup can be written as:

CSHTML

```
<label class="caption" asp-for="FirstName"></label>
```

With the Tag Helper version, as soon as you enter `<l` in the Visual Studio editor, IntelliSense displays matching elements:



IntelliSense helps you write the entire line.

The following code image shows the Form portion of the `Views/Account/Register.cshtml` Razor view generated from the ASP.NET 4.5.x MVC template included with Visual Studio.

```
@using (Html.BeginForm("Register", "Account", FormMethod.Post, new { @class = "form-horizo
{
    @Html.AntiForgeryToken()
    <h4>Create a new account.</h4>
    <hr />
    @Html.ValidationSummary("", new { @class = "text-danger" })
    <div class="form-group">
        @Html.LabelFor(m => m.Email, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.TextBoxFor(m => m.Email, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        @Html.LabelFor(m => m.ConfirmPassword, new { @class = "col-md-2 control-label" })
        <div class="col-md-10">
            @Html.PasswordFor(m => m.ConfirmPassword, new { @class = "form-control" })
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <input type="submit" class="btn btn-default" value="Register" />
        </div>
    </div>
}
```

The Visual Studio editor displays C# code with a grey background. For example, the `AntiForgeryToken` HTML Helper:

CSHTML

```
@Html.AntiForgeryToken()
```

is displayed with a grey background. Most of the markup in the Register view is C#. Compare that to the equivalent approach using Tag Helpers:

```
<form asp-controller="Account" asp-action="Register" method="post" class="form-hori
    <h4>Create a new account.</h4>
    <hr />
    <div asp-validation-summary="ValidationSummary.All" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Email" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Email" class="form-control" />
            <span asp-validation-for="Email" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Password" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Password" class="form-control" />
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ConfirmPassword" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ConfirmPassword" class="form-control" />
            <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <div class="col-md-offset-2 col-md-10">
            <button type="submit" class="btn btn-default">Register</button>
        </div>
    </div>
</form>
```

The markup is much cleaner and easier to read, edit, and maintain than the HTML Helpers approach. The C# code is reduced to the minimum that the server needs to know about. The Visual Studio editor displays markup targeted by a Tag Helper in a distinctive font.

Consider the *Email* group:

CSHTML

```
<div class="form-group">
    <label asp-for="Email" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Email" class="form-control" />
        <span asp-validation-for="Email" class="text-danger"></span>
    </div>
</div>
```

Each of the "asp-" attributes has a value of "Email", but "Email" isn't a string. In this context, "Email" is the C# model expression property for the `RegisterViewModel`.

The Visual Studio editor helps you write **all** of the markup in the Tag Helper approach of the register form, while Visual Studio provides no help for most of the code in the HTML

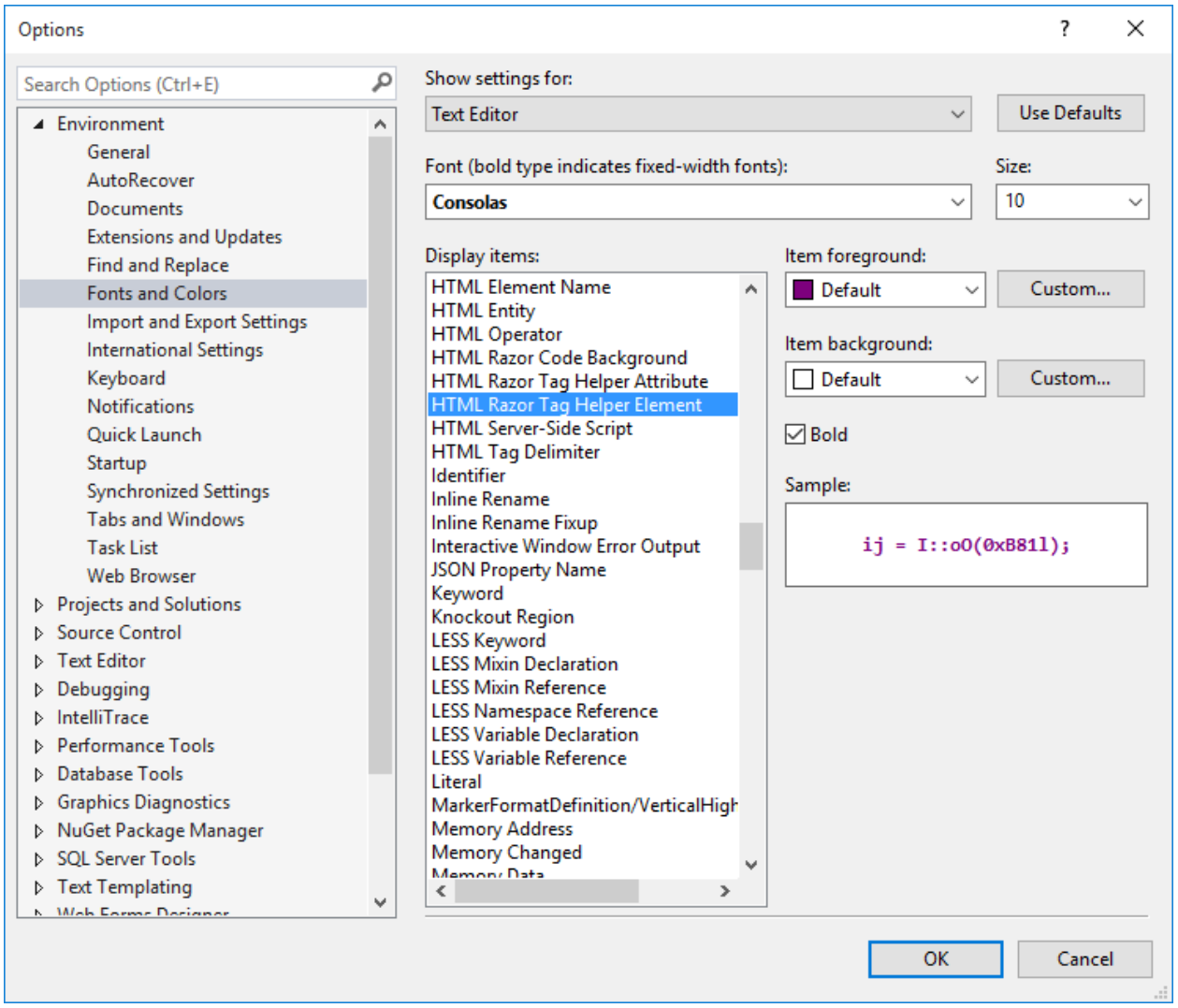
Helpers approach. IntelliSense support for Tag Helpers goes into detail on working with Tag Helpers in the Visual Studio editor.

Tag Helpers compared to Web Server Controls

- Tag Helpers don't own the element they're associated with; they simply participate in the rendering of the element and content. ASP.NET Web Server Controls are declared and invoked on a page.
- ASP.NET Web Server Controls have a non-trivial lifecycle that can make developing and debugging difficult.
- Web Server controls allow you to add functionality to the client DOM elements by using a client control. Tag Helpers have no DOM.
- Web Server controls include automatic browser detection. Tag Helpers have no knowledge of the browser.
- Multiple Tag Helpers can act on the same element (see Avoiding Tag Helper conflicts) while you typically can't compose Web Server controls.
- Tag Helpers can modify the tag and content of HTML elements that they're scoped to, but don't directly modify anything else on a page. Web Server controls have a less specific scope and can perform actions that affect other parts of your page; enabling unintended side effects.
- Web Server controls use type converters to convert strings into objects. With Tag Helpers, you work natively in C#, so you don't need to do type conversion.
- Web Server controls use System.ComponentModel to implement the run-time and design-time behavior of components and controls. `System.ComponentModel` includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components. Contrast that to Tag Helpers, which typically derive from `TagHelper`, and the `TagHelper` base class exposes only two methods, `Process` and `ProcessAsync`.

Customizing the Tag Helper element font

You can customize the font and colorization from **Tools** > **Options** > **Environment** > **Fonts and Colors**:



Built-in ASP.NET Core Tag Helpers

Anchor

Cache

Component

Distributed Cache

Environment

Form

Form Action

Image

Input

Label

Link

Partial

Persist Component State

Script

Select

Textarea

Validation Message

Validation Summary

Additional resources

- Author Tag Helpers
- Working with Forms
- TagHelperSamples on GitHub contains Tag Helper samples for working with Bootstrap.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Author Tag Helpers in ASP.NET Core

Article • 09/14/2022

By Rick Anderson

View or download sample code (how to download)

Get started with Tag Helpers

This tutorial provides an introduction to programming Tag Helpers. Introduction to Tag Helpers describes the benefits that Tag Helpers provide.

A tag helper is any class that implements the `ITagHelper` interface. However, when you author a tag helper, you generally derive from `TagHelper`, doing so gives you access to the `Process` method.

1. Create a new ASP.NET Core project called **AuthoringTagHelpers**. You won't need authentication for this project.
2. Create a folder to hold the Tag Helpers called *TagHelpers*. The *TagHelpers* folder is *not* required, but it's a reasonable convention. Now let's get started writing some simple tag helpers.

A minimal Tag Helper

In this section, you write a tag helper that updates an email tag. For example:

HTML

```
<email>Support</email>
```

The server will use our email tag helper to convert that markup into the following:

HTML

```
<a href="mailto:Support@contoso.com">Support@contoso.com</a>
```

That is, an anchor tag that makes this an email link. You might want to do this if you are writing a blog engine and need it to send email for marketing, support, and other contacts, all to the same domain.

1. Add the following `EmailTagHelper` class to the *TagHelpers* folder.

C#

```
using Microsoft.AspNetCore.Razor.TagHelpers;
using System.Threading.Tasks;

namespace AuthoringTagHelpers.TagHelpers
{
    public class EmailTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            output.TagName = "a";    // Replaces <email> with <a> tag
        }
    }
}
```

- Tag helpers use a naming convention that targets elements of the root class name (minus the *TagHelper* portion of the class name). In this example, the root name of **EmailTagHelper** is *email*, so the `<email>` tag will be targeted. This naming convention should work for most tag helpers, later on I'll show how to override it.
- The `EmailTagHelper` class derives from `TagHelper`. The `TagHelper` class provides methods and properties for writing Tag Helpers.
- The overridden `Process` method controls what the tag helper does when executed. The `TagHelper` class also provides an asynchronous version (`ProcessAsync`) with the same parameters.
- The context parameter to `Process` (and `ProcessAsync`) contains information associated with the execution of the current HTML tag.
- The output parameter to `Process` (and `ProcessAsync`) contains a stateful HTML element representative of the original source used to generate an HTML tag and content.
- Our class name has a suffix of **TagHelper**, which is *not* required, but it's considered a best practice convention. You could declare the class as:

C#

```
public class Email : TagHelper
```

2. To make the `EmailTagHelper` class available to all our Razor views, add the `addTagHelper` directive to the `Views/_ViewImports.cshtml` file:

```
CSHTML

@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, AuthoringTagHelpers
```

The code above uses the wildcard syntax to specify all the tag helpers in our assembly will be available. The first string after `@addTagHelper` specifies the tag helper to load (Use "*" for all tag helpers), and the second string "AuthoringTagHelpers" specifies the assembly the tag helper is in. Also, note that the second line brings in the ASP.NET Core MVC tag helpers using the wildcard syntax (those helpers are discussed in Introduction to Tag Helpers.) It's the `@addTagHelper` directive that makes the tag helper available to the Razor view. Alternatively, you can provide the fully qualified name (FQN) of a tag helper as shown below:

```
C#

@using AuthoringTagHelpers
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper AuthoringTagHelpers.TagHelpers.EmailTagHelper,
AuthoringTagHelpers
```

To add a tag helper to a view using a FQN, you first add the FQN (`AuthoringTagHelpers.TagHelpers.EmailTagHelper`), and then the **assembly name** (*AuthoringTagHelpers*, not necessarily the `namespace`). Most developers will prefer to use the wildcard syntax. Introduction to Tag Helpers goes into detail on tag helper adding, removing, hierarchy, and wildcard syntax.

1. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

```
CSHTML

@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
```

```
        <abbr title="Phone">P:</abbr>
        425.555.0100
    </address>

    <address>
        <strong>Support:</strong><email>Support</email><br />
        <strong>Marketing:</strong><email>Marketing</email>
    </address>
```

2. Run the app and use your favorite browser to view the HTML source so you can verify that the email tags are replaced with anchor markup (For example, `<a>Support</a>`). *Support* and *Marketing* are rendered as a links, but they don't have an `href` attribute to make them functional. We'll fix that in the next section.

SetAttribute and SetContent

In this section, we'll update the `EmailTagHelper` so that it will create a valid anchor tag for email. We'll update it to take information from a Razor view (in the form of a `mail-to` attribute) and use that in generating the anchor.

Update the `EmailTagHelper` class with the following:

C#

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";

    // Can be passed via <email mail-to="..." />.
    // PascalCase gets translated into kebab-case.
    public string MailTo { get; set; }

    public override void Process(TagHelperContext context, TagHelperOutput
output)
    {
        output.TagName = "a";    // Replaces <email> with <a> tag

        var address = MailTo + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + address);
        output.Content.SetContent(address);
    }
}
```

- Pascal-cased class and property names for tag helpers are translated into their kebab case. Therefore, to use the `MailTo` attribute, you'll use `<email mail-to="value"/>` equivalent.

- The last line sets the completed content for our minimally functional tag helper.
- The highlighted line shows the syntax for adding attributes:

C#

```
public override void Process(TagHelperContext context, TagHelperOutput
output)
{
    output.TagName = "a";    // Replaces <email> with <a> tag

    var address = MailTo + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
```

That approach works for the attribute "href" as long as it doesn't currently exist in the attributes collection. You can also use the `output.Attributes.Add` method to add a tag helper attribute to the end of the collection of tag attributes.

1. Update the markup in the `Views/Home/Contact.cshtml` file with these changes:

   CSHTML

   ```
   @{
       ViewData["Title"] = "Contact Copy";
   }
   <h2>@ViewData["Title"].</h2>
   <h3>@ViewData["Message"]</h3>

   <address>
       One Microsoft Way Copy Version <br />
       Redmond, WA 98052-6399<br />
       <abbr title="Phone">P:</abbr>
       425.555.0100
   </address>

   <address>
       <strong>Support:</strong><email mail-to="Support"></email><br />
       <strong>Marketing:</strong><email mail-to="Marketing"></email>
   </address>
   ```

2. Run the app and verify that it generates the correct links.

> **Note**
>
> If you were to write the email tag self-closing (`<email mail-to="Rick" />`), the final output would also be self-closing. To enable the ability to write the tag with only a

start tag (`<email mail-to="Rick">`) you must mark the class with the following:

C#

```
[HtmlTargetElement("email", TagStructure = TagStructure.WithoutEndTag)]
public class EmailVoidTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    // Code removed for brevity
```

With a self-closing email tag helper, the output would be `<a href="mailto:Rick@contoso.com" />`. Self-closing anchor tags are not valid HTML, so you wouldn't want to create one, but you might want to create a tag helper that's self-closing. Tag helpers set the type of the `TagMode` property after reading a tag.

You can also map a different attribute name to a property using the [HtmlAttributeName] attribute.

To map an attribute named `recipient` to the `MailTo` property:

C#

```
[HtmlAttributeName("recipient")]
public string? MailTo { get; set; }
```

Tag Helper for the `recipient` attribute:

HTML

```
<email recipient="…"/>
```

ProcessAsync

In this section, we'll write an asynchronous email helper.

1. Replace the `EmailTagHelper` class with the following code:

C#

```
public class EmailTagHelper : TagHelper
{
    private const string EmailDomain = "contoso.com";
    public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
```

```
    {
        output.TagName = "a";                                    //
Replaces <email> with <a> tag
        var content = await output.GetChildContentAsync();
        var target = content.GetContent() + "@" + EmailDomain;
        output.Attributes.SetAttribute("href", "mailto:" + target);
        output.Content.SetContent(target);
    }
}
```

Notes:

- This version uses the asynchronous `ProcessAsync` method. The asynchronous `GetChildContentAsync` returns a `Task` containing the `TagHelperContent`.
- Use the `output` parameter to get contents of the HTML element.

2. Make the following change to the `Views/Home/Contact.cshtml` file so the tag helper can get the target email.

CSHTML

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>
```

3. Run the app and verify that it generates valid email links.

RemoveAll, PreContent.SetHtmlContent and PostContent.SetHtmlContent

1. Add the following `BoldTagHelper` class to the *TagHelpers* folder.

C#

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = "bold")]
    public class BoldTagHelper : TagHelper
    {
        public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            output.Attributes.RemoveAll("bold");
            output.PreContent.SetHtmlContent("<strong>");
            output.PostContent.SetHtmlContent("</strong>");
        }
    }
}
```

- The `[HtmlTargetElement]` attribute passes an attribute parameter that specifies that any HTML element that contains an HTML attribute named "bold" will match, and the `Process` override method in the class will run. In our sample, the `Process` method removes the "bold" attribute and surrounds the containing markup with `<strong></strong>`.

- Because you don't want to replace the existing tag content, you must write the opening `<strong>` tag with the `PreContent.SetHtmlContent` method and the closing `</strong>` tag with the `PostContent.SetHtmlContent` method.

2. Modify the `About.cshtml` view to contain a `bold` attribute value. The completed code is shown below.

CSHTML

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>
```

3. Run the app. You can use your favorite browser to inspect the source and verify the markup.

   The `[HtmlTargetElement]` attribute above only targets HTML markup that provides an attribute name of "bold". The `<bold>` element wasn't modified by the tag

helper.

4. Comment out the `[HtmlTargetElement]` attribute line and it will default to targeting `<bold>` tags, that is, HTML markup of the form `<bold>`. Remember, the default naming convention will match the class name **Bold**TagHelper to `<bold>` tags.

5. Run the app and verify that the `<bold>` tag is processed by the tag helper.

Decorating a class with multiple `[HtmlTargetElement]` attributes results in a logical-OR of the targets. For example, using the code below, a bold tag or a bold attribute will match.

C#

```
[HtmlTargetElement("bold")]
[HtmlTargetElement(Attributes = "bold")]
public class BoldTagHelper : TagHelper
{
    public override void Process(TagHelperContext context, TagHelperOutput
output)
    {
        output.Attributes.RemoveAll("bold");
        output.PreContent.SetHtmlContent("<strong>");
        output.PostContent.SetHtmlContent("</strong>");
    }
}
```

When multiple attributes are added to the same statement, the runtime treats them as a logical-AND. For example, in the code below, an HTML element must be named "bold" with an attribute named "bold" (`<bold bold />`) to match.

C#

```
[HtmlTargetElement("bold", Attributes = "bold")]
```

You can also use the `[HtmlTargetElement]` to change the name of the targeted element. For example if you wanted the `BoldTagHelper` to target `<MyBold>` tags, you would use the following attribute:

C#

```
[HtmlTargetElement("MyBold")]
```

Pass a model to a Tag Helper

1. Add a *Models* folder.

2. Add the following `WebsiteContext` class to the *Models* folder:

C#

```
using System;

namespace AuthoringTagHelpers.Models
{
    public class WebsiteContext
    {
        public Version Version { get; set; }
        public int CopyrightYear { get; set; }
        public bool Approved { get; set; }
        public int TagsToShow { get; set; }
    }
}
```

3. Add the following `WebsiteInformationTagHelper` class to the *TagHelpers* folder.

C#

```
using System;
using AuthoringTagHelpers.Models;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    public class WebsiteInformationTagHelper : TagHelper
    {
        public WebsiteContext Info { get; set; }

      public override void Process(TagHelperContext context,
TagHelperOutput output)
      {
         output.TagName = "section";
         output.Content.SetHtmlContent(
$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
<li><strong>Copyright Year:</strong> {Info.CopyrightYear}</li>
<li><strong>Approved:</strong> {Info.Approved}</li>
<li><strong>Number of tags to show:</strong> {Info.TagsToShow}</li>
</ul>");
         output.TagMode = TagMode.StartTagAndEndTag;
      }
   }
}
```

- As mentioned previously, tag helpers translates Pascal-cased C# class names and properties for tag helpers into kebab case. Therefore, to use the

`WebsiteInformationTagHelper` in Razor, you'll write `<website-information />`.

- You are not explicitly identifying the target element with the `[HtmlTargetElement]` attribute, so the default of `website-information` will be targeted. If you applied the following attribute (note it's not kebab case but matches the class name):

```
C#

[HtmlTargetElement("WebsiteInformation")]
```

The kebab case tag `<website-information />` wouldn't match. If you want use the `[HtmlTargetElement]` attribute, you would use kebab case as shown below:

```
C#

[HtmlTargetElement("Website-Information")]
```

- Elements that are self-closing have no content. For this example, the Razor markup will use a self-closing tag, but the tag helper will be creating a section element (which isn't self-closing and you are writing content inside the `section` element). Therefore, you need to set `TagMode` to `StartTagAndEndTag` to write output. Alternatively, you can comment out the line setting `TagMode` and write markup with a closing tag. (Example markup is provided later in this tutorial.)

- The `$` (dollar sign) in the following line uses an interpolated string:

```
CSHTML

$@"<ul><li><strong>Version:</strong> {Info.Version}</li>
```

4. Add the following markup to the `About.cshtml` view. The highlighted markup displays the web site information.

```
CSHTML

@using AuthoringTagHelpers.Models
@{
    ViewData["Title"] = "About";
```

```
    WebsiteContext webContext = new WebsiteContext {
                                    Version = new Version(1, 3),
                                    CopyrightYear = 1638,
                                    Approved = true,
                                    TagsToShow = 131 };
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p bold>Use this area to provide additional information.</p>

<bold> Is this bold?</bold>

<h3> web site info </h3>
<website-information info="webContext" />
```

> **Note**
>
> In the Razor markup shown below:
>
> HTML
>
> ```
> <website-information info="webContext" />
> ```
>
> Razor knows the `info` attribute is a class, not a string, and you want to write C# code. Any non-string tag helper attribute should be written without the `@` character.

5. Run the app, and navigate to the About view to see the web site information.

> **Note**
>
> You can use the following markup with a closing tag and remove the line with `TagMode.StartTagAndEndTag` in the tag helper:
>
> HTML
>
> ```
> <website-information info="webContext" >
> </website-information>
> ```

Condition Tag Helper

The condition tag helper renders output when passed a true value.

1. Add the following `ConditionTagHelper` class to the *TagHelpers* folder.

C#

```
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace AuthoringTagHelpers.TagHelpers
{
    [HtmlTargetElement(Attributes = nameof(Condition))]
    public class ConditionTagHelper : TagHelper
    {
        public bool Condition { get; set; }

        public override void Process(TagHelperContext context,
TagHelperOutput output)
        {
            if (!Condition)
            {
                output.SuppressOutput();
            }
        }
    }
}
```

2. Replace the contents of the `Views/Home/Index.cshtml` file with the following markup:

CSHTML

```
@using AuthoringTagHelpers.Models
@model WebsiteContext

@{
    ViewData["Title"] = "Home Page";
}

<div>
    <h3>Information about our website (outdated):</h3>
    <Website-InforMation info="Model" />
    <div condition="Model.Approved">
        <p>
            This website has <strong
surround="em">@Model.Approved</strong> been approved yet.
            Visit www.contoso.com for more information.
        </p>
    </div>
</div>
```

3. Replace the `Index` method in the `Home` controller with the following code:

C#

```
public IActionResult Index(bool approved = false)
{
    return View(new WebsiteContext
    {
        Approved = approved,
        CopyrightYear = 2015,
        Version = new Version(1, 3, 3, 7),
        TagsToShow = 20
    });
}
```

4. Run the app and browse to the home page. The markup in the conditional `div` won't be rendered. Append the query string `?approved=true` to the URL (for example, `http://localhost:1235/Home/Index?approved=true`). `approved` is set to true and the conditional markup will be displayed.

> **Note**
>
> Use the **nameof** operator to specify the attribute to target rather than specifying a string as you did with the bold tag helper:

C#

```
[HtmlTargetElement(Attributes = nameof(Condition))]
 //   [HtmlTargetElement(Attributes = "condition")]
 public class ConditionTagHelper : TagHelper
{
   public bool Condition { get; set; }

   public override void Process(TagHelperContext context,
TagHelperOutput output)
   {
      if (!Condition)
      {
         output.SuppressOutput();
      }
   }
}
```

> The **nameof** operator will protect the code should it ever be refactored (we might want to change the name to `RedCondition`).

Avoid Tag Helper conflicts

In this section, you write a pair of auto-linking tag helpers. The first will replace markup containing a URL starting with HTTP to an HTML anchor tag containing the same URL (and thus yielding a link to the URL). The second will do the same for a URL starting with WWW.

Because these two helpers are closely related and you may refactor them in the future, we'll keep them in the same file.

1. Add the following `AutoLinkerHttpTagHelper` class to the *TagHelpers* folder.

C#

```
[HtmlTargetElement("p")]
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
    {
        var childContent = await output.GetChildContentAsync();
        // Find Urls in the content and replace them with their anchor
tag equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent.GetContent(),
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http
link version}
    }
}
```

> **Note**
>
> The `AutoLinkerHttpTagHelper` class targets `p` elements and uses **Regex** to create the anchor.

2. Add the following markup to the end of the `Views/Home/Contact.cshtml` file:

CSHTML

```
@{
    ViewData["Title"] = "Contact";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<address>
    One Microsoft Way<br />
    Redmond, WA 98052<br />
    <abbr title="Phone">P:</abbr>
```

```
    425.555.0100
</address>

<address>
    <strong>Support:</strong><email>Support</email><br />
    <strong>Marketing:</strong><email>Marketing</email>
</address>

<p>Visit us at http://docs.asp.net or at www.microsoft.com</p>
```

3. Run the app and verify that the tag helper renders the anchor correctly.

4. Update the `AutoLinker` class to include the `AutoLinkerWwwTagHelper` which will convert www text to an anchor tag that also contains the original www text. The updated code is highlighted below:

C#

```
    [HtmlTargetElement("p")]
    public class AutoLinkerHttpTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext
context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their
anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                 childContent.GetContent(),
                 @"\b(?:https?://)(\S+)\b",
                  "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http
link version}
        }
    }

    [HtmlTargetElement("p")]
    public class AutoLinkerWwwTagHelper : TagHelper
    {
        public override async Task ProcessAsync(TagHelperContext
context, TagHelperOutput output)
        {
            var childContent = await output.GetChildContentAsync();
            // Find Urls in the content and replace them with their
anchor tag equivalent.
            output.Content.SetHtmlContent(Regex.Replace(
                childContent.GetContent(),
                 @"\b(www\.)(\S+)\b",
                 "<a target=\"_blank\" href=\"http://$0\">$0</a>"));
// www version
        }
    }
}
```

5. Run the app. Notice the www text is rendered as a link but the HTTP text isn't. If you put a break point in both classes, you can see that the HTTP tag helper class runs first. The problem is that the tag helper output is cached, and when the WWW tag helper is run, it overwrites the cached output from the HTTP tag helper. Later in the tutorial we'll see how to control the order that tag helpers run in. We'll fix the code with the following:

```
C#
```

```
  public class AutoLinkerHttpTagHelper : TagHelper
  {
      public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
      {
          var childContent = output.Content.IsModified ?
output.Content.GetContent() :
              (await output.GetChildContentAsync()).GetContent();

          // Find Urls in the content and replace them with their
anchor tag equivalent.
          output.Content.SetHtmlContent(Regex.Replace(
               childContent,
               @"\b(?:https?://)(\S+)\b",
                "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http
link version}
      }
  }

  [HtmlTargetElement("p")]
  public class AutoLinkerWwwTagHelper : TagHelper
  {
      public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
      {
          var childContent = output.Content.IsModified ?
output.Content.GetContent() :
              (await output.GetChildContentAsync()).GetContent();

          // Find Urls in the content and replace them with their
anchor tag equivalent.
          output.Content.SetHtmlContent(Regex.Replace(
               childContent,
               @"\b(www\.)(\S+)\b",
               "<a target=\"_blank\" href=\"http://$0\">$0</a>"));  //
www version
      }
  }
```

> **Note**

In the first edition of the auto-linking tag helpers, you got the content of the target with the following code:

C#

```
var childContent = await output.GetChildContentAsync();
```

That is, you call `GetChildContentAsync` using the `TagHelperOutput` passed into the `ProcessAsync` method. As mentioned previously, because the output is cached, the last tag helper to run wins. You fixed that problem with the following code:

C#

```
var childContent = output.Content.IsModified ?
output.Content.GetContent() :
    (await output.GetChildContentAsync()).GetContent();
```

The code above checks to see if the content has been modified, and if it has, it gets the content from the output buffer.

6. Run the app and verify that the two links work as expected. While it might appear our auto linker tag helper is correct and complete, it has a subtle problem. If the WWW tag helper runs first, the www links won't be correct. Update the code by adding the `Order` overload to control the order that the tag runs in. The `Order` property determines the execution order relative to other tag helpers targeting the same element. The default order value is zero and instances with lower values are executed first.

   C#

   ```
   public class AutoLinkerHttpTagHelper : TagHelper
   {
       // This filter must run before the AutoLinkerWwwTagHelper as it
   searches and replaces http and
       // the AutoLinkerWwwTagHelper adds http to the markup.
       public override int Order
       {
           get  {  return int.MinValue;   }
       }
   ```

   The preceding code guarantees that the HTTP tag helper runs before the WWW tag helper. Change `Order` to `MaxValue` and verify that the markup generated for the WWW tag is incorrect.

Inspect and retrieve child content

The tag helpers provide several properties to retrieve content.

- The result of `GetChildContentAsync` can be appended to `output.Content`.
- You can inspect the result of `GetChildContentAsync` with `GetContent`.
- If you modify `output.Content`, the TagHelper body won't be executed or rendered unless you call `GetChildContentAsync` as in our auto-linker sample:

C#

```
public class AutoLinkerHttpTagHelper : TagHelper
{
    public override async Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
    {
        var childContent = output.Content.IsModified ?
output.Content.GetContent() :
            (await output.GetChildContentAsync()).GetContent();

        // Find Urls in the content and replace them with their anchor tag
equivalent.
        output.Content.SetHtmlContent(Regex.Replace(
             childContent,
             @"\b(?:https?://)(\S+)\b",
              "<a target=\"_blank\" href=\"$0\">$0</a>"));  // http link
version}
    }
}
```

- Multiple calls to `GetChildContentAsync` returns the same value and doesn't re-execute the `TagHelper` body unless you pass in a false parameter indicating not to use the cached result.

Load minified partial view TagHelper

In production environments, performance can be improved by loading minified partial views. To take advantage of minified partial view in production:

- Create/set up a pre-build process that minifies partial views.
- Use the following code to load minified partial views in non-development environments.

C#

```
public class MinifiedVersionPartialTagHelper : PartialTagHelper
    {
        public MinifiedVersionPartialTagHelper(ICompositeViewEngine
viewEngine,
                                    IViewBufferScope viewBufferScope)
                                   : base(viewEngine, viewBufferScope)
        {

        }

        public override Task ProcessAsync(TagHelperContext context,
TagHelperOutput output)
        {
            // Append ".min" to load the minified partial view.
            if (!IsDevelopment())
            {
                Name += ".min";
            }

            return base.ProcessAsync(context, output);
        }

        private bool IsDevelopment()
        {
            return
Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")
                                                    ==
EnvironmentName.Development;
        }
    }
```

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

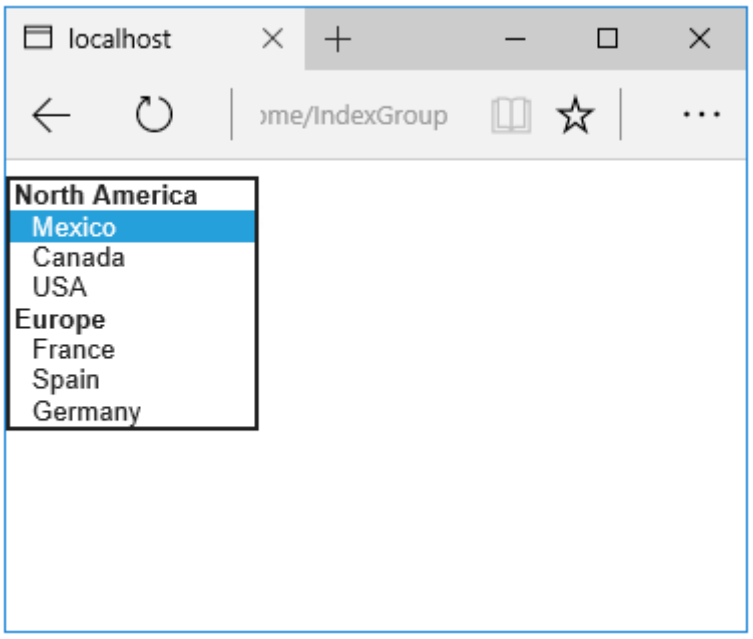
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Tag Helper Components in ASP.NET Core

Article • 06/03/2022

By Scott Addie and Fiyaz Bin Hasan

A Tag Helper Component is a Tag Helper that allows you to conditionally modify or add HTML elements from server-side code. This feature is available in ASP.NET Core 2.0 or later.

ASP.NET Core includes two built-in Tag Helper Components: `head` and `body`. They're located in the Microsoft.AspNetCore.Mvc.Razor.TagHelpers namespace and can be used in both MVC and Razor Pages. Tag Helper Components don't require registration with the app in `_ViewImports.cshtml`.

View or download sample code (how to download)

Use cases

Two common use cases of Tag Helper Components include:

1. Injecting a <link> into the <head>.
2. Injecting a <script> into the <body>.

The following sections describe these use cases.

Inject into HTML head element

Inside the HTML `<head>` element, CSS files are commonly imported with the HTML `<link>` element. The following code injects a `<link>` element into the `<head>` element using the `head` Tag Helper Component:

C#

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressStyleTagHelperComponent : TagHelperComponent
    {
        private readonly string _style =
```

```
            @"<link rel=""stylesheet"" href=""/css/address.css"" />";

        public override int Order => 1;

        public override Task ProcessAsync(TagHelperContext context,
                                          TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "head",
                              StringComparison.OrdinalIgnoreCase))
            {
                output.PostContent.AppendHtml(_style);
            }

            return Task.CompletedTask;
        }
    }
}
```

In the preceding code:

- `AddressStyleTagHelperComponent` implements TagHelperComponent. The abstraction:
  - Allows initialization of the class with a TagHelperContext.
  - Enables the use of Tag Helper Components to add or modify HTML elements.
- The Order property defines the order in which the Components are rendered. `Order` is necessary when there are multiple usages of Tag Helper Components in an app.
- ProcessAsync compares the execution context's TagName property value to `head`. If the comparison evaluates to true, the content of the `_style` field is injected into the HTML `<head>` element.

Inject into HTML body element

The `body` Tag Helper Component can inject a `<script>` element into the `<body>` element. The following code demonstrates this technique:

C#

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Razor.TagHelpers;

namespace RazorPagesSample.TagHelpers
{
    public class AddressScriptTagHelperComponent : TagHelperComponent
    {
```

```
        public override int Order => 2;

        public override async Task ProcessAsync(TagHelperContext context,
                                                TagHelperOutput output)
        {
            if (string.Equals(context.TagName, "body",
                              StringComparison.OrdinalIgnoreCase))
            {
                var script = await File.ReadAllTextAsync(
                    "TagHelpers/Templates/AddressToolTipScript.html");
                output.PostContent.AppendHtml(script);
            }
        }
    }
}
```

A separate HTML file is used to store the `<script>` element. The HTML file makes the code cleaner and more maintainable. The preceding code reads the contents of `TagHelpers/Templates/AddressToolTipScript.html` and appends it with the Tag Helper output. The `AddressToolTipScript.html` file includes the following markup:

HTML

```
<script>
$("address[printable]").hover(function() {
    $(this).attr({
        "data-toggle": "tooltip",
        "data-placement": "right",
        "title": "Home of Microsoft!"
    });
});
</script>
```

The preceding code binds a Bootstrap tooltip widget to any `<address>` element that includes a `printable` attribute. The effect is visible when a mouse pointer hovers over the element.

Register a Component

A Tag Helper Component must be added to the app's Tag Helper Components collection. There are three ways to add to the collection:

- Registration via services container
- Registration via Razor file
- Registration via Page Model or controller

Registration via services container

If the Tag Helper Component class isn't managed with ITagHelperComponentManager, it must be registered with the dependency injection (DI) system. The following `Startup.ConfigureServices` code registers the `AddressStyleTagHelperComponent` and `AddressScriptTagHelperComponent` classes with a transient lifetime:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddTransient<ITagHelperComponent,
        AddressScriptTagHelperComponent>();
    services.AddTransient<ITagHelperComponent,
        AddressStyleTagHelperComponent>();
}
```

Registration via Razor file

If the Tag Helper Component isn't registered with DI, it can be registered from a Razor Pages page or an MVC view. This technique is used for controlling the injected markup and the component execution order from a Razor file.

`ITagHelperComponentManager` is used to add Tag Helper Components or remove them from the app. The following code demonstrates this technique with `AddressTagHelperComponent`:

CSHTML

```
@using RazorPagesSample.TagHelpers;
@using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
@inject ITagHelperComponentManager manager;

@{
    string markup;

    if (Model.IsWeekend)
    {
        markup = "<em class='text-warning'>Office closed today!</em>";
```

```
    }
    else
    {
        markup = "<em class='text-info'>Office open today!</em>";
    }

    manager.Components.Add(new AddressTagHelperComponent(markup, 1));
}
```

In the preceding code:

- The `@inject` directive provides an instance of `ITagHelperComponentManager`. The instance is assigned to a variable named `manager` for access downstream in the Razor file.
- An instance of `AddressTagHelperComponent` is added to the app's Tag Helper Components collection.

`AddressTagHelperComponent` is modified to accommodate a constructor that accepts the `markup` and `order` parameters:

C#

```
private readonly string _markup;

public override int Order { get; }

public AddressTagHelperComponent(string markup = "", int order = 1)
{
    _markup = markup;
    Order = order;
}
```

The provided `markup` parameter is used in `ProcessAsync` as follows:

C#

```
public override async Task ProcessAsync(TagHelperContext context,
                                        TagHelperOutput output)
{
    if (string.Equals(context.TagName, "address",
            StringComparison.OrdinalIgnoreCase) &&
        output.Attributes.ContainsName("printable"))
    {
        TagHelperContent childContent = await output.GetChildContentAsync();
        string content = childContent.GetContent();
        output.Content.SetHtmlContent(
            $"<div>{content}<br>{_markup}</div>{_printableButton}");
```

```
    }
}
```

Registration via Page Model or controller

If the Tag Helper Component isn't registered with DI, it can be registered from a Razor Pages page model or an MVC controller. This technique is useful for separating C# logic from Razor files.

Constructor injection is used to access an instance of `ITagHelperComponentManager`. The Tag Helper Component is added to the instance's Tag Helper Components collection. The following Razor Pages page model demonstrates this technique with `AddressTagHelperComponent`:

C#

```
using System;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.RazorPages;
using RazorPagesSample.TagHelpers;

public class IndexModel : PageModel
{
    private readonly ITagHelperComponentManager _tagHelperComponentManager;

    public bool IsWeekend
    {
        get
        {
            var dayOfWeek = DateTime.Now.DayOfWeek;

            return dayOfWeek == DayOfWeek.Saturday ||
                   dayOfWeek == DayOfWeek.Sunday;
        }
    }

    public IndexModel(ITagHelperComponentManager tagHelperComponentManager)
    {
        _tagHelperComponentManager = tagHelperComponentManager;
    }

    public void OnGet()
    {
        string markup;

        if (IsWeekend)
        {
            markup = "<em class='text-warning'>Office closed today!</em>";
        }
        else
```

```
        {
            markup = "<em class='text-info'>Office open today!</em>";
        }

        _tagHelperComponentManager.Components.Add(
            new AddressTagHelperComponent(markup, 1));
    }
}
```

In the preceding code:

- Constructor injection is used to access an instance of `ITagHelperComponentManager`.
- An instance of `AddressTagHelperComponent` is added to the app's Tag Helper Components collection.

Create a Component

To create a custom Tag Helper Component:

- Create a public class deriving from TagHelperComponentTagHelper.
- Apply an [HtmlTargetElement] attribute to the class. Specify the name of the target HTML element.
- *Optional*: Apply an [EditorBrowsable(EditorBrowsableState.Never)] attribute to the class to suppress the type's display in IntelliSense.

The following code creates a custom Tag Helper Component that targets the `<address>` HTML element:

C#

```
using System.ComponentModel;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Microsoft.Extensions.Logging;

namespace RazorPagesSample.TagHelpers
{
    [HtmlTargetElement("address")]
    [EditorBrowsable(EditorBrowsableState.Never)]
    public class AddressTagHelperComponentTagHelper :
TagHelperComponentTagHelper
    {
        public AddressTagHelperComponentTagHelper(
            ITagHelperComponentManager componentManager,
            ILoggerFactory loggerFactory) : base(componentManager,
loggerFactory)
        {
        }
```

```
    }
}
```

Use the custom `address` Tag Helper Component to inject HTML markup as follows:

C#

```
public class AddressTagHelperComponent : TagHelperComponent
{
    private readonly string _printableButton =
        "<button type='button' class='btn btn-info' onclick=\"window.open("
+
        "'https://binged.it/2AXRRYw')\">" +
        "<span class='glyphicon glyphicon-road' aria-hidden='true'></span>"
+
        "</button>";

    public override int Order => 3;

    public override async Task ProcessAsync(TagHelperContext context,
                                            TagHelperOutput output)
    {
        if (string.Equals(context.TagName, "address",
                StringComparison.OrdinalIgnoreCase) &&
            output.Attributes.ContainsName("printable"))
        {
            var content = await output.GetChildContentAsync();
            output.Content.SetHtmlContent(
                $"<div>{content.GetContent()}</div>{_printableButton}");
        }
    }
}
```

The preceding `ProcessAsync` method injects the HTML provided to SetHtmlContent into the matching `<address>` element. The injection occurs when:

- The execution context's `TagName` property value equals `address`.
- The corresponding `<address>` element has a `printable` attribute.

For example, the `if` statement evaluates to true when processing the following `<address>` element:

CSHTML

```
<address printable>
    One Microsoft Way<br />
    Redmond, WA 98052-6399<br />
    <abbr title="Phone">P:</abbr>
```

```
    425.555.0100
</address>
```

Additional resources

- Dependency injection in ASP.NET Core
- Dependency injection into views in ASP.NET Core
- ASP.NET Core built-in Tag Helpers

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Anchor Tag Helper in ASP.NET Core

Article • 06/03/2022

By Peter Kellner and Scott Addie

The Anchor Tag Helper enhances the standard HTML anchor (`<a ... ></a>`) tag by adding new attributes. By convention, the attribute names are prefixed with `asp-`. The rendered anchor element's `href` attribute value is determined by the values of the `asp-` attributes.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

View or download sample code (how to download)

SpeakerController is used in samples throughout this document:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;

public class SpeakerController : Controller
{
    private List<Speaker> Speakers =
        new List<Speaker>
        {
            new Speaker {SpeakerId = 10},
            new Speaker {SpeakerId = 11},
            new Speaker {SpeakerId = 12}
        };

    [Route("Speaker/{id:int}")]
    public IActionResult Detail(int id) =>
        View(Speakers.FirstOrDefault(a => a.SpeakerId == id));

    [Route("/Speaker/Evaluations",
           Name = "speakerevals")]
    public IActionResult Evaluations() => View();

    [Route("/Speaker/EvaluationsCurrent",
           Name = "speakerevalscurrent")]
    public IActionResult Evaluations(
        int speakerId,
        bool currentYear) => View();

    public IActionResult Index() => View(Speakers);
}

public class Speaker
```

```
{
    public int SpeakerId { get; set; }
}
```

Anchor Tag Helper attributes

asp-controller

The asp-controller attribute assigns the controller used for generating the URL. The following markup lists all speakers:

CSHTML

```
<a asp-controller="Speaker"
   asp-action="Index">All Speakers</a>
```

The generated HTML:

HTML

```
<a href="/Speaker">All Speakers</a>
```

If the `asp-controller` attribute is specified and `asp-action` isn't, the default `asp-action` value is the controller action associated with the currently executing view. If `asp-action` is omitted from the preceding markup, and the Anchor Tag Helper is used in *HomeController*'s *Index* view (*/Home*), the generated HTML is:

HTML

```
<a href="/Home">All Speakers</a>
```

asp-action

The asp-action attribute value represents the controller action name included in the generated `href` attribute. The following markup sets the generated `href` attribute value to the speaker evaluations page:

CSHTML

```
<a asp-controller="Speaker"
   asp-action="Evaluations">Speaker Evaluations</a>
```

The generated HTML:

```HTML
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

If no `asp-controller` attribute is specified, the default controller calling the view executing the current view is used.

If the `asp-action` attribute value is `Index`, then no action is appended to the URL, leading to the invocation of the default `Index` action. The action specified (or defaulted), must exist in the controller referenced in `asp-controller`.

asp-route-{value}

The asp-route-{value} attribute enables a wildcard route prefix. Any value occupying the `{value}` placeholder is interpreted as a potential route parameter. If a default route isn't found, this route prefix is appended to the generated `href` attribute as a request parameter and value. Otherwise, it's substituted in the route template.

Consider the following controller action:

```C#
private List<Speaker> Speakers =
    new List<Speaker>
    {
        new Speaker {SpeakerId = 10},
        new Speaker {SpeakerId = 11},
        new Speaker {SpeakerId = 12}
    };

[Route("Speaker/{id:int}")]
public IActionResult Detail(int id) =>
    View(Speakers.FirstOrDefault(a => a.SpeakerId == id));
```

With a default route template defined in *Startup.Configure*:

```C#
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
                    template: "
{area:exists}/{controller=Home}/{action=Index}");
```

```
    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

The MVC view uses the model, provided by the action, as follows:

CSHTML

```
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-id="@Model.SpeakerId">SpeakerId: @Model.SpeakerId</a>
</body>
</html>
```

The default route's `{id?}` placeholder was matched. The generated HTML:

HTML

```
<a href="/Speaker/Detail/12">SpeakerId: 12</a>
```

Assume the route prefix isn't part of the matching routing template, as with the following MVC view:

CSHTML

```
@model Speaker
<!DOCTYPE html>
<html>
<body>
    <a asp-controller="Speaker"
       asp-action="Detail"
       asp-route-speakerid="@Model.SpeakerId">SpeakerId:
@Model.SpeakerId</a>
<body>
</html>
```

The following HTML is generated because `speakerid` wasn't found in the matching route:

HTML

```
<a href="/Speaker/Detail?speakerid=12">SpeakerId: 12</a>
```

If either `asp-controller` or `asp-action` aren't specified, then the same default processing is followed as is in the `asp-route` attribute.

asp-route

The asp-route attribute is used for creating a URL linking directly to a named route. Using routing attributes, a route can be named as shown in the `SpeakerController` and used in its `Evaluations` action:

C#

```
[Route("/Speaker/Evaluations",
       Name = "speakerevals")]
```

In the following markup, the `asp-route` attribute references the named route:

CSHTML

```
<a asp-route="speakerevals">Speaker Evaluations</a>
```

The Anchor Tag Helper generates a route directly to that controller action using the URL */Speaker/Evaluations*. The generated HTML:

HTML

```
<a href="/Speaker/Evaluations">Speaker Evaluations</a>
```

If `asp-controller` or `asp-action` is specified in addition to `asp-route`, the route generated may not be what you expect. To avoid a route conflict, `asp-route` shouldn't be used with the `asp-controller` and `asp-action` attributes.

asp-all-route-data

The asp-all-route-data attribute supports the creation of a dictionary of key-value pairs. The key is the parameter name, and the value is the parameter value.

In the following example, a dictionary is initialized and passed to a Razor view. Alternatively, the data could be passed in with your model.

CSHTML

```
@{
var parms = new Dictionary<string, string>
            {
                { "speakerId", "11" },
                { "currentYear", "true" }
            };
}

<a asp-route="speakerevalscurrent"
   asp-all-route-data="parms">Speaker Evaluations</a>
```

The preceding code generates the following HTML:

HTML

```
<a href="/Speaker/EvaluationsCurrent?speakerId=11&currentYear=true">Speaker
Evaluations</a>
```

The `asp-all-route-data` dictionary is flattened to produce a querystring meeting the requirements of the overloaded `Evaluations` action:

C#

```
public IActionResult Evaluations() => View();

[Route("/Speaker/EvaluationsCurrent",
       Name = "speakerevalscurrent")]
public IActionResult Evaluations(
```

If any keys in the dictionary match route parameters, those values are substituted in the route as appropriate. The other non-matching values are generated as request parameters.

asp-fragment

The asp-fragment attribute defines a URL fragment to append to the URL. The Anchor Tag Helper adds the hash character (#). Consider the following markup:

CSHTML

```
<a asp-controller="Speaker"
   asp-action="Evaluations"
   asp-fragment="SpeakerEvaluations">Speaker Evaluations</a>
```

The generated HTML:

```HTML
<a href="/Speaker/Evaluations#SpeakerEvaluations">Speaker Evaluations</a>
```

Hash tags are useful when building client-side apps. They can be used for easy marking and searching in JavaScript, for example.

asp-area

The asp-area attribute sets the area name used to set the appropriate route. The following examples depict how the `asp-area` attribute causes a remapping of routes.

Usage in Razor Pages

Razor Pages areas are supported in ASP.NET Core 2.1 or later.

Consider the following directory hierarchy:

- **{Project name}**
  - **wwwroot**
  - **Areas**
    - **Sessions**
      - **Pages**
        - *_ViewStart.cshtml*
        - `Index.cshtml`
        - `Index.cshtml.cs`
  - **Pages**

The markup to reference the *Sessions* area *Index* Razor Page is:

```CSHTML
<a asp-area="Sessions"
   asp-page="/Index">View Sessions</a>
```

The generated HTML:

```HTML
<a href="/Sessions">View Sessions</a>
```

> **Tip**
>
> To support areas in a Razor Pages app, do one of the following in `Startup.ConfigureServices`:
>
> - Set the compatibility version to 2.1 or later.
> - Set the RazorPagesOptions.AllowAreas property to `true`:
>
> ```
> C#
>
> services.AddMvc()
>         .AddRazorPagesOptions(options => options.AllowAreas =
> true);
> ```

Usage in MVC

Consider the following directory hierarchy:

- **{Project name}**
  - **wwwroot**
  - **Areas**
    - **Blogs**
      - **Controllers**
        - `HomeController.cs`
      - **Views**
        - **Home**
          - `AboutBlog.cshtml`
          - `Index.cshtml`
        - *_ViewStart.cshtml*
  - **Controllers**

Setting `asp-area` to "Blogs" prefixes the directory *Areas/Blogs* to the routes of the associated controllers and views for this anchor tag. The markup to reference the *AboutBlog* view is:

```
CSHTML

<a asp-area="Blogs"
   asp-controller="Home"
   asp-action="AboutBlog">About Blog</a>
```

The generated HTML:

```HTML
<a href="/Blogs/Home/AboutBlog">About Blog</a>
```

> **Tip**
>
> To support areas in an MVC app, the route template must include a reference to the area, if it exists. That template is represented by the second parameter of the `routes.MapRoute` method call in *Startup.Configure*:

```C#
app.UseMvc(routes =>
{
    // need route and attribute on controller: [Area("Blogs")]
    routes.MapRoute(name: "mvcAreaRoute",
                    template: "
{area:exists}/{controller=Home}/{action=Index}");

    // default route for non-areas
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

asp-protocol

The asp-protocol attribute is for specifying a protocol (such as `https`) in your URL. For example:

```CSHTML
<a asp-protocol="https"
   asp-controller="Home"
   asp-action="About">About</a>
```

The generated HTML:

```HTML
<a href="https://localhost/Home/About">About</a>
```

The host name in the example is localhost. The Anchor Tag Helper uses the website's public domain when generating the URL.

asp-host

The asp-host attribute is for specifying a host name in your URL. For example:

CSHTML

```
<a asp-protocol="https"
   asp-host="microsoft.com"
   asp-controller="Home"
   asp-action="About">About</a>
```

The generated HTML:

HTML

```
<a href="https://microsoft.com/Home/About">About</a>
```

asp-page

The asp-page attribute is used with Razor Pages. Use it to set an anchor tag's `href` attribute value to a specific page. Prefixing the page name with `/` creates a URL for a matching page from the root of the app:

With the sample code, the following markup creates a link to the attendee Razor Page:

CSHTML

```
<a asp-page="/Attendee">All Attendees</a>
```

The generated HTML:

HTML

```
<a href="/Attendee">All Attendees</a>
```

The `asp-page` attribute is mutually exclusive with the `asp-route`, `asp-controller`, and `asp-action` attributes. However, `asp-page` can be used with `asp-route-{value}` to control routing, as the following markup demonstrates:

CSHTML

```
<a asp-page="/Attendee"
   asp-route-attendeeid="10">View Attendee</a>
```

The generated HTML:

HTML

```
<a href="/Attendee?attendeeid=10">View Attendee</a>
```

If the referenced page doesn't exist, a link to the current page is generated using an ambient value from the request. No warning is indicated, except at the debug log level.

asp-page-handler

The asp-page-handler attribute is used with Razor Pages. It's intended for linking to specific page handlers.

Consider the following page handler:

C#

```
public void OnGetProfile(int attendeeId)
{
    ViewData["AttendeeId"] = attendeeId;

    // code omitted for brevity
}
```

The page model's associated markup links to the `OnGetProfile` page handler. Note the `On<Verb>` prefix of the page handler method name is omitted in the `asp-page-handler` attribute value. When the method is asynchronous, the `Async` suffix is omitted, too.

CSHTML

```
<a asp-page="/Attendee"
   asp-page-handler="Profile"
   asp-route-attendeeid="12">Attendee Profile</a>
```

The generated HTML:

HTML

```
<a href="/Attendee?attendeeid=12&handler=Profile">Attendee Profile</a>
```

Additional resources

- Areas in ASP.NET Core
- Introduction to Razor Pages in ASP.NET Core
- Compatibility version for ASP.NET Core MVC

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Cache Tag Helper in ASP.NET Core MVC

Article • 06/03/2022

By Peter Kellner

The Cache Tag Helper provides the ability to improve the performance of your ASP.NET Core app by caching its content to the internal ASP.NET Core cache provider.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

The following Razor markup caches the current date:

CSHTML

```
<cache>@DateTime.Now</cache>
```

The first request to the page that contains the Tag Helper displays the current date. Additional requests show the cached value until the cache expires (default 20 minutes) or until the cached date is evicted from the cache.

Cache Tag Helper Attributes

enabled

Expand table

Attribute Type	Examples	Default
Boolean	`true`, `false`	`true`

`enabled` determines if the content enclosed by the Cache Tag Helper is cached. The default is `true`. If set to `false`, the rendered output is **not** cached.

Example:

CSHTML

```
<cache enabled="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-on

Expand table

Attribute Type	Example
`DateTimeOffset`	`@new DateTime(2025,1,29,17,02,0)`

`expires-on` sets an absolute expiration date for the cached item.

The following example caches the contents of the Cache Tag Helper until 5:02 PM on January 29, 2025:

CSHTML

```
<cache expires-on="@new DateTime(2025,1,29,17,02,0)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

expires-after

Expand table

Attribute Type	Example	Default
`TimeSpan`	`@TimeSpan.FromSeconds(120)`	20 minutes

`expires-after` sets the length of time from the first request time to cache the contents.

Example:

CSHTML

```
<cache expires-after="@TimeSpan.FromSeconds(120)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

The Razor View Engine sets the default `expires-after` value to twenty minutes.

expires-sliding

Expand table

Attribute Type	Example
`TimeSpan`	`@TimeSpan.FromSeconds(60)`

Sets the time that a cache entry should be evicted if its value hasn't been accessed.

Example:

CSHTML

```
<cache expires-sliding="@TimeSpan.FromSeconds(60)">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-header

Expand table

Attribute Type	Examples
String	`User-Agent`, `User-Agent,content-encoding`

`vary-by-header` accepts a comma-delimited list of header values that trigger a cache refresh when they change.

The following example monitors the header value `User-Agent`. The example caches the content for every different `User-Agent` presented to the web server:

CSHTML

```
<cache vary-by-header="User-Agent">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-query

Expand table

Attribute Type	Examples
String	`Make`, `Make,Model`

`vary-by-query` accepts a comma-delimited list of Keys in a query string (Query) that trigger a cache refresh when the value of any listed key changes.

The following example monitors the values of `Make` and `Model`. The example caches the content for every different `Make` and `Model` presented to the web server:

CSHTML

```
<cache vary-by-query="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-route

Expand table

Attribute Type	Examples
String	`Make`, `Make,Model`

`vary-by-route` accepts a comma-delimited list of route parameter names that trigger a cache refresh when the route data parameter value changes.

Example:

`Startup.cs`:

C#

```
routes.MapRoute(
    name: "default",
    template: "{controller=Home}/{action=Index}/{Make?}/{Model?}");
```

`Index.cshtml`:

CSHTML

```
<cache vary-by-route="Make,Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-cookie

Expand table

Attribute Type	Examples
String	`.AspNetCore.Identity.Application`, `.AspNetCore.Identity.Application,HairColor`

`vary-by-cookie` accepts a comma-delimited list of cookie names that trigger a cache refresh when the cookie values change.

The following example monitors the cookie associated with ASP.NET Core Identity. When a user is authenticated, a change in the Identity cookie triggers a cache refresh:

CSHTML

```
<cache vary-by-cookie=".AspNetCore.Identity.Application">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

vary-by-user

Expand table

Attribute Type	Examples	Default
Boolean	`true`, `false`	`true`

`vary-by-user` specifies whether or not the cache resets when the signed-in user (or Context Principal) changes. The current user is also known as the Request Context Principal and can be viewed in a Razor view by referencing `@User.Identity.Name`.

The following example monitors the current logged in user to trigger a cache refresh:

CSHTML

```
<cache vary-by-user="true">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

Using this attribute maintains the contents in cache through a sign-in and sign-out cycle. When the value is set to `true`, an authentication cycle invalidates the cache for the authenticated user. The cache is invalidated because a new unique cookie value is generated when a user is authenticated. Cache is maintained for the anonymous state

when no cookie is present or the cookie has expired. If the user is **not** authenticated, the cache is maintained.

vary-by

Expand table

Attribute Type	Example
String	`@Model`

`vary-by` allows for customization of what data is cached. When the object referenced by the attribute's string value changes, the content of the Cache Tag Helper is updated. Often, a string-concatenation of model values are assigned to this attribute. Effectively, this results in a scenario where an update to any of the concatenated values invalidates the cache.

The following example assumes the controller method rendering the view sums the integer value of the two route parameters, `myParam1` and `myParam2`, and returns the sum as the single model property. When this sum changes, the content of the Cache Tag Helper is rendered and cached again.

Action:

C#

```
public IActionResult Index(string myParam1, string myParam2, string
myParam3)
{
    int num1;
    int num2;
    int.TryParse(myParam1, out num1);
    int.TryParse(myParam2, out num2);
    return View(viewName, num1 + num2);
}
```

`Index.cshtml`:

CSHTML

```
<cache vary-by="@Model">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

priority

Expand table

Attribute Type	Examples	Default
`CacheItemPriority`	`High`, `Low`, `NeverRemove`, `Normal`	`Normal`

`priority` provides cache eviction guidance to the built-in cache provider. The web server evicts `Low` cache entries first when it's under memory pressure.

Example:

CSHTML

```
<cache priority="High">
    Current Time Inside Cache Tag Helper: @DateTime.Now
</cache>
```

The `priority` attribute doesn't guarantee a specific level of cache retention. `CacheItemPriority` is only a suggestion. Setting this attribute to `NeverRemove` doesn't guarantee that cached items are always retained. See the topics in the Additional Resources section for more information.

The Cache Tag Helper is dependent on the memory cache service. The Cache Tag Helper adds the service if it hasn't been added.

Additional resources

- Cache in-memory in ASP.NET Core
- Introduction to Identity on ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Component Tag Helper in ASP.NET Core

Article • 11/15/2024

The Component Tag Helper renders a Razor component in a Razor Pages page or MVC view.

Prerequisites

Follow the guidance in the *Use non-routable components in pages or views* section of the Integrate ASP.NET Core Razor components article.

Component Tag Helper

To render a component from a page or view, use the Component Tag Helper (`<component>` tag).

RenderMode configures whether the component:

- Is prerendered into the page.
- Is rendered as static HTML on the page or if it includes the necessary information to bootstrap a Blazor app from the user agent.

Blazor WebAssembly app render modes are shown in the following table.

Expand table

Render Mode	Description
`WebAssembly`	Renders a marker for a Blazor WebAssembly app for use to include an interactive component when loaded in the browser. The component isn't prerendered. This option makes it easier to render different Blazor WebAssembly components on different pages.
`WebAssemblyPrerendered`	Prerenders the component into static HTML and includes a marker for a Blazor WebAssembly app for later use to make the component interactive when loaded in the browser.

Render modes are shown in the following table.

Expand table

Render Mode	Description
ServerPrerendered	Renders the component into static HTML and includes a marker for a server-side Blazor app. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Server	Renders a marker for a server-side Blazor app. Output from the component isn't included. When the user-agent starts, this marker is used to bootstrap a Blazor app.
Static	Renders the component into static HTML.

Additional characteristics include:

- Multiple Component Tag Helpers rendering multiple Razor components is allowed.
- Components can't be dynamically rendered after the app has started.
- While pages and views can use components, the converse isn't true. Components can't use view- and page-specific features, such as partial views and sections. To use logic from a partial view in a component, factor out the partial view logic into a component.
- Rendering server components from a static HTML page isn't supported.

The following Component Tag Helper renders the `EmbeddedCounter` component in a page or view in a server-side Blazor app with `ServerPrerendered`:

CSHTML

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}.Components

...

<component type="typeof(EmbeddedCounter)" render-mode="ServerPrerendered" />
```

The preceding example assumes that the `EmbeddedCounter` component is in the app's `Components` folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample.Components`).

The Component Tag Helper can also pass parameters to components. Consider the following `ColorfulCheckbox` component that sets the checkbox label's color and size.

`Components/ColorfulCheckbox.razor`:

razor

```
<label style="font-size:@(Size)px;color:@Color">
    <input @bind="Value"
           id="survey"
           name="blazor"
           type="checkbox" />
    Enjoying Blazor?
</label>

@code {
    [Parameter]
    public bool Value { get; set; }

    [Parameter]
    public int Size { get; set; } = 8;

    [Parameter]
    public string? Color { get; set; }

    protected override void OnInitialized()
    {
        Size += 10;
    }
}
```

The `Size` (`int`) and `Color` (`string`) component parameters can be set by the Component Tag Helper:

CSHTML

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}.Components

...

<component type="typeof(ColorfulCheckbox)" render-mode="ServerPrerendered"
    param-Size="14" param-Color="@("blue")" />
```

The preceding example assumes that the `ColorfulCheckbox` component is in the `Components` folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample.Components`).

The following HTML is rendered in the page or view:

HTML

```
<label style="font-size:24px;color:blue">
    <input id="survey" name="blazor" type="checkbox">
```

```
    Enjoying Blazor?
</label>
```

Passing a quoted string requires an explicit Razor expression, as shown for `param-Color` in the preceding example. The Razor parsing behavior for a `string` type value doesn't apply to a `param-*` attribute because the attribute is an `object` type.

All types of parameters are supported, except:

- Generic parameters.
- Non-serializable parameters.
- Inheritance in collection parameters.
- Parameters whose type is defined outside of the Blazor WebAssembly app or within a lazily-loaded assembly.
- For receiving a RenderFragment delegate for child content (for example, `param-ChildContent="..."`). For this scenario, we recommend creating a Razor component (`.razor`) that references the component you want to render with the child content you want to pass and then invoke the Razor component from the page or view with the Component Tag Helper.

The parameter type must be JSON serializable, which typically means that the type must have a default constructor and settable properties. For example, you can specify a value for `Size` and `Color` in the preceding example because the types of `Size` and `Color` are primitive types (`int` and `string`), which are supported by the JSON serializer.

In the following example, a class object is passed to the component:

`MyClass.cs`:

C#

```
public class MyClass
{
    public MyClass()
    {
    }

    public int MyInt { get; set; } = 999;
    public string MyString { get; set; } = "Initial value";
}
```

The class must have a public parameterless constructor.

`Components/ParameterComponent.razor`:

razor

```razor
<h2>ParameterComponent</h2>

<p>Int: @MyObject?.MyInt</p>
<p>String: @MyObject?.MyString</p>

@code
{
    [Parameter]
    public MyClass? MyObject { get; set; }
}
```

`Pages/MyPage.cshtml`:

CSHTML

```cshtml
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using {APP ASSEMBLY}
@using {APP ASSEMBLY}.Components

...

@{
    var myObject = new MyClass();
    myObject.MyInt = 7;
    myObject.MyString = "Set by MyPage";
}

<component type="typeof(ParameterComponent)" render-mode="ServerPrerendered"
    param-MyObject="@myObject" />
```

The preceding example assumes that the `ParameterComponent` component is in the app's `Components` folder. The placeholder `{APP ASSEMBLY}` is the app's assembly name (for example, `@using BlazorSample` and `@using BlazorSample.Components`). `MyClass` is in the app's namespace.

Additional resources

- Persist Component State Tag Helper in ASP.NET Core
- Prerender ASP.NET Core Razor components
- ComponentTagHelper
- Tag Helpers in ASP.NET Core
- ASP.NET Core Razor components

Distributed Cache Tag Helper in ASP.NET Core

Article • 06/03/2022

By Peter Kellner

The Distributed Cache Tag Helper provides the ability to dramatically improve the performance of your ASP.NET Core app by caching its content to a distributed cache source.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

The Distributed Cache Tag Helper inherits from the same base class as the Cache Tag Helper. All of the Cache Tag Helper attributes are available to the Distributed Tag Helper.

The Distributed Cache Tag Helper uses constructor injection. The IDistributedCache interface is passed into the Distributed Cache Tag Helper's constructor. If no concrete implementation of `IDistributedCache` is created in `Startup.ConfigureServices` (`Startup.cs`), the Distributed Cache Tag Helper uses the same in-memory provider for storing cached data as the Cache Tag Helper.

Distributed Cache Tag Helper Attributes

Attributes shared with the Cache Tag Helper

- `enabled`
- `expires-on`
- `expires-after`
- `expires-sliding`
- `vary-by-header`
- `vary-by-query`
- `vary-by-route`
- `vary-by-cookie`
- `vary-by-user`
- `vary-by`
- `priority`

The Distributed Cache Tag Helper inherits from the same class as Cache Tag Helper. For descriptions of these attributes, see the Cache Tag Helper.

name

Expand table

Attribute Type	Example
String	`my-distributed-cache-unique-key-101`

`name` is required. The `name` attribute is used as a key for each stored cache instance. Unlike the Cache Tag Helper that assigns a cache key to each instance based on the Razor page name and location in the Razor page, the Distributed Cache Tag Helper only bases its key on the attribute `name`.

Example:

CSHTML

```
<distributed-cache name="my-distributed-cache-unique-key-101">
    Time Inside Cache Tag Helper: @DateTime.Now
</distributed-cache>
```

Distributed Cache Tag Helper IDistributedCache implementations

There are two implementations of IDistributedCache built in to ASP.NET Core. One is based on SQL Server, and the other is based on Redis. Third-party implementations are also available, such as NCache. Details of these implementations can be found at Distributed caching in ASP.NET Core. Both implementations involve setting an instance of `IDistributedCache` in `Startup`.

There are no tag attributes specifically associated with using any specific implementation of `IDistributedCache`.

Additional resources

- Cache Tag Helper in ASP.NET Core MVC
- Dependency injection in ASP.NET Core
- Distributed caching in ASP.NET Core
- Cache in-memory in ASP.NET Core
- Introduction to Identity on ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Environment Tag Helper in ASP.NET Core

Article • 06/03/2022

By Peter Kellner and Hisham Bin Ateya

The Environment Tag Helper conditionally renders its enclosed content based on the current hosting environment. The Environment Tag Helper's single attribute, `names`, is a comma-separated list of environment names. If any of the provided environment names match the current environment, the enclosed content is rendered.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

Environment Tag Helper Attributes

names

`names` accepts a single hosting environment name or a comma-separated list of hosting environment names that trigger the rendering of the enclosed content.

Environment values are compared to the current value returned by IWebHostEnvironment.EnvironmentName. The comparison ignores case.

The following example uses an Environment Tag Helper. The content is rendered if the hosting environment is Staging or Production:

CSHTML

```
<environment names="Staging,Production">
    <strong>IWebHostEnvironment.EnvironmentName is Staging or
Production</strong>
</environment>
```

include and exclude attributes

`include` & `exclude` attributes control rendering the enclosed content based on the included or excluded hosting environment names.

include

The `include` property exhibits similar behavior to the `names` attribute. An environment listed in the `include` attribute value must match the app's hosting environment (IWebHostEnvironment.EnvironmentName) to render the content of the `<environment>` tag.

CSHTML

```
<environment include="Staging,Production">
    <strong>IWebHostEnvironment.EnvironmentName is Staging or
Production</strong>
</environment>
```

exclude

In contrast to the `include` attribute, the content of the `<environment>` tag is rendered when the hosting environment doesn't match an environment listed in the `exclude` attribute value.

CSHTML

```
<environment exclude="Development">
    <strong>IWebHostEnvironment.EnvironmentName is not Development</strong>
</environment>
```

Additional resources

- Use multiple environments in ASP.NET Core

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

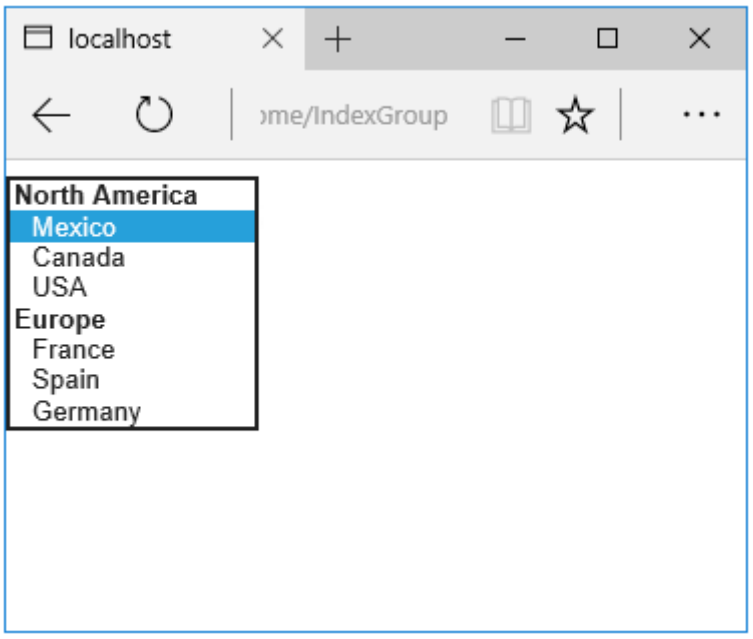
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

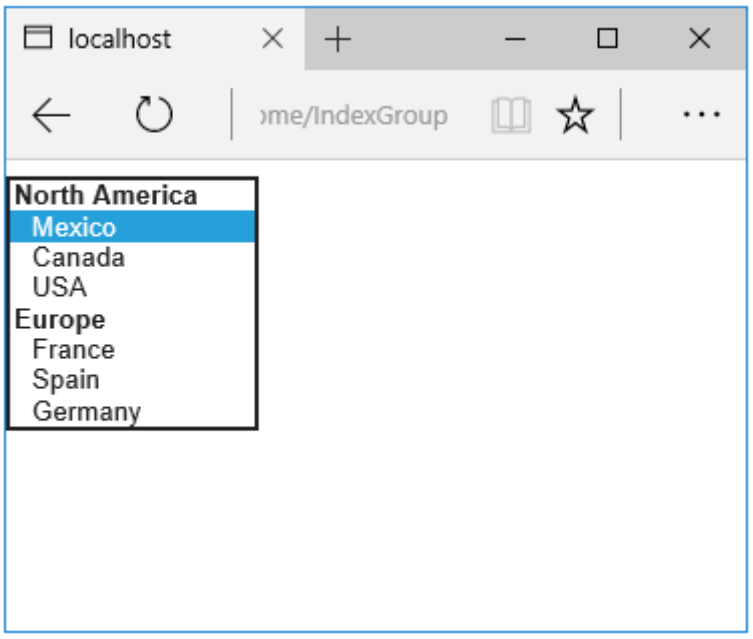
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Image Tag Helper in ASP.NET Core

Article • 06/03/2022

By Peter Kellner

The Image Tag Helper enhances the `<img>` tag to provide cache-busting behavior for static image files.

A cache-busting string is a unique value representing the hash of the static image file appended to the asset's URL. The unique string prompts clients (and some proxies) to reload the image from the host web server and not from the client's cache.

If the image source (`src`) is a static file on the host web server:

- A unique cache-busting string is appended as a query parameter to the image source.
- If the file on the host web server changes, a unique request URL is generated that includes the updated request parameter.

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

Image Tag Helper Attributes

src

To activate the Image Tag Helper, the `src` attribute is required on the `<img>` element.

The image source (`src`) must point to a physical static file on the server. If the `src` is a remote URI, the cache-busting query string parameter isn't generated.

asp-append-version

When `asp-append-version` is specified with a `true` value along with a `src` attribute, the Image Tag Helper is invoked.

The following example uses an Image Tag Helper:

CSHTML

```
<img src="~/images/asplogo.png" asp-append-version="true">
```

If the static file exists in the directory */wwwroot/images/*, the generated HTML is similar to the following (the hash will be different):

```HTML
<img src="/images/asplogo.png?
v=Kl_dqr9NVtnMdsM2MUg4qthUnWZm5T1fCEimBPWDNgM">
```

The value assigned to the parameter `v` is the hash value of the `asplogo.png` file on disk. If the web server is unable to obtain read access to the static file, no `v` parameter is added to the `src` attribute in the rendered markup.

For a Tag Helper to generate a version for a static file outside `wwwroot`, see Serve files from multiple locations

Hash caching behavior

The Image Tag Helper uses the cache provider on the local web server to store the calculated `Sha512` hash of a given file. If the file is requested multiple times, the hash isn't recalculated. The cache is invalidated by a file watcher that's attached to the file when the file's `Sha512` hash is calculated. When the file changes on disk, a new hash is calculated and cached.

Additional resources

- Cache in-memory in ASP.NET Core

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

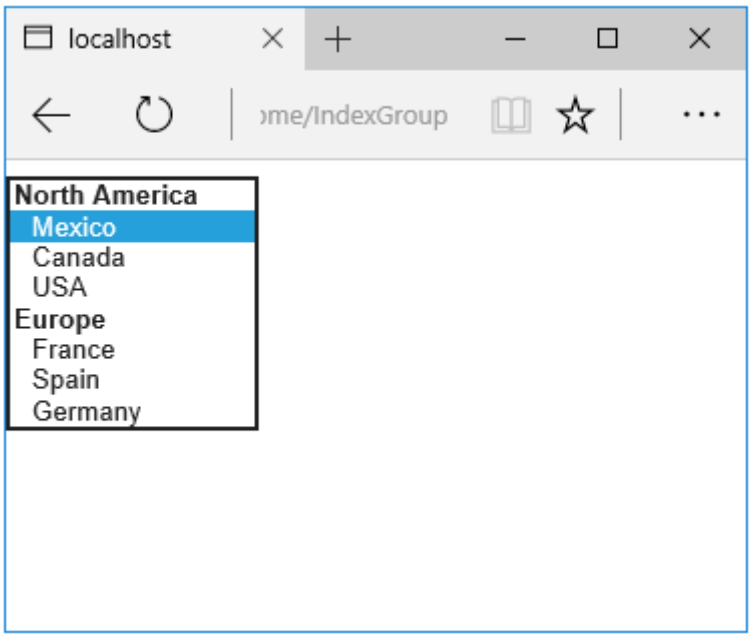
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

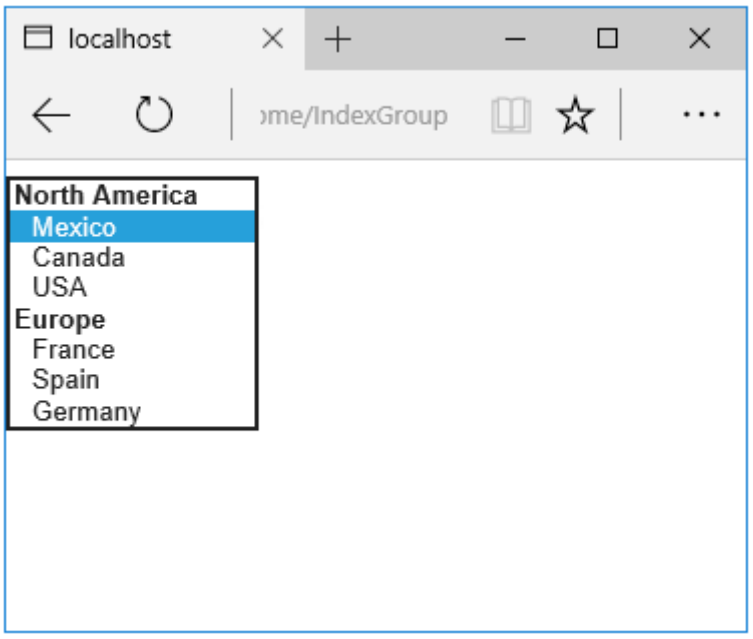
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Link Tag Helper in ASP.NET Core

Article • 06/27/2022

By Rick Anderson

The Link Tag Helper generates a link to a primary or fall back CSS file. Typically the primary CSS file is on a Content Delivery Network (CDN).

A CDN:

- Provides several performance advantages vs hosting the asset with the web app.
- Should not be relied on as the only source for the asset. CDNs are not always available, therefore a reliable fallback should be used. Typically the fallback is the site hosting the web app.

The Link Tag Helper allows you to specify a CDN for the CSS file and a fallback when the CDN is not available. The Link Tag Helper provides the performance advantage of a CDN with the robustness of local hosting.

The following Razor markup shows the `head` element of a layout file created with the ASP.NET Core web app template:

CSHTML

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebLinkTH</title>

    <environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css"
/>
    </environment>
    <environment exclude="Development">
        <link rel="stylesheet"
              href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.css"
              asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.css"
              asp-fallback-test-class="sr-only" asp-fallback-test-
property="position"
              asp-fallback-test-value="absolute"
              crossorigin="anonymous"
              integrity="sha256-
eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE=" />
    </environment>
```

```
    <link rel="stylesheet" href="~/css/site.css" />
</head>
```

The following is rendered HTML from the preceding code (in a non-Development environment):

HTML

```
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Home page - WebLinkTH</title>
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.1.3/css/bootstrap.css"
          crossorigin="anonymous" integrity="sha256-eS<snip>BE=" />
    <meta name="x-stylesheet-fallback-test" content="" class="sr-only" />
    <script>
        !function (a, b, c, d) {
            var e, f = document,
                g = f.getElementsByTagName("SCRIPT"),
                h = g[g.length - 1].previousElementSibling,
                i = f.defaultView && f.defaultView.getComputedStyle ?
f.defaultView.getComputedStyle(h) : h.currentStyle;
            if (i && i[a] !== b) for (e = 0; e < c.length; e++)
                f.write('<link href="' + c[e] + '" ' + d + "/>")
        }
            ("position", "absolute",
["\/lib\/bootstrap\/dist\/css\/bootstrap.css"],
                "rel=\u0022stylesheet\u0022
crossorigin=\u0022anonymous\u0022 integrity=\abc<snip>BE=\u0022 ");
    </script>

    <link rel="stylesheet" href="/css/site.css" />
</head>
```

In the preceding code, the Link Tag Helper generated the `<meta name="x-stylesheet-fallback-test" content="" class="sr-only" />` element and the following JavaScript which is used to verify the requested `bootstrap.css` file is available on the CDN. In this case, the CSS file was available so the Tag Helper generated the `<link />` element with the CDN CSS file.

Commonly used Link Tag Helper attributes

See Link Tag Helper for all the Link Tag Helper attributes, properties, and methods.

href

Preferred address of the linked resource. The address is passed thought to the generated HTML in all cases.

asp-fallback-href

The URL of a CSS stylesheet to fallback to in the case the primary URL fails.

asp-fallback-test-class

The class name defined in the stylesheet to use for the fallback test. For more information, see FallbackTestClass.

asp-fallback-test-property

The CSS property name to use for the fallback test. For more information, see FallbackTestProperty.

asp-fallback-test-value

The CSS property value to use for the fallback test. For more information, see FallbackTestValue.

Additional resources

- Tag Helpers in ASP.NET Core
- Areas in ASP.NET Core
- Introduction to Razor Pages in ASP.NET Core
- Compatibility version for ASP.NET Core MVC

Partial Tag Helper in ASP.NET Core

Article • 06/03/2022

By Scott Addie

For an overview of Tag Helpers, see Tag Helpers in ASP.NET Core.

View or download sample code (how to download)

Overview

The Partial Tag Helper is used for rendering a partial view in Razor Pages and MVC apps. Consider that it:

- Requires ASP.NET Core 2.1 or later.
- Is an alternative to HTML Helper syntax.
- Renders the partial view asynchronously.

The HTML Helper options for rendering a partial view include:

- @await Html.PartialAsync
- @await Html.RenderPartialAsync
- @Html.Partial
- @Html.RenderPartial

The *Product* model is used in samples throughout this document:

C#

```
namespace TagHelpersBuiltIn.Models
{
    public class Product
    {
        public int Number { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }
    }
}
```

An inventory of the Partial Tag Helper attributes follows.

name

The `name` attribute is required. It indicates the name or the path of the partial view to be rendered. When a partial view name is provided, the view discovery process is initiated. That process is bypassed when an explicit path is provided. For all acceptable `name` values, see Partial view discovery.

The following markup uses an explicit path, indicating that `_ProductPartial.cshtml` is to be loaded from the *Shared* folder. Using the for attribute, a model is passed to the partial view for binding.

CSHTML

```
<partial name="Shared/_ProductPartial.cshtml" for="Product">
```

for

The `for` attribute assigns a ModelExpression to be evaluated against the current model. A `ModelExpression` infers the `@Model.` syntax. For example, `for="Product"` can be used instead of `for="@Model.Product"`. This default inference behavior is overridden by using the `@` symbol to define an inline expression.

The following markup loads `_ProductPartial.cshtml`:

CSHTML

```
<partial name="_ProductPartial" for="Product">
```

The partial view is bound to the associated page model's `Product` property:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using TagHelpersBuiltIn.Models;

namespace TagHelpersBuiltIn.Pages
{
    public class ProductModel : PageModel
    {
        public Product Product { get; set; }

        public void OnGet()
        {
            Product = new Product
            {
                Number = 1,
                Name = "Test product",
```

```
                Description = "This is a test product"
            };
        }
    }
}
```

model

The `model` attribute assigns a model instance to pass to the partial view. The `model` attribute can't be used with the for attribute.

In the following markup, a new `Product` object is instantiated and passed to the `model` attribute for binding:

CSHTML

```
<partial name="_ProductPartial"
         model='new Product { Number = 1, Name = "Test product", Description
= "This is a test" }'>
```

view-data

The `view-data` attribute assigns a ViewDataDictionary to pass to the partial view. The following markup makes the entire ViewData collection accessible to the partial view:

CSHTML

```
@{
    ViewData["IsNumberReadOnly"] = true;
}

<partial name="_ProductViewDataPartial" for="Product" view-data="ViewData">
```

In the preceding code, the `IsNumberReadOnly` key value is set to `true` and added to the ViewData collection. Consequently, `ViewData["IsNumberReadOnly"]` is made accessible within the following partial view:

CSHTML

```
@model TagHelpersBuiltIn.Models.Product

<div class="form-group">
    <label asp-for="Number"></label>
    @if ((bool)ViewData["IsNumberReadOnly"])
```

```
    {
        <input asp-for="Number" type="number" class="form-control" readonly
/>
    }
    else
    {
        <input asp-for="Number" type="number" class="form-control" />
    }
</div>
<div class="form-group">
    <label asp-for="Name"></label>
    <input asp-for="Name" type="text" class="form-control" />
</div>
<div class="form-group">
    <label asp-for="Description"></label>
    <textarea asp-for="Description" rows="4" cols="50" class="form-control">
</textarea>
</div>
```

In this example, the value of `ViewData["IsNumberReadOnly"]` determines whether the *Number* field is displayed as read only.

Migrate from an HTML Helper

Consider the following asynchronous HTML Helper example. A collection of products is iterated and displayed. Per the `PartialAsync` method's first parameter, the `_ProductPartial.cshtml` partial view is loaded. An instance of the `Product` model is passed to the partial view for binding.

CSHTML

```
@foreach (var product in Model.Products)
{
    @await Html.PartialAsync("_ProductPartial", product)
}
```

The following Partial Tag Helper achieves the same asynchronous rendering behavior as the `PartialAsync` HTML Helper. The `model` attribute is assigned a `Product` model instance for binding to the partial view.

CSHTML

```
@foreach (var product in Model.Products)
{
    <partial name="_ProductPartial" model="@product" />
}
```

Additional resources

- Partial views in ASP.NET Core
- Views in ASP.NET Core MVC

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Persist Component State Tag Helper in ASP.NET Core

Article • 11/15/2024

The Persist Component State Tag Helper saves the state of non-routable Razor components rendered in a page or view of a Razor Pages or MVC app.

Prerequisites

Follow the guidance in the *Use non-routable components in pages or views* section of the Integrate ASP.NET Core Razor components article.

Persist state for prerendered components

To persist state for prerendered components, use the Persist Component State Tag Helper (reference source). Add the Tag Helper's tag, `<persist-component-state />`, inside the closing `</body>` tag of the layout in an app that prerenders components.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

In `Pages/Shared/_Layout.cshtml` for embedded components that are either WebAssembly prerendered (`WebAssemblyPrerendered`) or server prerendered (`ServerPrerendered`):

CSHTML

```
<body>
    ...

    <persist-component-state />
</body>
```

Decide what state to persist using the PersistentComponentState service. PersistentComponentState.RegisterOnPersisting registers a callback to persist the component state before the app is paused. The state is retrieved when the application resumes.

For more information and examples, see Prerender ASP.NET Core Razor components.

Additional resources

- Component Tag Helper in ASP.NET Core
- Prerender ASP.NET Core Razor components
- ComponentTagHelper
- Tag Helpers in ASP.NET Core
- ASP.NET Core Razor components

Script Tag Helper in ASP.NET Core

Article • 09/12/2024

By Rick Anderson

The Script Tag Helper generates a link to a primary or fall back script file. Typically the primary script file is on a Content Delivery Network (CDN).

A CDN:

- Provides several performance advantages vs hosting the asset with the web app.
- Should not be relied on as the only source for the asset. CDNs are not always available, therefore a reliable fallback should be used. Typically the fallback is the site hosting the web app.

The Script Tag Helper allows you to specify a CDN for the script file and a fallback when the CDN is not available. The Script Tag Helper provides the performance advantage of a CDN with the robustness of local hosting.

The following Razor markup shows a `script` element with a fallback:

HTML

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.3.1.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-
tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/8aZajjp4Bqd+V6D5IgvKT">
</script>
```

Don't use the `<script>` element's defer attribute to defer loading the CDN script. The Script Tag Helper renders JavaScript that immediately executes the asp-fallback-test expression. The expression fails if loading the CDN script is deferred.

Commonly used Script Tag Helper attributes

See Script Tag Helper for all the Script Tag Helper attributes, properties, and methods.

src

Address of the external script to use.

asp-append-version

When `asp-append-version` is specified with a `true` value along with a src attribute, a unique version is generated.

For a Tag Helper to generate a version for a static file outside `wwwroot`, see Serve files from multiple locations

asp-fallback-src

The URL of a Script tag to fallback to in the case the primary one fails.

asp-fallback-src-exclude

A comma-separated list of globbed file patterns of JavaScript scripts to exclude from the fallback list, in the case the primary one fails. The glob patterns are assessed relative to the application's `webroot` setting. Must be used in conjunction with `asp-fallback-src-include`.

asp-fallback-src-include

A comma-separated list of globbed file patterns of JavaScript scripts to fallback to in the case the primary one fails. The glob patterns are assessed relative to the application's `webroot` setting.

asp-fallback-test

The script method defined in the primary script to use for the fallback test. For more information, see FallbackTestExpression.

asp-order

When a set of `ITagHelper` instances are executed, their `Init(TagHelperContext)` methods are first invoked in the specified order; then their `ProcessAsync(TagHelperContext, TagHelperOutput)` methods are invoked in the specified order. Lower values are executed first.

asp-src-exclude

A comma-separated list of globbed file patterns of JavaScript scripts to exclude from loading. The glob patterns are assessed relative to the application's `webroot` setting. Must be used in conjunction with `asp-src-include`.

asp-src-include

A comma-separated list of globbed file patterns of JavaScript scripts to load. The glob patterns are assessed relative to the application's `webroot` setting.

asp-suppress-fallback-integrity

Boolean value that determines if an integrity hash will be compared with the asp-fallback-src value.

Additional resources

- Tag Helpers in ASP.NET Core
- Areas in ASP.NET Core
- Introduction to Razor Pages in ASP.NET Core
- Compatibility version for ASP.NET Core MVC

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

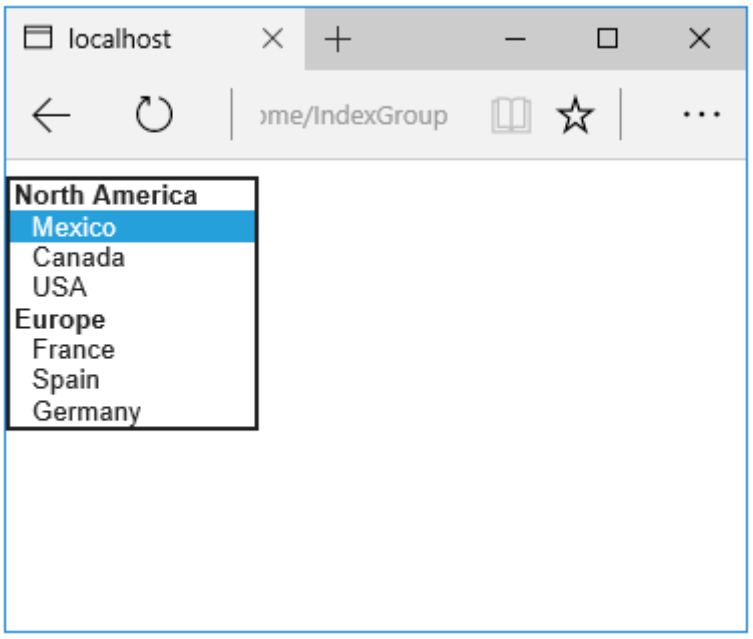
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

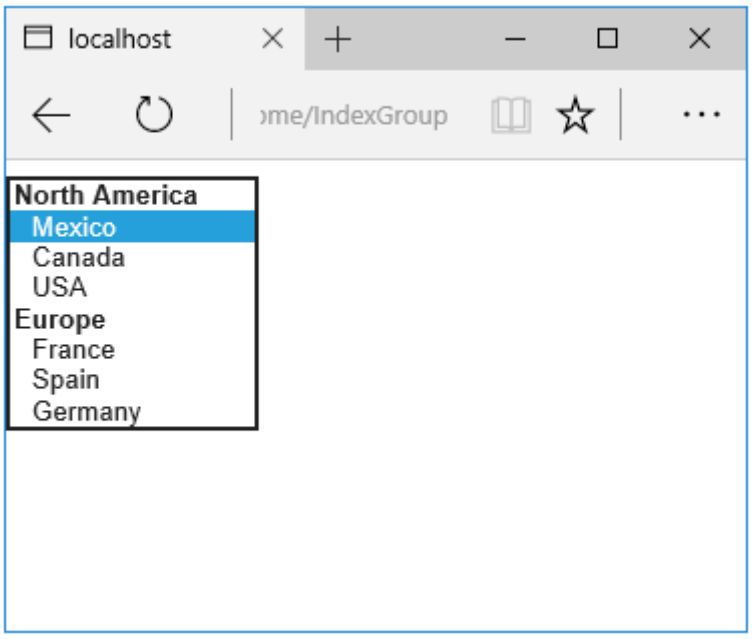
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

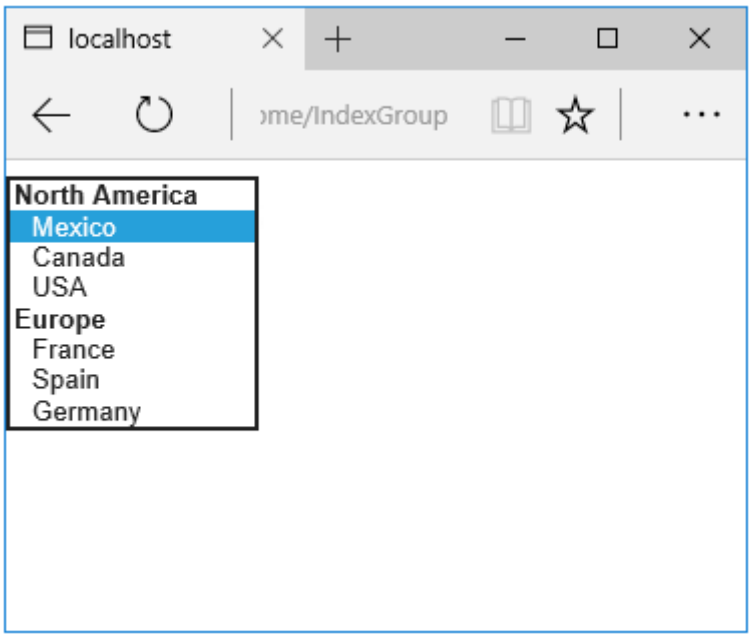
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Tag Helpers in forms in ASP.NET Core

Article • 07/10/2023

By Rick Anderson, N. Taylor Mullen, Dave Paquette, and Jerrie Pelser

This document demonstrates working with Forms and the HTML elements commonly used on a Form. The HTML Form element provides the primary mechanism web apps use to post back data to the server. Most of this document describes Tag Helpers and how they can help you productively create robust HTML forms. We recommend you read Introduction to Tag Helpers before you read this document.

In many cases, HTML Helpers provide an alternative approach to a specific Tag Helper, but it's important to recognize that Tag Helpers don't replace HTML Helpers and there's not a Tag Helper for each HTML Helper. When an HTML Helper alternative exists, it's mentioned.

The Form Tag Helper

The Form Tag Helper:

- Generates the HTML <FORM> `action` attribute value for a MVC controller action or named route
- Generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method)
- Provides the `asp-route-<Parameter Name>` attribute, where `<Parameter Name>` is added to the route values. The `routeValues` parameters to `Html.BeginForm` and `Html.BeginRouteForm` provide similar functionality.
- Has an HTML Helper alternative `Html.BeginForm` and `Html.BeginRouteForm`

Sample:

CSHTML

```
<form asp-controller="Demo" asp-action="Register" method="post">
    <!-- Input and Submit elements -->
</form>
```

The Form Tag Helper above generates the following HTML:

HTML

```
<form method="post" action="/Demo/Register">
    <!-- Input and Submit elements -->
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The MVC runtime generates the `action` attribute value from the Form Tag Helper attributes `asp-controller` and `asp-action`. The Form Tag Helper also generates a hidden Request Verification Token to prevent cross-site request forgery (when used with the `[ValidateAntiForgeryToken]` attribute in the HTTP Post action method). Protecting a pure HTML Form from cross-site request forgery is difficult, the Form Tag Helper provides this service for you.

Using a named route

The `asp-route` Tag Helper attribute can also generate markup for the HTML `action` attribute. An app with a route named `register` could use the following markup for the registration page:

CSHTML

```
<form asp-route="register" method="post">
    <!-- Input and Submit elements -->
</form>
```

Many of the views in the *Views/Account* folder (generated when you create a new web app with *Individual User Accounts*) contain the asp-route-returnurl attribute:

CSHTML

```
<form asp-controller="Account" asp-action="Login"
     asp-route-returnurl="@ViewData["ReturnUrl"]"
     method="post" class="form-horizontal" role="form">
```

> **Note**
>
> With the built in templates, `returnUrl` is only populated automatically when you try to access an authorized resource but are not authenticated or authorized. When you attempt an unauthorized access, the security middleware redirects you to the login page with the `returnUrl` set.

The Form Action Tag Helper

The Form Action Tag Helper generates the `formaction` attribute on the generated `<button ...>` or `<input type="image" ...>` tag. The `formaction` attribute controls where a form submits its data. It binds to <input> elements of type `image` and <button> elements. The Form Action Tag Helper enables the usage of several AnchorTagHelper `asp-` attributes to control what `formaction` link is generated for the corresponding element.

Supported AnchorTagHelper attributes to control the value of `formaction`:

Expand table

Attribute	Description
asp-controller	The name of the controller.
asp-action	The name of the action method.
asp-area	The name of the area.
asp-page	The name of the Razor page.
asp-page-handler	The name of the Razor page handler.
asp-route	The name of the route.
asp-route-{value}	A single URL route value. For example, `asp-route-id="1234"`.
asp-all-route-data	All route values.
asp-fragment	The URL fragment.

Submit to controller example

The following markup submits the form to the `Index` action of `HomeController` when the input or button are selected:

CSHTML

```
<form method="post">
    <button asp-controller="Home" asp-action="Index">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-controller="Home"
                                asp-action="Index">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home">
</form>
```

Submit to page example

The following markup submits the form to the `About` Razor Page:

CSHTML

```
<form method="post">
    <button asp-page="About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-page="About">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/About">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/About">
</form>
```

Submit to route example

Consider the `/Home/Test` endpoint:

C#

```
public class HomeController : Controller
{
    [Route("/Home/Test", Name = "Custom")]
    public string Test()
    {
        return "This is the test page";
    }
}
```

The following markup submits the form to the `/Home/Test` endpoint.

CSHTML

```
<form method="post">
    <button asp-route="Custom">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" asp-route="Custom">
</form>
```

The previous markup generates following HTML:

HTML

```
<form method="post">
    <button formaction="/Home/Test">Click Me</button>
    <input type="image" src="..." alt="Or Click Me" formaction="/Home/Test">
</form>
```

The Input Tag Helper

The Input Tag Helper binds an HTML <input> element to a model expression in your razor view.

Syntax:

CSHTML

```
<input asp-for="<Expression Name>">
```

The Input Tag Helper:

- Generates the `id` and `name` HTML attributes for the expression name specified in the `asp-for` attribute. `asp-for="Property1.Property2"` is equivalent to `m => m.Property1.Property2`. The name of the expression is what is used for the `asp-for` attribute value. See the Expression names section for additional information.
- Sets the HTML `type` attribute value based on the model type and data annotation attributes applied to the model property
- Won't overwrite the HTML `type` attribute value when one is specified
- Generates HTML5 validation attributes from data annotation attributes applied to model properties
- Has an HTML Helper feature overlap with `Html.TextBoxFor` and `Html.EditorFor`. See the HTML Helper alternatives to Input Tag Helper section for details.

- Provides strong typing. If the name of the property changes and you don't update the Tag Helper you'll get an error similar to the following:

```
An error occurred during the compilation of a resource required to
process
this request. Please review the following specific error details and
modify
your source code appropriately.

Type expected
'RegisterViewModel' does not contain a definition for 'Email' and no
extension method 'Email' accepting a first argument of type
'RegisterViewModel'
could be found (are you missing a using directive or an assembly
reference?)
```

The `Input` Tag Helper sets the HTML `type` attribute based on the .NET type. The following table lists some common .NET types and generated HTML type (not every .NET type is listed).

Expand table

.NET type	Input Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime-local"
Byte	type="number"
Int	type="number"
Single, Double	type="number"

The following table shows some common data annotations attributes that the input tag helper will map to specific input types (not every validation attribute is listed):

Expand table

Attribute	Input Type
[EmailAddress]	type="email"
[Url]	type="url"

Attribute	Input Type
[HiddenInput]	type="hidden"
[Phone]	type="tel"
[DataType(DataType.Password)]	type="password"
[DataType(DataType.Date)]	type="date"
[DataType(DataType.Time)]	type="time"

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterInput" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <button type="submit">Register</button>
</form>
```

The code above generates the following HTML:

HTML

```
<form method="post" action="/Demo/RegisterInput">
    Email:
    <input type="email" data-val="true"
```

```
            data-val-email="The Email Address field is not a valid email
address."
            data-val-required="The Email Address field is required."
            id="Email" name="Email" value=""><br>
    Password:
    <input type="password" data-val="true"
            data-val-required="The Password field is required."
            id="Password" name="Password"><br>
    <button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

The data annotations applied to the `Email` and `Password` properties generate metadata on the model. The Input Tag Helper consumes the model metadata and produces HTML5 `data-val-*` attributes (see Model Validation). These attributes describe the validators to attach to the input fields. This provides unobtrusive HTML5 and jQuery validation. The unobtrusive attributes have the format `data-val-rule="Error Message"`, where rule is the name of the validation rule (such as `data-val-required`, `data-val-email`, `data-val-maxlength`, etc.) If an error message is provided in the attribute, it's displayed as the value for the `data-val-rule` attribute. There are also attributes of the form `data-val-ruleName-argumentName="argumentValue"` that provide additional details about the rule, for example, `data-val-maxlength-max="1024"` .

When binding multiple `input` controls to the same property, the generated controls share the same `id`, which makes the generated mark-up invalid. To prevent duplicates, specify the `id` attribute for each control explicitly.

Checkbox hidden input rendering

Checkboxes in HTML5 don't submit a value when they're unchecked. To enable a default value to be sent for an unchecked checkbox, the Input Tag Helper generates an additional hidden input for checkboxes.

For example, consider the following Razor markup that uses the Input Tag Helper for a boolean model property `IsChecked`:

CSHTML

```
<form method="post">
    <input asp-for="@Model.IsChecked" />
    <button type="submit">Submit</button>
</form>
```

The preceding Razor markup generates HTML markup similar to the following:

HTML

```
<form method="post">
    <input name="IsChecked" type="checkbox" value="true" />
    <button type="submit">Submit</button>

    <input name="IsChecked" type="hidden" value="false" />
</form>
```

The preceding HTML markup shows an additional hidden input with a name of `IsChecked` and a value of `false`. By default, this hidden input is rendered at the end of the form. When the form is submitted:

- If the `IsChecked` checkbox input is checked, both `true` and `false` are submitted as values.
- If the `IsChecked` checkbox input is unchecked, only the hidden input value `false` is submitted.

The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

To configure the behavior of the hidden input rendering, set the CheckBoxHiddenInputRenderMode property on MvcViewOptions.HtmlHelperOptions. For example:

C#

```
services.Configure<MvcViewOptions>(options =>
    options.HtmlHelperOptions.CheckBoxHiddenInputRenderMode =
        CheckBoxHiddenInputRenderMode.None);
```

The preceding code disables hidden input rendering for checkboxes by setting `CheckBoxHiddenInputRenderMode` to CheckBoxHiddenInputRenderMode.None. For all available rendering modes, see the CheckBoxHiddenInputRenderMode enum.

HTML Helper alternatives to Input Tag Helper

`Html.TextBox`, `Html.TextBoxFor`, `Html.Editor` and `Html.EditorFor` have overlapping features with the Input Tag Helper. The Input Tag Helper will automatically set the `type` attribute; `Html.TextBox` and `Html.TextBoxFor` won't. `Html.Editor` and `Html.EditorFor` handle collections, complex objects and templates; the Input Tag Helper doesn't. The

Input Tag Helper, `Html.EditorFor` and `Html.TextBoxFor` are strongly typed (they use lambda expressions); `Html.TextBox` and `Html.Editor` are not (they use expression names).

HtmlAttributes

`@Html.Editor()` and `@Html.EditorFor()` use a special `ViewDataDictionary` entry named `htmlAttributes` when executing their default templates. This behavior is optionally augmented using `additionalViewData` parameters. The key "htmlAttributes" is case-insensitive. The key "htmlAttributes" is handled similarly to the `htmlAttributes` object passed to input helpers like `@Html.TextBox()`.

CSHTML

```
@Html.EditorFor(model => model.YourProperty,
  new { htmlAttributes = new { @class="myCssClass", style="Width:100px" } })
```

Expression names

The `asp-for` attribute value is a `ModelExpression` and the right hand side of a lambda expression. Therefore, `asp-for="Property1"` becomes `m => m.Property1` in the generated code which is why you don't need to prefix with `Model`. You can use the "@" character to start an inline expression and move before the `m.`:

CSHTML

```
@{
  var joe = "Joe";
}

<input asp-for="@joe">
```

Generates the following:

HTML

```
<input type="text" id="joe" name="joe" value="Joe">
```

With collection properties, `asp-for="CollectionProperty[23].Member"` generates the same name as `asp-for="CollectionProperty[i].Member"` when `i` has the value `23`.

When ASP.NET Core MVC calculates the value of `ModelExpression`, it inspects several sources, including `ModelState`. Consider `<input type="text" asp-for="Name">`. The calculated `value` attribute is the first non-null value from:

- `ModelState` entry with key "Name".
- Result of the expression `Model.Name`.

Navigating child properties

You can also navigate to child properties using the property path of the view model. Consider a more complex model class that contains a child `Address` property.

C#

```
public class AddressViewModel
{
    public string AddressLine1 { get; set; }
}
```

C#

```
public class RegisterAddressViewModel
{
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public AddressViewModel Address { get; set; }
}
```

In the view, we bind to `Address.AddressLine1`:

CSHTML

```
@model RegisterAddressViewModel

<form asp-controller="Demo" asp-action="RegisterAddress" method="post">
    <label>Email: <input asp-for="Email" /></label> <br />
    <label>Password: <input asp-for="Password" /></label><br />
    <label>Address: <input asp-for="Address.AddressLine1" /></label><br />
    <button type="submit">Register</button>
</form>
```

The following HTML is generated for `Address.AddressLine1`:

HTML

```
<input type="text" id="Address_AddressLine1" name="Address.AddressLine1"
value="">
```

Expression names and Collections

Sample, a model containing an array of `Colors`:

C#

```
public class Person
{
    public List<string> Colors { get; set; }

    public int Age { get; set; }
}
```

The action method:

C#

```
public IActionResult Edit(int id, int colorIndex)
{
    ViewData["Index"] = colorIndex;
    return View(GetPerson(id));
}
```

The following Razor shows how you access a specific `Color` element:

CSHTML

```
@model Person
@{
    var index = (int)ViewData["index"];
}

<form asp-controller="ToDo" asp-action="Edit" method="post">
    @Html.EditorFor(m => m.Colors[index])
    <label asp-for="Age"></label>
    <input asp-for="Age" /><br />
    <button type="submit">Post</button>
</form>
```

The `Views/Shared/EditorTemplates/String.cshtml` template:

CSHTML

```
@model string

<label asp-for="@Model"></label>
<input asp-for="@Model" /> <br />
```

Sample using `List<T>`:

C#

```
public class ToDoItem
{
    public string Name { get; set; }

    public bool IsDone { get; set; }
}
```

The following Razor shows how to iterate over a collection:

CSHTML

```
@model List<ToDoItem>

<form asp-controller="ToDo" asp-action="Edit" method="post">
    <table>
        <tr> <th>Name</th> <th>Is Done</th> </tr>

        @for (int i = 0; i < Model.Count; i++)
        {
            <tr>
                @Html.EditorFor(model => model[i])
            </tr>
        }

    </table>
    <button type="submit">Save</button>
</form>
```

The `Views/Shared/EditorTemplates/ToDoItem.cshtml` template:

CSHTML

```
@model ToDoItem

<td>
    <label asp-for="@Model.Name"></label>
    @Html.DisplayFor(model => model.Name)
</td>
<td>
    <input asp-for="@Model.IsDone" />
```

```
</td>

@*
    This template replaces the following Razor which evaluates the indexer
three times.
    <td>
         <label asp-for="@Model[i].Name"></label>
         @Html.DisplayFor(model => model[i].Name)
     </td>
     <td>
         <input asp-for="@Model[i].IsDone" />
     </td>
*@
```

`foreach` should be used if possible when the value is going to be used in an `asp-for` or `Html.DisplayFor` equivalent context. In general, `for` is better than `foreach` (if the scenario allows it) because it doesn't need to allocate an enumerator; however, evaluating an indexer in a LINQ expression can be expensive and should be minimized.

> ⓘ **Note**
>
> The commented sample code above shows how you would replace the lambda expression with the `@` operator to access each `ToDoItem` in the list.

The Textarea Tag Helper

The `Textarea Tag Helper` tag helper is similar to the Input Tag Helper.

- Generates the `id` and `name` attributes, and the data validation attributes from the model for a <textarea> element.
- Provides strong typing.
- HTML Helper alternative: `Html.TextAreaFor`

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class DescriptionViewModel
```

```
    {
        [MinLength(5)]
        [MaxLength(1024)]
        public string Description { get; set; }
    }
}
```

CSHTML

```
@model DescriptionViewModel

<form asp-controller="Demo" asp-action="RegisterTextArea" method="post">
    <textarea asp-for="Description"></textarea>
    <button type="submit">Test</button>
</form>
```

The following HTML is generated:

HTML

```
<form method="post" action="/Demo/RegisterTextArea">
  <textarea data-val="true"
   data-val-maxlength="The field Description must be a string or array type
with a maximum length of &#x27;1024&#x27;."
   data-val-maxlength-max="1024"
   data-val-minlength="The field Description must be a string or array type
with a minimum length of &#x27;5&#x27;."
   data-val-minlength-min="5"
   id="Description" name="Description">
  </textarea>
  <button type="submit">Test</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Label Tag Helper

- Generates the label caption and `for` attribute on a <label> element for an expression name
- HTML Helper alternative: `Html.LabelFor`.

The `Label Tag Helper` provides the following benefits over a pure HTML label element:

- You automatically get the descriptive label value from the `Display` attribute. The intended display name might change over time, and the combination of `Display` attribute and Label Tag Helper will apply the `Display` everywhere it's used.

- Less markup in source code
- Strong typing with the model property.

Sample:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class SimpleViewModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }
    }
}
```

CSHTML

```
@model SimpleViewModel

<form asp-controller="Demo" asp-action="RegisterLabel" method="post">
    <label asp-for="Email"></label>
    <input asp-for="Email" /> <br />
</form>
```

The following HTML is generated for the `<label>` element:

HTML

```
<label for="Email">Email Address</label>
```

The Label Tag Helper generated the `for` attribute value of "Email", which is the ID associated with the `<input>` element. The Tag Helpers generate consistent `id` and `for` elements so they can be correctly associated. The caption in this sample comes from the `Display` attribute. If the model didn't contain a `Display` attribute, the caption would be the property name of the expression. To override the default caption, add a caption inside the label tag.

The Validation Tag Helpers

There are two Validation Tag Helpers. The `Validation Message Tag Helper` (which displays a validation message for a single property on your model), and the `Validation Summary Tag Helper` (which displays a summary of validation errors). The `Input Tag Helper` adds HTML5 client side validation attributes to input elements based on data annotation attributes on your model classes. Validation is also performed on the server. The Validation Tag Helper displays these error messages when a validation error occurs.

The Validation Message Tag Helper

- Adds the HTML5 `data-valmsg-for="property"` attribute to the span element, which attaches the validation error messages on the input field of the specified model property. When a client side validation error occurs, jQuery displays the error message in the `<span>` element.
- Validation also takes place on the server. Clients may have JavaScript disabled and some validation can only be done on the server side.
- HTML Helper alternative: `Html.ValidationMessageFor`

The `Validation Message Tag Helper` is used with the `asp-validation-for` attribute on an HTML span element.

CSHTML

```
<span asp-validation-for="Email"></span>
```

The Validation Message Tag Helper will generate the following HTML:

HTML

```
<span class="field-validation-valid"
  data-valmsg-for="Email"
  data-valmsg-replace="true"></span>
```

You generally use the `Validation Message Tag Helper` after an `Input` Tag Helper for the same property. Doing so displays any validation error messages near the input that caused the error.

> ⓘ **Note**
>
> You must have a view with the correct JavaScript and jQuery script references in place for client side validation. See Model Validation for more information.

When a server side validation error occurs (for example when you have custom server side validation or client-side validation is disabled), MVC places that error message as the body of the `<span>` element.

HTML

```
<span class="field-validation-error" data-valmsg-for="Email"
            data-valmsg-replace="true">
   The Email Address field is required.
</span>
```

The Validation Summary Tag Helper

- Targets `<div>` elements with the `asp-validation-summary` attribute
- HTML Helper alternative: `@Html.ValidationSummary`

The `Validation Summary Tag Helper` is used to display a summary of validation messages. The `asp-validation-summary` attribute value can be any of the following:

Expand table

asp-validation-summary	Validation messages displayed
`All`	Property and model level
`ModelOnly`	Model
`None`	None

Sample

In the following example, the data model has `DataAnnotation` attributes, which generates validation error messages on the `<input>` element. When a validation error occurs, the Validation Tag Helper displays the error message:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public class RegisterViewModel
    {
        [Required]
```

```
        [EmailAddress]
        [Display(Name = "Email Address")]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

CSHTML

```
@model RegisterViewModel

<form asp-controller="Demo" asp-action="RegisterValidation" method="post">
    <div asp-validation-summary="ModelOnly"></div>
    <label>Email: <input asp-for="Email" /></label> <br />
    <span asp-validation-for="Email"></span><br />
    <label>Password: <input asp-for="Password" /></label><br />
    <span asp-validation-for="Password"></span><br />
    <button type="submit">Register</button>
</form>
```

The generated HTML (when the model is valid):

HTML

```
<form action="/DemoReg/Register" method="post">
  <label>Email: <input name="Email" id="Email" type="email" value=""
   data-val-required="The Email field is required."
   data-val-email="The Email field is not a valid email address."
   data-val="true"></label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Email"></span><br>
  <label>Password: <input name="Password" id="Password" type="password"
   data-val-required="The Password field is required." data-val="true">
</label><br>
  <span class="field-validation-valid" data-valmsg-replace="true"
   data-valmsg-for="Password"></span><br>
  <button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

The Select Tag Helper

- Generates select and associated option elements for properties of your model.
- Has an HTML Helper alternative `Html.DropDownListFor` and `Html.ListBoxFor`

The `Select Tag Helper` `asp-for` specifies the model property name for the select element and `asp-items` specifies the option elements. For example:

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Sample:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModel
    {
        public string Country { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA" },
        };
    }
}
```

The `Index` method initializes the `CountryViewModel`, sets the selected country and passes it to the `Index` view.

C#

```
public IActionResult Index()
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

The HTTP POST `Index` method displays the selection:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public IActionResult Index(CountryViewModel model)
{
    if (ModelState.IsValid)
    {
        var msg = model.Country + " selected";
        return RedirectToAction("IndexSuccess", new { message = msg });
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

The `Index` view:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="Index" method="post">
    <select asp-for="Country" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Which generates the following HTML (with "CA" selected):

HTML

```
<form method="post" action="/">
     <select id="Country" name="Country">
       <option value="MX">Mexico</option>
       <option selected="selected" value="CA">Canada</option>
       <option value="US">USA</option>
     </select>
       <br /><button type="submit">Register</button>
     <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
   </form>
```

> **Note**
>
> We don't recommend using `ViewBag` or `ViewData` with the Select Tag Helper. A view model is more robust at providing MVC metadata and generally less problematic.

The `asp-for` attribute value is a special case and doesn't require a `Model` prefix, the other Tag Helper attributes do (such as `asp-items`)

CSHTML

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

Enum binding

It's often convenient to use `<select>` with an `enum` property and generate the `SelectListItem` elements from the `enum` values.

Sample:

C#

```
public class CountryEnumViewModel
{
    public CountryEnum EnumCountry { get; set; }
}
```

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The `GetEnumSelectList` method generates a `SelectList` object for an enum.

CSHTML

```
@model CountryEnumViewModel

<form asp-controller="Home" asp-action="IndexEnum" method="post">
    <select asp-for="EnumCountry"
            asp-items="Html.GetEnumSelectList<CountryEnum>()">
    </select>
```

```
    <br /><button type="submit">Register</button>
</form>
```

You can mark your enumerator list with the `Display` attribute to get a richer UI:

C#

```
using System.ComponentModel.DataAnnotations;

namespace FormsTagHelper.ViewModels
{
    public enum CountryEnum
    {
        [Display(Name = "United Mexican States")]
        Mexico,
        [Display(Name = "United States of America")]
        USA,
        Canada,
        France,
        Germany,
        Spain
    }
}
```

The following HTML is generated:

HTML

```
<form method="post" action="/Home/IndexEnum">
    <select data-val="true" data-val-required="The EnumCountry field is
required."
            id="EnumCountry" name="EnumCountry">
        <option value="0">United Mexican States</option>
        <option value="1">United States of America</option>
        <option value="2">Canada</option>
        <option value="3">France</option>
        <option value="4">Germany</option>
        <option selected="selected" value="5">Spain</option>
    </select>
    <br /><button type="submit">Register</button>
    <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
</form>
```

Option Group

The HTML <optgroup> element is generated when the view model contains one or more `SelectListGroup` objects.

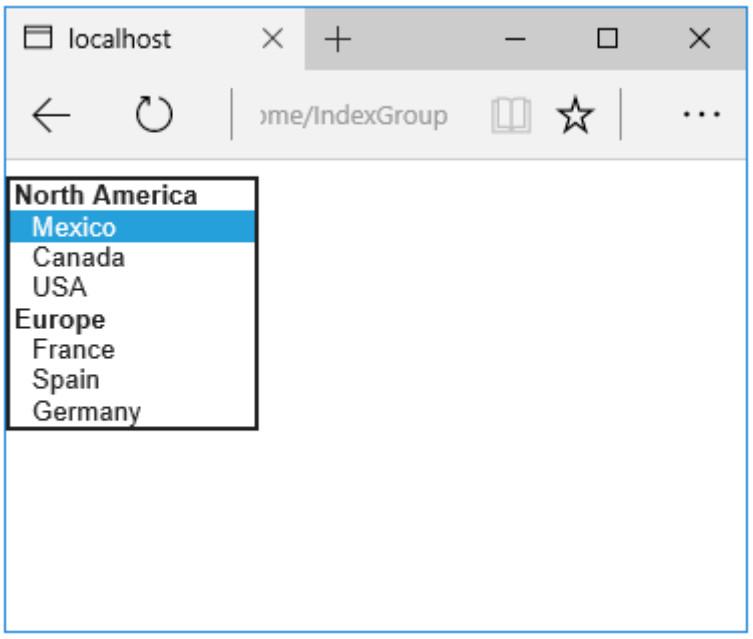
The `CountryViewModelGroup` groups the `SelectListItem` elements into the "North America" and "Europe" groups:

C#

```
public class CountryViewModelGroup
{
    public CountryViewModelGroup()
    {
        var NorthAmericaGroup = new SelectListGroup { Name = "North America"
};
        var EuropeGroup = new SelectListGroup { Name = "Europe" };

        Countries = new List<SelectListItem>
        {
            new SelectListItem
            {
                Value = "MEX",
                Text = "Mexico",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "CAN",
                Text = "Canada",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "US",
                Text = "USA",
                Group = NorthAmericaGroup
            },
            new SelectListItem
            {
                Value = "FR",
                Text = "France",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "ES",
                Text = "Spain",
                Group = EuropeGroup
            },
            new SelectListItem
            {
                Value = "DE",
                Text = "Germany",
                Group = EuropeGroup
            }
      };
    }
```

```
    public string Country { get; set; }

    public List<SelectListItem> Countries { get; }
```

The two groups are shown below:



The generated HTML:

HTML

```
 <form method="post" action="/Home/IndexGroup">
      <select id="Country" name="Country">
          <optgroup label="North America">
              <option value="MEX">Mexico</option>
              <option value="CAN">Canada</option>
              <option value="US">USA</option>
          </optgroup>
          <optgroup label="Europe">
              <option value="FR">France</option>
              <option value="ES">Spain</option>
              <option value="DE">Germany</option>
          </optgroup>
      </select>
      <br /><button type="submit">Register</button>
      <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
  </form>
```

Multiple select

The Select Tag Helper will automatically generate the multiple = "multiple" attribute if the property specified in the `asp-for` attribute is an `IEnumerable`. For example, given the following model:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace FormsTagHelper.ViewModels
{
    public class CountryViewModelIEnumerable
    {
        public IEnumerable<string> CountryCodes { get; set; }

        public List<SelectListItem> Countries { get; } = new
List<SelectListItem>
        {
            new SelectListItem { Value = "MX", Text = "Mexico" },
            new SelectListItem { Value = "CA", Text = "Canada" },
            new SelectListItem { Value = "US", Text = "USA"    },
            new SelectListItem { Value = "FR", Text = "France" },
            new SelectListItem { Value = "ES", Text = "Spain"  },
            new SelectListItem { Value = "DE", Text = "Germany"}
         };
    }
}
```

With the following view:

CSHTML

```
@model CountryViewModelIEnumerable

<form asp-controller="Home" asp-action="IndexMultiSelect" method="post">
    <select asp-for="CountryCodes" asp-items="Model.Countries"></select>
    <br /><button type="submit">Register</button>
</form>
```

Generates the following HTML:

HTML

```
<form method="post" action="/Home/IndexMultiSelect">
    <select id="CountryCodes"
    multiple="multiple"
    name="CountryCodes"><option value="MX">Mexico</option>
<option value="CA">Canada</option>
<option value="US">USA</option>
<option value="FR">France</option>
```

```
<option value="ES">Spain</option>
<option value="DE">Germany</option>
</select>
    <br /><button type="submit">Register</button>
  <input name="__RequestVerificationToken" type="hidden" value="<removed for
brevity>">
</form>
```

No selection

If you find yourself using the "not specified" option in multiple pages, you can create a template to eliminate repeating the HTML:

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    @Html.EditorForModel()
    <br /><button type="submit">Register</button>
</form>
```

The `Views/Shared/EditorTemplates/CountryViewModel.cshtml` template:

CSHTML

```
@model CountryViewModel

<select asp-for="Country" asp-items="Model.Countries">
    <option value="">--none--</option>
</select>
```

Adding HTML <option> elements isn't limited to the *No selection* case. For example, the following view and action method will generate HTML similar to the code above:

C#

```
public IActionResult IndexNone()
{
    var model = new CountryViewModel();
    model.Countries.Insert(0, new SelectListItem("<none>", ""));
    return View(model);
}
```

CSHTML

```
@model CountryViewModel

<form asp-controller="Home" asp-action="IndexEmpty" method="post">
    <select asp-for="Country">
        <option value="">&lt;none&gt;</option>
        <option value="MX">Mexico</option>
        <option value="CA">Canada</option>
        <option value="US">USA</option>
    </select>
    <br /><button type="submit">Register</button>
</form>
```

The correct `<option>` element will be selected (contain the `selected="selected"` attribute) depending on the current `Country` value.

C#

```
public IActionResult IndexOption(int id)
{
    var model = new CountryViewModel();
    model.Country = "CA";
    return View(model);
}
```

HTML

```
 <form method="post" action="/Home/IndexEmpty">
      <select id="Country" name="Country">
          <option value="">&lt;none&gt;</option>
          <option value="MX">Mexico</option>
          <option value="CA" selected="selected">Canada</option>
          <option value="US">USA</option>
      </select>
      <br /><button type="submit">Register</button>
   <input name="__RequestVerificationToken" type="hidden" value="<removed
for brevity>">
 </form>
```

Additional resources

- Tag Helpers in ASP.NET Core
- HTML Form element
- Request Verification Token
- Model Binding in ASP.NET Core
- Model validation in ASP.NET Core MVC
- IAttributeAdapter Interface

- Code snippets for this document

Share controllers, views, Razor Pages and more with Application Parts

Article • 06/03/2022

By Rick Anderson

View or download sample code (how to download)

An *Application Part* is an abstraction over the resources of an app. Application Parts allow ASP.NET Core to discover controllers, view components, tag helpers, Razor Pages, razor compilation sources, and more. AssemblyPart is an Application part. `AssemblyPart` encapsulates an assembly reference and exposes types and compilation references.

Feature providers work with application parts to populate the features of an ASP.NET Core app. The main use case for application parts is to configure an app to discover (or avoid loading) ASP.NET Core features from an assembly. For example, you might want to share common functionality between multiple apps. Using Application Parts, you can share an assembly (DLL) containing controllers, views, Razor Pages, razor compilation sources, Tag Helpers, and more with multiple apps. Sharing an assembly is preferred to duplicating code in multiple projects.

ASP.NET Core apps load features from ApplicationPart. The AssemblyPart class represents an application part that's backed by an assembly.

Load ASP.NET Core features

Use the Microsoft.AspNetCore.Mvc.ApplicationParts and AssemblyPart classes to discover and load ASP.NET Core features (controllers, view components, etc.). The ApplicationPartManager tracks the application parts and feature providers available. `ApplicationPartManager` is configured in `Startup.ConfigureServices`:

C#

```
// Requires using System.Reflection;
public void ConfigureServices(IServiceCollection services)
{
    var assembly = typeof(MySharedController).Assembly;
    services.AddControllersWithViews()
        .AddApplicationPart(assembly)
        .AddRazorRuntimeCompilation();

    services.Configure<MvcRazorRuntimeCompilationOptions>(options =>
```

```
    { options.FileProviders.Add(new EmbeddedFileProvider(assembly)); });
}
```

The following code provides an alternative approach to configuring `ApplicationPartManager` using `AssemblyPart`:

C#

```
// Requires using System.Reflection;
// Requires using Microsoft.AspNetCore.Mvc.ApplicationParts;
public void ConfigureServices(IServiceCollection services)
{
    var assembly = typeof(MySharedController).Assembly;
    // This creates an AssemblyPart, but does not create any related parts
for items such as views.
    var part = new AssemblyPart(assembly);
    services.AddControllersWithViews()
        .ConfigureApplicationPartManager(apm =>
apm.ApplicationParts.Add(part));
}
```

The preceding two code samples load the `SharedController` from an assembly. The `SharedController` is not in the app's project. See the WebAppParts solution sample download.

Include views

Use a Razor class library to include views in the assembly.

Prevent loading resources

Application parts can be used to *avoid* loading resources in a particular assembly or location. Add or remove members of the Microsoft.AspNetCore.Mvc.ApplicationParts collection to hide or make available resources. The order of the entries in the `ApplicationParts` collection isn't important. Configure the `ApplicationPartManager` before using it to configure services in the container. For example, configure the `ApplicationPartManager` before invoking `AddControllersAsServices`. Call `Remove` on the `ApplicationParts` collection to remove a resource.

The `ApplicationPartManager` includes parts for:

- The app's assembly and dependent assemblies.
- `Microsoft.AspNetCore.Mvc.ApplicationParts.CompiledRazorAssemblyPart`
- `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`

- `Microsoft.AspNetCore.Mvc.TagHelpers`.
- `Microsoft.AspNetCore.Mvc.Razor`.

Feature providers

Application feature providers examine application parts and provide features for those parts. There are built-in feature providers for the following ASP.NET Core features:

- ControllerFeatureProvider
- TagHelperFeatureProvider
- MetadataReferenceFeatureProvider
- ViewsFeatureProvider
- `internal class` RazorCompiledItemFeatureProvider

Feature providers inherit from IApplicationFeatureProvider<TFeature>, where `T` is the type of the feature. Feature providers can be implemented for any of the previously listed feature types. The order of feature providers in the `ApplicationPartManager.FeatureProviders` can impact run time behavior. Later added providers can react to actions taken by earlier added providers.

Display available features

The features available to an app can be enumerated by requesting an `ApplicationPartManager` through dependency injection:

C#

```
using AppPartsSample.ViewModels;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using Microsoft.AspNetCore.Mvc.Controllers;
using System.Linq;
using Microsoft.AspNetCore.Mvc.Razor.Compilation;
using Microsoft.AspNetCore.Mvc.Razor.TagHelpers;
using Microsoft.AspNetCore.Mvc.ViewComponents;

namespace AppPartsSample.Controllers
{
    public class FeaturesController : Controller
    {
        private readonly ApplicationPartManager _partManager;

        public FeaturesController(ApplicationPartManager partManager)
        {
            _partManager = partManager;
        }
```

```
        public IActionResult Index()
        {
            var viewModel = new FeaturesViewModel();

            var controllerFeature = new ControllerFeature();
            _partManager.PopulateFeature(controllerFeature);
            viewModel.Controllers = controllerFeature.Controllers.ToList();

            var tagHelperFeature = new TagHelperFeature();
            _partManager.PopulateFeature(tagHelperFeature);
            viewModel.TagHelpers = tagHelperFeature.TagHelpers.ToList();

            var viewComponentFeature = new ViewComponentFeature();
            _partManager.PopulateFeature(viewComponentFeature);
            viewModel.ViewComponents =
viewComponentFeature.ViewComponents.ToList();

            return View(viewModel);
        }
    }
}
```

The download sample uses the preceding code to display the app features:

text

```
Controllers:
    - FeaturesController
    - HomeController
    - HelloController
    - GenericController`1
    - GenericController`1
Tag Helpers:
    - PrerenderTagHelper
    - AnchorTagHelper
    - CacheTagHelper
    - DistributedCacheTagHelper
    - EnvironmentTagHelper
    - Additional Tag Helpers omitted for brevity.
View Components:
    - SampleViewComponent
```

Discovery in application parts

HTTP 404 errors are not uncommon when developing with application parts. These errors are typically caused by missing an essential requirement for how applications parts are discovered. If your app returns an HTTP 404 error, verify the following requirements have been met:

- The `applicationName` setting needs to be set to the root assembly used for discovery. The root assembly used for discovery is normally the entry point assembly.
- The root assembly needs to have a reference to the parts used for discovery. The reference can be direct or transitive.
- The root assembly needs to reference the Web SDK. The framework has logic that stamps attributes into the root assembly that are used for discovery.

Work with the application model in ASP.NET Core

Article • 06/03/2022

By Steve Smith

ASP.NET Core MVC defines an *application model* representing the components of an MVC app. Read and manipulate this model to modify how MVC elements behave. By default, MVC follows certain conventions to determine which classes are considered controllers, which methods on those classes are actions, and how parameters and routing behave. Customize this behavior to suit an app's needs by creating custom conventions and applying them globally or as attributes.

Models and Providers (`IApplicationModelProvider`)

The ASP.NET Core MVC application model includes both abstract interfaces and concrete implementation classes that describe an MVC application. This model is the result of MVC discovering the app's controllers, actions, action parameters, routes, and filters according to default conventions. By working with the application model, modify an app to follow different conventions from the default MVC behavior. The parameters, names, routes, and filters are all used as configuration data for actions and controllers.

The ASP.NET Core MVC Application Model has the following structure:

- ApplicationModel
  - Controllers (ControllerModel)
    - Actions (ActionModel)
      - Parameters (ParameterModel)

Each level of the model has access to a common `Properties` collection, and lower levels can access and overwrite property values set by higher levels in the hierarchy. The properties are persisted to the ActionDescriptor.Properties when the actions are created. Then when a request is being handled, any properties a convention added or modified can be accessed through ActionContext.ActionDescriptor. Using properties is a great way to configure filters, model binders, and other app model aspects on a per-action basis.

> **Note**

The ActionDescriptor.Properties collection isn't thread safe (for writes) after app startup. Conventions are the best way to safely add data to this collection.

ASP.NET Core MVC loads the application model using a provider pattern, defined by the IApplicationModelProvider interface. This section covers some of the internal implementation details of how this provider functions. Use of the provider pattern is an advanced subject, primarily for framework use. Most apps should use conventions, not the provider pattern.

Implementations of the IApplicationModelProvider interface "wrap" one another, where each implementation calls OnProvidersExecuting in ascending order based on its Order property. The OnProvidersExecuted method is then called in reverse order. The framework defines several providers:

First (`Order=-1000`):

- `DefaultApplicationModelProvider`

Then (`Order=-990`):

- `AuthorizationApplicationModelProvider`
- `CorsApplicationModelProvider`

> ⓘ Note
>
> The order in which two providers with the same value for `Order` are called is undefined and shouldn't be relied upon.

> ⓘ Note
>
> IApplicationModelProvider is an advanced concept for framework authors to extend. In general, apps should use conventions, and frameworks should use providers. The key distinction is that providers always run before conventions.

The `DefaultApplicationModelProvider` establishes many of the default behaviors used by ASP.NET Core MVC. Its responsibilities include:

- Adding global filters to the context
- Adding controllers to the context
- Adding public controller methods as actions
- Adding action method parameters to the context

- Applying route and other attributes

Some built-in behaviors are implemented by the `DefaultApplicationModelProvider`. This provider is responsible for constructing the ControllerModel, which in turn references ActionModel, PropertyModel, and ParameterModel instances. The `DefaultApplicationModelProvider` class is an internal framework implementation detail that may change in the future.

The `AuthorizationApplicationModelProvider` is responsible for applying the behavior associated with the AuthorizeFilter and AllowAnonymousFilter attributes. For more information, see Simple authorization in ASP.NET Core.

The `CorsApplicationModelProvider` implements behavior associated with IEnableCorsAttribute and IDisableCorsAttribute. For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core.

Information on the framework's internal providers described in this section aren't available via the .NET API browser. However, the providers may be inspected in the ASP.NET Core reference source (dotnet/aspnetcore GitHub repository) . Use GitHub search to find the providers by name and select the version of the source with the **Switch branches/tags** dropdown list.

Conventions

The application model defines convention abstractions that provide a simpler way to customize the behavior of the models than overriding the entire model or provider. These abstractions are the recommended way to modify an app's behavior. Conventions provide a way to write code that dynamically applies customizations. While filters provide a means of modifying the framework's behavior, customizations permit control over how the whole app works together.

The following conventions are available:

- IApplicationModelConvention
- IControllerModelConvention
- IActionModelConvention
- IParameterModelConvention

Conventions are applied by adding them to MVC options or by implementing attributes and applying them to controllers, actions, or action parameters (similar to filters).Unlike filters, conventions are only executed when the app is starting, not as part of each request.

> **Note**
>
> For information on Razor Pages route and application model provider conventions, see Razor Pages route and app conventions in ASP.NET Core.

Modify the `ApplicationModel`

The following convention is used to add a property to the application model:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ApplicationDescription : IApplicationModelConvention
    {
        private readonly string _description;

        public ApplicationDescription(string description)
        {
            _description = description;
        }

        public void Apply(ApplicationModel application)
        {
            application.Properties["description"] = _description;
        }
    }
}
```

Application model conventions are applied as options when MVC is added in `Startup.ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application
Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
    });
}
```

Properties are accessible from the ActionDescriptor.Properties collection within controller actions:

C#

```
public class AppModelController : Controller
{
    public string Description()
    {
        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

Modify the `ControllerModel` description

The controller model can also include custom properties. Custom properties override existing properties with the same name specified in the application model. The following convention attribute adds a description at the controller level:

C#

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ControllerDescriptionAttribute : Attribute,
IControllerModelConvention
    {
        private readonly string _description;

        public ControllerDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ControllerModel controllerModel)
        {
            controllerModel.Properties["description"] = _description;
        }
    }
}
```

This convention is applied as an attribute on a controller:

C#

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }
```

Modify the `ActionModel` description

A separate attribute convention can be applied to individual actions, overriding behavior already applied at the application or controller level:

C#

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class ActionDescriptionAttribute : Attribute,
IActionModelConvention
    {
        private readonly string _description;

        public ActionDescriptionAttribute(string description)
        {
            _description = description;
        }

        public void Apply(ActionModel actionModel)
        {
            actionModel.Properties["description"] = _description;
        }
    }
}
```

Applying this to an action within the controller demonstrates how it overrides the controller-level convention:

C#

```
[ControllerDescription("Controller Description")]
public class DescriptionAttributesController : Controller
{
    public string Index()
    {
```

```
        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }

    [ActionDescription("Action Description")]
    public string UseActionDescriptionAttribute()
    {
        return "Description: " +
ControllerContext.ActionDescriptor.Properties["description"];
    }
}
```

Modify the `ParameterModel`

The following convention can be applied to action parameters to modify their BindingInfo. The following convention requires that the parameter be a route parameter. Other potential binding sources, such as query string values, are ignored:

C#

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace AppModelSample.Conventions
{
    public class MustBeInRouteParameterModelConvention : Attribute,
IParameterModelConvention
    {
        public void Apply(ParameterModel model)
        {
            if (model.BindingInfo == null)
            {
                model.BindingInfo = new BindingInfo();
            }
            model.BindingInfo.BindingSource = BindingSource.Path;
        }
    }
}
```

The attribute may be applied to any action parameter:

C#

```
public class ParameterModelController : Controller
{
    // Will bind:  /ParameterModel/GetById/123
    // WON'T bind: /ParameterModel/GetById?id=123
    public string GetById([MustBeInRouteParameterModelConvention]int id)
```

```
    {
        return $"Bound to id: {id}";
    }
}
```

To apply the convention to all action parameters, add the `MustBeInRouteParameterModelConvention` to MvcOptions in `Startup.ConfigureServices`:

C#

```
options.Conventions.Add(new MustBeInRouteParameterModelConvention());
```

Modify the `ActionModel` name

The following convention modifies the ActionModel to update the *name* of the action to which it's applied. The new name is provided as a parameter to the attribute. This new name is used by routing, so it affects the route used to reach this action method:

C#

```
using System;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class CustomActionNameAttribute : Attribute,
IActionModelConvention
    {
        private readonly string _actionName;

        public CustomActionNameAttribute(string actionName)
        {
            _actionName = actionName;
        }

        public void Apply(ActionModel actionModel)
        {
            // this name will be used by routing
            actionModel.ActionName = _actionName;
        }
    }
}
```

This attribute is applied to an action method in the `HomeController`:

C#

```
// Route: /Home/MyCoolAction
[CustomActionName("MyCoolAction")]
public string SomeName()
{
    return ControllerContext.ActionDescriptor.ActionName;
}
```

Even though the method name is `SomeName`, the attribute overrides the MVC convention of using the method name and replaces the action name with `MyCoolAction`. Thus, the route used to reach this action is `/Home/MyCoolAction`.

> ⓘ **Note**
>
> This example in this section is essentially the same as using the built-in ActionNameAttribute.

Custom routing convention

Use an IApplicationModelConvention to customize how routing works. For example, the following convention incorporates controllers' namespaces into their routes, replacing `.` in the namespace with `/` in the route:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;

namespace AppModelSample.Conventions
{
    public class NamespaceRoutingConvention : IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            foreach (var controller in application.Controllers)
            {
                var hasAttributeRouteModels = controller.Selectors
                    .Any(selector => selector.AttributeRouteModel != null);

                if (!hasAttributeRouteModels
                    && controller.ControllerName.Contains("Namespace")) //
affect one controller in this sample
                {
                    // Replace the . in the namespace with a / to create the
attribute route
                    // Ex: MySite.Admin namespace will correspond to
MySite/Admin attribute route
```

```
                    // Then attach [controller], [action] and optional {id?}
token.
                    // [Controller] and [action] is replaced with the
controller and action
                    // name to generate the final template
                    controller.Selectors[0].AttributeRouteModel = new
AttributeRouteModel()
                    {
                        Template =
controller.ControllerType.Namespace.Replace('.', '/') +
"/[controller]/[action]/{id?}"
                    };
                }
            }

            // You can continue to put attribute route templates for the
controller actions depending on the way you want them to behave
        }
    }
}
```

The convention is added as an option in `Startup.ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Conventions.Add(new ApplicationDescription("My Application
Description"));
        options.Conventions.Add(new NamespaceRoutingConvention());
    });
}
```

Tip

Add conventions to **middleware** via **MvcOptions** using the following approach. The `{CONVENTION}` placeholder is the convention to add:

C#

```
services.Configure<MvcOptions>(c => c.Conventions.Add({CONVENTION}));
```

The following example applies a convention to routes that aren't using attribute routing where the controller has `Namespace` in its name:

C#

```
using Microsoft.AspNetCore.Mvc;

namespace AppModelSample.Controllers
{
    public class NamespaceRoutingController : Controller
    {
        // using NamespaceRoutingConvention
        // route: /AppModelSample/Controllers/NamespaceRouting/Index
        public string Index()
        {
            return "This demonstrates namespace routing.";
        }
    }
}
```

Use `ApiExplorer` to document an app

The application model exposes an ApiExplorerModel property at each level that can be used to traverse the app's structure. This can be used to generate help pages for web APIs using tools like Swagger. The `ApiExplorer` property exposes an IsVisible property that can be set to specify which parts of the app's model should be exposed. Configure this setting using a convention:

C#

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace AppModelSample.Conventions
{
    public class EnableApiExplorerApplicationConvention :
IApplicationModelConvention
    {
        public void Apply(ApplicationModel application)
        {
            application.ApiExplorer.IsVisible = true;
        }
    }
}
```

Using this approach (and additional conventions if required), API visibility is enabled or disabled at any level within an app.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Areas in ASP.NET Core

Article • 06/03/2022

By Dhananjay Kumar and Rick Anderson

Areas are an ASP.NET feature used to organize related functionality into a group as a separate:

- Namespace for routing.
- Folder structure for views and Razor Pages.

Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area`, to `controller` and `action` or a Razor Page `page`.

Areas provide a way to partition an ASP.NET Core Web app into smaller functional groups, each with its own set of Razor Pages, controllers, views, and models. An area is effectively a structure inside an app. In an ASP.NET Core web project, logical components like Pages, Model, Controller, and View are kept in different folders. The ASP.NET Core runtime uses naming conventions to create the relationship between these components. For a large app, it may be advantageous to partition the app into separate high level areas of functionality. For instance, an e-commerce app with multiple business units, such as checkout, billing, and search. Each of these units have their own area to contain views, controllers, Razor Pages, and models.

Consider using Areas in a project when:

- The app is made of multiple high-level functional components that can be logically separated.
- You want to partition the app so that each functional area can be worked on independently.

If you're using Razor Pages, see Areas with Razor Pages in this document.

Areas for controllers with views

A typical ASP.NET Core web app using areas, controllers, and views contains the following:

- An Area folder structure.
- Controllers with the [Area] attribute to associate the controller with the area:

```
C#
```

```
[Area("Products")]
public class ManageController : Controller
{
```

- The area route added to Program.cs:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "MyArea",
    pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Area folder structure

Consider an app that has two logical groups, *Products* and *Services*. Using areas, the folder structure would be similar to the following:

- Project name
  - Areas
    - Products
      - Controllers
        - HomeController.cs
        - ManageController.cs

        - Views
            - Home
                - Index.cshtml
            - Manage
                - Index.cshtml
                - About.cshtml
    - Services
        - Controllers
            - HomeController.cs
        - Views
            - Home
                - Index.cshtml

While the preceding layout is typical when using Areas, only the view files are required to use this folder structure. View discovery searches for a matching area view file in the following order:

```text
/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml
/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml
/Views/Shared/<Action-Name>.cshtml
/Pages/Shared/<Action-Name>.cshtml
```

Associate the controller with an Area

Area controllers are designated with the [Area] attribute:

```C#
using Microsoft.AspNetCore.Mvc;
using Microsoft.Docs.Samples;

namespace MVCareas.Areas.Products.Controllers;

[Area("Products")]
public class ManageController : Controller
{
    public IActionResult Index()
    {
        ViewData["routeInfo"] = ControllerContext.MyDisplayRouteInfo();
        return View();
    }

    public IActionResult About()
    {
```

```
        ViewData["routeInfo"] = ControllerContext.MyDisplayRouteInfo();
        return View();
    }
}
```

Add Area route

Area routes typically use conventional routing rather than attribute routing. Conventional routing is order-dependent. In general, routes with areas should be placed earlier in the route table as they're more specific than routes without an area.

`{area:...}` can be used as a token in route templates if url space is uniform across all areas:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "MyArea",
    pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

In the preceding code, `exists` applies a constraint that the route must match an area. Using `{area:...}` with `MapControllerRoute`:

- Is the least complicated mechanism to adding routing to areas.

- Matches all controllers with the `[Area("Area name")]` attribute.

The following code uses MapAreaControllerRoute to create two named area routes:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapAreaControllerRoute(
    name: "MyAreaProducts",
    areaName: "Products",
    pattern: "Products/{controller=Home}/{action=Index}/{id?}");

app.MapAreaControllerRoute(
    name: "MyAreaServices",
    areaName: "Services",
    pattern: "Services/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

For more information, see Area routing.

Link generation with MVC areas

The following code from the sample download shows link generation with the area specified:

CSHTML

```
<li>Anchor Tag Helper links</li>
<ul>
    <li>
        <a asp-area="Products" asp-controller="Home" asp-action="About">
            Products/Home/About
        </a>
    </li>
    <li>
        <a asp-area="Services" asp-controller="Home" asp-action="About">
            Services About
        </a>
    </li>
    <li>
        <a asp-area="" asp-controller="Home" asp-action="About">
            /Home/About
        </a>
    </li>
</ul>
<li>Html.ActionLink generated links</li>
<ul>
    <li>
        @Html.ActionLink("Product/Manage/About", "About", "Manage",
                                                 new { area = "Products" })
    </li>
</ul>
<li>Url.Action generated links</li>
<ul>
    <li>
        <a href='@Url.Action("About", "Manage", new { area = "Products" })'>
            Products/Manage/About
        </a>
    </li>
</ul>
```

The sample download includes a partial view that contains:

- The preceding links.
- Links similar to the preceding except `area` is not specified.

The partial view is referenced in the layout file, so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area and controller.

When the area or controller is not specified, routing depends on the ambient values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links with the markup that doesn't specify the area.

For more information, see Routing to controller actions.

Shared layout for Areas using the _ViewStart.cshtml file

To share a common layout for the entire app, keep the *_ViewStart.cshtml* in the application root folder. For more information, see Layout in ASP.NET Core

Application root folder

The application root folder is the folder containing the `Program.cs` file in a web app created with the ASP.NET Core templates.

_ViewImports.cshtml

/Views/_ViewImports.cshtml, for MVC, and */Pages/_ViewImports.cshtml* for Razor Pages, is not imported to views in areas. Use one of the following approaches to provide view imports to all views:

- Add *_ViewImports.cshtml* to the application root folder. A *_ViewImports.cshtml* in the application root folder will apply to all views in the app.
- Copy the *_ViewImports.cshtml* file to the appropriate view folder under areas. For example, a Razor Pages app created with individual user accounts has a *_ViewImports.cshtml* file in the following folders:
  - */Areas/Identity/Pages/_ViewImports.cshtml*
  - */Pages/_ViewImports.cshtml*

The *_ViewImports.cshtml* file typically contains Tag Helpers imports, `@using`, and `@inject` statements. For more information, see Importing Shared Directives.

Change default area folder where views are stored

The following code changes the default area folder from `"Areas"` to `"MyAreas"`:

C#

```
using Microsoft.AspNetCore.Mvc.Razor;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.Configure<RazorViewEngineOptions>(options =>
{
    options.AreaViewLocationFormats.Clear();

options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/{1}/{0}.cshtml");

options.AreaViewLocationFormats.Add("/MyAreas/{2}/Views/Shared/{0}.cshtml");
    options.AreaViewLocationFormats.Add("/Views/Shared/{0}.cshtml");
});
```

```
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "MyArea",
    pattern: "{area:exists}/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Areas with Razor Pages

Areas with Razor Pages require an `Areas/<area name>/Pages` folder in the root of the app. The following folder structure is used with the sample app:

- Project name
  - Areas
    - Products
      - Pages
        - _ViewImports
        - About
        - Index

- Services
  - Pages
    - Manage
      - About
      - Index

Link generation with Razor Pages and areas

The following code from the sample download shows link generation with the area specified (for example, `asp-area="Products"`):

CSHTML

```
<li>Anchor Tag Helper links</li>
<ul>
    <li>
        <a asp-area="Products" asp-page="/About">
            Products/About
        </a>
    </li>
    <li>
        <a asp-area="Services" asp-page="/Manage/About">
            Services/Manage/About
        </a>
    </li>
    <li>
        <a asp-area="" asp-page="/About">
            /About
        </a>
    </li>
</ul>
<li>Url.Page generated links</li>
<ul>
    <li>
        <a href='@Url.Page("/Manage/About", new { area = "Services" })'>
            Services/Manage/About
        </a>
    </li>
    <li>
        <a href='@Url.Page("/About", new { area = "Products" })'>
            Products/About
        </a>
    </li>
</ul>
```

The sample download includes a partial view that contains the preceding links and the same links without specifying the area. The partial view is referenced in the layout file, so every page in the app displays the generated links. The links generated without specifying the area are only valid when referenced from a page in the same area.

When the area is not specified, routing depends on the *ambient* values. The current route values of the current request are considered ambient values for link generation. In many cases for the sample app, using the ambient values generates incorrect links. For example, consider the links generated from the following code:

CSHTML

```
<li>
    <a asp-page="/Manage/About">
        Services/Manage/About
    </a>
</li>
<li>
    <a asp-page="/About">
        /About
    </a>
</li>
```

For the preceding code:

- The link generated from `<a asp-page="/Manage/About">` is correct only when the last request was for a page in `Services` area. For example, `/Services/Manage/`, `/Services/Manage/Index`, or `/Services/Manage/About`.
- The link generated from `<a asp-page="/About">` is correct only when the last request was for a page in `/Home`.
- The code is from the sample download.

Import namespace and Tag Helpers with _ViewImports file

A *_ViewImports.cshtml* file can be added to each area *Pages* folder to import the namespace and Tag Helpers to each Razor Page in the folder.

Consider the *Services* area of the sample code, which doesn't contain a *_ViewImports.cshtml* file. The following markup shows the */Services/Manage/About* Razor Page:

CSHTML

```
@page
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@model RPareas.Areas.Services.Pages.Manage.AboutModel
@{
    ViewData["Title"] = "Srv Mng About";
}
```

```
<div>
  ViewData["routeInfo"]:  @ViewData["routeInfo"]
</div>

<a asp-area="Products" asp-page="/Index">
    Products/Index
</a>
```

In the preceding markup:

- The fully qualified class name must be used to specify the model (`@model RPareas.Areas.Services.Pages.Manage.AboutModel`).
- Tag Helpers are enabled by `@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers`

In the sample download, the Products area contains the following *_ViewImports.cshtml* file:

CSHTML

```
@namespace RPareas.Areas.Products.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The following markup shows the */Products/About* Razor Page:

CSHTML

```
@page
@model AboutModel
@{
    ViewData["Title"] = "Prod About";
}
```

In the preceding file, the namespace and `@addTagHelper` directive is imported to the file by the *Areas/Products/Pages/_ViewImports.cshtml* file.

For more information, see Managing Tag Helper scope and Importing Shared Directives.

Shared layout for Razor Pages Areas

To share a common layout for the entire app, move the *_ViewStart.cshtml* to the application root folder.

Publishing Areas

All *.cshtml files and files within the *wwwroot* directory are published to output when `<Project Sdk="Microsoft.NET.Sdk.Web">` is included in the *.csproj file.

Add MVC Area with Visual Studio

In Solution Explorer, right click the project and select **ADD** > **New Scaffolded Item**, then select **MVC Area**.

Additional resources

- View or download sample code (how to download). The download sample provides a basic app for testing areas.
- MyDisplayRouteInfo and ToCtxString are provided by the Rick.Docs.Samples.RouteInfo NuGet package. The methods display `Controller` and `Razor Page` route information.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Filters in ASP.NET Core

Article • 06/20/2023

By Kirk Larkin, Rick Anderson, Tom Dykstra, and Steve Smith

Filters in ASP.NET Core allow code to run before or after specific stages in the request processing pipeline.

Built-in filters handle tasks such as:

- Authorization, preventing access to resources a user isn't authorized for.
- Response caching, short-circuiting the request pipeline to return a cached response.

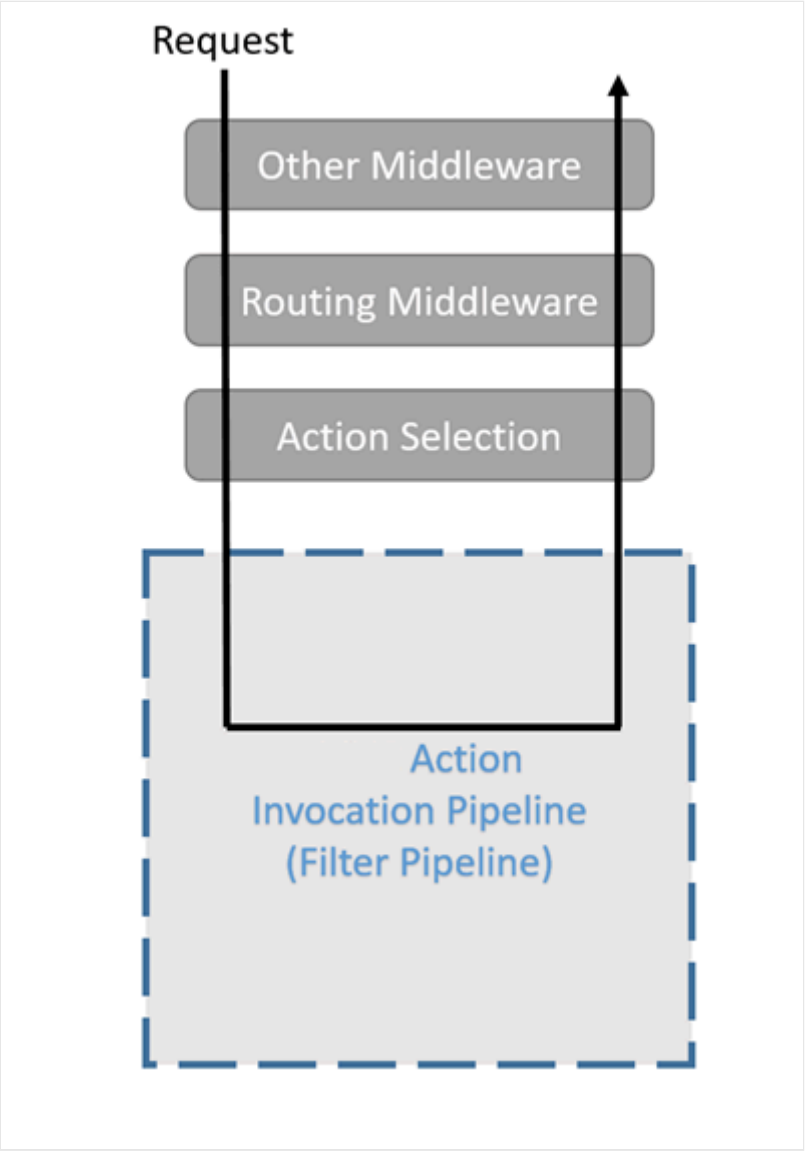
Custom filters can be created to handle cross-cutting concerns. Examples of cross-cutting concerns include error handling, caching, configuration, authorization, and logging. Filters avoid duplicating code. For example, an error handling exception filter could consolidate error handling.

This document applies to Razor Pages, API controllers, and controllers with views. Filters don't work directly with Razor components. A filter can only indirectly affect a component when:

- The component is embedded in a page or view.
- The page or controller and view uses the filter.

How filters work

Filters run within the *ASP.NET Core action invocation pipeline*, sometimes referred to as the *filter pipeline*. The filter pipeline runs after ASP.NET Core selects the action to execute:



Filter types

Each filter type is executed at a different stage in the filter pipeline:

- Authorization filters:
  - Run first.
  - Determine whether the user is authorized for the request.
  - Short-circuit the pipeline if the request is not authorized.
- Resource filters:
  - Run after authorization.
  - OnResourceExecuting runs code before the rest of the filter pipeline. For example, `OnResourceExecuting` runs code before model binding.
  - OnResourceExecuted runs code after the rest of the pipeline has completed.
- Action filters:
  - Run immediately before and after an action method is called.
  - Can change the arguments passed into an action.

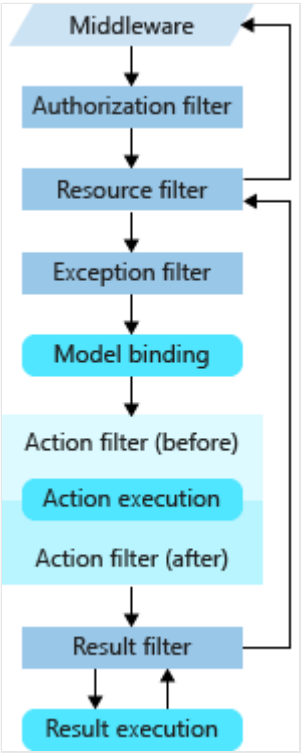
    - Can change the result returned from the action.
    - Are **not** supported in Razor Pages.

- Endpoint filters:
    - Run immediately before and after an action method is called.
    - Can change the arguments passed into an action.
    - Can change the result returned from the action.
    - Are **not** supported in Razor Pages.
    - Can be invoked on both actions and route handler-based endpoints.

- Exception filters apply global policies to unhandled exceptions that occur before the response body has been written to.

- Result filters:
    - Run immediately before and after the execution of action results.
    - Run only when the action method executes successfully.
    - Are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline:



Razor Pages also support Razor Page filters, which run before and after a Razor Page handler.

Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions.

Synchronous filters run before and after their pipeline stage. For example, OnActionExecuting is called before the action method is called. OnActionExecuted is called after the action method returns:

```
C#
```

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

Asynchronous filters define an `On-Stage-ExecutionAsync` method. For example, OnActionExecutionAsync:

```
C#
```

```
public class SampleAsyncActionFilter : IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context, ActionExecutionDelegate next)
    {
        // Do something before the action executes.
        await next();
        // Do something after the action executes.
    }
}
```

In the preceding code, the `SampleAsyncActionFilter` has an ActionExecutionDelegate, `next`, which executes the action method.

Multiple filter stages

Interfaces for multiple filter stages can be implemented in a single class. For example, the ActionFilterAttribute class implements:

- Synchronous: IActionFilter and IResultFilter
- Asynchronous: IAsyncActionFilter and IAsyncResultFilter
- IOrderedFilter

Implement **either** the synchronous or the async version of a filter interface, **not** both. The runtime checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both asynchronous and synchronous interfaces are implemented in one class, only the async method is called. When using abstract classes like ActionFilterAttribute, override only the synchronous methods or the asynchronous methods for each filter type.

Built-in filter attributes

ASP.NET Core includes built-in attribute-based filters that can be subclassed and customized. For example, the following result filter adds a header to the response:

C#

```
public class ResponseHeaderAttribute : ActionFilterAttribute
{
    private readonly string _name;
    private readonly string _value;

    public ResponseHeaderAttribute(string name, string value) =>
        (_name, _value) = (name, value);

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_name, _value);

        base.OnResultExecuting(context);
    }
}
```

Attributes allow filters to accept arguments, as shown in the preceding example. Apply the `ResponseHeaderAttribute` to a controller or action method and specify the name and value of the HTTP header:

C#

```
[ResponseHeader("Filter-Header", "Filter Value")]
public class ResponseHeaderController : ControllerBase
{
    public IActionResult Index() =>
        Content("Examine the response headers using the F12 developer
tools.");

    // ...
```

Use a tool such as the browser developer tools to examine the headers. Under **Response Headers**, `filter-header: Filter Value` is displayed.

The following code applies `ResponseHeaderAttribute` to both a controller and an action:

```
C#

[ResponseHeader("Filter-Header", "Filter Value")]
public class ResponseHeaderController : ControllerBase
{
    public IActionResult Index() =>
        Content("Examine the response headers using the F12 developer
tools.");

    // ...

    [ResponseHeader("Another-Filter-Header", "Another Filter Value")]
    public IActionResult Multiple() =>
        Content("Examine the response headers using the F12 developer
tools.");
}
```

Responses from the `Multiple` action include the following headers:

- `filter-header: Filter Value`
- `another-filter-header: Another Filter Value`

Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

- ActionFilterAttribute
- ExceptionFilterAttribute
- ResultFilterAttribute
- FormatFilterAttribute
- ServiceFilterAttribute
- TypeFilterAttribute

Filters cannot be applied to Razor Page handler methods. They can be applied either to the Razor Page model or globally.

Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*:

- Using an attribute on a controller or Razor Page.
- Using an attribute on a controller action. Filter attributes cannot be applied to Razor Pages handler methods.
- Globally for all controllers, actions, and Razor Pages as shown in the following code:

```
C#

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add<GlobalSampleActionFilter>();
});
```

Default order of execution

When there are multiple filters for a particular stage of the pipeline, scope determines the default order of filter execution. Global filters surround class filters, which in turn surround method filters.

As a result of filter nesting, the *after* code of filters runs in the reverse order of the *before* code. The filter sequence:

- The *before* code of global filters.
  - The *before* code of controller filters.
    - The *before* code of action method filters.
    - The *after* code of action method filters.
  - The *after* code of controller filters.
- The *after* code of global filters.

The following example illustrates the order in which filter methods run for synchronous action filters:

Expand table

Sequence	Filter scope	Filter method
1	Global	`OnActionExecuting`
2	Controller	`OnActionExecuting`
3	Action	`OnActionExecuting`

Sequence	Filter scope	Filter method
4	Action	`OnActionExecuted`
5	Controller	`OnActionExecuted`
6	Global	`OnActionExecuted`

Controller level filters

Every controller that inherits from Controller includes the OnActionExecuting, OnActionExecutionAsync, and OnActionExecuted methods. These methods wrap the filters that run for a given action:

- `OnActionExecuting` runs before any of the action's filters.
- `OnActionExecuted` runs after all of the action's filters.
- `OnActionExecutionAsync` runs before any of the action's filters. Code after a call to `next` runs after the action's filters.

The following `ControllerFiltersController` class:

- Applies the `SampleActionFilterAttribute` (`[SampleActionFilter]`) to the controller.
- Overrides `OnActionExecuting` and `OnActionExecuted`.

C#

```
[SampleActionFilter]
public class ControllerFiltersController : Controller
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine(
            $"- {nameof(ControllerFiltersController)}.
{nameof(OnActionExecuting)}");

        base.OnActionExecuting(context);
    }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
        Console.WriteLine(
            $"- {nameof(ControllerFiltersController)}.
{nameof(OnActionExecuted)}");

        base.OnActionExecuted(context);
    }

    public IActionResult Index()
```

```
    {
        Console.WriteLine(
            $"- {nameof(ControllerFiltersController)}.{nameof(Index)}");

        return Content("Check the Console.");
    }
}
```

Navigating to `https://localhost:<port>/ControllerFilters` runs the following code:

- `ControllerFiltersController.OnActionExecuting`
  - `GlobalSampleActionFilter.OnActionExecuting`
    - `SampleActionFilterAttribute.OnActionExecuting`
      - `ControllerFiltersController.Index`
    - `SampleActionFilterAttribute.OnActionExecuted`
  - `GlobalSampleActionFilter.OnActionExecuted`
- `ControllerFiltersController.OnActionExecuted`

Controller level filters set the Order property to `int.MinValue`. Controller level filters can **not** be set to run after filters applied to methods. Order is explained in the next section.

For Razor Pages, see Implement Razor Page filters by overriding filter methods.

Override the default order

The default sequence of execution can be overridden by implementing IOrderedFilter. `IOrderedFilter` exposes the Order property that takes precedence over scope to determine the order of execution. A filter with a lower `Order` value:

- Runs the *before* code before that of a filter with a higher value of `Order`.
- Runs the *after* code after that of a filter with a higher `Order` value.

In the Controller level filters example, `GlobalSampleActionFilter` has global scope so it runs before `SampleActionFilterAttribute`, which has controller scope. To make `SampleActionFilterAttribute` run first, set its order to `int.MinValue`:

C#

```
[SampleActionFilter(Order = int.MinValue)]
public class ControllerFiltersController : Controller
{
    // ...
}
```

To make the global filter `GlobalSampleActionFilter` run first, set its `Order` to `int.MinValue`:

```
C#

builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add<GlobalSampleActionFilter>(int.MinValue);
});
```

Cancellation and short-circuiting

The filter pipeline can be short-circuited by setting the Result property on the ResourceExecutingContext parameter provided to the filter method. For example, the following Resource filter prevents the rest of the pipeline from executing:

```
C#

public class ShortCircuitingResourceFilterAttribute : Attribute,
IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        context.Result = new ContentResult
        {
            Content = nameof(ShortCircuitingResourceFilterAttribute)
        };
    }

    public void OnResourceExecuted(ResourceExecutedContext context) { }
}
```

In the following code, both the `[ShortCircuitingResourceFilter]` and the `[ResponseHeader]` filter target the `Index` action method. The `ShortCircuitingResourceFilterAttribute` filter:

- Runs first, because it's a Resource Filter and `ResponseHeaderAttribute` is an Action Filter.
- Short-circuits the rest of the pipeline.

Therefore the `ResponseHeaderAttribute` filter never runs for the `Index` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilterAttribute` ran first. The `ShortCircuitingResourceFilterAttribute` runs first because of its filter type:

C#

```
[ResponseHeader("Filter-Header", "Filter Value")]
public class ShortCircuitingController : Controller
{
    [ShortCircuitingResourceFilter]
    public IActionResult Index() =>
        Content($"- {nameof(ShortCircuitingController)}.{nameof(Index)}");
}
```

Dependency injection

Filters can be added by type or by instance. If an instance is added, that instance is used for every request. If a type is added, it's type-activated. A type-activated filter means:

- An instance is created for each request.
- Any constructor dependencies are populated by dependency injection (DI).

Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by dependency injection (DI). Constructor dependencies cannot be provided by DI because attributes must have their constructor parameters supplied where they're applied.

The following filters support constructor dependencies provided from DI:

- ServiceFilterAttribute
- TypeFilterAttribute
- IFilterFactory implemented on the attribute.

The preceding filters can be applied to a controller or an action.

Loggers are available from DI. However, avoid creating and using filters purely for logging purposes. The built-in framework logging typically provides what's needed for logging. Logging added to filters:

- Should focus on business domain concerns or behavior specific to the filter.
- Should **not** log actions or other framework events. The built-in filters already log actions and framework events.

ServiceFilterAttribute

Service filter implementation types are registered in `Program.cs`. A ServiceFilterAttribute retrieves an instance of the filter from DI.

The following code shows the `LoggingResponseHeaderFilterService` class, which uses DI:

C#

```
public class LoggingResponseHeaderFilterService : IResultFilter
{
    private readonly ILogger _logger;

    public LoggingResponseHeaderFilterService(
            ILogger<LoggingResponseHeaderFilterService> logger) =>
        _logger = logger;

    public void OnResultExecuting(ResultExecutingContext context)
    {
        _logger.LogInformation(
            $"- {nameof(LoggingResponseHeaderFilterService)}.
{nameof(OnResultExecuting)}");

        context.HttpContext.Response.Headers.Add(
            nameof(OnResultExecuting),
nameof(LoggingResponseHeaderFilterService));
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        _logger.LogInformation(
            $"- {nameof(LoggingResponseHeaderFilterService)}.
{nameof(OnResultExecuted)}");
    }
}
```

In the following code, `LoggingResponseHeaderFilterService` is added to the DI container:

C#

```
builder.Services.AddScoped<LoggingResponseHeaderFilterService>();
```

In the following code, the `ServiceFilter` attribute retrieves an instance of the `LoggingResponseHeaderFilterService` filter from DI:

C#

```
[ServiceFilter<LoggingResponseHeaderFilterService>]
public IActionResult WithServiceFilter() =>
    Content($"- {nameof(FilterDependenciesController)}.
{nameof(WithServiceFilter)}");
```

When using `ServiceFilterAttribute`, setting ServiceFilterAttribute.IsReusable:

- Provides a hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime doesn't guarantee:
  - That a single instance of the filter will be created.
  - The filter will not be re-requested from the DI container at some later point.
- Shouldn't be used with a filter that depends on services with a lifetime other than singleton.

ServiceFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from DI.

TypeFilterAttribute

TypeFilterAttribute is similar to ServiceFilterAttribute, but its type isn't resolved directly from the DI container. It instantiates the type by using Microsoft.Extensions.DependencyInjection.ObjectFactory.

Because `TypeFilterAttribute` types aren't resolved directly from the DI container:

- Types that are referenced using the `TypeFilterAttribute` don't need to be registered with the DI container. They do have their dependencies fulfilled by the DI container.
- `TypeFilterAttribute` can optionally accept constructor arguments for the type.

When using `TypeFilterAttribute`, setting TypeFilterAttribute.IsReusable:

- Provides hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime provides no guarantees that a single instance of the filter will be created.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

The following example shows how to pass arguments to a type using `TypeFilterAttribute`:

C#

```
[TypeFilter(typeof(LoggingResponseHeaderFilter),
    Arguments = new object[] { "Filter-Header", "Filter Value" })]
public IActionResult WithTypeFilter() =>
    Content($"- {nameof(FilterDependenciesController)}.
{nameof(WithTypeFilter)}");
```

Authorization filters

Authorization filters:

- Are the first filters run in the filter pipeline.
- Control access to action methods.
- Have a before method, but no after method.

Custom authorization filters require a custom authorization framework. Prefer configuring the authorization policies or writing a custom authorization policy over writing a custom filter. The built-in authorization filter:

- Calls the authorization system.
- Does not authorize requests.

Do **not** throw exceptions within authorization filters:

- The exception will not be handled.
- Exception filters will not handle the exception.

Consider issuing a challenge when an exception occurs in an authorization filter.

Learn more about Authorization.

Resource filters

Resource filters:

- Implement either the IResourceFilter or IAsyncResourceFilter interface.
- Execution wraps most of the filter pipeline.
- Only Authorization filters run before resource filters.

Resource filters are useful to short-circuit most of the pipeline. For example, a caching filter can avoid the rest of the pipeline on a cache hit.

Resource filter examples:

- The short-circuiting resource filter shown previously.
- DisableFormValueModelBindingAttribute:
  - Prevents model binding from accessing the form data.
  - Used for large file uploads to prevent the form data from being read into memory.

Action filters

Action filters do **not** apply to Razor Pages. Razor Pages supports IPageFilter and IAsyncPageFilter. For more information, see Filter methods for Razor Pages.

Action filters:

- Implement either the IActionFilter or IAsyncActionFilter interface.
- Their execution surrounds the execution of action methods.

The following code shows a sample action filter:

C#

```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

The ActionExecutingContext provides the following properties:

- ActionArguments - enables reading the inputs to an action method.
- Controller - enables manipulating the controller instance.
- Result - setting `Result` short-circuits execution of the action method and subsequent action filters.

Throwing an exception in an action method:

- Prevents running of subsequent filters.
- Unlike setting `Result`, is treated as a failure instead of a successful result.

The ActionExecutedContext provides `Controller` and `Result` plus the following properties:

- Canceled - True if the action execution was short-circuited by another filter.
- Exception - Non-null if the action or a previously run action filter threw an exception. Setting this property to null:
  - Effectively handles the exception.

    - `Result` is executed as if it was returned from the action method.

For an `IAsyncActionFilter`, a call to the ActionExecutionDelegate:

- Executes any subsequent action filters and the action method.
- Returns `ActionExecutedContext`.

To short-circuit, assign Microsoft.AspNetCore.Mvc.Filters.ActionExecutingContext.Result to a result instance and don't call `next` (the `ActionExecutionDelegate`).

The framework provides an abstract ActionFilterAttribute that can be subclassed.

The `OnActionExecuting` action filter can be used to:

- Validate model state.
- Return an error if the state is invalid.

C#

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

> **Note**
>
> Controllers annotated with the `[ApiController]` attribute automatically validate model state and return a 400 response. For more information, see Automatic HTTP 400 responses.

The `OnActionExecuted` method runs after the action method:

- And can see and manipulate the results of the action through the Result property.
- Canceled is set to true if the action execution was short-circuited by another filter.
- Exception is set to a non-null value if the action or a subsequent action filter threw an exception. Setting `Exception` to null:
    - Effectively handles an exception.
    - `ActionExecutedContext.Result` is executed as if it were returned normally from the action method.

Exception filters

Exception filters:

- Implement IExceptionFilter or IAsyncExceptionFilter.
- Can be used to implement common error handling policies.

The following sample exception filter displays details about exceptions that occur when the app is in development:

C#

```
public class SampleExceptionFilter : IExceptionFilter
{
    private readonly IHostEnvironment _hostEnvironment;

    public SampleExceptionFilter(IHostEnvironment hostEnvironment) =>
        _hostEnvironment = hostEnvironment;

    public void OnException(ExceptionContext context)
    {
        if (!_hostEnvironment.IsDevelopment())
        {
            // Don't display exception details unless running in
Development.
            return;
        }

        context.Result = new ContentResult
        {
            Content = context.Exception.ToString()
        };
    }
}
```

The following code tests the exception filter:

C#

```
[TypeFilter<SampleExceptionFilter>]
public class ExceptionController : Controller
{
    public IActionResult Index() =>
        Content($"- {nameof(ExceptionController)}.{nameof(Index)}");
}
```

Exception filters:

- Don't have before and after events.

- Implement OnException or OnExceptionAsync.
- Handle unhandled exceptions that occur in Razor Page or controller creation, model binding, action filters, or action methods.
- Do **not** catch exceptions that occur in resource filters, result filters, or MVC result execution.

To handle an exception, set the ExceptionHandled property to `true` or assign the Result property. This stops propagation of the exception. An exception filter can't turn an exception into a "success". Only an action filter can do that.

Exception filters:

- Are good for trapping exceptions that occur within actions.
- Are not as flexible as error handling middleware.

Prefer middleware for exception handling. Use exception filters only where error handling *differs* based on which action method is called. For example, an app might have action methods for both API endpoints and for views/HTML. The API endpoints could return error information as JSON, while the view-based actions could return an error page as HTML.

Result filters

Result filters:

- Implement an interface:
  - IResultFilter or IAsyncResultFilter
  - IAlwaysRunResultFilter or IAsyncAlwaysRunResultFilter
- Their execution surrounds the execution of action results.

IResultFilter and IAsyncResultFilter

The following code shows a sample result filter:

C#

```
public class SampleResultFilter : IResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        // Do something before the result executes.
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
```

```
        // Do something after the result executes.
    }
}
```

The kind of result being executed depends on the action. An action returning a view includes all razor processing as part of the ViewResult being executed. An API method might perform some serialization as part of the execution of the result. Learn more about action results.

Result filters are only executed when an action or action filter produces an action result. Result filters are not executed when:

- An authorization filter or resource filter short-circuits the pipeline.
- An exception filter handles an exception by producing an action result.

The Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuting method can short-circuit execution of the action result and subsequent result filters by setting Microsoft.AspNetCore.Mvc.Filters.ResultExecutingContext.Cancel to `true`. Write to the response object when short-circuiting to avoid generating an empty response. Throwing an exception in `IResultFilter.OnResultExecuting`:

- Prevents execution of the action result and subsequent filters.
- Is treated as a failure instead of a successful result.

When the Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuted method runs, the response has probably already been sent to the client. If the response has already been sent to the client, it cannot be changed.

`ResultExecutedContext.Canceled` is set to `true` if the action result execution was short-circuited by another filter.

`ResultExecutedContext.Exception` is set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `Exception` to null effectively handles an exception and prevents the exception from being thrown again later in the pipeline. There is no reliable way to write data to a response when handling an exception in a result filter. If the headers have been flushed to the client when an action result throws an exception, there's no reliable mechanism to send a failure code.

For an IAsyncResultFilter, a call to `await next` on the ResultExecutionDelegate executes any subsequent result filters and the action result. To short-circuit, set ResultExecutingContext.Cancel to `true` and don't call the `ResultExecutionDelegate`:

```
C#
```

```
public class SampleAsyncResultFilter : IAsyncResultFilter
{
    public async Task OnResultExecutionAsync(
        ResultExecutingContext context, ResultExecutionDelegate next)
    {
        if (context.Result is not EmptyResult)
        {
            await next();
        }
        else
        {
            context.Cancel = true;
        }
    }
}
```

The framework provides an abstract `ResultFilterAttribute` that can be subclassed. The ResponseHeaderAttribute class shown previously is an example of a result filter attribute.

IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter

The IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter interfaces declare an IResultFilter implementation that runs for all action results. This includes action results produced by:

- Authorization filters and resource filters that short-circuit.
- Exception filters.

For example, the following filter always runs and sets an action result (ObjectResult) with a *422 Unprocessable Entity* status code when content negotiation fails:

C#

```
public class UnprocessableResultFilter : IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        if (context.Result is StatusCodeResult statusCodeResult
            && statusCodeResult.StatusCode ==
StatusCodes.Status415UnsupportedMediaType)
        {
            context.Result = new ObjectResult("Unprocessable")
            {
                StatusCode = StatusCodes.Status422UnprocessableEntity
            };
        }
    }
```

```
    public void OnResultExecuted(ResultExecutedContext context) { }
}
```

IFilterFactory

IFilterFactory implements IFilterMetadata. Therefore, an `IFilterFactory` instance can be used as an `IFilterMetadata` instance anywhere in the filter pipeline. When the runtime prepares to invoke the filter, it attempts to cast it to an `IFilterFactory`. If that cast succeeds, the CreateInstance method is called to create the `IFilterMetadata` instance that is invoked. This provides a flexible design, since the precise filter pipeline doesn't need to be set explicitly when the app starts.

`IFilterFactory.IsReusable`:

- Is a hint by the factory that the filter instance created by the factory may be reused outside of the request scope it was created within.
- Should ***not*** be used with a filter that depends on services with a lifetime other than singleton.

The ASP.NET Core runtime doesn't guarantee:

- That a single instance of the filter will be created.
- The filter will not be re-requested from the DI container at some later point.

> ⚠ **Warning**
>
> Only configure **IFilterFactory.IsReusable** to return `true` if the source of the filters is unambiguous, the filters are stateless, and the filters are safe to use across multiple HTTP requests. For instance, don't return filters from DI that are registered as scoped or transient if `IFilterFactory.IsReusable` returns `true`.

`IFilterFactory` can be implemented using custom attribute implementations as another approach to creating filters:

C#

```
public class ResponseHeaderFilterFactory : Attribute, IFilterFactory
{
    public bool IsReusable => false;
```

```
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
=>
        new InternalResponseHeaderFilter();

    private class InternalResponseHeaderFilter : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context) =>
            context.HttpContext.Response.Headers.Add(
                nameof(OnActionExecuting),
nameof(InternalResponseHeaderFilter));

        public void OnActionExecuted(ActionExecutedContext context) { }
    }
```

The filter is applied in the following code:

C#

```
[ResponseHeaderFilterFactory]
public IActionResult Index() =>
    Content($"- {nameof(FilterFactoryController)}.{nameof(Index)}");
```

IFilterFactory implemented on an attribute

Filters that implement `IFilterFactory` are useful for filters that:

- Don't require passing parameters.
- Have constructor dependencies that need to be filled by DI.

TypeFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from the services container (DI).

C#

```
public class SampleActionTypeFilterAttribute : TypeFilterAttribute
{
    public SampleActionTypeFilterAttribute()
         : base(typeof(InternalSampleActionFilter)) { }

    private class InternalSampleActionFilter : IActionFilter
    {
        private readonly ILogger<InternalSampleActionFilter> _logger;

        public
InternalSampleActionFilter(ILogger<InternalSampleActionFilter> logger) =>
            _logger = logger;

        public void OnActionExecuting(ActionExecutingContext context)
```

```
        {
            _logger.LogInformation(
                $"- {nameof(InternalSampleActionFilter)}.
{nameof(OnActionExecuting)}");
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            _logger.LogInformation(
                $"- {nameof(InternalSampleActionFilter)}.
{nameof(OnActionExecuted)}");
        }
    }
}
```

The following code shows three approaches to applying the filter:

C#

```
[SampleActionTypeFilter]
public IActionResult WithDirectAttribute() =>
    Content($"- {nameof(FilterFactoryController)}.
{nameof(WithDirectAttribute)}");

[TypeFilter<SampleActionTypeFilterAttribute>]
public IActionResult WithTypeFilterAttribute() =>
    Content($"- {nameof(FilterFactoryController)}.
{nameof(WithTypeFilterAttribute)}");

[ServiceFilter<SampleActionTypeFilterAttribute>]
public IActionResult WithServiceFilterAttribute() =>
    Content($"- {nameof(FilterFactoryController)}.
{nameof(WithServiceFilterAttribute)}");
```

In the preceding code, the first approach to applying the filter is preferred.

Use middleware in the filter pipeline

Resource filters work like middleware in that they surround the execution of everything that comes later in the pipeline. But filters differ from middleware in that they're part of the runtime, which means that they have access to context and constructs.

To use middleware as a filter, create a type with a `Configure` method that specifies the middleware to inject into the filter pipeline. The following example uses middleware to set a response header:

C#

```
public class FilterMiddlewarePipeline
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            context.Response.Headers.Add("Pipeline", "Middleware");

            await next();
        });
    }
}
```

Use the MiddlewareFilterAttribute to run the middleware:

C#

```
[MiddlewareFilter<FilterMiddlewarePipeline>]
public class FilterMiddlewareController : Controller
{
    public IActionResult Index() =>
        Content($"- {nameof(FilterMiddlewareController)}.{nameof(Index)}");
}
```

Middleware filters run at the same stage of the filter pipeline as Resource filters, before model binding and after the rest of the pipeline.

Thread safety

When passing an *instance* of a filter into `Add`, instead of its `Type`, the filter is a singleton and is **not** thread-safe.

Additional resources

- View or download sample (how to download).
- Filter methods for Razor Pages in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

ASP.NET Core Razor SDK

Article • 06/21/2023

By Rick Anderson

Overview

The .NET 6.0 SDK includes the `Microsoft.NET.Sdk.Razor` MSBuild SDK (Razor SDK). The Razor SDK:

- Is required to build, package, and publish projects containing Razor files for ASP.NET Core MVC-based or Blazor projects.
- Includes a set of predefined properties, and items that allow customizing the compilation of Razor (`.cshtml` or `.razor`) files.

The Razor SDK includes `Content` items with `Include` attributes set to the `**\*.cshtml` and `**\*.razor` globbing patterns. Matching files are published.

Prerequisites

.NET 6.0 SDK

Use the Razor SDK

Most web apps aren't required to explicitly reference the Razor SDK.

To use the Razor SDK to build class libraries containing Razor views or Razor Pages, we recommend starting with the Razor class library (RCL) project template. An RCL that's used to build Blazor (`.razor`) files minimally requires a reference to the Microsoft.AspNetCore.Components package. An RCL that's used to build Razor views or pages (`.cshtml` files) minimally requires targeting `netcoreapp3.0` or later and has a `FrameworkReference` to the Microsoft.AspNetCore.App metapackage in its project file.

Properties

The following properties control the Razor's SDK behavior as part of a project build:

- `RazorCompileOnBuild`: When `true`, compiles and emits the Razor assembly as part of building the project. Defaults to `true`.

- `RazorCompileOnPublish`: When `true`, compiles and emits the Razor assembly as part of publishing the project. Defaults to `true`.
- `UseRazorSourceGenerator`: Defaults to `true`. When `true`:
  - Compiles using source generation.
  - Doesn't create `<app_name>.Views.dll`. Views are included in `<app_name>.dll`.
  - Supports .NET Hot Reload.

The properties and items in the following table are used to configure inputs and output to the Razor SDK.

Expand table

Items	Description
`RazorGenerate`	Item elements (`.cshtml` files) that are inputs to code generation.
`RazorComponent`	Item elements (`.razor` files) that are inputs to Razor component code generation.
`RazorCompile`	Item elements (`.cs` files) that are inputs to Razor compilation targets. Use this `ItemGroup` to specify additional files to be compiled into the Razor assembly.
`RazorEmbeddedResource`	Item elements added as embedded resources to the generated Razor assembly.

Expand table

Property	Description
`RazorOutputPath`	The Razor output directory.
`RazorCompileToolset`	Used to determine the toolset used to build the Razor assembly. Valid values are `Implicit`, `RazorSDK`, and `PrecompilationTool`.
EnableDefaultContentItems	Default is `true`. When `true`, includes *web.config*, `.json`, and `.cshtml` files as content in the project. When referenced via `Microsoft.NET.Sdk.Web`, files under *wwwroot* and config files are also included.
`EnableDefaultRazorGenerateItems`	When `true`, includes `.cshtml` files from `Content` items in `RazorGenerate` items.
`GenerateRazorTargetAssemblyInfo`	Not used in .NET 6 and later.

Property	Description
`EnableDefaultRazorTargetAssemblyInfoAttributes`	Not used in .NET 6 and later.
`CopyRazorGenerateFilesToPublishDirectory`	When `true`, copies `RazorGenerate` items (`.cshtml`) files to the publish directory. Typically, Razor files aren't required for a published app if they participate in compilation at build-time or publish-time. Defaults to `false`.
`PreserveCompilationReferences`	When `true`, copy reference assembly items to the publish directory. Typically, reference assemblies aren't required for a published app if Razor compilation occurs at build-time or publish-time. Set to `true` if your published app requires runtime compilation. For example, set the value to `true` if the app modifies `.cshtml` files at runtime or uses embedded views. Defaults to `false`.
`IncludeRazorContentInPack`	When `true`, all Razor content items (`.cshtml` files) are marked for inclusion in the generated NuGet package. Defaults to `false`.
`EmbedRazorGenerateSources`	When `true`, adds RazorGenerate (`.cshtml`) items as embedded files to the generated Razor assembly. Defaults to `false`.
`GenerateMvcApplicationPartsAssemblyAttributes`	Not used in .NET 6 and later.
`DefaultWebContentItemExcludes`	A globbing pattern for item elements that are to be excluded from the `Content` item group in projects targeting the Web or Razor SDK
`ExcludeConfigFilesFromBuildOutput`	When `true`, *.config* and `.json` files do not get copied to the build output directory.
`AddRazorSupportForMvc`	When `true`, configures the Razor SDK to add support for the MVC configuration that is required when building applications containing MVC views or Razor Pages. This property is implicitly set for .NET Core 3.0 or later projects targeting the Web SDK
`RazorLangVersion`	The version of the Razor Language to target.
`EmitCompilerGeneratedFiles`	When set to `true`, the generated source files are written to disk. Setting to `true` is useful

Property	Description
	when debugging the compiler. The default is `false`.

For more information on properties, see MSBuild properties.

Runtime compilation of Razor views

- By default, the Razor SDK doesn't publish reference assemblies that are required to perform runtime compilation. This results in compilation failures when the application model relies on runtime compilation—for example, the app uses embedded views or changes views after the app is published. Set `CopyRefAssembliesToPublishDirectory` to `true` to continue publishing reference assemblies. Both code generation and compilation are supported by a single call to the compiler. A single assembly is produced that contains the app types and the generated views.
- For a web app, ensure your app is targeting the `Microsoft.NET.Sdk.Web` SDK.

Razor language version

When targeting the `Microsoft.NET.Sdk.Web` SDK, the Razor language version is inferred from the app's target framework version. For projects targeting the `Microsoft.NET.Sdk.Razor` SDK or in the rare case that the app requires a different Razor language version than the inferred value, a version can be configured by setting the `<RazorLangVersion>` property in the app's project file:

XML

```
<PropertyGroup>
  <RazorLangVersion>{VERSION}</RazorLangVersion>
</PropertyGroup>
```

Razor's language version is tightly integrated with the version of the runtime that it was built for. Targeting a language version that isn't designed for the runtime is unsupported and likely produces build errors.

Additional resources

- Additions to the csproj format for .NET Core

- Common MSBuild project items

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

View components in ASP.NET Core

Article • 09/25/2023

By Rick Anderson

View components

View components are similar to partial views, but they're much more powerful. View components don't use model binding, they depend on the data passed when calling the view component. This article was written using controllers and views, but view components work with Razor Pages.

A view component:

- Renders a chunk rather than a whole response.
- Includes the same separation-of-concerns and testability benefits found between a controller and view.
- Can have parameters and business logic.
- Is typically invoked from a layout page.

View components are intended anywhere reusable rendering logic that's too complex for a partial view, such as:

- Dynamic navigation menus
- Tag cloud, where it queries the database
- Sign in panel
- Shopping cart
- Recently published articles
- Sidebar content on a blog
- A sign in panel that would be rendered on every page and show either the links to sign out or sign in, depending on the sign in state of the user

A view component consists of two parts:

- The class, typically derived from ViewComponent
- The result it returns, typically a view.

Like controllers, a view component can be a POCO, but most developers take advantage of the methods and properties available by deriving from ViewComponent.

When considering if view components meet an app's specifications, consider using Razor components instead. Razor components also combine markup with C# code to

produce reusable UI units. Razor components are designed for developer productivity when providing client-side UI logic and composition. For more information, see ASP.NET Core Razor components. For information on how to incorporate Razor components into an MVC or Razor Pages app, see Integrate ASP.NET Core Razor components.

Create a view component

This section contains the high-level requirements to create a view component. Later in the article, we'll examine each step in detail and create a view component.

The view component class

A view component class can be created by any of the following:

- Deriving from ViewComponent
- Decorating a class with the [ViewComponent] attribute, or deriving from a class with the `[ViewComponent]` attribute
- Creating a class where the name ends with the suffix ViewComponent

Like controllers, view components must be public, non-nested, and non-abstract classes. The view component name is the class name with the `ViewComponent` suffix removed. It can also be explicitly specified using the Name property.

A view component class:

- Supports constructor dependency injection
- Doesn't take part in the controller lifecycle, therefore filters can't be used in a view component

To prevent a class that has a case-insensitive `ViewComponent` suffix from being treated as a view component, decorate the class with the [NonViewComponent] attribute:

C#

```
using Microsoft.AspNetCore.Mvc;

[NonViewComponent]
public class ReviewComponent
{
    public string Status(string name) => JobStatus.GetCurrentStatus(name);
}
```

View component methods

A view component defines its logic in an:

- `InvokeAsync` method that returns `Task<IViewComponentResult>`.
- `Invoke` synchronous method that returns an IViewComponentResult.

Parameters come directly from invocation of the view component, not from model binding. A view component never directly handles a request. Typically, a view component initializes a model and passes it to a view by calling the `View` method. In summary, view component methods:

- Define an `InvokeAsync` method that returns a `Task<IViewComponentResult>` or a synchronous `Invoke` method that returns an `IViewComponentResult`.
- Typically initializes a model and passes it to a view by calling the ViewComponent.View method.
- Parameters come from the calling method, not HTTP. There's no model binding.
- Aren't reachable directly as an HTTP endpoint. They're typically invoked in a view. A view component never handles a request.
- Are overloaded on the signature rather than any details from the current HTTP request.

View search path

The runtime searches for the view in the following paths:

- /Views/{Controller Name}/Components/{View Component Name}/{View Name}
- /Views/Shared/Components/{View Component Name}/{View Name}
- /Pages/Shared/Components/{View Component Name}/{View Name}
- /Areas/{Area Name}/Views/Shared/Components/{View Component Name}/{View Name}

The search path applies to projects using controllers + views and Razor Pages.

The default view name for a view component is `Default`, which means view files will typically be named `Default.cshtml`. A different view name can be specified when creating the view component result or when calling the `View` method.

We recommend naming the view file `Default.cshtml` and using the *Views/Shared/Components/{View Component Name}/{View Name}* path. The `PriorityList` view component used in this sample uses `Views/Shared/Components/PriorityList/Default.cshtml` for the view component view.

Customize the view search path

To customize the view search path, modify Razor's ViewLocationFormats collection. For example, to search for views within the path `/Components/{View Component Name}/{View Name}`, add a new item to the collection:

C#

```
using Microsoft.EntityFrameworkCore;
using ViewComponentSample.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews()
    .AddRazorOptions(options =>
    {
        options.ViewLocationFormats.Add("/{0}.cshtml");
    });

builder.Services.AddDbContext<ToDoContext>(options =>
        options.UseInMemoryDatabase("db"));

var app = builder.Build();

// Remaining code removed for brevity.
```

In the preceding code, the placeholder `{0}` represents the path `Components/{View Component Name}/{View Name}`.

Invoke a view component

To use the view component, call the following inside a view:

CSHTML

```
@await Component.InvokeAsync("Name of view component",
                              {Anonymous Type Containing Parameters})
```

The parameters are passed to the `InvokeAsync` method. The `PriorityList` view component developed in the article is invoked from the `Views/ToDo/Index.cshtml` view file. In the following code, the `InvokeAsync` method is called with two parameters:

CSHTML

```
</table>

<div>
    Maximum Priority: @ViewData["maxPriority"] <br />
    Is Complete:  @ViewData["isDone"]
```

```
    @await Component.InvokeAsync("PriorityList",
                      new {
                          maxPriority =  ViewData["maxPriority"],
                          isDone = ViewData["isDone"]  }
                      )
</div>
```

Invoke a view component as a Tag Helper

A View Component can be invoked as a Tag Helper:

CSHTML

```
<div>
       Maxium Priority: @ViewData["maxPriority"] <br />
       Is Complete:  @ViewData["isDone"]
    @{
        int maxPriority = Convert.ToInt32(ViewData["maxPriority"]);
        bool isDone = Convert.ToBoolean(ViewData["isDone"]);
    }
    <vc:priority-list max-priority=maxPriority is-done=isDone>
    </vc:priority-list>
</div>
```

Pascal-cased class and method parameters for Tag Helpers are translated into their kebab case. The Tag Helper to invoke a view component uses the `<vc></vc>` element. The view component is specified as follows:

CSHTML

```
<vc:[view-component-name]
  parameter1="parameter1 value"
  parameter2="parameter2 value">
</vc:[view-component-name]>
```

To use a view component as a Tag Helper, register the assembly containing the view component using the `@addTagHelper` directive. If the view component is in an assembly called `MyWebApp`, add the following directive to the `_ViewImports.cshtml` file:

CSHTML

```
@addTagHelper *, MyWebApp
```

A view component can be registered as a Tag Helper to any file that references the view component. See Managing Tag Helper Scope for more information on how to register

Tag Helpers.

The `InvokeAsync` method used in this tutorial:

CSHTML

```
@await Component.InvokeAsync("PriorityList",
                  new {
                      maxPriority =  ViewData["maxPriority"],
                      isDone = ViewData["isDone"]  }
                  )
```

In the preceding markup, the `PriorityList` view component becomes `priority-list`. The parameters to the view component are passed as attributes in kebab case.

Invoke a view component directly from a controller

View components are typically invoked from a view, but they can be invoked directly from a controller method. While view components don't define endpoints like controllers, a controller action that returns the content of a `ViewComponentResult` can be implemented.

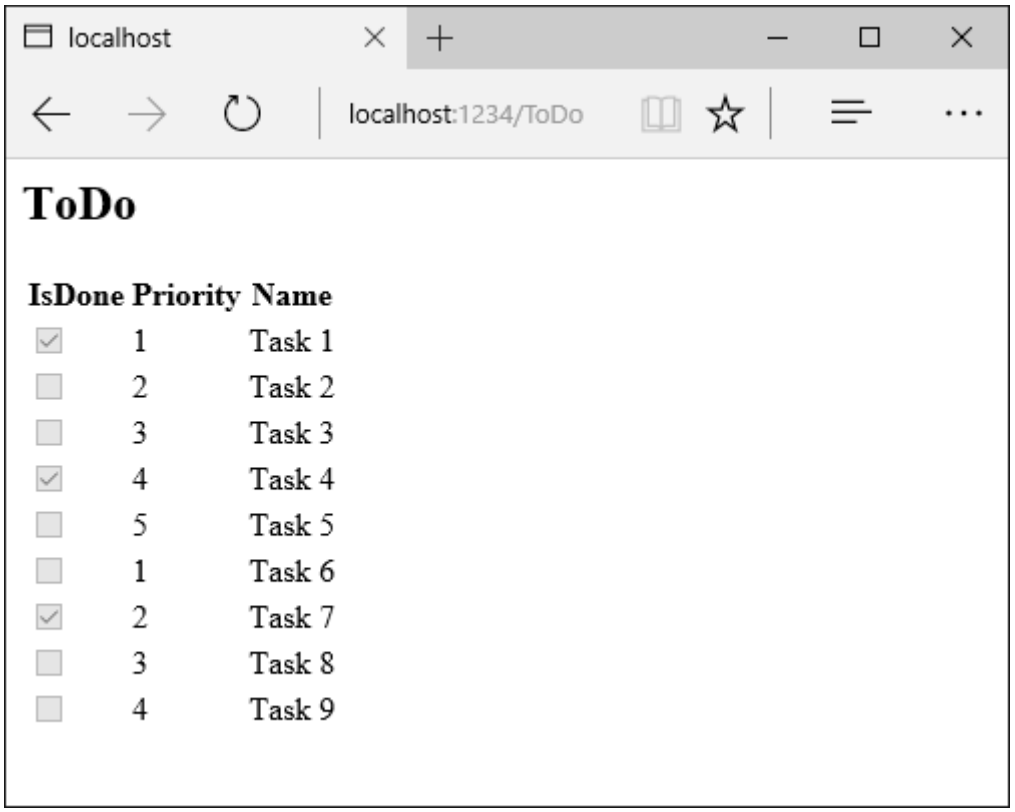
In the following example, the view component is called directly from the controller:

C#

```
public IActionResult IndexVC(int maxPriority = 2, bool isDone = false)
{
    return ViewComponent("PriorityList",
        new {
           maxPriority = maxPriority,
           isDone = isDone
        });
}
```

Create a basic view component

Download, build and test the starter code. It's a basic project with a `ToDo` controller that displays a list of *ToDo* items.



Update the controller to pass in priority and completion status

Update the `Index` method to use priority and completion status parameters:

C#

```
using Microsoft.AspNetCore.Mvc;
using ViewComponentSample.Models;

namespace ViewComponentSample.Controllers;
public class ToDoController : Controller
{
    private readonly ToDoContext _ToDoContext;

    public ToDoController(ToDoContext context)
    {
        _ToDoContext = context;
        _ToDoContext.Database.EnsureCreated();
    }

    public IActionResult Index(int maxPriority = 2, bool isDone = false)
    {
        var model = _ToDoContext!.ToDo!.ToList();
        ViewData["maxPriority"] = maxPriority;
        ViewData["isDone"] = isDone;
        return View(model);
    }
```

Add a ViewComponent class

Add a ViewComponent class to `ViewComponents/PriorityListViewComponent.cs`:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents;

public class PriorityListViewComponent : ViewComponent
{
    private readonly ToDoContext db;

    public PriorityListViewComponent(ToDoContext context) => db = context;

    public async Task<IViewComponentResult> InvokeAsync(
                                            int maxPriority, bool isDone)
    {
        var items = await GetItemsAsync(maxPriority, isDone);
        return View(items);
    }

    private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
    {
        return db!.ToDo!.Where(x => x.IsDone == isDone &&
                               x.Priority <= maxPriority).ToListAsync();
    }
}
```

Notes on the code:

- View component classes can be contained in **any** folder in the project.
- Because the class name PriorityList**ViewComponent** ends with the suffix **ViewComponent**, the runtime uses the string `PriorityList` when referencing the class component from a view.
- The [ViewComponent] attribute can change the name used to reference a view component. For example, the class could have been named `XYZ` with the following `[ViewComponent]` attribute:

  C#

  ```
  [ViewComponent(Name = "PriorityList")]
     public class XYZ : ViewComponent
  ```

- The `[ViewComponent]` attribute in the preceding code tells the view component selector to use:
  - The name `PriorityList` when looking for the views associated with the component
  - The string "PriorityList" when referencing the class component from a view.
- The component uses dependency injection to make the data context available.
- `InvokeAsync` exposes a method that can be called from a view, and it can take an arbitrary number of arguments.
- The `InvokeAsync` method returns the set of `ToDo` items that satisfy the `isDone` and `maxPriority` parameters.

Create the view component Razor view

- Create the *Views/Shared/Components* folder. This folder **must** be named *Components*.
- Create the *Views/Shared/Components/PriorityList* folder. This folder name must match the name of the view component class, or the name of the class minus the suffix. If the `ViewComponent` attribute is used, the class name would need to match the attribute designation.
- Create a `Views/Shared/Components/PriorityList/Default.cshtml` Razor view:

  CSHTML

  ```
  @model IEnumerable<ViewComponentSample.Models.TodoItem>

  <h3>Priority Items</h3>
  <ul>
      @foreach (var todo in Model)
      {
          <li>@todo.Name</li>
      }
  </ul>
  ```

  The Razor view takes a list of `TodoItem` and displays them. If the view component `InvokeAsync` method doesn't pass the name of the view, *Default* is used for the view name by convention. To override the default styling for a specific controller, add a view to the controller-specific view folder (for example *Views/ToDo/Components/PriorityList/Default.cshtml*).

If the view component is controller-specific, it can be added to the controller-specific folder. For example, `Views/ToDo/Components/PriorityList/Default.cshtml` is controller-specific.

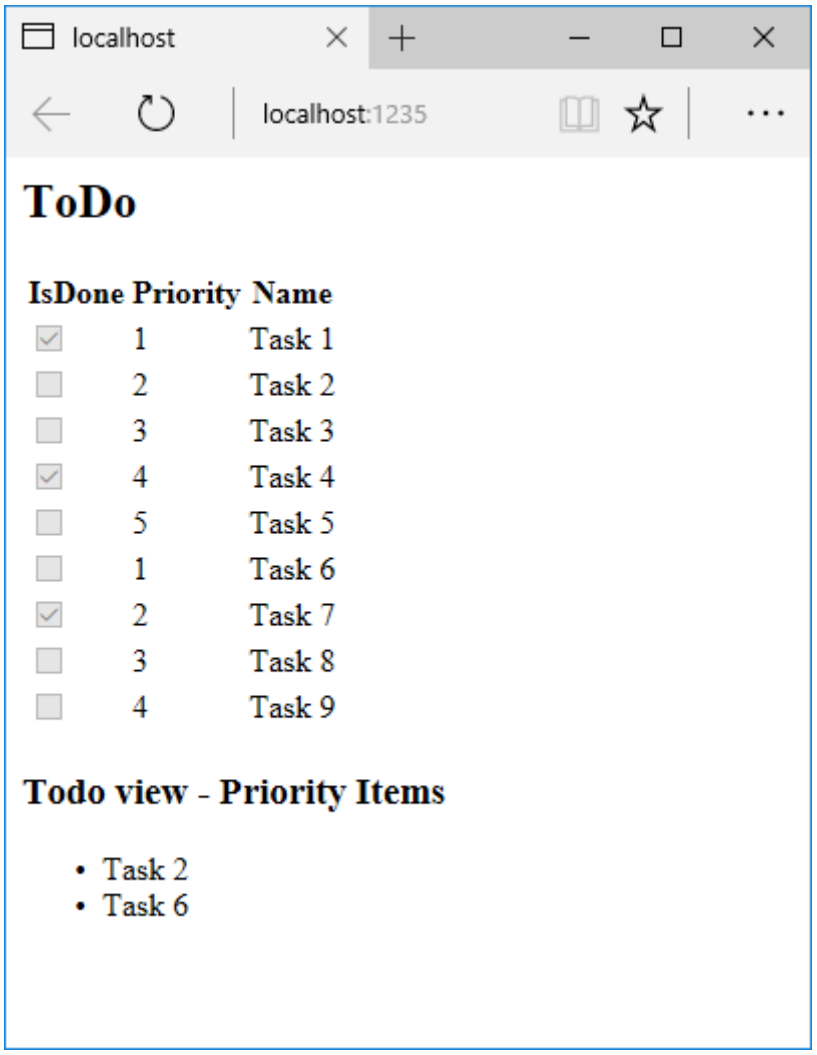
- Add a `div` containing a call to the priority list component to the bottom of the `Views/ToDo/index.cshtml` file:

CSHTML

```
</table>

<div>
    Maximum Priority: @ViewData["maxPriority"] <br />
    Is Complete:  @ViewData["isDone"]
    @await Component.InvokeAsync("PriorityList",
                    new {
                        maxPriority =  ViewData["maxPriority"],
                        isDone = ViewData["isDone"]  }
                    )
</div>
```

The markup `@await Component.InvokeAsync` shows the syntax for calling view components. The first argument is the name of the component we want to invoke or call. Subsequent parameters are passed to the component. `InvokeAsync` can take an arbitrary number of arguments.

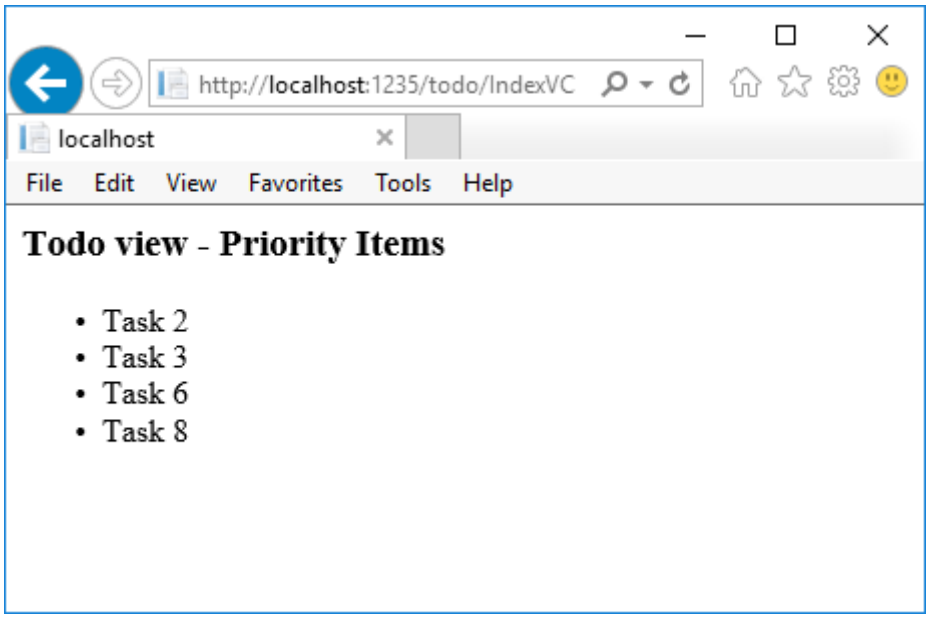
Test the app. The following image shows the ToDo list and the priority items:



The view component can be called directly from the controller:

C#

```
public IActionResult IndexVC(int maxPriority = 2, bool isDone = false)
{
    return ViewComponent("PriorityList",
        new {
           maxPriority = maxPriority,
           isDone = isDone
        });
}
```



Specify a view component name

A complex view component might need to specify a non-default view under some conditions. The following code shows how to specify the "PVC" view from the `InvokeAsync` method. Update the `InvokeAsync` method in the `PriorityListViewComponent` class.

C#

```
public async Task<IViewComponentResult> InvokeAsync(
                                            int maxPriority, bool isDone)
{
    string MyView = "Default";
    // If asking for all completed tasks, render with the "PVC" view.
    if (maxPriority > 3 && isDone == true)
    {
        MyView = "PVC";
    }
    var items = await GetItemsAsync(maxPriority, isDone);
    return View(MyView, items);
}
```

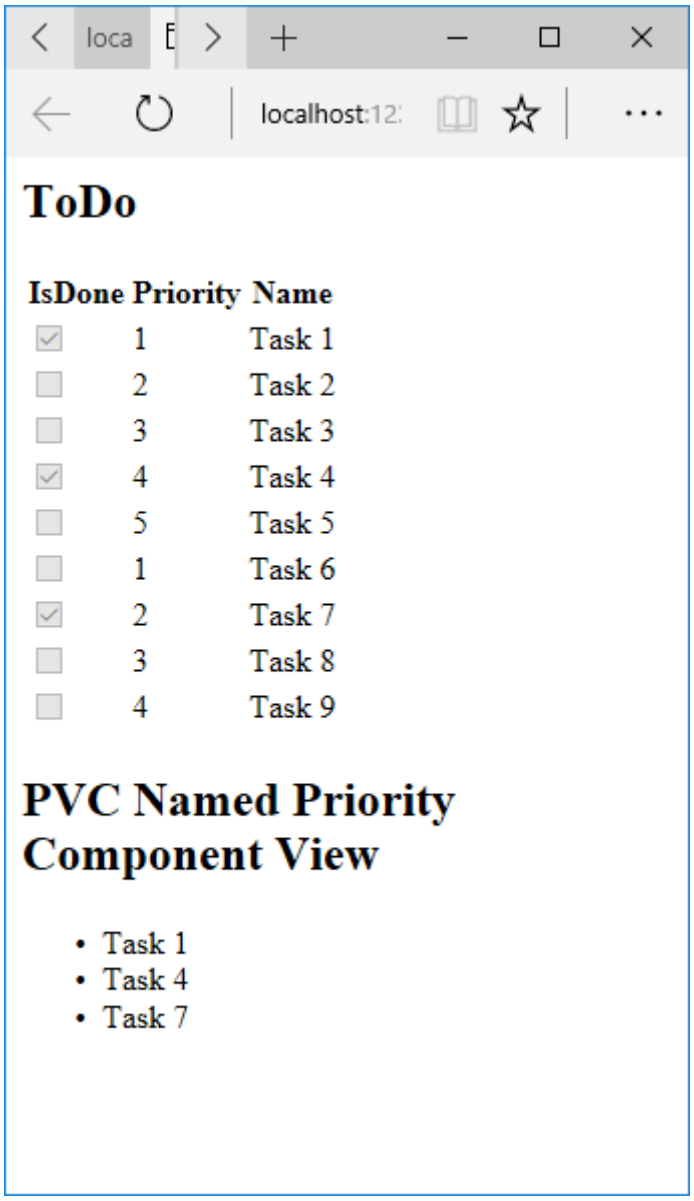
Copy the `Views/Shared/Components/PriorityList/Default.cshtml` file to a view named `Views/Shared/Components/PriorityList/PVC.cshtml`. Add a heading to indicate the PVC view is being used.

CSHTML

```
@model IEnumerable<ViewComponentSample.Models.TodoItem>

<h2> PVC Named Priority Component View</h2>
<h4>@ViewBag.PriorityMessage</h4>
```

```
<ul>
    @foreach (var todo in Model)
    {
        <li>@todo.Name</li>
    }
</ul>
```

Run the app and verify PVC view.



If the PVC view isn't rendered, verify the view component with a priority of 4 or higher is called.

Examine the view path

- Change the priority parameter to three or less so the priority view isn't returned.
- Temporarily rename the `Views/ToDo/Components/PriorityList/Default.cshtml` to `1Default.cshtml`.

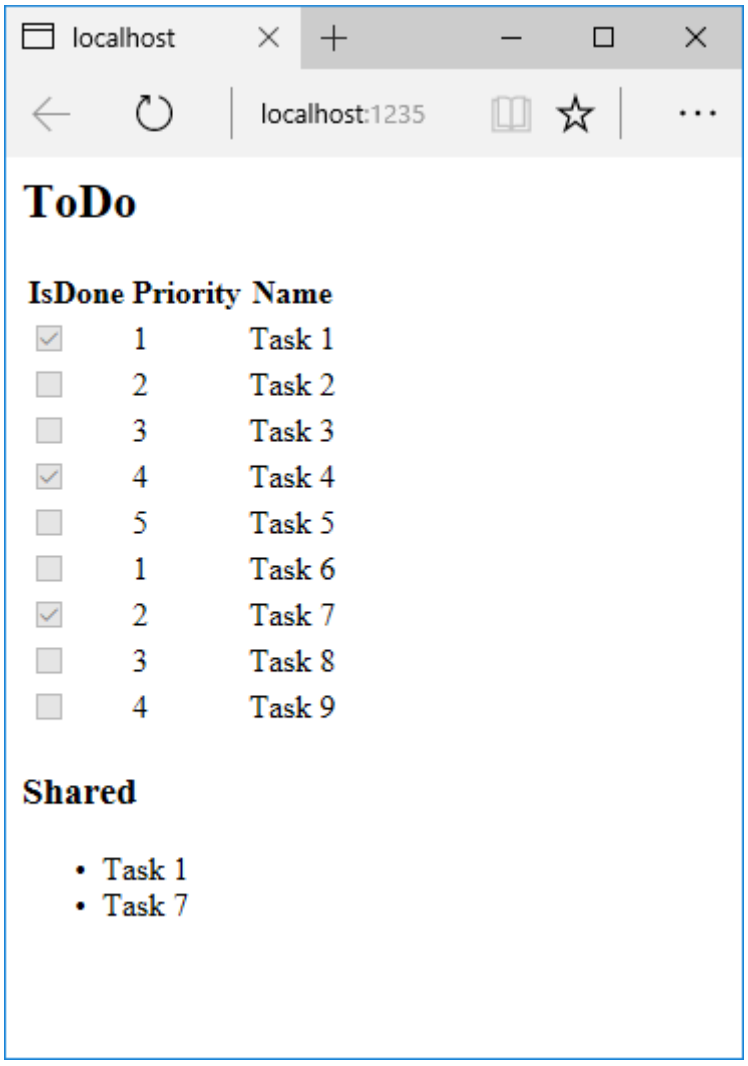
- Test the app, the following error occurs:

```txt
An unhandled exception occurred while processing the request.
InvalidOperationException: The view 'Components/PriorityList/Default'
wasn't found. The following locations were searched:
/Views/ToDo/Components/PriorityList/Default.cshtml
/Views/Shared/Components/PriorityList/Default.cshtml
```

- Copy `Views/ToDo/Components/PriorityList/1Default.cshtml` to `Views/Shared/Components/PriorityList/Default.cshtml`.

- Add some markup to the *Shared* ToDo view component view to indicate the view is from the *Shared* folder.

- Test the **Shared** component view.



Avoid hard-coded strings

For compile time safety, replace the hard-coded view component name with the class name. Update the *PriorityListViewComponent.cs* file to not use the "ViewComponent" suffix:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents;

public class PriorityList : ViewComponent
{
    private readonly ToDoContext db;

    public PriorityList(ToDoContext context)
    {
        db = context;
    }

    public async Task<IViewComponentResult> InvokeAsync(
                                                int maxPriority, bool isDone)
    {
        var items = await GetItemsAsync(maxPriority, isDone);
        return View(items);
    }

    private Task<List<TodoItem>> GetItemsAsync(int maxPriority, bool isDone)
    {
        return db!.ToDo!.Where(x => x.IsDone == isDone &&
                               x.Priority <= maxPriority).ToListAsync();
    }
}
```

The view file:

CSHTML

```
</table>

<div>
    Testing nameof(PriorityList) <br />

    Maxium Priority: @ViewData["maxPriority"] <br />
    Is Complete:  @ViewData["isDone"]
    @await Component.InvokeAsync(nameof(PriorityList),
                        new {
                            maxPriority =  ViewData["maxPriority"],
                            isDone = ViewData["isDone"]  }
```

```
                    )
</div>
```

An overload of `Component.InvokeAsync` method that takes a CLR type uses the `typeof` operator:

CSHTML

```
</table>

<div>
    Testing typeof(PriorityList) <br />

    Maxium Priority: @ViewData["maxPriority"] <br />
    Is Complete:  @ViewData["isDone"]
    @await Component.InvokeAsync(typeof(PriorityList),
                    new {
                        maxPriority =  ViewData["maxPriority"],
                        isDone = ViewData["isDone"]  }
                    )
</div>
```

Perform synchronous work

The framework handles invoking a synchronous `Invoke` method if asynchronous work isn't required. The following method creates a synchronous `Invoke` view component:

C#

```
using Microsoft.AspNetCore.Mvc;
using ViewComponentSample.Models;

namespace ViewComponentSample.ViewComponents
{
    public class PriorityListSync : ViewComponent
    {
        private readonly ToDoContext db;

        public PriorityListSync(ToDoContext context)
        {
            db = context;
        }

        public IViewComponentResult Invoke(int maxPriority, bool isDone)
        {

            var x = db!.ToDo!.Where(x => x.IsDone == isDone &&
                                    x.Priority <= maxPriority).ToList();
            return View(x);
```

```
        }
    }
}
```

The view component's Razor file:

CSHTML

```
<div>
    Testing nameof(PriorityList) <br />

    Maxium Priority: @ViewData["maxPriority"] <br />
    Is Complete:  @ViewData["isDone"]
    @await Component.InvokeAsync(nameof(PriorityListSync),
                    new {
                        maxPriority =  ViewData["maxPriority"],
                        isDone = ViewData["isDone"]  }
                    )
</div>
```

The view component is invoked in a Razor file (for example, `Views/Home/Index.cshtml`) using one of the following approaches:

- IViewComponentHelper
- Tag Helper

To use the IViewComponentHelper approach, call `Component.InvokeAsync`:

CSHTML

```
@await Component.InvokeAsync(nameof(PriorityList),
                              new { maxPriority = 4, isDone = true })
```

To use the Tag Helper, register the assembly containing the View Component using the `@addTagHelper` directive (the view component is in an assembly called `MyWebApp`):

CSHTML

```
@addTagHelper *, MyWebApp
```

Use the view component Tag Helper in the Razor markup file:

CSHTML

```
<vc:priority-list max-priority="999" is-done="false">
</vc:priority-list>
```

The method signature of `PriorityList.Invoke` is synchronous, but Razor finds and calls the method with `Component.InvokeAsync` in the markup file.

Additional resources

- View or download sample code (how to download)
- Dependency injection into views
- View Components in Razor Pages
- Why You Should Use View Components, not Partial Views, in ASP.NET Core

Razor file compilation in ASP.NET Core

Article • 08/30/2024

Razor files with a `.cshtml` extension are compiled at both build and publish time using the Razor SDK. Runtime compilation may be optionally enabled by configuring the project.

> **Note**
>
> Runtime compilation:
>
> - Isn't supported for Razor components of Blazor apps.
> - Doesn't support global using directives.
> - Doesn't support implicit using directives.
> - Disables .NET Hot Reload.
> - Is recommended for development, not for production.

Razor compilation

Build-time and publish-time compilation of Razor files is enabled by default by the Razor SDK. When enabled, runtime compilation complements build-time compilation, allowing Razor files to be updated if they're edited while the app is running.

Updating Razor views and Razor Pages during development while the app is running is also supported using .NET Hot Reload.

> **Note**
>
> When enabled, runtime compilation disables .NET Hot Reload. We recommend using Hot Reload instead of Razor runtime compilation during development.

Enable runtime compilation for all environments

To enable runtime compilation for all environments:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.

2. Call AddRazorRuntimeCompilation in `Program.cs`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages()
    .AddRazorRuntimeCompilation();
```

Enable runtime compilation conditionally

Runtime compilation can be enabled conditionally, which ensures that the published output:

- Uses compiled views.
- Doesn't enable file watchers in production.

To enable runtime compilation only for the Development environment:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.
2. Call AddRazorRuntimeCompilation in `Program.cs` when the current environment is set to Development:

```
C#

var builder = WebApplication.CreateBuilder(args);

var mvcBuilder = builder.Services.AddRazorPages();

if (builder.Environment.IsDevelopment())
{
    mvcBuilder.AddRazorRuntimeCompilation();
}
```

Runtime compilation can also be enabled with a hosting startup assembly. To enable runtime compilation in the Development environment for specific launch profiles:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.
2. Modify the launch profile's `environmentVariables` section in `launchSettings.json`:
    - Verify that `ASPNETCORE_ENVIRONMENT` is set to `"Development"`.

- Set `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` to `"Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"`. For example, the following `launchSettings.json` enables runtime compilation for the `ViewCompilationSample` and `IIS Express` launch profiles:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:7098",
      "sslPort": 44332
    }
  },
  "profiles": {
    "ViewCompilationSample": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl":
"https://localhost:7173;http://localhost:5251",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES":
"Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES":
"Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation"
      }
    }
  }
}
```

With this approach, no code changes are needed in `Program.cs`. At runtime, ASP.NET Core searches for an assembly-level HostingStartup attribute in `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation`. The `HostingStartup` attribute specifies the app startup code to execute and that startup code enables runtime compilation.

Enable runtime compilation for a Razor class library

Consider a scenario in which a Razor Pages project references a Razor class library (RCL) named *MyClassLib*. The RCL contains a `_Layout.cshtml` file consumed by MVC and Razor Pages projects. To enable runtime compilation for the `_Layout.cshtml` file in that RCL, make the following changes in the Razor Pages project:

1. Enable runtime compilation with the instructions at Enable runtime compilation conditionally.

2. Configure MvcRazorRuntimeCompilationOptions in `Program.cs`:

   C#

   ```
   var builder = WebApplication.CreateBuilder(args);

   builder.Services.AddRazorPages();

   builder.Services.Configure<MvcRazorRuntimeCompilationOptions>(options
   =>
   {
       var libraryPath = Path.GetFullPath(
           Path.Combine(builder.Environment.ContentRootPath, "..",
   "MyClassLib"));

       options.FileProviders.Add(new PhysicalFileProvider(libraryPath));
   });
   ```

   The preceding code builds an absolute path to the *MyClassLib* RCL. The PhysicalFileProvider API is used to locate directories and files at that absolute path. Finally, the `PhysicalFileProvider` instance is added to a file providers collection, which allows access to the RCL's `.cshtml` files.

Additional resources

- RazorCompileOnBuild and RazorCompileOnPublish properties
- Introduction to Razor Pages in ASP.NET Core
- Views in ASP.NET Core MVC
- ASP.NET Core Razor SDK

Display and Editor templates in ASP.NET Core

Article • 06/15/2023

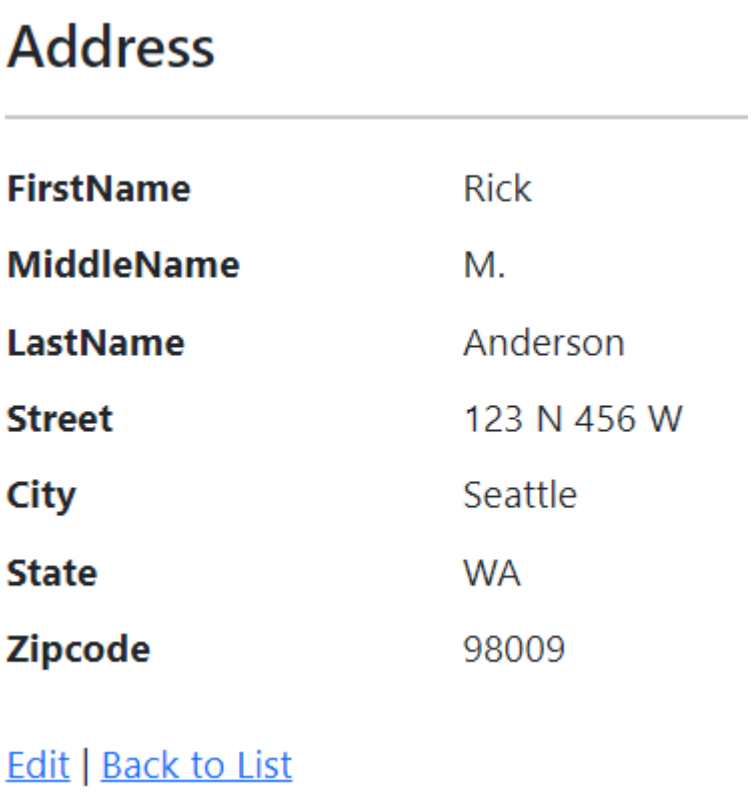
By Alexander Wicht

Display and Editor templates specify the user interface layout of custom types. Consider the following `Address` model:

C#

```
public class Address
{
    public int Id { get; set; }
    public string FirstName { get; set; } = null!;
    public string MiddleName { get; set; } = null!;
    public string LastName { get; set; } = null!;
    public string Street { get; set; } = null!;
    public string City { get; set; } = null!;
    public string State { get; set; } = null!;
    public string Zipcode { get; set; } = null!;
}
```

A project that scaffolds the `Address` model displays the `Address` in the following form:



A web site could use a Display Template to show the `Address` in standard format:



Display and Editor templates can also reduce code duplication and maintenance costs. Consider a web site that displays the `Address` model on 20 different pages. If the `Address` model changes, the 20 pages will all need to be updated. If a Display Template is used for the `Address` model, only the Display Template needs to be updated. For example, the `Address` model might be updated to include the country or region.

Tag Helpers provide an alternative way that enables server-side code to participate in creating and rendering HTML elements in Razor files. For more information, see Tag Helpers compared to HTML Helpers.

Display templates

`DisplayTemplates` customize the display of model fields or create a layer of abstraction between the model values and their display.

A `DisplayTemplate` is a Razor file placed in the `DisplayTemplates` folder:

- For Razor Pages apps, in the `Pages/Shared/DisplayTemplates` folder.
- For MVC apps, in the `Views/Shared/DisplayTemplates` folder or the `Views/ControllerName/DisplayTemplates` folder. Display templates in the `Views/Shared/DisplayTemplates` are used by all controllers in the app. Display templates in the `Views/ControllerName/DisplayTemplates` folder are resolved only by the `ControllerName` controller.

By convention, the `DisplayTemplate` file is named after the type to be displayed. The `Address.cshtml` template used in this sample:

CSHTML

```
@model Address

<dl>
    <dd>@Model.FirstName @Model.MiddleName @Model.LastName</dd>
    <dd>@Model.Street</dd>
    <dd>@Model.City @Model.State @Model.Zipcode</dd>
</dl>
```

The view engine automatically looks for a file in the `DisplayTemplates` folder that matches the name of the type. If it doesn't find a matching template, it falls back to the built in templates.

The following code shows the Details view of the scaffolded project:

CSHTML

```
@page
@model WebAddress.Pages.Adr.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Address</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.FirstName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.FirstName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.MiddleName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.MiddleName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.Street)
        </dt>
```

```
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.Street)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.City)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.City)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.State)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.State)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Address.Zipcode)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Address.Zipcode)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Address?.Id">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The following code shows the Details view using the Address Display Template:

CSHTML

```
@page
@model WebAddress.Pages.Adr2.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Address DM</h4>
    <hr />
    <dl class="row">
        <dd class="col-sm-10">
           @Html.DisplayFor(model => model.Address)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Address?.Id">Edit</a> |
```

```
    <a asp-page="./Index">Back to List</a>
</div>
```

To reference a template whose name doesn't match the type name, use the `templateName` parameter in the DisplayFor method. For example, the following markup displays the `Address` model with the `AddressShort` template:

CSHTML

```
@page
@model WebAddress.Pages.Adr2.DetailsCCModel

@{
    ViewData["Title"] = "Details Short";
}

<h1>Details Short</h1>

<div>
    <h4>Address Short</h4>
    <hr />
    <dl class="row">
        <dd class="col-sm-10">
           @Html.DisplayFor(model => model.Address,"AddressShort")
        </dd>
    </dl>
</div>
```

Use one of the available DisplayFor overloads that expose the `additionalViewData` parameter to pass additional view data that is merged into the View Data Dictionary instance created for the template.

Editor templates

Editor templates are used in form controls when the model is edited or updated.

An `EditorTemplate` is a Razor file placed in the `EditorTemplates` folder:

- For Razor Pages apps, in the `Pages/Shared/EditorTemplates` folder.
- For MVC apps, in the `Views/Shared/EditorTemplates` folder or the `Views/ControllerName/EditorTemplates` folder.

The following markup shows the `Pages/Shared/EditorTemplates/Address.cshtml` used in the sample:

CSHTML

```
@model Address

<dl>
    <dd>    Name:
        <input asp-for="FirstName" /> <input asp-for="MiddleName" /> <input
asp-for="LastName" />
    </dd>
        <dd>    Street:
        <input asp-for="Street" />
    </dd>

        <dd>    city/state/zip:
        <input asp-for="City" /> <input asp-for="State" /> <input asp-
for="Zipcode" />
    </dd>

</dl>
```

The following markup shows the *Edit.cshtml* page which uses the `Pages/Shared/EditorTemplates/Address.cshtml` template:

CSHTML

```
@page
@model WebAddress.Pages.Adr.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h1>Edit</h1>

<h4>Address</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Address.Id" />
             @Html.EditorFor(model => model.Address)
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>
```

```
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Additional resources

- View or download sample code (how to download)
- Tag Helpers
- Tag Helpers compared to HTML Helpers

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Upload files in ASP.NET Core

Article • 01/09/2023

By Rutger Storm

ASP.NET Core supports uploading one or more files using buffered model binding for smaller files and unbuffered streaming for larger files.

View or download sample code (how to download)

Security considerations

Use caution when providing users with the ability to upload files to a server. Cyberattackers may attempt to:

- Execute denial of service attacks.
- Upload viruses or malware.
- Compromise networks and servers in other ways.

Security steps that reduce the likelihood of a successful attack are:

- Upload files to a dedicated file upload area, preferably to a non-system drive. A dedicated location makes it easier to impose security restrictions on uploaded files. Disable execute permissions on the file upload location.†
- Do **not** persist uploaded files in the same directory tree as the app.†
- Use a safe file name determined by the app. Don't use a file name provided by the user or the untrusted file name of the uploaded file.† HTML encode the untrusted file name when displaying it. For example, logging the file name or displaying in UI (Razor automatically HTML encodes output).
- Allow only approved file extensions for the app's design specification.†
- Verify that client-side checks are performed on the server.† Client-side checks are easy to circumvent.
- Check the size of an uploaded file. Set a maximum size limit to prevent large uploads.†
- When files shouldn't be overwritten by an uploaded file with the same name, check the file name against the database or physical storage before uploading the file.
- **Run a virus/malware scanner on uploaded content before the file is stored.**

†The sample app demonstrates an approach that meets the criteria.

> ⚠ **Warning**

Uploading malicious code to a system is frequently the first step to executing code that can:

- Completely gain control of a system.
- Overload a system with the result that the system crashes.
- Compromise user or system data.
- Apply graffiti to a public UI.

For information on reducing vulnerabilities when accepting files from users, see the following resources:

- Unrestricted File Upload
- Azure Security: Ensure appropriate controls are in place when accepting files from users

For more information on implementing security measures, including examples from the sample app, see the Validation section.

Storage scenarios

Common storage options for files include:

- Database
  - For small file uploads, a database is often faster than physical storage (file system or network share) options.
  - A database is often more convenient than physical storage options because retrieval of a database record for user data can concurrently supply the file content (for example, an avatar image).
  - A database is potentially less expensive than using a cloud data storage service.
- Physical storage (file system or network share)
  - For large file uploads:
    - Database limits may restrict the size of the upload.
    - Physical storage is often less economical than storage in a database.
  - Physical storage is potentially less expensive than using a cloud data storage service.
  - The app's process must have read and write permissions to the storage location. **Never grant execute permission.**
- Cloud data storage service, for example, Azure Blob Storage.

- Services usually offer improved scalability and resiliency over on-premises solutions that are usually subject to single points of failure.
- Services are potentially lower cost in large storage infrastructure scenarios.

For more information, see Quickstart: Use .NET to create a blob in object storage.

Small and large files

The definition of small and large files depend on the computing resources available. Apps should benchmark the storage approach used to ensure it can handle the expected sizes. Benchmark memory, CPU, disk, and database performance.

While specific boundaries can't be provided on what is small versus large for your deployment, here are some of ASP.NET Core's related defaults for FormOptions (API documentation):

- By default, HttpRequest.Form doesn't buffer the entire request body (BufferBody), but it does buffer any multipart form files included.
- MultipartBodyLengthLimit is the maximum size for buffered form files (default: 128 MB).
- MemoryBufferThreshold indicates the buffering threshold in memory before transitioning to a buffer file on disk (default: 64 KB). `MemoryBufferThreshold` acts as a boundary between small and large files, which is raised or lowered depending on the apps resources and scenarios.

For more information on FormOptions, see the FormOptions class in the ASP.NET Core reference source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

File upload scenarios

Two general approaches for uploading files are buffering and streaming.

Buffering

The entire file is read into an IFormFile. `IFormFile` is a C# representation of the file used to process or save the file.

The disk and memory used by file uploads depend on the number and size of concurrent file uploads. If an app attempts to buffer too many uploads, the site crashes when it runs out of memory or disk space. If the size or frequency of file uploads is exhausting app resources, use streaming.

Any single buffered file exceeding 64 KB is moved from memory to a temp file on disk.

Temporary files for larger requests are written to the location named in the `ASPNETCORE_TEMP` environment variable. If `ASPNETCORE_TEMP` is not defined, the files are written to the current user's temporary folder.

Buffering small files is covered in the following sections of this topic:

- Physical storage
- Database

Streaming

The file is received from a multipart request and directly processed or saved by the app. Streaming doesn't improve performance significantly. Streaming reduces the demands for memory or disk space when uploading files.

Streaming large files is covered in the Upload large files with streaming section.

Upload small files with buffered model binding to physical storage

To upload small files, use a multipart form or construct a POST request using JavaScript.

The following example demonstrates the use of a Razor Pages form to upload a single file (`Pages/BufferedSingleFileUploadPhysical.cshtml` in the sample app):

CSHTML

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file" />
            <span asp-validation-for="FileUpload.FormFile"></span>
        </dd>
```

```
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit"
value="Upload" />
</form>
```

The following example is analogous to the prior example except that:

- JavaScript's (Fetch API) is used to submit the form's data.
- There's no validation.

CSHTML

```
<form action="BufferedSingleFileUploadPhysical/?handler=Upload"
      enctype="multipart/form-data" onsubmit="AJAXSubmit(this);return
false;"
      method="post">
    <dl>
        <dt>
            <label for="FileUpload_FormFile">File</label>
        </dt>
        <dd>
            <input id="FileUpload_FormFile" type="file"
                name="FileUpload.FormFile" />
        </dd>
    </dl>

    <input class="btn" type="submit" value="Upload" />

    <div style="margin-top:15px">
        <output name="result"></output>
    </div>
</form>

<script>
  async function AJAXSubmit (oFormElement) {
    var resultElement = oFormElement.elements.namedItem("result");
    const formData = new FormData(oFormElement);

    try {
    const response = await fetch(oFormElement.action, {
      method: 'POST',
      body: formData
    });

    if (response.ok) {
      window.location.href = '/';
    }

    resultElement.value = 'Result: ' + response.status + ' ' +
      response.statusText;
    } catch (error) {
      console.error('Error:', error);
    }
```

```
  }
</script>
```

To perform the form POST in JavaScript for clients that don't support the Fetch API, use one of the following approaches:

- Use a Fetch Polyfill (for example, window.fetch polyfill (github/fetch)).
- Use `XMLHttpRequest`. For example:

JavaScript

```
<script>
  "use strict";

  function AJAXSubmit (oFormElement) {
    var oReq = new XMLHttpRequest();
    oReq.onload = function(e) {
    oFormElement.elements.namedItem("result").value =
      'Result: ' + this.status + ' ' + this.statusText;
    };
    oReq.open("post", oFormElement.action);
    oReq.send(new FormData(oFormElement));
  }
</script>
```

In order to support file uploads, HTML forms must specify an encoding type (`enctype`) of `multipart/form-data`.

For a `files` input element to support uploading multiple files provide the `multiple` attribute on the `<input>` element:

CSHTML

```
<input asp-for="FileUpload.FormFiles" type="file" multiple />
```

The individual files uploaded to the server can be accessed through Model Binding using IFormFile. The sample app demonstrates multiple buffered file uploads for database and physical storage scenarios.

> ⚠ **Warning**
>
> Do **not** use the `FileName` property of **IFormFile** other than for display and logging. When displaying or logging, HTML encode the file name. A cyberattacker can

provide a malicious filename, including full paths or relative paths. Applications should:

- Remove the path from the user-supplied filename.
- Save the HTML-encoded, path-removed filename for UI or logging.
- Generate a new random filename for storage.

The following code removes the path from the file name:

```
C#
string untrustedFileName = Path.GetFileName(pathName);
```

The examples provided thus far don't take into account security considerations. Additional information is provided by the following sections and the sample app:

- Security considerations
- Validation

When uploading files using model binding and IFormFile, the action method can accept:

- A single IFormFile.
- Any of the following collections that represent several files:
  - IFormFileCollection
  - IEnumerable<IFormFile>
  - List<IFormFile>

> **Note**
>
> Binding matches form files by name. For example, the HTML `name` value in `<input type="file" name="formFile">` must match the C# parameter/property bound (`FormFile`). For more information, see the Match name attribute value to parameter name of POST method section.

The following example:

- Loops through one or more uploaded files.
- Uses Path.GetTempFileName to return a full path for a file, including the file name.
- Saves the files to the local file system using a file name generated by the app.
- Returns the total number and size of files uploaded.

```
C#
```

```
public async Task<IActionResult> OnPostUploadAsync(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            var filePath = Path.GetTempFileName();

            using (var stream = System.IO.File.Create(filePath))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    // Process uploaded files
    // Don't rely on or trust the FileName property without validation.

    return Ok(new { count = files.Count, size });
}
```

Use `Path.GetRandomFileName` to generate a file name without a path. In the following example, the path is obtained from configuration:

C#

```
foreach (var formFile in files)
{
    if (formFile.Length > 0)
    {
        var filePath = Path.Combine(_config["StoredFilesPath"],
            Path.GetRandomFileName());

        using (var stream = System.IO.File.Create(filePath))
        {
            await formFile.CopyToAsync(stream);
        }
    }
}
```

The path passed to the FileStream *must* include the file name. If the file name isn't provided, an UnauthorizedAccessException is thrown at runtime.

Files uploaded using the IFormFile technique are buffered in memory or on disk on the server before processing. Inside the action method, the IFormFile contents are accessible as a Stream. In addition to the local file system, files can be saved to a network share or to a file storage service, such as Azure Blob storage.

For another example that loops over multiple files for upload and uses safe file names, see `Pages/BufferedMultipleFileUploadPhysical.cshtml.cs` in the sample app.

> ⚠ **Warning**
>
> Path.GetTempFileName throws an IOException if more than 65,535 files are created without deleting previous temporary files. The limit of 65,535 files is a per-server limit. For more information on this limit on Windows OS, see the remarks in the following topics:
>
> - GetTempFileNameA function
> - GetTempFileName

Upload small files with buffered model binding to a database

To store binary file data in a database using Entity Framework, define a Byte array property on the entity:

C#

```
public class AppFile
{
    public int Id { get; set; }
    public byte[] Content { get; set; }
}
```

Specify a page model property for the class that includes an IFormFile:

C#

```
public class BufferedSingleFileUploadDbModel : PageModel
{
    ...

    [BindProperty]
    public BufferedSingleFileUploadDb FileUpload { get; set; }

    ...
}

public class BufferedSingleFileUploadDb
{
    [Required]
    [Display(Name="File")]
```

```
    public IFormFile FormFile { get; set; }
}
```

> ⓘ Note
>
> IFormFile can be used directly as an action method parameter or as a bound model property. The prior example uses a bound model property.

The `FileUpload` is used in the Razor Pages form:

CSHTML

```
<form enctype="multipart/form-data" method="post">
    <dl>
        <dt>
            <label asp-for="FileUpload.FormFile"></label>
        </dt>
        <dd>
            <input asp-for="FileUpload.FormFile" type="file">
        </dd>
    </dl>
    <input asp-page-handler="Upload" class="btn" type="submit"
value="Upload">
</form>
```

When the form is POSTed to the server, copy the IFormFile to a stream and save it as a byte array in the database. In the following example, `_dbContext` stores the app's database context:

C#

```
public async Task<IActionResult> OnPostUploadAsync()
{
    using (var memoryStream = new MemoryStream())
    {
        await FileUpload.FormFile.CopyToAsync(memoryStream);

        // Upload the file if less than 2 MB
        if (memoryStream.Length < 2097152)
        {
            var file = new AppFile()
            {
                Content = memoryStream.ToArray()
            };

            _dbContext.File.Add(file);

            await _dbContext.SaveChangesAsync();
```

```
        }
        else
        {
            ModelState.AddModelError("File", "The file is too large.");
        }
    }

    return Page();
}
```

The preceding example is similar to a scenario demonstrated in the sample app:

- `Pages/BufferedSingleFileUploadDb.cshtml`
- `Pages/BufferedSingleFileUploadDb.cshtml.cs`

> ⚠ **Warning**
>
> Use caution when storing binary data in relational databases, as it can adversely impact performance.
>
> Don't rely on or trust the `FileName` property of **IFormFile** without validation. The `FileName` property should only be used for display purposes and only after HTML encoding.
>
> The examples provided don't take into account security considerations. Additional information is provided by the following sections and the sample app:
>
> - Security considerations
> - Validation

Upload large files with streaming

The 3.1 example demonstrates how to use JavaScript to stream a file to a controller action. The file's antiforgery token is generated using a custom filter attribute and passed to the client HTTP headers instead of in the request body. Because the action method processes the uploaded data directly, form model binding is disabled by another custom filter. Within the action, the form's contents are read using a `MultipartReader`, which reads each individual `MultipartSection`, processing the file or storing the contents as appropriate. After the multipart sections are read, the action performs its own model binding.

The initial page response loads the form and saves an antiforgery token in a cookie (via the `GenerateAntiforgeryTokenCookieAttribute` attribute). The attribute uses ASP.NET

Core's built-in antiforgery support to set a cookie with a request token:

C#

```
public class GenerateAntiforgeryTokenCookieAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(ResultExecutingContext context)
    {
        var antiforgery =
context.HttpContext.RequestServices.GetService<IAntiforgery>();

        // Send the request token as a JavaScript-readable cookie
        var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);

        context.HttpContext.Response.Cookies.Append(
            "RequestVerificationToken",
            tokens.RequestToken,
            new CookieOptions() { HttpOnly = false });
    }

    public override void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

The `DisableFormValueModelBindingAttribute` is used to disable model binding:

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class DisableFormValueModelBindingAttribute : Attribute,
IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        var factories = context.ValueProviderFactories;
        factories.RemoveType<FormValueProviderFactory>();
        factories.RemoveType<FormFileValueProviderFactory>();
        factories.RemoveType<JQueryFormValueProviderFactory>();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

In the sample app, `GenerateAntiforgeryTokenCookieAttribute` and `DisableFormValueModelBindingAttribute` are applied as filters to the page application models of `/StreamedSingleFileUploadDb` and `/StreamedSingleFileUploadPhysical` in `Startup.ConfigureServices` using Razor Pages conventions:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/StreamedSingleFileUploadDb",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
    options.Conventions

.AddPageApplicationModelConvention("/StreamedSingleFileUploadPhysical",
            model =>
            {
                model.Filters.Add(
                    new GenerateAntiforgeryTokenCookieAttribute());
                model.Filters.Add(
                    new DisableFormValueModelBindingAttribute());
            });
});
```

Since model binding doesn't read the form, parameters that are bound from the form don't bind (query, route, and header continue to work). The action method works directly with the `Request` property. A `MultipartReader` is used to read each section. Key/value data is stored in a `KeyValueAccumulator`. After the multipart sections are read, the contents of the `KeyValueAccumulator` are used to bind the form data to a model type.

The complete `StreamingController.UploadDatabase` method for streaming to a database with EF Core:

C#

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadDatabase()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
        // Log error

        return BadRequest(ModelState);
    }
```

```
    // Accumulate the form data key-value pairs in the request
(formAccumulator).
    var formAccumulator = new KeyValueAccumulator();
    var trustedFileNameForDisplay = string.Empty;
    var untrustedFileNameForStorage = string.Empty;
    var streamedFileContent = Array.Empty<byte>();

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);

    var section = await reader.ReadNextSectionAsync();

    while (section != null)
    {
        var hasContentDispositionHeader =
            ContentDispositionHeaderValue.TryParse(
                section.ContentDisposition, out var contentDisposition);

        if (hasContentDispositionHeader)
        {
            if (MultipartRequestHelper
                .HasFileContentDisposition(contentDisposition))
            {
                untrustedFileNameForStorage =
contentDisposition.FileName.Value;
                // Don't trust the file name sent by the client. To display
                // the file name, HTML-encode the value.
                trustedFileNameForDisplay = WebUtility.HtmlEncode(
                        contentDisposition.FileName.Value);

                streamedFileContent =
                    await FileHelpers.ProcessStreamedFile(section,
contentDisposition,
                        ModelState, _permittedExtensions, _fileSizeLimit);

                if (!ModelState.IsValid)
                {
                    return BadRequest(ModelState);
                }
            }
            else if (MultipartRequestHelper
                .HasFormDataContentDisposition(contentDisposition))
            {
                // Don't limit the key name length because the
                // multipart headers length limit is already in effect.
                var key = HeaderUtilities
                    .RemoveQuotes(contentDisposition.Name).Value;
                var encoding = GetEncoding(section);

                if (encoding == null)
                {
                    ModelState.AddModelError("File",
```

```
                        $"The request couldn't be processed (Error 2).");
                    // Log error

                    return BadRequest(ModelState);
                }

                using (var streamReader = new StreamReader(
                    section.Body,
                    encoding,
                    detectEncodingFromByteOrderMarks: true,
                    bufferSize: 1024,
                    leaveOpen: true))
                {
                    // The value length limit is enforced by
                    // MultipartBodyLengthLimit
                    var value = await streamReader.ReadToEndAsync();

                    if (string.Equals(value, "undefined",
                        StringComparison.OrdinalIgnoreCase))
                    {
                        value = string.Empty;
                    }

                    formAccumulator.Append(key, value);

                    if (formAccumulator.ValueCount >
                        _defaultFormOptions.ValueCountLimit)
                    {
                        // Form key count limit of
                        // _defaultFormOptions.ValueCountLimit
                        // is exceeded.
                        ModelState.AddModelError("File",
                            $"The request couldn't be processed (Error
3).");
                        // Log error

                        return BadRequest(ModelState);
                    }
                }
            }
        }

        // Drain any remaining section body that hasn't been consumed and
        // read the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    // Bind form data to the model
    var formData = new FormData();
    var formValueProvider = new FormValueProvider(
        BindingSource.Form,
        new FormCollection(formAccumulator.GetResults()),
        CultureInfo.CurrentCulture);
    var bindingSuccessful = await TryUpdateModelAsync(formData, prefix: "",
        valueProvider: formValueProvider);
```

```
    if (!bindingSuccessful)
    {
        ModelState.AddModelError("File",
            "The request couldn't be processed (Error 5).");
        // Log error

        return BadRequest(ModelState);
    }

    // **WARNING!**
    // In the following example, the file is saved without
    // scanning the file's contents. In most production
    // scenarios, an anti-virus/anti-malware scanner API
    // is used on the file before making the file available
    // for download or for use by other systems.
    // For more information, see the topic that accompanies
    // this sample app.

    var file = new AppFile()
    {
        Content = streamedFileContent,
        UntrustedName = untrustedFileNameForStorage,
        Note = formData.Note,
        Size = streamedFileContent.Length,
        UploadDT = DateTime.UtcNow
    };

    _context.File.Add(file);
    await _context.SaveChangesAsync();

    return Created(nameof(StreamingController), null);
}
```

`MultipartRequestHelper` (`Utilities/MultipartRequestHelper.cs`):

C#

```
using System;
using System.IO;
using Microsoft.Net.Http.Headers;

namespace SampleApp.Utilities
{
    public static class MultipartRequestHelper
    {
        // Content-Type: multipart/form-data; boundary="----
WebKitFormBoundarymx2fSWqWSd0OxQqq"
        // The spec at https://tools.ietf.org/html/rfc2046#section-5.1
states that 70 characters is a reasonable limit.
        public static string GetBoundary(MediaTypeHeaderValue contentType,
int lengthLimit)
        {
```

```
            var boundary =
HeaderUtilities.RemoveQuotes(contentType.Boundary).Value;

            if (string.IsNullOrWhiteSpace(boundary))
            {
                throw new InvalidDataException("Missing content-type
boundary.");
            }

            if (boundary.Length > lengthLimit)
            {
                throw new InvalidDataException(
                    $"Multipart boundary length limit {lengthLimit}
exceeded.");
            }

            return boundary;
        }

        public static bool IsMultipartContentType(string contentType)
        {
            return !string.IsNullOrEmpty(contentType)
                   && contentType.IndexOf("multipart/",
StringComparison.OrdinalIgnoreCase) >= 0;
        }

        public static bool
HasFormDataContentDisposition(ContentDispositionHeaderValue
contentDisposition)
        {
            // Content-Disposition: form-data; name="key";
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && string.IsNullOrEmpty(contentDisposition.FileName.Value)
                &&
string.IsNullOrEmpty(contentDisposition.FileNameStar.Value);
        }

        public static bool
HasFileContentDisposition(ContentDispositionHeaderValue contentDisposition)
        {
            // Content-Disposition: form-data; name="myfile1";
filename="Misc 002.jpg"
            return contentDisposition != null
                && contentDisposition.DispositionType.Equals("form-data")
                && (!string.IsNullOrEmpty(contentDisposition.FileName.Value)
                    ||
!string.IsNullOrEmpty(contentDisposition.FileNameStar.Value));
        }
    }
}
```

The complete `StreamingController.UploadPhysical` method for streaming to a physical location:

C#

```
[HttpPost]
[DisableFormValueModelBinding]
[ValidateAntiForgeryToken]
public async Task<IActionResult> UploadPhysical()
{
    if (!MultipartRequestHelper.IsMultipartContentType(Request.ContentType))
    {
        ModelState.AddModelError("File",
            $"The request couldn't be processed (Error 1).");
        // Log error

        return BadRequest(ModelState);
    }

    var boundary = MultipartRequestHelper.GetBoundary(
        MediaTypeHeaderValue.Parse(Request.ContentType),
        _defaultFormOptions.MultipartBoundaryLengthLimit);
    var reader = new MultipartReader(boundary, HttpContext.Request.Body);
    var section = await reader.ReadNextSectionAsync();

    while (section != null)
    {
        var hasContentDispositionHeader =
            ContentDispositionHeaderValue.TryParse(
                section.ContentDisposition, out var contentDisposition);

        if (hasContentDispositionHeader)
        {
            // This check assumes that there's a file
            // present without form data. If form data
            // is present, this method immediately fails
            // and returns the model error.
            if (!MultipartRequestHelper
                .HasFileContentDisposition(contentDisposition))
            {
                ModelState.AddModelError("File",
                    $"The request couldn't be processed (Error 2).");
                // Log error

                return BadRequest(ModelState);
            }
            else
            {
                // Don't trust the file name sent by the client. To display
                // the file name, HTML-encode the value.
                var trustedFileNameForDisplay = WebUtility.HtmlEncode(
                        contentDisposition.FileName.Value);
                var trustedFileNameForFileStorage =
Path.GetRandomFileName();
```

```
                // **WARNING!**
                // In the following example, the file is saved without
                // scanning the file's contents. In most production
                // scenarios, an anti-virus/anti-malware scanner API
                // is used on the file before making the file available
                // for download or for use by other systems.
                // For more information, see the topic that accompanies
                // this sample.

                var streamedFileContent = await
FileHelpers.ProcessStreamedFile(
                    section, contentDisposition, ModelState,
                    _permittedExtensions, _fileSizeLimit);

                if (!ModelState.IsValid)
                {
                    return BadRequest(ModelState);
                }

                using (var targetStream = System.IO.File.Create(
                    Path.Combine(_targetFilePath,
trustedFileNameForFileStorage)))
                {
                    await targetStream.WriteAsync(streamedFileContent);

                    _logger.LogInformation(
                        "Uploaded file '{TrustedFileNameForDisplay}' saved
to " +
                        "'{TargetFilePath}' as
{TrustedFileNameForFileStorage}",
                        trustedFileNameForDisplay, _targetFilePath,
                        trustedFileNameForFileStorage);
                }
            }
        }

        // Drain any remaining section body that hasn't been consumed and
        // read the headers for the next section.
        section = await reader.ReadNextSectionAsync();
    }

    return Created(nameof(StreamingController), null);
}
```

In the sample app, validation checks are handled by `FileHelpers.ProcessStreamedFile`.

Validation

The sample app's `FileHelpers` class demonstrates several checks for buffered IFormFile and streamed file uploads. For processing IFormFile buffered file uploads in the sample

app, see the `ProcessFormFile` method in the `Utilities/FileHelpers.cs` file. For processing streamed files, see the `ProcessStreamedFile` method in the same file.

> ⚠ **Warning**
>
> The validation processing methods demonstrated in the sample app don't scan the content of uploaded files. In most production scenarios, a virus/malware scanner API is used on the file before making the file available to users or other systems.
>
> Although the topic sample provides a working example of validation techniques, don't implement the `FileHelpers` class in a production app unless you:
>
> - Fully understand the implementation.
> - Modify the implementation as appropriate for the app's environment and specifications.
>
> **Never indiscriminately implement security code in an app without addressing these requirements.**

Content validation

Use a third party virus/malware scanning API on uploaded content.

Scanning files is demanding on server resources in high volume scenarios. If request processing performance is diminished due to file scanning, consider offloading the scanning work to a background service, possibly a service running on a server different from the app's server. Typically, uploaded files are held in a quarantined area until the background virus scanner checks them. When a file passes, the file is moved to the normal file storage location. These steps are usually performed in conjunction with a database record that indicates the scanning status of a file. By using such an approach, the app and app server remain focused on responding to requests.

File extension validation

The uploaded file's extension should be checked against a list of permitted extensions. For example:

C#

```
private string[] permittedExtensions = { ".txt", ".pdf" };

var ext = Path.GetExtension(uploadedFileName).ToLowerInvariant();
```

```
if (string.IsNullOrEmpty(ext) || !permittedExtensions.Contains(ext))
{
    // The extension is invalid ... discontinue processing the file
}
```

File signature validation

A file's signature is determined by the first few bytes at the start of a file. These bytes can be used to indicate if the extension matches the content of the file. The sample app checks file signatures for a few common file types. In the following example, the file signature for a JPEG image is checked against the file:

C#

```
private static readonly Dictionary<string, List<byte[]>> _fileSignature =
    new Dictionary<string, List<byte[]>>
{
    { ".jpeg", new List<byte[]>
        {
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE0 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE2 },
            new byte[] { 0xFF, 0xD8, 0xFF, 0xE3 },
        }
    },
};

using (var reader = new BinaryReader(uploadedFileData))
{
    var signatures = _fileSignature[ext];
    var headerBytes = reader.ReadBytes(signatures.Max(m => m.Length));

    return signatures.Any(signature =>
        headerBytes.Take(signature.Length).SequenceEqual(signature));
}
```

To obtain additional file signatures, use a file signatures database (Google search result) and official file specifications. Consulting official file specifications may ensure that the selected signatures are valid.

File name security

Never use a client-supplied file name for saving a file to physical storage. Create a safe file name for the file using Path.GetRandomFileName or Path.GetTempFileName to create a full path (including the file name) for temporary storage.

Razor automatically HTML encodes property values for display. The following code is safe to use:

CSHTML

```
@foreach (var file in Model.DatabaseFiles) {
    <tr>
        <td>
            @file.UntrustedName
        </td>
    </tr>
}
```

Outside of Razor, always HtmlEncode file name content from a user's request.

Many implementations must include a check that the file exists; otherwise, the file is overwritten by a file of the same name. Supply additional logic to meet your app's specifications.

Size validation

Limit the size of uploaded files.

In the sample app, the size of the file is limited to 2 MB (indicated in bytes). The limit is supplied via Configuration from the `appsettings.json` file:

JSON

```
{
  "FileSizeLimit": 2097152
}
```

The `FileSizeLimit` is injected into `PageModel` classes:

C#

```
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    private readonly long _fileSizeLimit;

    public BufferedSingleFileUploadPhysicalModel(IConfiguration config)
    {
        _fileSizeLimit = config.GetValue<long>("FileSizeLimit");
    }
```

```
    ...
}
```

When a file size exceeds the limit, the file is rejected:

C#

```
if (formFile.Length > _fileSizeLimit)
{
    // The file is too large ... discontinue processing the file
}
```

Match name attribute value to parameter name of POST method

In non-Razor forms that POST form data or use JavaScript's `FormData` directly, the name specified in the form's element or `FormData` must match the name of the parameter in the controller's action.

In the following example:

- When using an `<input>` element, the `name` attribute is set to the value `battlePlans`:

  HTML

  ```
  <input type="file" name="battlePlans" multiple>
  ```

- When using `FormData` in JavaScript, the name is set to the value `battlePlans`:

  JavaScript

  ```
  var formData = new FormData();

  for (var file in files) {
    formData.append("battlePlans", file, file.name);
  }
  ```

Use a matching name for the parameter of the C# method (`battlePlans`):

- For a Razor Pages page handler method named `Upload`:

  C#

```
public async Task<IActionResult> OnPostUploadAsync(List<IFormFile>
battlePlans)
```

- For an MVC POST controller action method:

```
C#

public async Task<IActionResult> Post(List<IFormFile> battlePlans)
```

Server and app configuration

Multipart body length limit

MultipartBodyLengthLimit sets the limit for the length of each multipart body. Form sections that exceed this limit throw an InvalidDataException when parsed. The default is 134,217,728 (128 MB). Customize the limit using the MultipartBodyLengthLimit setting in `Startup.ConfigureServices`:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.Configure<FormOptions>(options =>
    {
        // Set the limit to 256 MB
        options.MultipartBodyLengthLimit = 268435456;
    });
}
```

RequestFormLimitsAttribute is used to set the MultipartBodyLengthLimit for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

```
C#

services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",
            model.Filters.Add(
                new RequestFormLimitsAttribute()
                {
                    // Set the limit to 256 MB
```

```
                    MultipartBodyLengthLimit = 268435456
                });
});
```

In a Razor Pages app or an MVC app, apply the filter to the page model or action method:

C#

```
// Set the limit to 256 MB
[RequestFormLimits(MultipartBodyLengthLimit = 268435456)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

Kestrel maximum request body size

For apps hosted by Kestrel, the default maximum request body size is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit using the MaxRequestBodySize Kestrel server option:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel((context, options) =>
            {
                // Handle requests up to 50 MB
                options.Limits.MaxRequestBodySize = 52428800;
            })
            .UseStartup<Startup>();
        });
```

RequestSizeLimitAttribute is used to set the MaxRequestBodySize for a single page or action.

In a Razor Pages app, apply the filter with a convention in `Startup.ConfigureServices`:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions
        .AddPageApplicationModelConvention("/FileUploadPage",
```

```
            model =>
            {
                // Handle requests up to 50 MB
                model.Filters.Add(
                    new RequestSizeLimitAttribute(52428800));
            });
});
```

In a Razor pages app or an MVC app, apply the filter to the page handler class or action method:

C#

```
// Handle requests up to 50 MB
[RequestSizeLimit(52428800)]
public class BufferedSingleFileUploadPhysicalModel : PageModel
{
    ...
}
```

The `RequestSizeLimitAttribute` can also be applied using the @attribute Razor directive:

CSHTML

```
@attribute [RequestSizeLimitAttribute(52428800)]
```

Other Kestrel limits

Other Kestrel limits may apply for apps hosted by Kestrel:

- Maximum client connections
- Request and response data rates

IIS

The default request limit (`maxAllowedContentLength`) is 30,000,000 bytes, which is approximately 28.6 MB. Customize the limit in the `web.config` file. In the following example, the limit is set to 50 MB (52,428,800 bytes):

XML

```
<system.webServer>
  <security>
    <requestFiltering>
```

```
      <requestLimits maxAllowedContentLength="52428800" />
    </requestFiltering>
  </security>
</system.webServer>
```

The `maxAllowedContentLength` setting only applies to IIS. For more information, see Request Limits <requestLimits>.

Troubleshoot

Below are some common problems encountered when working with uploading files and their possible solutions.

Not Found error when deployed to an IIS server

The following error indicates that the uploaded file exceeds the server's configured content length:

```
HTTP 404.13 - Not Found
The request filtering module is configured to deny a request that exceeds
the request content length.
```

For more information, see the IIS section.

Connection failure

A connection error and a reset server connection probably indicates that the uploaded file exceeds Kestrel's maximum request body size. For more information, see the Kestrel maximum request body size section. Kestrel client connection limits may also require adjustment.

Null Reference Exception with IFormFile

If the controller is accepting uploaded files using IFormFile but the value is `null`, confirm that the HTML form is specifying an `enctype` value of `multipart/form-data`. If this attribute isn't set on the `<form>` element, the file upload doesn't occur and any bound IFormFile arguments are `null`. Also confirm that the upload naming in form data matches the app's naming.

Stream was too long

The examples in this topic rely upon MemoryStream to hold the uploaded file's content. The size limit of a `MemoryStream` is `int.MaxValue`. If the app's file upload scenario requires holding file content larger than 50 MB, use an alternative approach that doesn't rely upon a single `MemoryStream` for holding an uploaded file's content.

Additional resources

- HTTP connection request draining

- Unrestricted File Upload
- Azure Security: Security Frame: Input Validation | Mitigations
- Azure Cloud Design Patterns: Valet Key pattern

ASP.NET Core Web SDK

Article • 04/11/2023

Overview

`Microsoft.NET.Sdk.Web` is an MSBuild project SDK for building ASP.NET Core apps. It's possible to build an ASP.NET Core app without this SDK, however, the Web SDK is:

- Tailored towards providing a first-class experience.
- The recommended target for most users.

Use the Web.SDK in a project:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">
  <!-- omitted for brevity -->
</Project>
```

Features enabled by using the Web SDK:

- Implicitly references:
  - The ASP.NET Core shared framework.
  - Analyzers designed for building ASP.NET Core apps.
- The Web SDK imports MSBuild targets that enable the use of publish profiles and publishing using WebDeploy.

Properties

Expand table

Property	Description
`DisableImplicitFrameworkReferences`	Disables implicit reference to the `Microsoft.AspNetCore.App` shared framework.
`DisableImplicitAspNetCoreAnalyzers`	Disables implicit reference to ASP.NET Core analyzers.
`DisableImplicitComponentsAnalyzers`	Disables implicit reference to Razor Components analyzers when building Blazor (server) applications.

For more information on tasks, targets, properties, implicit blobs, globs, publishing, methods, and more, see the README file in the WebSdk repository.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

ASP.NET Core code generator tool (`aspnet-codegenerator`)

Article • 08/06/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

The `dotnet aspnet-codegenerator` command runs the ASP.NET Core scaffolding engine. Running the `dotnet aspnet-codegenerator` command is required to scaffold from the command line or when using Visual Studio Code. The command isn't required to use scaffolding with Visual Studio, which includes the scaffolding engine by default.

Install and update the code generator tool

Install the .NET SDK.

`dotnet aspnet-codegenerator` is a global tool that must be installed. The following command installs the latest stable version of the ASP.NET Core code generator tool:

.NET CLI

```
dotnet tool install -g dotnet-aspnet-codegenerator
```

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

If the tool is already installed, the following command updates the tool to the latest stable version available from the installed .NET Core SDKs:

.NET CLI

```
dotnet tool update -g dotnet-aspnet-codegenerator
```

Uninstall the code generator tool

It may be necessary to uninstall the ASP.NET Core code generator tool to resolve problems. For example, if you installed a preview version of the tool, uninstall it before installing the released version.

The following commands uninstall the ASP.NET Core code generator tool and install the latest stable version:

.NET CLI

```
dotnet tool uninstall -g dotnet-aspnet-codegenerator
dotnet tool install -g dotnet-aspnet-codegenerator
```

Synopsis

```
dotnet aspnet-codegenerator [arguments] [-b|--build-base-path] [-c|--
configuration] [-n|--nuget-package-dir] [--no-build] [-p|--project] [-tfm|--
target-framework]
dotnet aspnet-codegenerator [-h|--help]
```

Description

The `dotnet aspnet-codegenerator` global command runs the ASP.NET Core code generator and scaffolding engine.

Arguments

`generator`

The code generator to run. The available generators are shown in the following table.

Expand table

Generator	Operation
`area`	Scaffolds an area.
`blazor`	Scaffolds Blazor create, read, update, delete, and list pages.

Generator	Operation
`blazor-identity`	Generates Blazor Identity files.
`controller`	Scaffolds a controller.
`identity`	Scaffolds Identity.
`minimalapi`	Generates an endpoints file (with CRUD API endpoints) given a model and optional database context.
`razorpage`	Scaffolds Razor Pages.
`view`	Scaffolds a view.

Options

`-b|--build-base-path`

The build base path.

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`.

`-h|--help`

Prints out a short help for the command.

`-n|--nuget-package-dir`

Specifies the NuGet package directory.

`--no-build`

Doesn't build the project before running. Passing `--no-build` also implicitly sets the `--no-restore` flag.

`-p|--project <PATH>`

Specifies the path of the project file to run (folder name or full path). If not specified, the tool defaults to the current directory.

`-tfm|--target-framework`

The target framework to use.

Generator options

The following sections detail the options available for the supported generators:

- Area (area)
- Controller (controller)
- Blazor (blazor)
- Blazor Identity (blazor-identity)
- Identity (identity)
- Minimal API (minimalapi)
- Razor page (razorpage)
- View (view)

Area options

Usage: `dotnet aspnet-codegenerator area {AREA NAME}`

The `{AREA NAME}` placeholder is the name of the area to generate.

The preceding command generates the following folders:

- `Areas`
  - `{AREA NAME}`
    - `Controllers`
    - `Data`
    - `Models`
    - `Views`

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator area -h
```

Blazor options

Razor components can be individually scaffolded for Blazor apps by specifying the name of the template to use. The supported templates are:

- `Empty`
- `Create`
- `Edit`

- `Delete`
- `Details`
- `List`
- `CRUD`: *CRUD* is an acronym for Create, Read, Update, and Delete. The `CRUD` template produces `Create`, `Edit`, `Delete`, `Details`, and `Index` (`List`) components for the app.

The options for the `blazor` generator are shown in the following table.

Expand table

Option	Description
`-dbProvider|--databaseProvider`	Database provider to use. Options include `sqlserver` (default), `sqlite`, `cosmos`, or `postgres`.
`-dc|--dataContext`	Database context class to use.
`-m|--model`	Model class to use.
`-ns|--namespaceName`	Specify the name of the namespace to use for the generated Endpoints file.
`--relativeFolderPath|-outDir`	Relative output folder path. If not specified, files are generated in the project folder.

The following example:

- Uses the `Edit` template to generate an `Edit` component (`Edit.razor`) in the `Components/Pages/MoviePages` folder of the app. If the `MoviePages` folder doesn't exist, the tool creates the folder automatically.
- Uses the SQLite database provider.
- Uses `BlazorWebAppMovies.Data.BlazorWebAppMoviesContext` for the database context.
- Uses the `Movie` model.

.NET CLI

```
dotnet aspnet-codegenerator blazor Edit -dbProvider sqlite -dc
BlazorWebAppMovies.Data.BlazorWebAppMoviesContext -m Movie -outDir
Components/Pages
```

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator blazor -h
```

For an example that uses the `blazor` generator, see Build a Blazor movie database app (Overview).

For more information, see ASP.NET Core Blazor QuickGrid component.

Blazor Identity options

Scaffold Identity Razor components into a Blazor app with the `blazor-identity` generator.

The options for the `blazor-identity` template are shown in the following table.

Expand table

Option	Description
`-dbProvider|--databaseProvider`	Database provider to use. Options include `sqlserver` (default) and `sqlite`.
`-dc|--dataContext`	Database context class to use.
`-f|--force`	Use this option to overwrite existing files.
`-fi|--files`	List of semicolon separated files to scaffold. Use the `-lf|--listFiles` option to see the available options.
`-lf|--listFiles`	Lists the files that can be scaffolded by using the `-fi|--files` option.
`-rn|--rootNamespace`	Root namespace to use for generating Identity code.
`-u|--userClass`	Name of the user class to generate.

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator blazor-identity -h
```

Controller options

General options are shown in the following table.

Expand table

Option	Description
`-b|--bootstrapVersion`	Specifies the bootstrap version and creates a `wwwroot` folder for the Bootstrap assets if the folder isn't present.
`-dbProvider|--databaseProvider`	Database provider to use. Options include `sqlserver` (default), `sqlite`, `cosmos`, `postgres`.
`-dc|--dataContext`	The database context class to use or the name of the class to generate.
`-f|--force`	Overwrite existing files.
`-l|--layout`	Custom layout page to use.
`-m|--model`	Model class to use.
`-outDir|--relativeFolderPath`	Relative output folder path. If not specified, files are generated in the project folder.
`-scripts|--referenceScriptLibraries`	Reference script libraries in the generated views. Adds `_ValidationScriptsPartial` to `Edit` and `Create` pages.
`-sqlite|--useSqlite`	Flag to specify if the database context should use SQLite instead of SQL Server.
`-udl|--useDefaultLayout`	Use the default layout for the views.

The options unique to `controller` are shown in the following table.

Expand table

Option	Description
`-actions|--readWriteActions`	Generate controller with read/write actions without a model.
`-api|--restWithNoViews`	Generate a controller with REST style API. `noViews` is assumed and any view related options are ignored.
`-async|--useAsyncActions`	Generate asynchronous controller actions.
`-name|--controllerName`	Name of the controller.
`-namespace|--controllerNamespace`	Specify the name of the namespace to use for the generated controller.
`-nv|--noViews`	Generate **no** views.

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator controller -h
```

For an example that uses the `controller` generator, see Part 4, add a model to an ASP.NET Core MVC app.

Identity options

For more information, see Scaffold Identity in ASP.NET Core projects.

Minimal API options

Scaffold a Minimal API backend with the `minimalapi` template.

The options for `minimalapi` are shown in the following table.

Expand table

Option	Description
`-dbProvider|--databaseProvider`	Database provider to use. Options include `sqlserver` (default), `sqlite`, `cosmos`, or `postgres`.
`-dc|--dataContext`	Database context class to use.
`-e|--endpoints`	Endpoints class to use (not the file name).
`-m|--model`	Model class to use.
`-namespace|--endpointsNamespace`	Specify the name of the namespace to use for the generated endpoints file.
`-o|--open`	Use this option to enable OpenAPI.
`-outDir|--relativeFolderPath`	Relative output folder path. If not specified, files are generated in the project folder.
`-sqlite|--useSqlite`	Flag to specify if the database context should use SQLite instead of SQL Server.

The following example:

- Generates an endpoints class named `SpeakersEndpoints` with API endpoints that map to database operations using the `ApplicationDbContext` database context class and the `BackEnd.Models.Speaker` model.
- Adds `app.MapSpeakerEndpoints();` to the `Program` file (`Program.cs`) to register the endpoints class.

.NET CLI

```
dotnet aspnet-codegenerator minimalapi -dc ApplicationDbContext -e
SpeakerEndpoints -m BackEnd.Models.Speaker -o
```

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator minimalapi -h
```

Razor page options

Razor Pages can be individually scaffolded by specifying the name of the new page and the template to use. The supported templates are:

- `Empty`
- `Create`
- `Edit`
- `Delete`
- `Details`
- `List`

Typically, the template and generated file name isn't specified, which creates the following templates:

- `Create`
- `Edit`
- `Delete`
- `Details`
- `List`

General options are shown in the following table.

Expand table

Option	Description
`-b|--bootstrapVersion`	Specifies the bootstrap version and creates a `wwwroot` folder for the Bootstrap assets if the folder isn't present.
`-dbProvider|--databaseProvider`	Database provider to use. Options include `sqlserver` (default), `sqlite`, `cosmos`, `postgres`.
`-dc|--dataContext`	The database context class to use or the name of the class to generate.
`-f|--force`	Overwrite existing files.
`-l|--layout`	Custom layout page to use.
`-m|--model`	Model class to use.
`-outDir|--relativeFolderPath`	Relative output folder path. If not specified, files are generated in the project folder.
`-scripts|--referenceScriptLibraries`	Reference script libraries in the generated views. Adds `_ValidationScriptsPartial` to `Edit` and `Create` pages.
`-sqlite|--useSqlite`	Flag to specify if the database context should use SQLite instead of SQL Server.
`-udl|--useDefaultLayout`	Use the default layout for the views.

The options unique to `razorpage` are shown in the following table.

Expand table

Option	Description
`-namespace|--namespaceName`	The name of the namespace to use for the generated `PageModel` class.
`-npm|--noPageModel`	Don't generate a `PageModel` class for the `Empty` template.
`-partial|--partialView`	Generate a partial view. Layout options `-l` and `-udl` are ignored if this is specified.

The following example uses the `Edit` template to generate `CustomEditPage.cshtml` and `CustomEditPage.cshtml.cs` in the `Pages/Movies` folder:

.NET CLI

```
dotnet aspnet-codegenerator razorpage CustomEditPage Edit -dc
```

```
RazorPagesMovieContext -m Movie -outDir Pages/Movies
```

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator razorpage -h
```

For an example that uses the `razorpage` generator, see Part 2, add a model.

View options

Views can be individually scaffolded by specifying the name of the view and the template. The supported templates are:

- `Empty`
- `Create`
- `Edit`
- `Delete`
- `Details`
- `List`

General options are shown in the following table.

Expand table

Option	Description
`-b|--bootstrapVersion`	Specifies the bootstrap version and creates a `wwwroot` folder for the Bootstrap assets if the folder isn't present.
`-dbProvider|--databaseProvider`	Database provider to use. Options include `sqlserver` (default), `sqlite`, `cosmos`, `postgres`.
`-dc|--dataContext`	The database context class to use or the name of the class to generate.
`-f|--force`	Overwrite existing files.
`-l|--layout`	Custom layout page to use.
`-m|--model`	Model class to use.
`-outDir|--relativeFolderPath`	Relative output folder path. If not specified, files are generated in the project folder.

Option	Description
`-scripts|--referenceScriptLibraries`	Reference script libraries in the generated views. Adds `_ValidationScriptsPartial` to `Edit` and `Create` pages.
`-sqlite|--useSqlite`	Flag to specify if the database context should use SQLite instead of SQL Server.
`-udl|--useDefaultLayout`	Use the default layout for the views.

The options unique to `view` are shown in the following table.

Expand table

Option	Description
`-namespace|--controllerNamespace`	Specify the name of the namespace to use for the generated controller.
`-partial|--partialView`	Generate a partial view. Other layout options (`-l` and `-udl`) are ignored if this is specified.

The following example uses the `Edit` template to generate `CustomEditView.cshtml` in the `Views/Movies` folder:

.NET CLI

```
dotnet aspnet-codegenerator view CustomEditView Edit -dc MovieContext -m
Movie -outDir Views/Movies
```

Use the `-h|--help` option for help:

.NET CLI

```
dotnet aspnet-codegenerator view -h
```

Choose between controller-based APIs and minimal APIs

Article • 04/11/2023

ASP.NET Core supports two approaches to creating APIs: a controller-based approach and minimal APIs. *Controllers* in an API project are classes that derive from ControllerBase. *Minimal APIs* define endpoints with logical handlers in lambdas or methods. This article points out differences between the two approaches.

The design of minimal APIs hides the host class by default and focuses on configuration and extensibility via extension methods that take functions as lambda expressions. Controllers are classes that can take dependencies via constructor injection or property injection, and generally follow object-oriented patterns. Minimal APIs support dependency injection through other approaches such as accessing the service provider.

Here's sample code for an API based on controllers:

C#

```
namespace APIWithControllers;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        var app = builder.Build();

        app.UseHttpsRedirection();

        app.MapControllers();

        app.Run();
    }
}
```

C#

```
using Microsoft.AspNetCore.Mvc;

namespace APIWithControllers.Controllers;
[ApiController]
[Route("[controller]")]
```

```
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy",
"Hot", "Sweltering", "Scorching"
    };

    private readonly ILogger<WeatherForecastController> _logger;

    public WeatherForecastController(ILogger<WeatherForecastController>
logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}
```

The following code provides the same functionality in a minimal API project. Notice that the minimal API approach involves including the related code in lambda expressions.

C#

```
namespace MinimalAPI;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        var app = builder.Build();

        app.UseHttpsRedirection();

        var summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm",
"Balmy", "Hot", "Sweltering", "Scorching"
        };
```

```
        app.MapGet("/weatherforecast", (HttpContext httpContext) =>
        {
            var forecast = Enumerable.Range(1, 5).Select(index =>
                new WeatherForecast
                {
                    Date =
DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
                    TemperatureC = Random.Shared.Next(-20, 55),
                    Summary =
summaries[Random.Shared.Next(summaries.Length)]
                })
                .ToArray();
            return forecast;
        });

        app.Run();
    }
}
```

Both API projects refer to the following class:

C#

```
namespace APIWithControllers;

public class WeatherForecast
{
    public DateOnly Date { get; set; }

    public int TemperatureC { get; set; }

    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

    public string? Summary { get; set; }
}
```

Minimal APIs have many of the same capabilities as controller-based APIs. They support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses. There are a few capabilities available with controller-based APIs that are not yet supported or implemented by minimal APIs. These include:

- No built-in support for model binding (IModelBinderProvider, IModelBinder). Support can be added with a custom binding shim.
- No built-in support for validation (IModelValidator).
- No support for application parts or the application model. There's no way to apply or build your own conventions.

- No built-in view rendering support. We recommend using Razor Pages for rendering views.
- No support for JsonPatch
- No support for OData

See also

- Create web APIs with ASP.NET Core.
- Tutorial: Create a web API with ASP.NET Core
- Minimal APIs overview
- Tutorial: Create a minimal API with ASP.NET Core

Create web APIs with ASP.NET Core

Article • 05/29/2024

ASP.NET Core supports creating web APIs using controllers or using minimal APIs. *Controllers* in a web API are classes that derive from ControllerBase. Controllers are activated and disposed on a per request basis.

This article shows how to use controllers for handling web API requests. For information on creating web APIs without controllers, see Tutorial: Create a minimal API with ASP.NET Core.

ControllerBase class

A controller-based web API consists of one or more controller classes that derive from ControllerBase. The web API project template provides a starter controller:

C#

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Web API controllers should typically derive from ControllerBase rather from Controller. `Controller` derives from ControllerBase and adds support for views, so it's for handling web pages, not web API requests. If the same controller must support views and web APIs, derive from `Controller`.

The `ControllerBase` class provides many properties and methods that are useful for handling HTTP requests. For example, CreatedAtAction returns a 201 status code:

C#

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
             _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

The following table contains examples of methods in `ControllerBase`.

Expand table

Method	Notes
BadRequest	Returns 400 status code.
NotFound	Returns 404 status code.
PhysicalFile	Returns a file.
TryUpdateModelAsync	Invokes model binding.
TryValidateModel	Invokes model validation.

For a list of all available methods and properties, see ControllerBase.

Attributes

The Microsoft.AspNetCore.Mvc namespace provides attributes that can be used to configure the behavior of web API controllers and action methods. The following example uses attributes to specify the supported HTTP action verb and any known HTTP status codes that could be returned:

C#

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
             _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);

    return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);
}
```

Here are some more examples of attributes that are available.

Expand table

Attribute	Notes
[Route]	Specifies URL pattern for a controller or action.

Attribute	Notes
[Bind]	Specifies prefix and properties to include for model binding.
[HttpGet]	Identifies an action that supports the HTTP GET action verb.
[Consumes]	Specifies data types that an action accepts.
[Produces]	Specifies data types that an action returns.

For a list that includes the available attributes, see the Microsoft.AspNetCore.Mvc namespace.

ApiController attribute

The [ApiController] attribute can be applied to a controller class to enable the following opinionated, API-specific behaviors:

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference
- Problem details for error status codes

Attribute on specific controllers

The `[ApiController]` attribute can be applied to specific controllers, as in the following example from the project template:

C#

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Attribute on multiple controllers

One approach to using the attribute on more than one controller is to create a custom base controller class annotated with the [ApiController] attribute. The following example shows a custom base class and a controller that derives from it:

C#

```
[ApiController]
public class MyControllerBase : ControllerBase
{
}
```

C#

```
[Produces(MediaTypeNames.Application.Json)]
[Route("[controller]")]
public class PetsController : MyControllerBase
```

Attribute on an assembly

The `[ApiController]` attribute can be applied to an assembly. When the `[ApiController]` attribute is applied to an assembly, all controllers in the assembly have the `[ApiController]` attribute applied. There's no way to opt out for individual controllers. Apply the assembly-level attribute to the `Program.cs` file:

C#

```
using Microsoft.AspNetCore.Mvc;
[assembly: ApiController]
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Attribute routing requirement

The `[ApiController]` attribute makes attribute routing a requirement. For example:

C#

```
[ApiController]
[Route("[controller]")]
```

```
public class WeatherForecastController : ControllerBase
```

Actions are inaccessible via conventional routes defined by `UseEndpoints`, UseMvc, or UseMvcWithDefaultRoute.

Automatic HTTP 400 responses

The `[ApiController]` attribute makes model validation errors automatically trigger an HTTP 400 response. Consequently, the following code is unnecessary in an action method:

C#

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

ASP.NET Core MVC uses the ModelStateInvalidFilter action filter to do the preceding check.

Default BadRequest response

The default response type for an HTTP 400 response is ValidationProblemDetails. The following response body is an example of the serialized type:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "|7fb5e16a-4c8f23bbfc974667.",
  "errors": {
    "": [
      "A non-empty request body is required."
    ]
  }
}
```

The `ValidationProblemDetails` type:

- Provides a machine-readable format for specifying errors in web API responses.
- Complies with the RFC 7807 specification.

To make automatic and custom responses consistent, call the ValidationProblem method instead of BadRequest. `ValidationProblem` returns a ValidationProblemDetails object as well as the automatic response.

Log automatic 400 responses

To log automatic 400 responses, set the InvalidModelStateResponseFactory delegate property to perform custom processing. By default, `InvalidModelStateResponseFactory` uses ProblemDetailsFactory to create an instance of ValidationProblemDetails.

The following example shows how to retrieve an instance of ILogger<TCategoryName> to log information about an automatic 400 response:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
      // To preserve the default behavior, capture the original delegate to
call later.
        var builtInFactory = options.InvalidModelStateResponseFactory;

        options.InvalidModelStateResponseFactory = context =>
        {
            var logger = context.HttpContext.RequestServices
                                .GetRequiredService<ILogger<Program>>();

            // Perform logging here.
            // ...

            // Invoke the default behavior, which produces a
ValidationProblemDetails
            // response.
            // To produce a custom response, return a different
implementation of
            // IActionResult instead.
            return builtInFactory(context);
        };
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();
```

```
app.Run();
```

Disable automatic 400 response

To disable the automatic 400 behavior, set the SuppressModelStateInvalidFilter property to `true`. Add the following highlighted code:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Binding source parameter inference

A binding source attribute defines the location at which an action parameter's value is found. The following binding source attributes exist:

Expand table

Attribute	Binding source
[FromBody]	Request body
[FromForm]	Form data in the request body

Attribute	Binding source
[FromHeader]	Request header
[FromQuery]	Request query string parameter
[FromRoute]	Route data from the current request
[FromServices]	The request service injected as an action parameter
[AsParameters]	Method parameters

> ⚠ **Warning**
>
> Don't use `[FromRoute]` when values might contain `%2f` (that is `/`). `%2f` won't be unescaped to `/`. Use `[FromQuery]` if the value might contain `%2f`.

Without the `[ApiController]` attribute or binding source attributes like `[FromQuery]`, the ASP.NET Core runtime attempts to use the complex object model binder. The complex object model binder pulls data from value providers in a defined order.

In the following example, the `[FromQuery]` attribute indicates that the `discontinuedOnly` parameter value is provided in the request URL's query string:

C#

```
[HttpGet]
public ActionResult<List<Product>> Get(
    [FromQuery] bool discontinuedOnly = false)
{
    List<Product> products = null;

    if (discontinuedOnly)
    {
        products = _productsInMemoryStore.Where(p =>
p.IsDiscontinued).ToList();
    }
    else
    {
        products = _productsInMemoryStore;
    }

    return products;
}
```

The `[ApiController]` attribute applies inference rules for the default data sources of action parameters. These rules save you from having to identify binding sources

manually by applying attributes to the action parameters. The binding source inference rules behave as follows:

- `[FromServices]` is inferred for complex type parameters registered in the DI Container.
- `[FromBody]` is inferred for complex type parameters not registered in the DI Container. An exception to the `[FromBody]` inference rule is any complex, built-in type with a special meaning, such as IFormCollection and CancellationToken. The binding source inference code ignores those special types.
- `[FromForm]` is inferred for action parameters of type IFormFile and IFormFileCollection. It's not inferred for any simple or user-defined types.
- `[FromRoute]` is inferred for any action parameter name matching a parameter in the route template. When more than one route matches an action parameter, any route value is considered `[FromRoute]`.
- `[FromQuery]` is inferred for any other action parameters.

FromBody inference notes

`[FromBody]` isn't inferred for simple types such as `string` or `int`. Therefore, the `[FromBody]` attribute should be used for simple types when that functionality is needed.

When an action has more than one parameter bound from the request body, an exception is thrown. For example, all of the following action method signatures cause an exception:

- `[FromBody]` inferred on both because they're complex types.

  C#

  ```
  [HttpPost]
  public IActionResult Action1(Product product, Order order)
  ```

- `[FromBody]` attribute on one, inferred on the other because it's a complex type.

  C#

  ```
  [HttpPost]
  public IActionResult Action2(Product product, [FromBody] Order order)
  ```

- `[FromBody]` attribute on both.

  C#

```
[HttpPost]
public IActionResult Action3([FromBody] Product product, [FromBody]
Order order)
```

FromServices inference notes

Parameter binding binds parameters through dependency injection when the type is configured as a service. This means it's not required to explicitly apply the [FromServices] attribute to a parameter. In the following code, both actions return the time:

C#

```
[Route("[controller]")]
[ApiController]
public class MyController : ControllerBase
{
    public ActionResult GetWithAttribute([FromServices] IDateTime dateTime)
                                                        => Ok(dateTime.Now);

    [Route("noAttribute")]
    public ActionResult Get(IDateTime dateTime) => Ok(dateTime.Now);
}
```

In rare cases, automatic DI can break apps that have a type in DI that is also accepted in an API controller's action methods. It's not common to have a type in DI and as an argument in an API controller action.

To disable `[FromServices]` inference for a single action parameter, apply the desired binding source attribute to the parameter. For example, apply the `[FromBody]` attribute to an action parameter that should be bound from the body of the request.

To disable `[FromServices]` inference globally, set DisableImplicitFromServicesParameters to `true`:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddSingleton<IDateTime, SystemDateTime>();
```

```
builder.Services.Configure<ApiBehaviorOptions>(options =>
{
    options.DisableImplicitFromServicesParameters = true;
});

var app = builder.Build();

app.MapControllers();

app.Run();
```

Types are checked at app startup with IServiceProviderIsService to determine if an argument in an API controller action comes from DI or from the other sources.

The mechanism to infer binding source of API Controller action parameters uses the following rules:

- A previously specified BindingInfo.BindingSource is never overwritten.
- A complex type parameter, registered in the DI container, is assigned BindingSource.Services.
- A complex type parameter, not registered in the DI container, is assigned BindingSource.Body.
- A parameter with a name that appears as a route value in ***any*** route template is assigned BindingSource.Path.
- All other parameters are BindingSource.Query.

Disable inference rules

To disable binding source inference, set SuppressInferBindingSourcesForParameters to `true`:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
        options.DisableImplicitFromServicesParameters = true;
    });
```

```
var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Multipart/form-data request inference

The `[ApiController]` attribute applies an inference rule for action parameters of type IFormFile and IFormFileCollection. The `multipart/form-data` request content type is inferred for these types.

To disable the default behavior, set the SuppressConsumesConstraintForFormFileParameters property to `true`:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Problem details for error status codes

MVC transforms an error result (a result with status code 400 or higher) to a result with ProblemDetails. The `ProblemDetails` type is based on the RFC 7807 specification for providing machine-readable error details in an HTTP response.

Consider the following code in a controller action:

C#

```
if (pet == null)
{
    return NotFound();
}
```

The `NotFound` method produces an HTTP 404 status code with a `ProblemDetails` body. For example:

JSON

```
{
  type: "https://tools.ietf.org/html/rfc7231#section-6.5.4",
  title: "Not Found",
  status: 404,
  traceId: "0HLHLV31KRN83:00000001"
}
```

Disable ProblemDetails response

The automatic creation of a `ProblemDetails` for error status codes is disabled when the SuppressMapClientErrors property is set to `true`. Add the following code:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressConsumesConstraintForFormFileParameters = true;
        options.SuppressInferBindingSourcesForParameters = true;
        options.SuppressModelStateInvalidFilter = true;
        options.SuppressMapClientErrors = true;
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
```

```
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Define supported request content types with the [Consumes] attribute

By default, an action supports all available request content types. For example, if an app is configured to support both JSON and XML input formatters, an action supports multiple content types, including `application/json` and `application/xml`.

The [Consumes] attribute allows an action to limit the supported request content types. Apply the `[Consumes]` attribute to an action or controller, specifying one or more content types:

C#

```
[HttpPost]
[Consumes("application/xml")]
public IActionResult CreateProduct(Product product)
```

In the preceding code, the `CreateProduct` action specifies the content type `application/xml`. Requests routed to this action must specify a `Content-Type` header of `application/xml`. Requests that don't specify a `Content-Type` header of `application/xml` result in a 415 Unsupported Media Type response.

The `[Consumes]` attribute also allows an action to influence its selection based on an incoming request's content type by applying a type constraint. Consider the following example:

C#

```
[ApiController]
[Route("api/[controller]")]
public class ConsumesController : ControllerBase
{
    [HttpPost]
```

```
    [Consumes("application/json")]
    public IActionResult PostJson(IEnumerable<int> values) =>
        Ok(new { Consumes = "application/json", Values = values });

    [HttpPost]
    [Consumes("application/x-www-form-urlencoded")]
    public IActionResult PostForm([FromForm] IEnumerable<int> values) =>
        Ok(new { Consumes = "application/x-www-form-urlencoded", Values =
values });
}
```

In the preceding code, `ConsumesController` is configured to handle requests sent to the `https://localhost:5001/api/Consumes` URL. Both of the controller's actions, `PostJson` and `PostForm`, handle POST requests with the same URL. Without the `[Consumes]` attribute applying a type constraint, an ambiguous match exception is thrown.

The `[Consumes]` attribute is applied to both actions. The `PostJson` action handles requests sent with a `Content-Type` header of `application/json`. The `PostForm` action handles requests sent with a `Content-Type` header of `application/x-www-form-urlencoded`.

Additional resources

- View or download sample code. (How to download).
- Controller action return types in ASP.NET Core web API
- Handle errors in ASP.NET Core controller-based web APIs
- Custom formatters in ASP.NET Core Web API
- Format response data in ASP.NET Core Web API
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Routing to controller actions in ASP.NET Core
- Use port tunneling Visual Studio to debug web APIs
- Create a web API with ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Tutorial: Create a web API with ASP.NET Core

Article • 08/23/2024

By Rick Anderson and Kirk Larkin

This tutorial teaches the basics of building a controller-based web API that uses a database. Another approach to creating APIs in ASP.NET Core is to create *minimal APIs*. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating a minimal API, see Tutorial: Create a minimal API with ASP.NET Core.

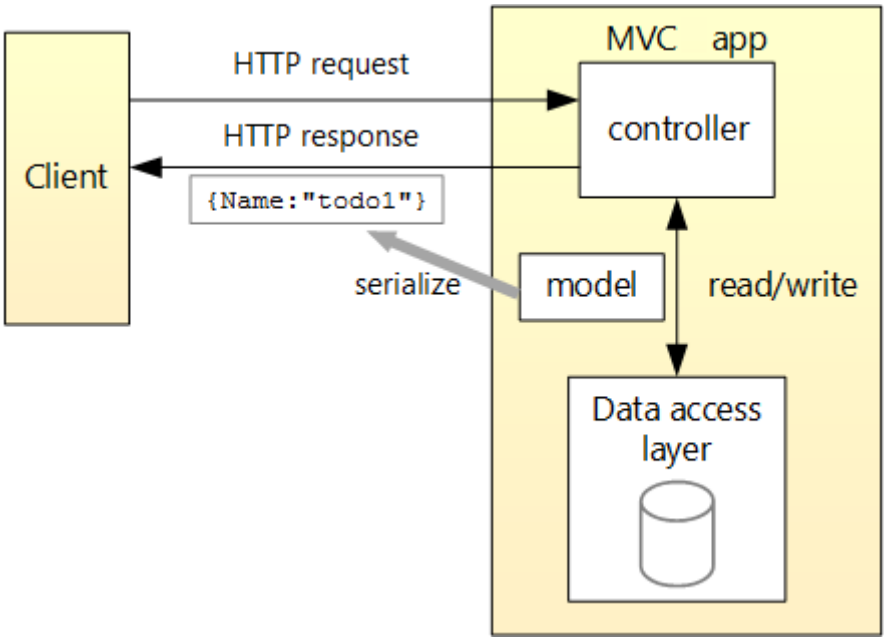
Overview

This tutorial creates the following API:

Expand table

API	Description	Request body	Response body
`GET /api/todoitems`	Get all to-do items	None	Array of to-do items
`GET /api/todoitems/{id}`	Get an item by ID	None	To-do item
`POST /api/todoitems`	Add a new item	To-do item	To-do item
`PUT /api/todoitems/{id}`	Update an existing item	To-do item	None
`DELETE /api/todoitems/{id}`	Delete an item	None	None

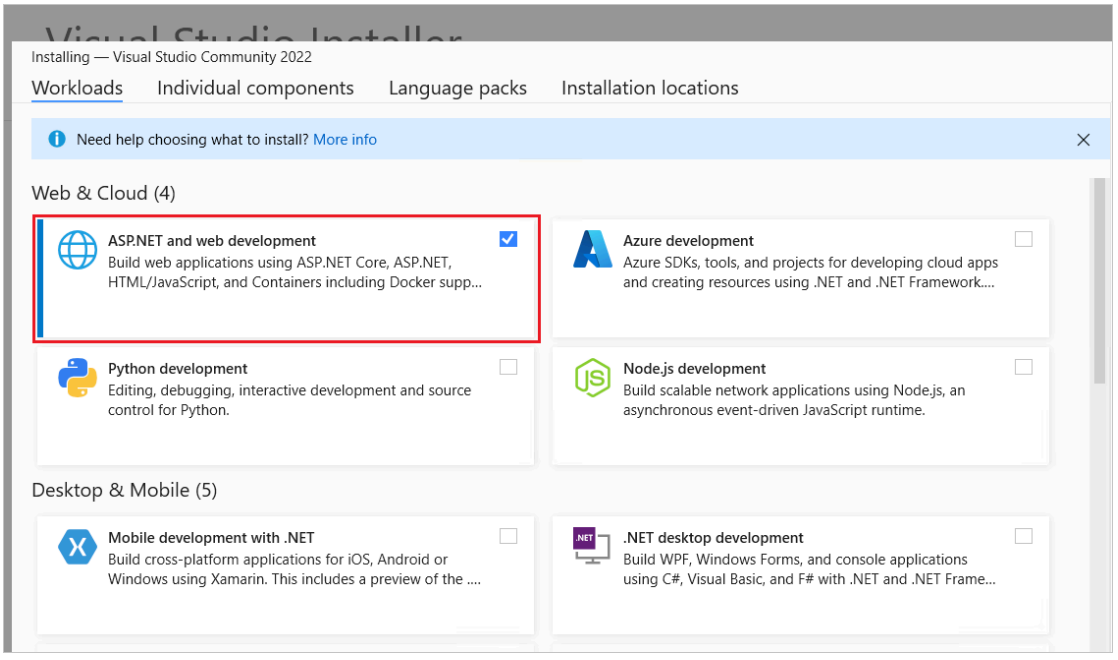
The following diagram shows the design of the app.



Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a web project

Visual Studio

- From the **File** menu, select **New** > **Project**.
- Enter *Web API* in the search box.
- Select the **ASP.NET Core Web API** template and select **Next**.
- In the **Configure your new project dialog**, name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Confirm the **Framework** is **.NET 8.0 (Long Term Support)**.
  - Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
  - Confirm the checkbox for **Enable OpenAPI support** is checked.
  - Select **Create**.

Add a NuGet package

A NuGet package must be added to support the database used in this tutorial.

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab.
- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select `Microsoft.EntityFrameworkCore.InMemory`.
- Select the **Project** checkbox in the right pane and then select **Install**.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

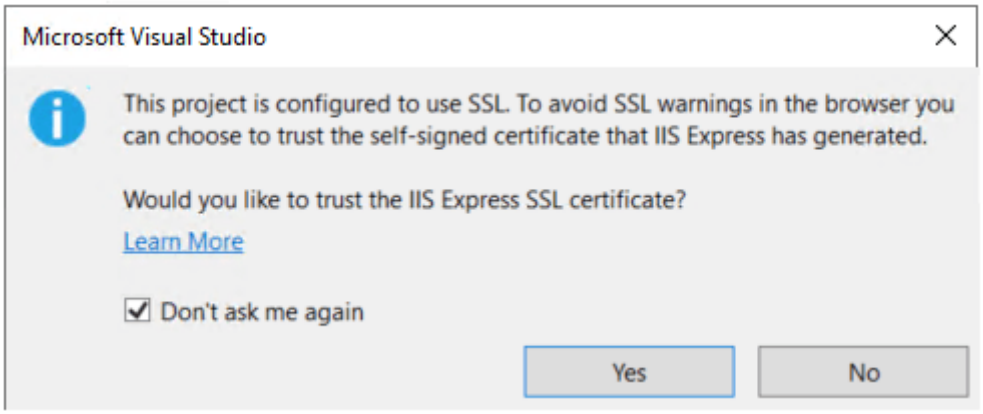
Test the project

The project template creates a `WeatherForecast` API with support for Swagger.

Visual Studio

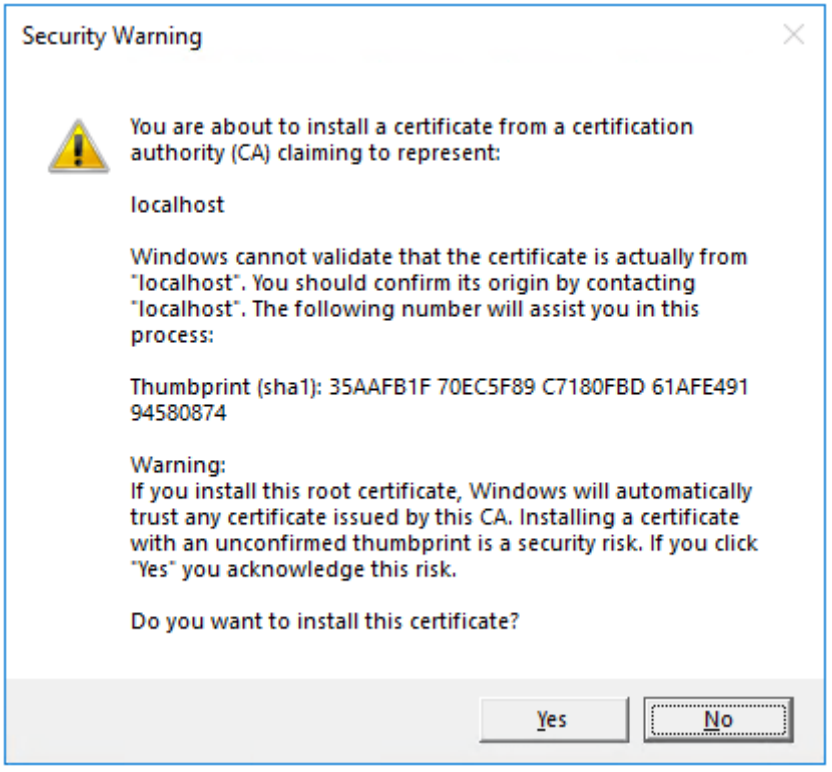
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog when a project is not yet configured to use SSL:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the default browser and navigates to `https://localhost:<port>/swagger/index.html`, where `<port>` is a randomly chosen port number set at the project creation.

The Swagger page `/swagger/index.html` is displayed. Select **GET** > **Try it out** > **Execute**. The page displays:

- The Curl command to test the WeatherForecast API.
- The URL to test the WeatherForecast API.
- The response code, body, and headers.
- A drop-down list box with media types and the example value and schema.

If the Swagger page doesn't appear, see this GitHub issue.

Swagger is used to generate useful documentation and help pages for web APIs. This tutorial uses Swagger to test the app. For more information on Swagger, see ASP.NET Core web API documentation with Swagger / OpenAPI.

Copy and paste the **Request URL** in the browser: `https://localhost:<port>/weatherforecast`

JSON similar to the following example is returned:

JSON


```
[
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
```

```
        "summary": "Chilly"
    }
]
```

Add a model class

A *model* is a set of classes that represent the data that the app manages. The model for this app is the `TodoItem` class.

Visual Studio

- In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder `Models`.
- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoItem* and select **Add**.
- Replace the template code with the following:

C#

```
namespace TodoApi.Models;

public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The `Id` property functions as the unique key in a relational database.

Model classes can go anywhere in the project, but the `Models` folder is used by convention.

Add a database context

The *database context* is the main class that coordinates Entity Framework functionality for a data model. This class is created by deriving from the Microsoft.EntityFrameworkCore.DbContext class.

Visual Studio

- Right-click the `Models` folder and select **Add** > **Class**. Name the class *TodoContext* and click **Add**.

- Enter the following code:

```
C#

using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models;

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

Register the database context

In ASP.NET Core, services such as the DB context must be registered with the dependency injection (DI) container. The container provides the service to controllers.

Update `Program.cs` with the following highlighted code:

```
C#

using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddDbContext<TodoContext>(opt =>
    opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
```

```
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Adds `using` directives.
- Adds the database context to the DI container.
- Specifies that the database context will use an in-memory database.

Scaffold a controller

Visual Studio

- Right-click the `Controllers` folder.
- Select **Add** > **New Scaffolded Item**.
- Select **API Controller with actions, using Entity Framework**, and then select **Add**.
- In the **Add API Controller with actions, using Entity Framework** dialog:
  - Select **TodoItem (TodoApi.Models)** in the **Model class**.
  - Select **TodoContext (TodoApi.Models)** in the **Data context class**.
  - Select **Add**.

  If the scaffolding operation fails, select **Add** to try scaffolding a second time.

The generated code:

- Marks the class with the [ApiController] attribute. This attribute indicates that the controller responds to web API requests. For information about specific behaviors that the attribute enables, see Create web APIs with ASP.NET Core.
- Uses DI to inject the database context (`TodoContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

The ASP.NET Core templates for:

- Controllers with views include `[action]` in the route template.
- API controllers don't include `[action]` in the route template.

When the `[action]` token isn't in the route template, the action name (method name) isn't included in the endpoint. That is, the action's associated method name isn't used in the matching route.

Update the PostTodoItem create method

Update the return statement in the `PostTodoItem` to use the nameof operator:

C#

```
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //    return CreatedAtAction("GetTodoItem", new { id = todoItem.Id },
todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id },
todoItem);
}
```

The preceding code is an `HTTP POST` method, as indicated by the [HttpPost] attribute. The method gets the value of the `TodoItem` from the body of the HTTP request.

For more information, see Attribute routing with Http[Verb] attributes.

The CreatedAtAction method:

- Returns an HTTP 201 status code if successful. `HTTP 201` is the standard response for an `HTTP POST` method that creates a new resource on the server.
- Adds a Location header to the response. The `Location` header specifies the URI of the newly created to-do item. For more information, see 10.2.2 201 Created.
- References the `GetTodoItem` action to create the `Location` header's URI. The C# `nameof` keyword is used to avoid hard-coding the action name in the `CreatedAtAction` call.

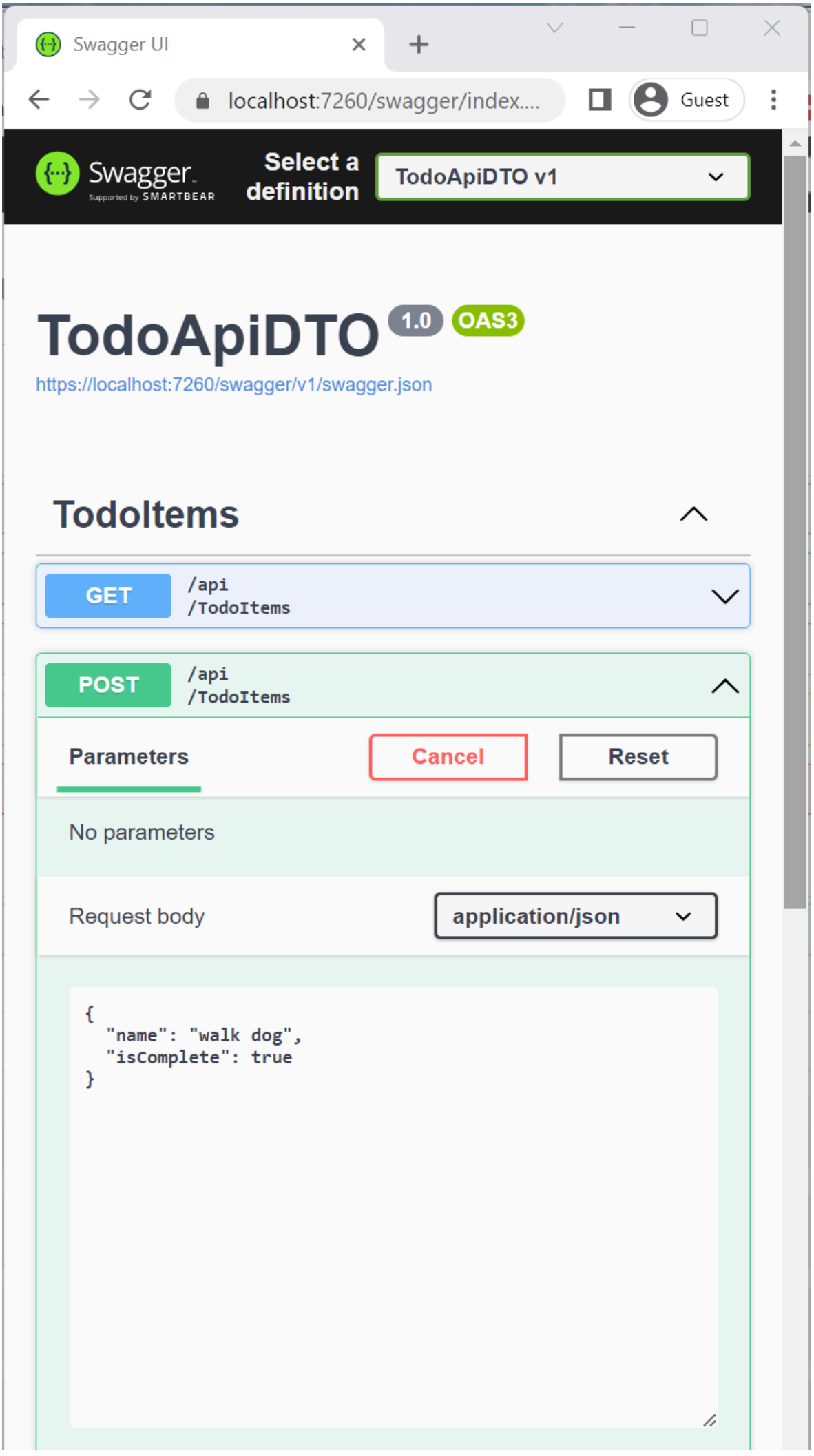
Test PostTodoItem

- Press Ctrl+F5 to run the app.

- In the Swagger browser window, select **POST /api/TodoItems**, and then select **Try it out**.
- In the **Request body** input window, update the JSON. For example,

```JSON
{
  "name": "walk dog",
  "isComplete": true
}
```

- Select **Execute**


Swagger UI
localhost:7260/swagger/index....
Guest
Swagger
Supported by SMARTBEAR
Select a definition
TodoApiDTO v1
TodoApiDTO
1.0
OAS3
https://localhost:7260/swagger/v1/swagger.json
TodoItems
GET
/api
/TodoItems
POST
/api
/TodoItems
Parameters
Cancel
Reset
No parameters
Request body
application/json
{
"name": "walk dog",
"isComplete": true
}



Test the location header URI

In the preceding POST, the Swagger UI shows the location header under **Response headers**. For example, `location: https://localhost:7260/api/TodoItems/1`. The location header shows the URI to the created resource.

To test the location header:

- In the Swagger browser window, select **GET /api/TodoItems/{id}**, and then select **Try it out**.
- Enter `1` in the `id` input box, and then select **Execute**.

GET /api/TodoItems/{id}

Parameters Cancel

Name	Description
id * required integer($int64) (path)	1

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'https://localhost:7260/api/TodoItems/1' \
  -H 'accept: text/plain'
```

Request URL

```
https://localhost:7260/api/TodoItems/1
```

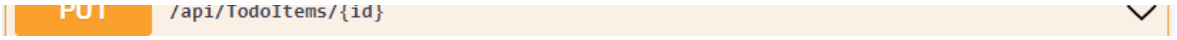
Server response

Code	Details

200	Response body
	`{ "id": 1, "name": "walk dog", "isComplete": true }` Download
	Response headers
	`content-type: application/json; charset=utf-8`
	`date: Fri,21 Oct 2022 00:32:45 GMT`
	`server: Kestrel`

Responses

Code	Description	Links
200	Success Media type: text/plain Controls Accept header. Example Value | Schema `{ "id": 0, "name": "string", "isComplete": true }`	*No links*



Examine the GET methods

Two GET endpoints are implemented:

- `GET /api/todoitems`
- `GET /api/todoitems/{id}`

The previous section showed an example of the `/api/todoitems/{id}` route.

Follow the POST instructions to add another todo item, and then test the `/api/todoitems` route using Swagger.

This app uses an in-memory database. If the app is stopped and started, the preceding GET request doesn't return any data. If no data is returned, POST data to the app.

Routing and URL paths

The [HttpGet] attribute denotes a method that responds to an `HTTP GET` request. The URL path for each method is constructed as follows:

- Start with the template string in the controller's `Route` attribute:

```
C#

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
```

- Replace `[controller]` with the name of the controller, which by convention is the controller class name minus the "Controller" suffix. For this sample, the controller class name is **TodoItems**Controller, so the controller name is "TodoItems". ASP.NET Core routing is case insensitive.

- If the `[HttpGet]` attribute has a route template (for example, `[HttpGet("products")]`), append that to the path. This sample doesn't use a template. For more information, see Attribute routing with Http[Verb] attributes.

In the following `GetTodoItem` method, `"{id}"` is a placeholder variable for the unique identifier of the to-do item. When `GetTodoItem` is invoked, the value of `"{id}"` in the URL is provided to the method in its `id` parameter.

C#

```
[HttpGet("{id}")]
public async Task<ActionResult<TodoItem>> GetTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null)
    {
        return NotFound();
    }

    return todoItem;
}
```

Return values

The return type of the `GetTodoItems` and `GetTodoItem` methods is ActionResult<T> type. ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

`ActionResult` return types can represent a wide range of HTTP status codes. For example, `GetTodoItem` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an `HTTP 200` response.

The PutTodoItem method

Examine the `PutTodoItem` method:

C#

```
[HttpPut("{id}")]
public async Task<IActionResult> PutTodoItem(long id, TodoItem todoItem)
{
    if (id != todoItem.Id)
    {
        return BadRequest();
    }

    _context.Entry(todoItem).State = EntityState.Modified;
```

```
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoItemExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

`PutTodoItem` is similar to `PostTodoItem`, except it uses `HTTP PUT`. The response is 204 (No Content). According to the HTTP specification, a `PUT` request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PutTodoItem method

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Using the Swagger UI, use the PUT button to update the `TodoItem` that has Id = 1 and set its name to `"feed fish"`. Note the response is HTTP 204 No Content.

The DeleteTodoItem method

Examine the `DeleteTodoItem` method:

C#

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteTodoItem(long id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);
    if (todoItem == null)
    {
        return NotFound();
```

```
    }

    _context.TodoItems.Remove(todoItem);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Test the DeleteTodoItem method

Use the Swagger UI to delete the `TodoItem` that has Id = 1. Note the response is HTTP 204 No Content.

Test with other tools

There are many other tools that can be used to test web APIs, for example:

- Visual Studio Endpoints Explorer and .http files
- http-repl
- curl. Swagger uses `curl` and shows the `curl` commands it submits.
- Fiddler

For more information, see:

- Minimal API tutorial: test with .http files and Endpoints Explorer
- Install and test APIs with http-repl

Prevent over-posting

Currently the sample app exposes the entire `TodoItem` object. Production apps typically limit the data that's input and returned using a subset of the model. There are multiple reasons behind this, and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this tutorial.

A DTO may be used to:

- Prevent over-posting.
- Hide properties that clients are not supposed to view.
- Omit some properties in order to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `TodoItem` class to include a secret field:

C#

```
namespace TodoApi.Models
{
    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
        public string? Secret { get; set; }
    }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a DTO model:

C#

```
namespace TodoApi.Models;

public class TodoItemDTO
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Update the `TodoItemsController` to use `TodoItemDTO`:

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi.Controllers;

[Route("api/[controller]")]
[ApiController]
public class TodoItemsController : ControllerBase
{
    private readonly TodoContext _context;

    public TodoItemsController(TodoContext context)
```

```
    {
        _context = context;
    }

    // GET: api/TodoItems
    [HttpGet]
    public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
        return await _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

    // GET: api/TodoItems/5
    // <snippet_GetByID>
    [HttpGet("{id}")]
    public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

        if (todoItem == null)
        {
            return NotFound();
        }

        return ItemToDTO(todoItem);
    }
    // </snippet_GetByID>

    // PUT: api/TodoItems/5
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Update>
    [HttpPut("{id}")]
    public async Task<IActionResult> PutTodoItem(long id, TodoItemDTO
todoDTO)
    {
        if (id != todoDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoDTO.Name;
        todoItem.IsComplete = todoDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
```

```
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }
    // </snippet_Update>

    // POST: api/TodoItems
    // To protect from overposting attacks, see
https://go.microsoft.com/fwlink/?linkid=2123754
    // <snippet_Create>
    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> PostTodoItem(TodoItemDTO
todoDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoDTO.IsComplete,
            Name = todoDTO.Name
        };

        _context.TodoItems.Add(todoItem);
        await _context.SaveChangesAsync();

        return CreatedAtAction(
            nameof(GetTodoItem),
            new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }
    // </snippet_Create>

    // DELETE: api/TodoItems/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);
        if (todoItem == null)
        {
            return NotFound();
        }

        _context.TodoItems.Remove(todoItem);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    private bool TodoItemExists(long id)
    {
        return _context.TodoItems.Any(e => e.Id == id);
    }

    private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
```

```
        new TodoItemDTO
        {
            Id = todoItem.Id,
            Name = todoItem.Name,
            IsComplete = todoItem.IsComplete
        };
}
```

Verify you can't post or get the secret field.

Call the web API with JavaScript

See Tutorial: Call an ASP.NET Core web API with JavaScript.

Web API video series

See Video: Beginner's Series to: Web APIs.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see **Migrate from ASP.NET Core 5.0 to 6.0**.

For more information, see the Duende Identity Server documentation (Duende Software website).

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app.

Additional resources

View or download sample code for this tutorial. See how to download.

For more information, see the following resources:

- Create web APIs with ASP.NET Core
- Tutorial: Create a minimal API with ASP.NET Core
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8
- Routing to controller actions in ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Deploy ASP.NET Core apps to Azure App Service
- Host and deploy ASP.NET Core
- Create a web API with ASP.NET Core

Create a web API with ASP.NET Core and MongoDB

Article • 04/17/2024

By Pratik Khandelwal and Scott Addie

This tutorial creates a web API that runs Create, Read, Update, and Delete (CRUD) operations on a MongoDB NoSQL database.

In this tutorial, you learn how to:

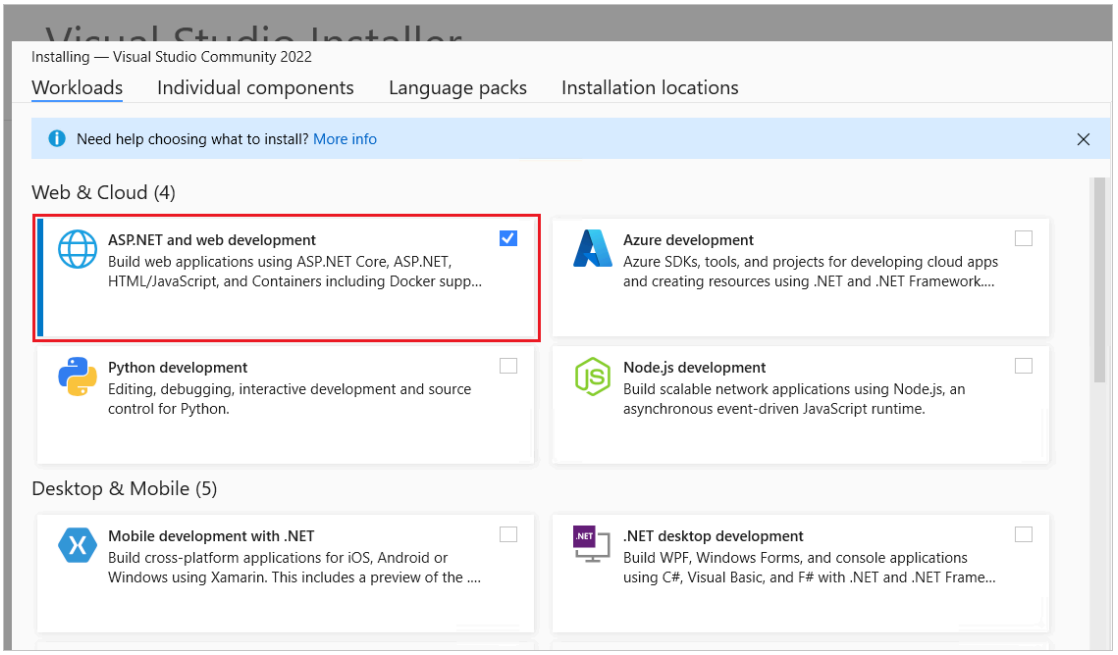
- ✓ Configure MongoDB
- ✓ Create a MongoDB database
- ✓ Define a MongoDB collection and schema
- ✓ Perform MongoDB CRUD operations from a web API
- ✓ Customize JSON serialization

Prerequisites

- MongoDB 6.0.5 or later
- MongoDB Shell

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Configure MongoDB

Enable MongoDB and MongoDB Shell access from anywhere on the development machine (Windows/Linux/macOS):

1. Download and Install MongoDB Shell:

    - macOS/Linux: Choose a directory to extract the MongoDB Shell to. Add the resulting path for `mongosh` to the `PATH` environment variable.
    - Windows: MongoDB Shell (mongosh.exe) is installed at *C:\Users<user>\AppData\Local\Programs\mongosh*. Add the resulting path for `mongosh.exe` to the `PATH` environment variable.

2. Download and Install MongoDB:

    - macOS/Linux: Verify the directory that MongoDB was installed at, usually in */usr/local/mongodb*. Add the resulting path for `mongodb` to the `PATH` environment variable.
    - Windows: MongoDB is installed at *C:\Program Files\MongoDB* by default. Add *C:\Program Files\MongoDB\Server\<version_number>\bin* to the `PATH` environment variable.

3. Choose a Data Storage Directory: Select a directory on your development machine for storing data. Create the directory if it doesn't exist. The MongoDB Shell doesn't create new directories:

    - macOS/Linux: For example, `/usr/local/var/mongodb`.

- Windows: For example, `C:\\BooksData`.

4. In the OS command shell (not the MongoDB Shell), use the following command to connect to MongoDB on default port 27017. Replace `<data_directory_path>` with the directory chosen in the previous step.

```
Console

mongod --dbpath <data_directory_path>
```

Use the previously installed MongoDB Shell in the following steps to create a database, make collections, and store documents. For more information on MongoDB Shell commands, see mongosh.

1. Open a MongoDB command shell instance by launching `mongosh.exe`.

2. In the command shell, connect to the default test database by running the following command:

```
Console

mongosh
```

3. Run the following command in the command shell:

```
Console

use BookStore
```

   A database named *BookStore* is created if it doesn't already exist. If the database does exist, its connection is opened for transactions.

4. Create a `Books` collection using following command:

```
Console

db.createCollection('Books')
```

   The following result is displayed:

```
Console

{ "ok" : 1 }
```

5. Define a schema for the `Books` collection and insert two documents using the following command:

```
Console

db.Books.insertMany([{ "Name": "Design Patterns", "Price": 54.93,
"Category": "Computers", "Author": "Ralph Johnson" }, { "Name": "Clean
Code", "Price": 43.15, "Category": "Computers","Author": "Robert C.
Martin" }])
```

A result similar to the following is displayed:

```
Console

{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("61a6058e6c43f32854e51f51"),
        ObjectId("61a6058e6c43f32854e51f52")
    ]
}
```

> **Note**
>
> The `ObjectId`s shown in the preceding result won't match those shown in the command shell.

6. View the documents in the database using the following command:

```
Console

db.Books.find().pretty()
```

A result similar to the following is displayed:

```
Console

{
    "_id" : ObjectId("61a6058e6c43f32854e51f51"),
    "Name" : "Design Patterns",
    "Price" : 54.93,
    "Category" : "Computers",
    "Author" : "Ralph Johnson"
}
{
    "_id" : ObjectId("61a6058e6c43f32854e51f52"),
```

```
    "Name" : "Clean Code",
    "Price" : 43.15,
    "Category" : "Computers",
    "Author" : "Robert C. Martin"
}
```

The schema adds an autogenerated `_id` property of type `ObjectId` for each document.

Create the ASP.NET Core web API project

Visual Studio

1. Go to **File** > **New** > **Project**.
2. Select the **ASP.NET Core Web API** project type, and select **Next**.
3. Name the project *BookStoreApi*, and select **Next**.
4. Select the **.NET 8.0 (Long Term support)** framework and select **Create**.
5. In the **Package Manager Console** window, navigate to the project root. Run the following command to install the .NET driver for MongoDB:

PowerShell

```
Install-Package MongoDB.Driver
```

Add an entity model

1. Add a *Models* directory to the project root.
2. Add a `Book` class to the *Models* directory with the following code:

C#

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace BookStoreApi.Models;

public class Book
{
```

```
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; set; }

    [BsonElement("Name")]
    public string BookName { get; set; } = null!;

    public decimal Price { get; set; }

    public string Category { get; set; } = null!;

    public string Author { get; set; } = null!;
}
```

In the preceding class, the `Id` property is:

- Required for mapping the Common Language Runtime (CLR) object to the MongoDB collection.
- Annotated with [BsonId] to make this property the document's primary key.
- Annotated with [BsonRepresentation(BsonType.ObjectId)] to allow passing the parameter as type `string` instead of an ObjectId structure. Mongo handles the conversion from `string` to `ObjectId`.

The `BookName` property is annotated with the [BsonElement] attribute. The attribute's value of `Name` represents the property name in the MongoDB collection.

Add a configuration model

1. Add the following database configuration values to `appsettings.json`:

JSON

```
{
    "BookStoreDatabase": {
        "ConnectionString": "mongodb://localhost:27017",
        "DatabaseName": "BookStore",
        "BooksCollectionName": "Books"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft.AspNetCore": "Warning"
        }
    },
    "AllowedHosts": "*"
}
```

2. Add a `BookStoreDatabaseSettings` class to the *Models* directory with the following code:

```C#
namespace BookStoreApi.Models;

public class BookStoreDatabaseSettings
{
    public string ConnectionString { get; set; } = null!;

    public string DatabaseName { get; set; } = null!;

    public string BooksCollectionName { get; set; } = null!;
}
```

   The preceding `BookStoreDatabaseSettings` class is used to store the `appsettings.json` file's `BookStoreDatabase` property values. The JSON and C# property names are named identically to ease the mapping process.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));
```

   In the preceding code, the configuration instance to which the `appsettings.json` file's `BookStoreDatabase` section binds is registered in the Dependency Injection (DI) container. For example, the `BookStoreDatabaseSettings` object's `ConnectionString` property is populated with the `BookStoreDatabase:ConnectionString` property in `appsettings.json`.

4. Add the following code to the top of `Program.cs` to resolve the `BookStoreDatabaseSettings` reference:

```C#
using BookStoreApi.Models;
```

Add a CRUD operations service

1. Add a *Services* directory to the project root.

2. Add a `BooksService` class to the *Services* directory with the following code:

C#

```
using BookStoreApi.Models;
using Microsoft.Extensions.Options;
using MongoDB.Driver;

namespace BookStoreApi.Services;

public class BooksService
{
    private readonly IMongoCollection<Book> _booksCollection;

    public BooksService(
        IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
    {
        var mongoClient = new MongoClient(
            bookStoreDatabaseSettings.Value.ConnectionString);

        var mongoDatabase = mongoClient.GetDatabase(
            bookStoreDatabaseSettings.Value.DatabaseName);

        _booksCollection = mongoDatabase.GetCollection<Book>(
            bookStoreDatabaseSettings.Value.BooksCollectionName);
    }

    public async Task<List<Book>> GetAsync() =>
        await _booksCollection.Find(_ => true).ToListAsync();

    public async Task<Book?> GetAsync(string id) =>
        await _booksCollection.Find(x => x.Id ==
id).FirstOrDefaultAsync();

    public async Task CreateAsync(Book newBook) =>
        await _booksCollection.InsertOneAsync(newBook);

    public async Task UpdateAsync(string id, Book updatedBook) =>
        await _booksCollection.ReplaceOneAsync(x => x.Id == id,
updatedBook);

    public async Task RemoveAsync(string id) =>
        await _booksCollection.DeleteOneAsync(x => x.Id == id);
}
```

In the preceding code, a `BookStoreDatabaseSettings` instance is retrieved from DI via constructor injection. This technique provides access to the `appsettings.json` configuration values that were added in the Add a configuration model section.

3. Add the following highlighted code to `Program.cs`:

```C#
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.Configure<BookStoreDatabaseSettings>(
    builder.Configuration.GetSection("BookStoreDatabase"));

builder.Services.AddSingleton<BooksService>();
```

In the preceding code, the `BooksService` class is registered with DI to support constructor injection in consuming classes. The singleton service lifetime is most appropriate because `BooksService` takes a direct dependency on `MongoClient`. Per the official Mongo Client reuse guidelines, `MongoClient` should be registered in DI with a singleton service lifetime.

4. Add the following code to the top of `Program.cs` to resolve the `BooksService` reference:

```C#
using BookStoreApi.Services;
```

The `BooksService` class uses the following `MongoDB.Driver` members to run CRUD operations against the database:

- MongoClient: Reads the server instance for running database operations. The constructor of this class is provided in the MongoDB connection string:

```C#
public BooksService(
    IOptions<BookStoreDatabaseSettings> bookStoreDatabaseSettings)
{
    var mongoClient = new MongoClient(
        bookStoreDatabaseSettings.Value.ConnectionString);

    var mongoDatabase = mongoClient.GetDatabase(
        bookStoreDatabaseSettings.Value.DatabaseName);

    _booksCollection = mongoDatabase.GetCollection<Book>(
        bookStoreDatabaseSettings.Value.BooksCollectionName);
}
```

- IMongoDatabase: Represents the Mongo database for running operations. This tutorial uses the generic GetCollection<TDocument>(collection) method on the interface to gain access to data in a specific collection. Run CRUD operations against the collection after this method is called. In the `GetCollection<TDocument>(collection)` method call:
  - `collection` represents the collection name.
  - `TDocument` represents the CLR object type stored in the collection.

`GetCollection<TDocument>(collection)` returns a MongoCollection object representing the collection. In this tutorial, the following methods are invoked on the collection:

- DeleteOneAsync: Deletes a single document matching the provided search criteria.
- Find<TDocument>: Returns all documents in the collection matching the provided search criteria.
- InsertOneAsync: Inserts the provided object as a new document in the collection.
- ReplaceOneAsync: Replaces the single document matching the provided search criteria with the provided object.

Add a controller

Add a `BooksController` class to the *Controllers* directory with the following code:

C#

```
using BookStoreApi.Models;
using BookStoreApi.Services;
using Microsoft.AspNetCore.Mvc;

namespace BookStoreApi.Controllers;

[ApiController]
[Route("api/[controller]")]
public class BooksController : ControllerBase
{
    private readonly BooksService _booksService;

    public BooksController(BooksService booksService) =>
        _booksService = booksService;

    [HttpGet]
    public async Task<List<Book>> Get() =>
        await _booksService.GetAsync();

    [HttpGet("{id:length(24)}")]
```

```
    public async Task<ActionResult<Book>> Get(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        return book;
    }

    [HttpPost]
    public async Task<IActionResult> Post(Book newBook)
    {
        await _booksService.CreateAsync(newBook);

        return CreatedAtAction(nameof(Get), new { id = newBook.Id },
newBook);
    }

    [HttpPut("{id:length(24)}")]
    public async Task<IActionResult> Update(string id, Book updatedBook)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        updatedBook.Id = book.Id;

        await _booksService.UpdateAsync(id, updatedBook);

        return NoContent();
    }

    [HttpDelete("{id:length(24)}")]
    public async Task<IActionResult> Delete(string id)
    {
        var book = await _booksService.GetAsync(id);

        if (book is null)
        {
            return NotFound();
        }

        await _booksService.RemoveAsync(id);

        return NoContent();
    }
}
```

The preceding web API controller:

- Uses the `BooksService` class to run CRUD operations.
- Contains action methods to support GET, POST, PUT, and DELETE HTTP requests.
- Calls CreatedAtAction in the `Create` action method to return an HTTP 201 response. Status code 201 is the standard response for an HTTP POST method that creates a new resource on the server. `CreatedAtAction` also adds a `Location` header to the response. The `Location` header specifies the URI of the newly created book.

Test the web API

1. Build and run the app.

2. Navigate to `https://localhost:<port>/api/books`, where `<port>` is the automatically assigned port number for the app, to test the controller's parameterless `Get` action method, select **Try it out** > **Execute**. A JSON response similar to the following is displayed:

   JSON

   ```
   [
     {
       "id": "61a6058e6c43f32854e51f51",
       "bookName": "Design Patterns",
       "price": 54.93,
       "category": "Computers",
       "author": "Ralph Johnson"
     },
     {
       "id": "61a6058e6c43f32854e51f52",
       "bookName": "Clean Code",
       "price": 43.15,
       "category": "Computers",
       "author": "Robert C. Martin"
     }
   ]
   ```

3. Navigate to `https://localhost:<port>/api/books/{id here}` to test the controller's overloaded `Get` action method. A JSON response similar to the following is displayed:

   JSON

   ```
   {
     "id": "61a6058e6c43f32854e51f52",
   ```

```
  "bookName": "Clean Code",
  "price": 43.15,
  "category": "Computers",
  "author": "Robert C. Martin"
}
```

Configure JSON serialization options

There are two details to change about the JSON responses returned in the Test the web API section:

- The property names' default camel casing should be changed to match the Pascal casing of the CLR object's property names.
- The `bookName` property should be returned as `Name`.

To satisfy the preceding requirements, make the following changes:

1. In `Program.cs`, chain the following highlighted code on to the `AddControllers` method call:

   C#

   ```
   var builder = WebApplication.CreateBuilder(args);

   // Add services to the container.
   builder.Services.Configure<BookStoreDatabaseSettings>(
       builder.Configuration.GetSection("BookStoreDatabase"));

   builder.Services.AddSingleton<BooksService>();

   builder.Services.AddControllers()
       .AddJsonOptions(
           options => options.JsonSerializerOptions.PropertyNamingPolicy =
   null);
   ```

   With the preceding change, property names in the web API's serialized JSON response match their corresponding property names in the CLR object type. For example, the `Book` class's `Author` property serializes as `Author` instead of `author`.

2. In `Models/Book.cs`, annotate the `BookName` property with the [JsonPropertyName] attribute:

   C#

   ```
   [BsonElement("Name")]
   [JsonPropertyName("Name")]
   ```

```
public string BookName { get; set; } = null!;
```

The `[JsonPropertyName]` attribute's value of `Name` represents the property name in the web API's serialized JSON response.

3. Add the following code to the top of `Models/Book.cs` to resolve the `[JsonProperty]` attribute reference:

```
C#

using System.Text.Json.Serialization;
```

4. Repeat the steps defined in the Test the web API section. Notice the difference in JSON property names.

Add authentication support to a web API

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

For more information, see the Duende Identity Server documentation (Duende Software website).

Additional resources

- View or download sample code (how to download)
- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Create a web API with ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Controller action return types in ASP.NET Core web API

Article • 01/22/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

View or download sample code (how to download)

ASP.NET Core provides the following options for web API controller action return types:

- Specific type
- IActionResult
- ActionResult<T>
- HttpResults

This article explains when it's most appropriate to use each return type.

Specific type

The most basic action returns a primitive or complex data type, for example, `string` or a custom object. Consider the following action, which returns a collection of custom `Product` objects:

C#

```
[HttpGet]
public Task<List<Product>> Get() =>
    _productContext.Products.OrderBy(p => p.Name).ToListAsync();
```

Without known conditions to safeguard against, returning a specific type could suffice. The preceding action accepts no parameters, so parameter constraints validation isn't needed.

When multiple return types are possible, it's common to mix an ActionResult return type with the primitive or complex return type. Either IActionResult or ActionResult<T> are necessary to accommodate this type of action. Several samples of multiple return types are provided in this article.

Return IEnumerable<T> or IAsyncEnumerable<T>

See Return IEnumerable<T> or IAsyncEnumerable<T> for performance considerations.

ASP.NET Core buffers the result of actions that return IEnumerable<T> before writing them to the response. Consider declaring the action signature's return type as IAsyncEnumerable<T> to guarantee asynchronous iteration. Ultimately, the iteration mode is based on the underlying concrete type being returned and the selected formatter affects how the result is processed:

- When using `System.Text.Json` formatter, MVC relies on the support that `System.Text.Json` added to **stream** the result.
- When using `Newtonsoft.Json` or with `XML-based` formatters the result is buffered.

Consider the following action, which returns sale-priced product records as `IEnumerable<Product>`:

C#

```
[HttpGet("syncsale")]
public IEnumerable<Product> GetOnSaleProducts()
{
    var products = _productContext.Products.OrderBy(p => p.Name).ToList();

    foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

The `IAsyncEnumerable<Product>` equivalent of the preceding action is:

C#

```
[HttpGet("asyncsale")]
public async IAsyncEnumerable<Product> GetOnSaleProductsAsync()
{
    var products = _productContext.Products.OrderBy(p =>
p.Name).AsAsyncEnumerable();

    await foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
```

```
        }
    }
}
```

IActionResult type

The IActionResult return type is appropriate when multiple `ActionResult` return types are possible in an action. The `ActionResult` types represent various HTTP status codes. Any non-abstract class deriving from `ActionResult` qualifies as a valid return type. Some common return types in this category are BadRequestResult (400), NotFoundResult (404), and OkObjectResult (200). Alternatively, convenience methods in the ControllerBase class can be used to return `ActionResult` types from an action. For example, `return BadRequest();` is a shorthand form of `return new BadRequestResult();`.

Because there are multiple return types and paths in this type of action, liberal use of the [ProducesResponseType] attribute is necessary. This attribute produces more descriptive response details for web API help pages generated by tools like Swagger. `[ProducesResponseType]` indicates the known types and HTTP status codes to be returned by the action.

Synchronous action

Consider the following synchronous action in which there are two possible return types:

C#

```
[HttpGet("{id}")]
[ProducesResponseType<Product>(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IActionResult GetById_IActionResult(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? NotFound() : Ok(product);
}
```

In the preceding action:

- A 404 status code is returned when the product represented by `id` doesn't exist in the underlying data store. The NotFound convenience method is invoked as shorthand for `return new NotFoundResult();`.
- A 200 status code is returned with the `Product` object when the product does exist. The Ok convenience method is invoked as shorthand for `return new`

`OkObjectResult(product);`.

Asynchronous action

Consider the following asynchronous action in which there are two possible return types:

```
C#

[HttpPost()]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> CreateAsync_IActionResult(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    return CreatedAtAction(nameof(CreateAsync_IActionResult), new { id =
product.Id }, product);
}
```

In the preceding action:

- A 400 status code is returned when the product description contains "XYZ Widget". The BadRequest convenience method is invoked as shorthand for `return new BadRequestResult();`.
- A 201 status code is generated by the CreatedAtAction convenience method when a product is created. The following code is an alternative to calling `CreatedAtAction`:

  ```
  C#

  return new CreatedAtActionResult(nameof(GetByIdAsync),
                                   "Products",
                                   new { id = product.Id },
                                   product);
  ```

  In the preceding code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

For example, the following model indicates that requests must include the `Name` and `Description` properties. Failure to provide `Name` and `Description` in the request causes model validation to fail.

C#

```
public class Product
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; } = string.Empty;

    [Required]
    public string Description { get; set; } = string.Empty;

    public bool IsOnSale { get; set; }
}
```

If the [ApiController] attribute is applied, model validation errors result in a 400 status code. For more information, see Automatic HTTP 400 responses.

ActionResult vs IActionResult

The following section compares `ActionResult` to `IActionResult`

ActionResult<T> type

ASP.NET Core includes the ActionResult<T> return type for web API controller actions. It enables returning a type deriving from ActionResult or return a specific type. `ActionResult<T>` offers the following benefits over the IActionResult type:

- The [ProducesResponseType] attribute's `Type` property can be excluded. For example, `[ProducesResponseType(200, Type = typeof(Product))]` is simplified to `[ProducesResponseType(200)]`. The action's expected return type is inferred from the `T` in `ActionResult<T>`.
- Implicit cast operators support the conversion of both `T` and `ActionResult` to `ActionResult<T>`. `T` converts to ObjectResult, which means `return new ObjectResult(T);` is simplified to `return T;`.

C# doesn't support implicit cast operators on interfaces. Consequently, conversion of the interface to a concrete type is necessary to use `ActionResult<T>`. For example, use of `IEnumerable` in the following example doesn't work:

C#

```
[HttpGet]
public ActionResult<IEnumerable<Product>> Get() =>
    _repository.GetProducts();
```

One option to fix the preceding code is to return `_repository.GetProducts().ToList();`.

Most actions have a specific return type. Unexpected conditions can occur during action execution, in which case the specific type isn't returned. For example, an action's input parameter may fail model validation. In such a case, it's common to return the appropriate `ActionResult` type instead of the specific type.

Synchronous action

Consider a synchronous action in which there are two possible return types:

C#

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public ActionResult<Product> GetById_ActionResultOfT(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? NotFound() : product;
}
```

In the preceding action:

- A 404 status code is returned when the product doesn't exist in the database.
- A 200 status code is returned with the corresponding `Product` object when the product does exist.

Asynchronous action

Consider an asynchronous action in which there are two possible return types:

C#

```
[HttpPost()]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<ActionResult<Product>> CreateAsync_ActionResultOfT(Product
product)
```

```
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    return CreatedAtAction(nameof(CreateAsync_ActionResultOfT), new { id =
product.Id }, product);
}
```

In the preceding action:

- A 400 status code (BadRequest) is returned by the ASP.NET Core runtime when:
  - The [ApiController] attribute has been applied and model validation fails.
  - The product description contains "XYZ Widget".
- A 201 status code is generated by the CreatedAtAction method when a product is created. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

HttpResults type

In addition to the MVC-specific built-in result types (IActionResult and ActionResult<T>), ASP.NET Core includes the HttpResults types that can be used in both Minimal APIs and Web API.

Different than the MVC-specific result types, the `HttpResults`:

- Are a results implementation that is processed by a call to IResult.ExecuteAsync.
- Does ***not*** leverage the configured Formatters. Not leveraging the configured formatters means:
  - Some features like `Content negotiation` aren't available.
  - The produced `Content-Type` is decided by the `HttpResults` implementation.

The `HttpResults` can be useful when sharing code between Minimal APIs and Web API.

IResult type

The Microsoft.AspNetCore.Http.HttpResults namespace contains classes that implement the IResult interface. The `IResult` interface defines a contract that represents the result

of an HTTP endpoint. The static Results class is used to create varying `IResult` objects that represent different types of responses.

The Built-in results table shows the common result helpers.

Consider the following code:

C#

```
[HttpGet("{id}")]
[ProducesResponseType<Product>(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IResult GetById(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? Results.NotFound() : Results.Ok(product);
}
```

In the preceding action:

- A 404 status code is returned when the product doesn't exist in the database.
- A 200 status code is returned with the corresponding `Product` object when the product does exist, generated by the Results.Ok<T>().

Consider the following code:

C#

```
[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType<Product>(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IResult> CreateAsync(Product product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return Results.BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    var location = Url.Action(nameof(CreateAsync), new { id = product.Id })
?? $"/{product.Id}";
    return Results.Created(location, product);
}
```

In the preceding action:

- A 400 status code is returned when:
  - The [ApiController] attribute has been applied and model validation fails.
  - The product description contains "XYZ Widget".
- A 201 status code is generated by the Results.Create method when a product is created. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

Results<TResult1, TResultN> type

The static TypedResults class returns the concrete `IResult` implementation that allows using `IResult` as return type. The usage of the concrete `IResult` implementation offers the following benefit over the IResult type:

- All the [ProducesResponseType] attributes can be excluded, since the `HttpResult` implementation contributes automatically to the endpoint metadata.

When multiple `IResult` return types are needed, returning Results<TResult1, TResultN> is preferred over returning `IResult`. Returning `Results<TResult1, TResultN>` is preferred because generic union types automatically retain the endpoint metadata.

The `Results<TResult1, TResultN>` union types implement implicit cast operators so that the compiler can automatically convert the types specified in the generic arguments to an instance of the union type. This has the added benefit of providing compile-time checking that a route handler actually only returns the results that it declares it does. Attempting to return a type that isn't declared as one of the generic arguments to `Results<>` results in a compilation error.

Consider the following code:

```
C#

[HttpGet("{id}")]
public Results<NotFound, Ok<Product>> GetById(int id)
{
    var product = _productContext.Products.Find(id);
    return product == null ? TypedResults.NotFound() :
TypedResults.Ok(product);
}
```

In the preceding action:

- A 404 status code is returned when the product doesn't exist in the database.

- A 200 status code is returned with the corresponding `Product` object when the product does exist, generated by the TypedResults.Ok<T>.

C#

```
[HttpPost]
public async Task<Results<BadRequest, Created<Product>>> CreateAsync(Product
product)
{
    if (product.Description.Contains("XYZ Widget"))
    {
        return TypedResults.BadRequest();
    }

    _productContext.Products.Add(product);
    await _productContext.SaveChangesAsync();

    var location = Url.Action(nameof(CreateAsync), new { id = product.Id })
?? $"/{product.Id}";
    return TypedResults.Created(location, product);
}
```

In the preceding action:

- A 400 status code is returned when:
  - The [ApiController] attribute was applied and model validation fails.
  - The product description contains "XYZ Widget".
- A 201 status code is generated by the TypedResults.Created method when a product is created. In this code path, the `Product` object is provided in the response body. A `Location` response header containing the newly created product's URL is provided.

Additional resources

- Handle requests with controllers in ASP.NET Core MVC
- Model validation in ASP.NET Core MVC
- ASP.NET Core web API documentation with Swagger / OpenAPI

JsonPatch in ASP.NET Core web API

Article • 03/08/2023

This article explains how to handle JSON Patch requests in an ASP.NET Core web API.

Package installation

JSON Patch support in ASP.NET Core web API is based on `Newtonsoft.Json` and requires the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package. To enable JSON Patch support:

- Install the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package.
- Call AddNewtonsoftJson. For example:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddNewtonsoftJson();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

`AddNewtonsoftJson` replaces the default `System.Text.Json`-based input and output formatters used for formatting ***all*** JSON content. This extension method is compatible with the following MVC service registration methods:

- AddRazorPages
- AddControllersWithViews
- AddControllers

JsonPatch requires setting the `Content-Type` header to `application/json-patch+json`.

Add support for JSON Patch when using System.Text.Json

The `System.Text.Json`-based input formatter doesn't support JSON Patch. To add support for JSON Patch using `Newtonsoft.Json`, while leaving the other input and output formatters unchanged:

- Install the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package.
- Update `Program.cs`:

C#

```
using JsonPatchSample;
using Microsoft.AspNetCore.Mvc.Formatters;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.InputFormatters.Insert(0,
MyJPIF.GetJsonPatchInputFormatter());
});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Formatters;
using Microsoft.Extensions.Options;

namespace JsonPatchSample;

public static class MyJPIF
{
    public static NewtonsoftJsonPatchInputFormatter
GetJsonPatchInputFormatter()
    {
        var builder = new ServiceCollection()
            .AddLogging()
```

```
            .AddMvc()
            .AddNewtonsoftJson()
            .Services.BuildServiceProvider();

        return builder
            .GetRequiredService<IOptions<MvcOptions>>()
            .Value
            .InputFormatters
            .OfType<NewtonsoftJsonPatchInputFormatter>()
            .First();
    }
}
```

The preceding code creates an instance of NewtonsoftJsonPatchInputFormatter and inserts it as the first entry in the MvcOptions.InputFormatters collection. This order of registration ensures that:

- `NewtonsoftJsonPatchInputFormatter` processes JSON Patch requests.
- The existing `System.Text.Json`-based input and formatters process all other JSON requests and responses.

Use the `Newtonsoft.Json.JsonConvert.SerializeObject` method to serialize a JsonPatchDocument.

PATCH HTTP request method

The PUT and PATCH methods are used to update an existing resource. The difference between them is that PUT replaces the entire resource, while PATCH specifies only the changes.

JSON Patch

JSON Patch is a format for specifying updates to be applied to a resource. A JSON Patch document has an array of *operations*. Each operation identifies a particular type of change. Examples of such changes include adding an array element or replacing a property value.

For example, the following JSON documents represent a resource, a JSON Patch document for the resource, and the result of applying the Patch operations.

Resource example

JSON

```
{
  "customerName": "John",
  "orders": [
    {
      "orderName": "Order0",
      "orderType": null
    },
    {
      "orderName": "Order1",
      "orderType": null
    }
  ]
}
```

JSON patch example

JSON

```
[
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "add",
    "path": "/orders/-",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

In the preceding JSON:

- The `op` property indicates the type of operation.
- The `path` property indicates the element to update.
- The `value` property provides the new value.

Resource after patch

Here's the resource after applying the preceding JSON Patch document:

JSON

```
{
  "customerName": "Barry",
```

```
  "orders": [
    {
      "orderName": "Order0",
      "orderType": null
    },
    {
      "orderName": "Order1",
      "orderType": null
    },
    {
      "orderName": "Order2",
      "orderType": null
    }
  ]
}
```

The changes made by applying a JSON Patch document to a resource are atomic. If any operation in the list fails, no operation in the list is applied.

Path syntax

The path property of an operation object has slashes between levels. For example, `"/address/zipCode"`.

Zero-based indexes are used to specify array elements. The first element of the `addresses` array would be at `/addresses/0`. To `add` to the end of an array, use a hyphen (`-`) rather than an index number: `/addresses/-`.

Operations

The following table shows supported operations as defined in the JSON Patch specification:

Operation	Notes
`add`	Add a property or array element. For existing property: set value.
`remove`	Remove a property or array element.
`replace`	Same as `remove` followed by `add` at same location.
`move`	Same as `remove` from source followed by `add` to destination using value from source.
`copy`	Same as `add` to destination using value from source.
`test`	Return success status code if value at `path` = provided `value`.

JSON Patch in ASP.NET Core

The ASP.NET Core implementation of JSON Patch is provided in the Microsoft.AspNetCore.JsonPatch NuGet package.

Action method code

In an API controller, an action method for JSON Patch:

- Is annotated with the `HttpPatch` attribute.
- Accepts a JsonPatchDocument<TModel>, typically with [FromBody].
- Calls ApplyTo(Object) on the patch document to apply the changes.

Here's an example:

C#

```
[HttpPatch]
public IActionResult JsonPatchWithModelState(
    [FromBody] JsonPatchDocument<Customer> patchDoc)
{
    if (patchDoc != null)
    {
        var customer = CreateCustomer();

        patchDoc.ApplyTo(customer, ModelState);

        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        return new ObjectResult(customer);
    }
    else
    {
        return BadRequest(ModelState);
    }
}
```

This code from the sample app works with the following `Customer` model:

C#

```
namespace JsonPatchSample.Models;

public class Customer
{
```

```
    public string? CustomerName { get; set; }
    public List<Order>? Orders { get; set; }
}
```

C#

```
namespace JsonPatchSample.Models;

public class Order
{
    public string OrderName { get; set; }
    public string OrderType { get; set; }
}
```

The sample action method:

- Constructs a `Customer`.
- Applies the patch.
- Returns the result in the body of the response.

In a real app, the code would retrieve the data from a store such as a database and update the database after applying the patch.

Model state

The preceding action method example calls an overload of `ApplyTo` that takes model state as one of its parameters. With this option, you can get error messages in responses. The following example shows the body of a 400 Bad Request response for a `test` operation:

JSON

```
{
  "Customer": [
    "The current value 'John' at path 'customerName' != test value 'Nancy'."
  ]
}
```

Dynamic objects

The following action method example shows how to apply a patch to a dynamic object:

C#

```
[HttpPatch]
public IActionResult JsonPatchForDynamic([FromBody]JsonPatchDocument patch)
{
    dynamic obj = new ExpandoObject();
    patch.ApplyTo(obj);

    return Ok(obj);
}
```

The add operation

- If `path` points to an array element: inserts new element before the one specified by `path`.
- If `path` points to a property: sets the property value.
- If `path` points to a nonexistent location:
  - If the resource to patch is a dynamic object: adds a property.
  - If the resource to patch is a static object: the request fails.

The following sample patch document sets the value of `CustomerName` and adds an `Order` object to the end of the `Orders` array.

JSON

```
[
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "add",
    "path": "/orders/-",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

The remove operation

- If `path` points to an array element: removes the element.
- If `path` points to a property:
  - If resource to patch is a dynamic object: removes the property.

- If resource to patch is a static object:
  - If the property is nullable: sets it to null.
  - If the property is non-nullable, sets it to `default<T>`.

The following sample patch document sets `CustomerName` to null and deletes `Orders[0]`:

JSON

```
[
  {
    "op": "remove",
    "path": "/customerName"
  },
  {
    "op": "remove",
    "path": "/orders/0"
  }
]
```

The replace operation

This operation is functionally the same as a `remove` followed by an `add`.

The following sample patch document sets the value of `CustomerName` and replaces `Orders[0]` with a new `Order` object:

JSON

```
[
  {
    "op": "replace",
    "path": "/customerName",
    "value": "Barry"
  },
  {
    "op": "replace",
    "path": "/orders/0",
    "value": {
      "orderName": "Order2",
      "orderType": null
    }
  }
]
```

The move operation

- If `path` points to an array element: copies `from` element to location of `path` element, then runs a `remove` operation on the `from` element.
- If `path` points to a property: copies value of `from` property to `path` property, then runs a `remove` operation on the `from` property.
- If `path` points to a nonexistent property:
  - If the resource to patch is a static object: the request fails.
  - If the resource to patch is a dynamic object: copies `from` property to location indicated by `path`, then runs a `remove` operation on the `from` property.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Sets `Orders[0].OrderName` to null.
- Moves `Orders[1]` to before `Orders[0]`.

JSON

```
[
  {
    "op": "move",
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "move",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The copy operation

This operation is functionally the same as a `move` operation without the final `remove` step.

The following sample patch document:

- Copies the value of `Orders[0].OrderName` to `CustomerName`.
- Inserts a copy of `Orders[1]` before `Orders[0]`.

JSON

```
[
  {
    "op": "copy",
```

```
    "from": "/orders/0/orderName",
    "path": "/customerName"
  },
  {
    "op": "copy",
    "from": "/orders/1",
    "path": "/orders/0"
  }
]
```

The test operation

If the value at the location indicated by `path` is different from the value provided in `value`, the request fails. In that case, the whole PATCH request fails even if all other operations in the patch document would otherwise succeed.

The `test` operation is commonly used to prevent an update when there's a concurrency conflict.

The following sample patch document has no effect if the initial value of `CustomerName` is "John", because the test fails:

JSON

```
[
  {
    "op": "test",
    "path": "/customerName",
    "value": "Nancy"
  },
  {
    "op": "add",
    "path": "/customerName",
    "value": "Barry"
  }
]
```

Get the code

View or download sample code. (How to download).

To test the sample, run the app and send HTTP requests with the following settings:

- URL: `http://localhost:{port}/jsonpatch/jsonpatchwithmodelstate`
- HTTP method: `PATCH`

- Header: `Content-Type: application/json-patch+json`
- Body: Copy and paste one of the JSON patch document samples from the *JSON* project folder.

Additional resources

- IETF RFC 5789 PATCH method specification
- IETF RFC 6902 JSON Patch specification
- IETF RFC 6901 JSON Pointer
- JSON Patch documentation. Includes links to resources for creating JSON Patch documents.
- ASP.NET Core JSON Patch source code

Format response data in ASP.NET Core Web API

Article • 04/11/2023

ASP.NET Core MVC supports formatting response data, using specified formats or in response to a client's request.

Format-specific Action Results

Some action result types are specific to a particular format, such as JsonResult and ContentResult. Actions can return results that always use a specified format, ignoring a client's request for a different format. For example, returning `JsonResult` returns JSON-formatted data and returning `ContentResult` returns plain-text-formatted string data.

An action isn't required to return any specific type. ASP.NET Core supports any object return value. Results from actions that return objects that aren't IActionResult types are serialized using the appropriate IOutputFormatter implementation. For more information, see Controller action return types in ASP.NET Core web API.

By default, the built-in helper method ControllerBase.Ok returns JSON-formatted data:

C#

```
[HttpGet]
public IActionResult Get()
    => Ok(_todoItemStore.GetList());
```

The sample code returns a list of todo items. Using the F12 browser developer tools or http-repl with the previous code displays:

- The response header containing **content-type:** `application/json; charset=utf-8`.
- The request headers. For example, the `Accept` header. The `Accept` header is ignored by the preceding code.

To return plain text formatted data, use ContentResult and the Content helper:

C#

```
[HttpGet("Version")]
public ContentResult GetVersion()
    => Content("v1.0.0");
```

In the preceding code, the `Content-Type` returned is `text/plain`.

For actions with multiple return types, return `IActionResult`. For example, when returning different HTTP status codes based on the result of the operation.

Content negotiation

Content negotiation occurs when the client specifies an Accept header. The default format used by ASP.NET Core is JSON. Content negotiation is:

- Implemented by ObjectResult.
- Built into the status code-specific action results returned from the helper methods. The action results helper methods are based on `ObjectResult`.

When a model type is returned, the return type is `ObjectResult`.

The following action method uses the `Ok` and `NotFound` helper methods:

C#

```
[HttpGet("{id:long}")]
public IActionResult GetById(long id)
{
    var todo = _todoItemStore.GetById(id);

    if (todo is null)
    {
        return NotFound();
    }

    return Ok(todo);
}
```

By default, ASP.NET Core supports the following media types:

- `application/json`
- `text/json`
- `text/plain`

Tools such as Fiddler or http-repl can set the `Accept` request header to specify the return format. When the `Accept` header contains a type the server supports, that type is returned. The next section shows how to add additional formatters.

Controller actions can return POCOs (Plain Old CLR Objects). When a POCO is returned, the runtime automatically creates an `ObjectResult` that wraps the object. The client gets

the formatted serialized object. If the object being returned is `null`, a `204 No Content` response is returned.

The following example returns an object type:

```
C#

[HttpGet("{id:long}")]
public TodoItem? GetById(long id)
    => _todoItemStore.GetById(id);
```

In the preceding code, a request for a valid todo item returns a `200 OK` response. A request for an invalid todo item returns a `204 No Content` response.

The Accept header

Content *negotiation* takes place when an `Accept` header appears in the request. When a request contains an accept header, ASP.NET Core:

- Enumerates the media types in the accept header in preference order.
- Tries to find a formatter that can produce a response in one of the formats specified.

If no formatter is found that can satisfy the client's request, ASP.NET Core:

- Returns `406 Not Acceptable` if MvcOptions.ReturnHttpNotAcceptable is set to `true`, or -
- Tries to find the first formatter that can produce a response.

If no formatter is configured for the requested format, the first formatter that can format the object is used. If no `Accept` header appears in the request:

- The first formatter that can handle the object is used to serialize the response.
- There isn't any negotiation taking place. The server is determining what format to return.

If the Accept header contains `*/*`, the Header is ignored unless `RespectBrowserAcceptHeader` is set to true on MvcOptions.

Browsers and content negotiation

Unlike typical API clients, web browsers supply `Accept` headers. Web browsers specify many formats, including wildcards. By default, when the framework detects that the

request is coming from a browser:

- The `Accept` header is ignored.
- The content is returned in JSON, unless otherwise configured.

This approach provides a more consistent experience across browsers when consuming APIs.

To configure an app to respect browser accept headers, set the RespectBrowserAcceptHeader property to `true`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.RespectBrowserAcceptHeader = true;
});
```

Configure formatters

Apps that need to support extra formats can add the appropriate NuGet packages and configure support. There are separate formatters for input and output. Input formatters are used by Model Binding. Output formatters are used to format responses. For information on creating a custom formatter, see Custom Formatters.

Add XML format support

To configure XML formatters implemented using XmlSerializer, call AddXmlSerializerFormatters:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddXmlSerializerFormatters();
```

When using the preceding code, controller methods return the appropriate format based on the request's `Accept` header.

Configure `System.Text.Json`-based formatters

To configure features for the `System.Text.Json`-based formatters, use Microsoft.AspNetCore.Mvc.JsonOptions.JsonSerializerOptions. The following highlighted code configures PascalCase formatting instead of the default camelCase formatting:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.PropertyNamingPolicy = null;
    });
```

To configure output serialization options for specific actions, use JsonResult. For example:

C#

```
[HttpGet]
public IActionResult Get()
    => new JsonResult(
        _todoItemStore.GetList(),
        new JsonSerializerOptions { PropertyNamingPolicy = null });
```

Add `Newtonsoft.Json`-based JSON format support

The default JSON formatters use `System.Text.Json`. To use the `Newtonsoft.Json`-based formatters, install the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package and configure it in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddNewtonsoftJson();
```

In the preceding code, the call to `AddNewtonsoftJson` configures the following Web API, MVC, and Razor Pages features to use `Newtonsoft.Json`:

- Input and output formatters that read and write JSON
- JsonResult
- JSON Patch
- IJsonHelper

- TempData

Some features may not work well with `System.Text.Json`-based formatters and require a reference to the `Newtonsoft.Json`-based formatters. Continue using the `Newtonsoft.Json`-based formatters when the app:

- Uses `Newtonsoft.Json` attributes. For example, `[JsonProperty]` or `[JsonIgnore]`.
- Customizes the serialization settings.
- Relies on features that `Newtonsoft.Json` provides.

To configure features for the `Newtonsoft.Json`-based formatters, use SerializerSettings:

C#

```
builder.Services.AddControllers()
    .AddNewtonsoftJson(options =>
    {
        options.SerializerSettings.ContractResolver = new
DefaultContractResolver();
    });
```

To configure output serialization options for specific actions, use `JsonResult`. For example:

C#

```
[HttpGet]
public IActionResult GetNewtonsoftJson()
    => new JsonResult(
        _todoItemStore.GetList(),
        new JsonSerializerSettings { ContractResolver = new
DefaultContractResolver() });
```

Format `ProblemDetails` and `ValidationProblemDetails` responses

The following action method calls ControllerBase.Problem to create a ProblemDetails response:

C#

```
[HttpGet("Error")]
public IActionResult GetError()
    => Problem("Something went wrong.");
```

A `ProblemDetails` response is always camelCase, even when the app sets the format to PascalCase. `ProblemDetails` follows RFC 7807, which specifies lowercase.

When the [ApiController] attribute is applied to a controller class, the controller creates a ValidationProblemDetails response when Model Validation fails. This response includes a dictionary that uses the model's property names as error keys, unchanged. For example, the following model includes a single property that requires validation:

C#

```
public class SampleModel
{
    [Range(1, 10)]
    public int Value { get; set; }
}
```

By default, the `ValidationProblemDetails` response returned when the `Value` property is invalid uses an error key of `Value`, as shown in the following example:

C#

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "00-00000000000000000000000000000000-0000000000000000-00",
  "errors": {
    "Value": [
      "The field Value must be between 1 and 10."
    ]
  }
}
```

To format the property names used as error keys, add an implementation of IMetadataDetailsProvider to the MvcOptions.ModelMetadataDetailsProviders collection. The following example adds a `System.Text.Json`-based implementation, `SystemTextJsonValidationMetadataProvider`, which formats property names as camelCase by default:

C#

```
builder.Services.AddControllers();

builder.Services.Configure<MvcOptions>(options =>
{
    options.ModelMetadataDetailsProviders.Add(
```

```
        new SystemTextJsonValidationMetadataProvider());
});
```

`SystemTextJsonValidationMetadataProvider` also accepts an implementation of JsonNamingPolicy in its constructor, which specifies a custom naming policy for formatting property names.

To set a custom name for a property within a model, use the [JsonPropertyName] attribute on the property:

C#

```
public class SampleModel
{
    [Range(1, 10)]
    [JsonPropertyName("sampleValue")]
    public int Value { get; set; }
}
```

The `ValidationProblemDetails` response returned for the preceding model when the `Value` property is invalid uses an error key of `sampleValue`, as shown in the following example:

C#

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "traceId": "00-00000000000000000000000000000000-0000000000000000-00",
  "errors": {
    "sampleValue": [
      "The field Value must be between 1 and 10."
    ]
  }
}
```

To format the `ValidationProblemDetails` response using `Newtonsoft.Json`, use `NewtonsoftJsonValidationMetadataProvider`:

C#

```
builder.Services.AddControllers()
    .AddNewtonsoftJson();

builder.Services.Configure<MvcOptions>(options =>
{
```

```
    options.ModelMetadataDetailsProviders.Add(
        new NewtonsoftJsonValidationMetadataProvider());
});
```

By default, `NewtonsoftJsonValidationMetadataProvider` formats property names as camelCase. `NewtonsoftJsonValidationMetadataProvider` also accepts an implementation of `NamingPolicy` in its constructor, which specifies a custom naming policy for formatting property names. To set a custom name for a property within a model, use the `[JsonProperty]` attribute.

Specify a format

To restrict the response formats, apply the [Produces] filter. Like most Filters, `[Produces]` can be applied at the action, controller, or global scope:

C#

```
[ApiController]
[Route("api/[controller]")]
[Produces("application/json")]
public class TodoItemsController : ControllerBase
```

The preceding [Produces] filter:

- Forces all actions within the controller to return JSON-formatted responses for POCOs (Plain Old CLR Objects) or ObjectResult and its derived types.
- Return JSON-formatted responses even if other formatters are configured and the client specifies a different format.

For more information, see Filters.

Special case formatters

Some special cases are implemented using built-in formatters. By default, `string` return types are formatted as *text/plain* (*text/html* if requested via the `Accept` header). This behavior can be deleted by removing the StringOutputFormatter. Formatters are removed in `Program.cs`. Actions that have a model object return type return `204 No Content` when returning `null`. This behavior can be deleted by removing the HttpNoContentOutputFormatter. The following code removes the `StringOutputFormatter` and `HttpNoContentOutputFormatter`.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    // using Microsoft.AspNetCore.Mvc.Formatters;
    options.OutputFormatters.RemoveType<StringOutputFormatter>();
    options.OutputFormatters.RemoveType<HttpNoContentOutputFormatter>();
});
```

Without the `StringOutputFormatter`, the built-in JSON formatter formats `string` return types. If the built-in JSON formatter is removed and an XML formatter is available, the XML formatter formats `string` return types. Otherwise, `string` return types return `406 Not Acceptable`.

Without the `HttpNoContentOutputFormatter`, null objects are formatted using the configured formatter. For example:

- The JSON formatter returns a response with a body of `null`.
- The XML formatter returns an empty XML element with the attribute `xsi:nil="true"` set.

Response format URL mappings

Clients can request a particular format as part of the URL, for example:

- In the query string or part of the path.
- By using a format-specific file extension such as .xml or .json.

The mapping from request path should be specified in the route the API is using. For example:

C#

```
[ApiController]
[Route("api/[controller]")]
[FormatFilter]
public class TodoItemsController : ControllerBase
{
    private readonly TodoItemStore _todoItemStore;

    public TodoItemsController(TodoItemStore todoItemStore)
        => _todoItemStore = todoItemStore;

    [HttpGet("{id:long}.{format?}")]
```

```
    public TodoItem? GetById(long id)
        => _todoItemStore.GetById(id);
```

The preceding route allows the requested format to be specified using an optional file extension. The [FormatFilter] attribute checks for the existence of the format value in the `RouteData` and maps the response format to the appropriate formatter when the response is created.

Expand table

Route	Formatter
`/api/todoitems/5`	The default output formatter
`/api/todoitems/5.json`	The JSON formatter (if configured)
`/api/todoitems/5.xml`	The XML formatter (if configured)

Polymorphic deserialization

Built-in features provide a limited range of polymorphic serialization but no support for deserialization at all. Deserialization requires a custom converter. See Polymorphic deserialization for a complete sample of polymorphic deserialization.

Additional resources

- View or download sample code (how to download)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Custom formatters in ASP.NET Core Web API

Article • 04/11/2023

ASP.NET Core MVC supports data exchange in Web APIs using input and output formatters. Input formatters are used by Model Binding. Output formatters are used to format responses.

The framework provides built-in input and output formatters for JSON and XML. It provides a built-in output formatter for plain text, but doesn't provide an input formatter for plain text.

This article shows how to add support for additional formats by creating custom formatters. For an example of a custom plain text input formatter, see TextPlainInputFormatter on GitHub.

View or download sample code (how to download)

When to use a custom formatter

Use a custom formatter to add support for a content type that isn't handled by the built-in formatters.

Overview of how to create a custom formatter

To create a custom formatter:

- For serializing data sent to the client, create an output formatter class.
- For deserializing data received from the client, create an input formatter class.
- Add instances of formatter classes to the InputFormatters and OutputFormatters collections in MvcOptions.

Create a custom formatter

To create a formatter:

- Derive the class from the appropriate base class. The sample app derives from TextOutputFormatter and TextInputFormatter.
- Specify supported media types and encodings in the constructor.
- Override the CanReadType and CanWriteType methods.

- Override the ReadRequestBodyAsync and WriteResponseBodyAsync methods.

The following code shows the `VcardOutputFormatter` class from the sample:

C#

```
public class VcardOutputFormatter : TextOutputFormatter
{
    public VcardOutputFormatter()
    {
        SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

        SupportedEncodings.Add(Encoding.UTF8);
        SupportedEncodings.Add(Encoding.Unicode);
    }

    protected override bool CanWriteType(Type? type)
        => typeof(Contact).IsAssignableFrom(type)
            || typeof(IEnumerable<Contact>).IsAssignableFrom(type);

    public override async Task WriteResponseBodyAsync(
        OutputFormatterWriteContext context, Encoding selectedEncoding)
    {
        var httpContext = context.HttpContext;
        var serviceProvider = httpContext.RequestServices;

        var logger =
serviceProvider.GetRequiredService<ILogger<VcardOutputFormatter>>();
        var buffer = new StringBuilder();

        if (context.Object is IEnumerable<Contact> contacts)
        {
            foreach (var contact in contacts)
            {
                FormatVcard(buffer, contact, logger);
            }
        }
        else
        {
            FormatVcard(buffer, (Contact)context.Object!, logger);
        }

        await httpContext.Response.WriteAsync(buffer.ToString(),
selectedEncoding);
    }

    private static void FormatVcard(
        StringBuilder buffer, Contact contact, ILogger logger)
    {
        buffer.AppendLine("BEGIN:VCARD");
        buffer.AppendLine("VERSION:2.1");
        buffer.AppendLine($"N:{contact.LastName};{contact.FirstName}");
        buffer.AppendLine($"FN:{contact.FirstName} {contact.LastName}");
        buffer.AppendLine($"UID:{contact.Id}");
```

```
            buffer.AppendLine("END:VCARD");

            logger.LogInformation("Writing {FirstName} {LastName}",
                contact.FirstName, contact.LastName);
        }
    }
```

Derive from the appropriate base class

For text media types (for example, vCard), derive from the TextInputFormatter or TextOutputFormatter base class:

C#

```
public class VcardOutputFormatter : TextOutputFormatter
```

For binary types, derive from the InputFormatter or OutputFormatter base class.

Specify supported media types and encodings

In the constructor, specify supported media types and encodings by adding to the SupportedMediaTypes and SupportedEncodings collections:

C#

```
public VcardOutputFormatter()
{
    SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

    SupportedEncodings.Add(Encoding.UTF8);
    SupportedEncodings.Add(Encoding.Unicode);
}
```

A formatter class can **not** use constructor injection for its dependencies. For example, `ILogger<VcardOutputFormatter>` can't be added as a parameter to the constructor. To access services, use the context object that gets passed in to the methods. A code example in this article and the sample show how to do this.

Override CanReadType and CanWriteType

Specify the type to deserialize into or serialize from by overriding the CanReadType or CanWriteType methods. For example, to create vCard text from a `Contact` type and vice versa:

C#

```
protected override bool CanWriteType(Type? type)
    => typeof(Contact).IsAssignableFrom(type)
        || typeof(IEnumerable<Contact>).IsAssignableFrom(type);
```

The CanWriteResult method

In some scenarios, CanWriteResult must be overridden rather than CanWriteType. Use `CanWriteResult` if the following conditions are true:

- The action method returns a model class.
- There are derived classes that might be returned at runtime.
- The derived class returned by the action must be known at runtime.

For example, suppose the action method:

- Signature returns a `Person` type.
- Can return a `Student` or `Instructor` type that derives from `Person`.

For the formatter to handle only `Student` objects, check the type of Object in the context object provided to the `CanWriteResult` method. When the action method returns IActionResult:

- It's not necessary to use `CanWriteResult`.
- The `CanWriteType` method receives the runtime type.

Override ReadRequestBodyAsync and WriteResponseBodyAsync

Deserialization or serialization is performed in ReadRequestBodyAsync or WriteResponseBodyAsync. The following example shows how to get services from the dependency injection container. Services can't be obtained from constructor parameters:

C#

```
public override async Task WriteResponseBodyAsync(
    OutputFormatterWriteContext context, Encoding selectedEncoding)
{
    var httpContext = context.HttpContext;
    var serviceProvider = httpContext.RequestServices;

    var logger =
```

```
serviceProvider.GetRequiredService<ILogger<VcardOutputFormatter>>();
    var buffer = new StringBuilder();

    if (context.Object is IEnumerable<Contact> contacts)
    {
        foreach (var contact in contacts)
        {
            FormatVcard(buffer, contact, logger);
        }
    }
    else
    {
        FormatVcard(buffer, (Contact)context.Object!, logger);
    }

    await httpContext.Response.WriteAsync(buffer.ToString(),
selectedEncoding);
}

private static void FormatVcard(
    StringBuilder buffer, Contact contact, ILogger logger)
{
    buffer.AppendLine("BEGIN:VCARD");
    buffer.AppendLine("VERSION:2.1");
    buffer.AppendLine($"N:{contact.LastName};{contact.FirstName}");
    buffer.AppendLine($"FN:{contact.FirstName} {contact.LastName}");
    buffer.AppendLine($"UID:{contact.Id}");
    buffer.AppendLine("END:VCARD");

    logger.LogInformation("Writing {FirstName} {LastName}",
        contact.FirstName, contact.LastName);
}
```

Configure MVC to use a custom formatter

To use a custom formatter, add an instance of the formatter class to the MvcOptions.InputFormatters or MvcOptions.OutputFormatters collection:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.InputFormatters.Insert(0, new VcardInputFormatter());
    options.OutputFormatters.Insert(0, new VcardOutputFormatter());
});
```

Formatters are evaluated in the order they're inserted, where the first one takes precedence.

The complete `VcardInputFormatter` class

The following code shows the `VcardInputFormatter` class from the sample:

C#

```
public class VcardInputFormatter : TextInputFormatter
{
    public VcardInputFormatter()
    {
        SupportedMediaTypes.Add(MediaTypeHeaderValue.Parse("text/vcard"));

        SupportedEncodings.Add(Encoding.UTF8);
        SupportedEncodings.Add(Encoding.Unicode);
    }

    protected override bool CanReadType(Type type)
        => type == typeof(Contact);

    public override async Task<InputFormatterResult> ReadRequestBodyAsync(
        InputFormatterContext context, Encoding effectiveEncoding)
    {
        var httpContext = context.HttpContext;
        var serviceProvider = httpContext.RequestServices;

        var logger =
serviceProvider.GetRequiredService<ILogger<VcardInputFormatter>>();

        using var reader = new StreamReader(httpContext.Request.Body,
effectiveEncoding);
        string? nameLine = null;

        try
        {
            await ReadLineAsync("BEGIN:VCARD", reader, context, logger);
            await ReadLineAsync("VERSION:", reader, context, logger);

            nameLine = await ReadLineAsync("N:", reader, context, logger);

            var split = nameLine.Split(";".ToCharArray());
            var contact = new Contact(FirstName: split[1], LastName:
split[0].Substring(2));

            await ReadLineAsync("FN:", reader, context, logger);
            await ReadLineAsync("END:VCARD", reader, context, logger);

            logger.LogInformation("nameLine = {nameLine}", nameLine);

            return await InputFormatterResult.SuccessAsync(contact);
        }
        catch
        {
            logger.LogError("Read failed: nameLine = {nameLine}", nameLine);
```

```
            return await InputFormatterResult.FailureAsync();
        }
    }

    private static async Task<string> ReadLineAsync(
        string expectedText, StreamReader reader, InputFormatterContext
context,
        ILogger logger)
    {
        var line = await reader.ReadLineAsync();

        if (line is null || !line.StartsWith(expectedText))
        {
            var errorMessage = $"Looked for '{expectedText}' and got
'{line}'";

            context.ModelState.TryAddModelError(context.ModelName,
errorMessage);
            logger.LogError(errorMessage);

            throw new Exception(errorMessage);
        }

        return line;
    }
}
```

Test the app

Run the sample app for this article, which implements basic vCard input and output formatters. The app reads and writes vCards similar to the following format:

```
BEGIN:VCARD
VERSION:2.1
N:Davolio;Nancy
FN:Nancy Davolio
END:VCARD
```

To see vCard output, run the app and send a Get request with Accept header `text/vcard` to `https://localhost:<port>/api/contacts`.

To add a vCard to the in-memory collection of contacts:

- Send a `Post` request to `/api/contacts` with a tool like http-repl.
- Set the `Content-Type` header to `text/vcard`.
- Set `vCard` text in the body, formatted like the preceding example.

Additional resources

- Format response data in ASP.NET Core Web API
- Manage Protobuf references with dotnet-grpc

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use web API analyzers

Article • 04/11/2023

ASP.NET Core provides an MVC analyzers package intended for use with web API projects. The analyzers work with controllers annotated with ApiControllerAttribute, while building on web API conventions.

The analyzers package notifies you of any controller action that:

- Returns an undeclared status code.
- Returns an undeclared success result.
- Documents a status code that isn't returned.
- Includes an explicit model validation check.

Reference the analyzer package

The analyzers are included in the .NET Core SDK. To enable the analyzer in your project, include the `IncludeOpenAPIAnalyzers` property in the project file:

XML

```
<PropertyGroup>
 <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
</PropertyGroup>
```

Analyzers for web API conventions

OpenAPI documents contain status codes and response types that an action may return. In ASP.NET Core MVC, attributes such as ProducesResponseTypeAttribute and ProducesAttribute are used to document an action. ASP.NET Core web API documentation with Swagger / OpenAPI goes into further detail on documenting your web API.

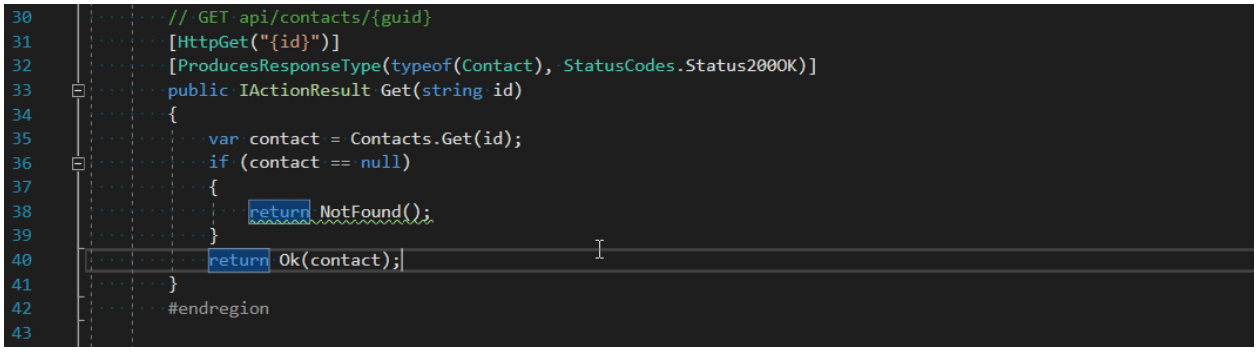
One of the analyzers in the package inspects controllers annotated with ApiControllerAttribute and identifies actions that don't entirely document their responses. Consider the following example:

C#

```
// GET api/contacts/{guid}
[HttpGet("{id}", Name = "GetById")]
[ProducesResponseType(typeof(Contact), StatusCodes.Status200OK)]
```

```
public IActionResult Get(string id)
{
    var contact = _contacts.Get(id);

    if (contact == null)
    {
        return NotFound();
    }

    return Ok(contact);
}
```

The preceding action documents the HTTP 200 success return type but doesn't document the HTTP 404 failure status code. The analyzer reports the missing documentation for the HTTP 404 status code as a warning. An option to fix the problem is provided.



Analyzers require Microsoft.NET.Sdk.Web

Analyzers don't work with library projects or projects referencing `Sdk="Microsoft.NET.Sdk"`.

Additional resources

- Use web API conventions
- ASP.NET Core web API documentation with Swagger / OpenAPI
- Create web APIs with ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Use web API conventions

Article • 04/11/2023

Common API documentation can be extracted and applied to multiple actions, controllers, or all controllers within an assembly. Web API conventions are a substitute for decorating individual actions with [ProducesResponseType].

A convention allows you to:

- Define the most common return types and status codes returned from a specific type of action.
- Identify actions that deviate from the defined standard.

Default conventions are available from Microsoft.AspNetCore.Mvc.DefaultApiConventions. The conventions are demonstrated with the `ValuesController.cs` added to an **API** project template:

C#

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace WebApp1.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ValuesController : ControllerBase
    {
        // GET api/values
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        [HttpGet("{id}")]
        public ActionResult<string> Get(int id)
        {
            return "value";
        }

        // POST api/values
        [HttpPost]
        public void Post([FromBody] string value)
        {
        }

        // PUT api/values/5
```

```
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] string value)
        {
        }

        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
}
```

Actions that follow the patterns in the `ValuesController.cs` work with the default conventions. If the default conventions don't meet your needs, see Create web API conventions.

At runtime, Microsoft.AspNetCore.Mvc.ApiExplorer understands conventions. `ApiExplorer` is MVC's abstraction to communicate with OpenAPI (also known as Swagger) document generators. Attributes from the applied convention are associated with an action and are included in the action's OpenAPI documentation. API analyzers also understand conventions. If your action is unconventional (for example, it returns a status code that isn't documented by the applied convention), a warning encourages you to document the status code.

View or download sample code (how to download)

Apply web API conventions

Conventions don't compose; each action may be associated with exactly one convention. More specific conventions take precedence over less specific conventions. The selection is non-deterministic when two or more conventions of the same priority apply to an action. The following options exist to apply a convention to an action, from the most specific to the least specific:

1. `Microsoft.AspNetCore.Mvc.ApiConventionMethodAttribute` — Applies to individual actions and specifies the convention type and the convention method that applies.

   In the following example, the default convention type's `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` convention method is applied to the `Update` action:

   ```
   C#
   ```

```
// PUT api/contactsconvention/{guid}
[HttpPut("{id}")]
[ApiConventionMethod(typeof(DefaultApiConventions),
                     nameof(DefaultApiConventions.Put))]
public IActionResult Update(string id, Contact contact)
{
    var contactToUpdate = _contacts.Get(id);

    if (contactToUpdate == null)
    {
        return NotFound();
    }

    _contacts.Update(contact);

    return NoContent();
}
```

The `Microsoft.AspNetCore.Mvc.DefaultApiConventions.Put` convention method applies the following attributes to the action:

C#

```
[ProducesDefaultResponseType]
[ProducesResponseType(StatusCodes.Status204NoContent)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
```

For more information on `[ProducesDefaultResponseType]`, see Default Response.

2. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` applied to a controller — Applies the specified convention type to all actions on the controller. A convention method is marked with hints that determine the actions to which the convention method applies. For more information on hints, see Create web API conventions).

   In the following example, the default set of conventions is applied to all actions in *ContactsConventionController*:

   C#

   ```
   [ApiController]
   [ApiConventionType(typeof(DefaultApiConventions))]
   [Route("api/[controller]")]
   public class ContactsConventionController : ControllerBase
   {
   ```

3. `Microsoft.AspNetCore.Mvc.ApiConventionTypeAttribute` applied to an assembly — Applies the specified convention type to all controllers in the current assembly. As a recommendation, apply assembly-level attributes in the `Startup.cs` file.

   In the following example, the default set of conventions is applied to all controllers in the assembly:

   ```
   C#

   [assembly: ApiConventionType(typeof(DefaultApiConventions))]
   namespace ApiConventions
   {
       public class Startup
       {
   ```

Create web API conventions

If the default API conventions don't meet your needs, create your own conventions. A convention is:

- A static type with methods.
- Capable of defining response types and naming requirements on actions.

Response types

These methods are annotated with `[ProducesResponseType]` or `[ProducesDefaultResponseType]` attributes. For example:

```
C#

public static class MyAppConventions
{
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status404NotFound)]
    public static void Find(int id)
    {
    }
}
```

If more specific metadata attributes are absent, applying this convention to an assembly enforces that:

- The convention method applies to any action named `Find`.
- A parameter named `id` is present on the `Find` action.

Naming requirements

The `[ApiConventionNameMatch]` and `[ApiConventionTypeMatch]` attributes can be applied to the convention method that determines the actions to which they apply. For example:

C#

```
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ApiConventionNameMatch(ApiConventionNameMatchBehavior.Prefix)]
public static void Find(
    [ApiConventionNameMatch(ApiConventionNameMatchBehavior.Suffix)]
    int id)
{ }
```

In the preceding example:

- The `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Prefix` option applied to the method indicates that the convention matches any action prefixed with "Find". Examples of matching actions include `Find`, `FindPet`, and `FindById`.
- The `Microsoft.AspNetCore.Mvc.ApiExplorer.ApiConventionNameMatchBehavior.Suffix` applied to the parameter indicates that the convention matches methods with exactly one parameter ending in the suffix identifier. Examples include parameters such as `id` or `petId`. `ApiConventionTypeMatch` can be similarly applied to types to constrain the parameter type. A `params[]` argument indicates remaining parameters that don't need to be explicitly matched.

Additional resources

- Video: Create metadata for NSwagClient
- Video: Beginner's Series to: Web APIs
- Use web API analyzers
- ASP.NET Core web API documentation with Swagger / OpenAPI

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

Handle errors in ASP.NET Core controller-based web APIs

Article • 06/01/2024

This article describes how to handle errors and customize error handling in controller-based ASP.NET Core web APIs. For information about error handling in minimal APIs, see Handle errors in ASP.NET Core and Handle errors in minimal APIs.

Developer Exception Page

The *Developer Exception Page* displays detailed information about unhandled request exceptions. It uses DeveloperExceptionPageMiddleware to capture synchronous and asynchronous exceptions from the HTTP pipeline and to generate error responses. The developer exception page runs early in the middleware pipeline, so that it can catch unhandled exceptions thrown in middleware that follows.

ASP.NET Core apps enable the developer exception page by default when both:

- Running in the Development environment.
- The app was created with the current templates, that is, by using WebApplication.CreateBuilder.

Apps created using earlier templates, that is, by using WebHost.CreateDefaultBuilder, can enable the developer exception page by calling app.UseDeveloperExceptionPage.

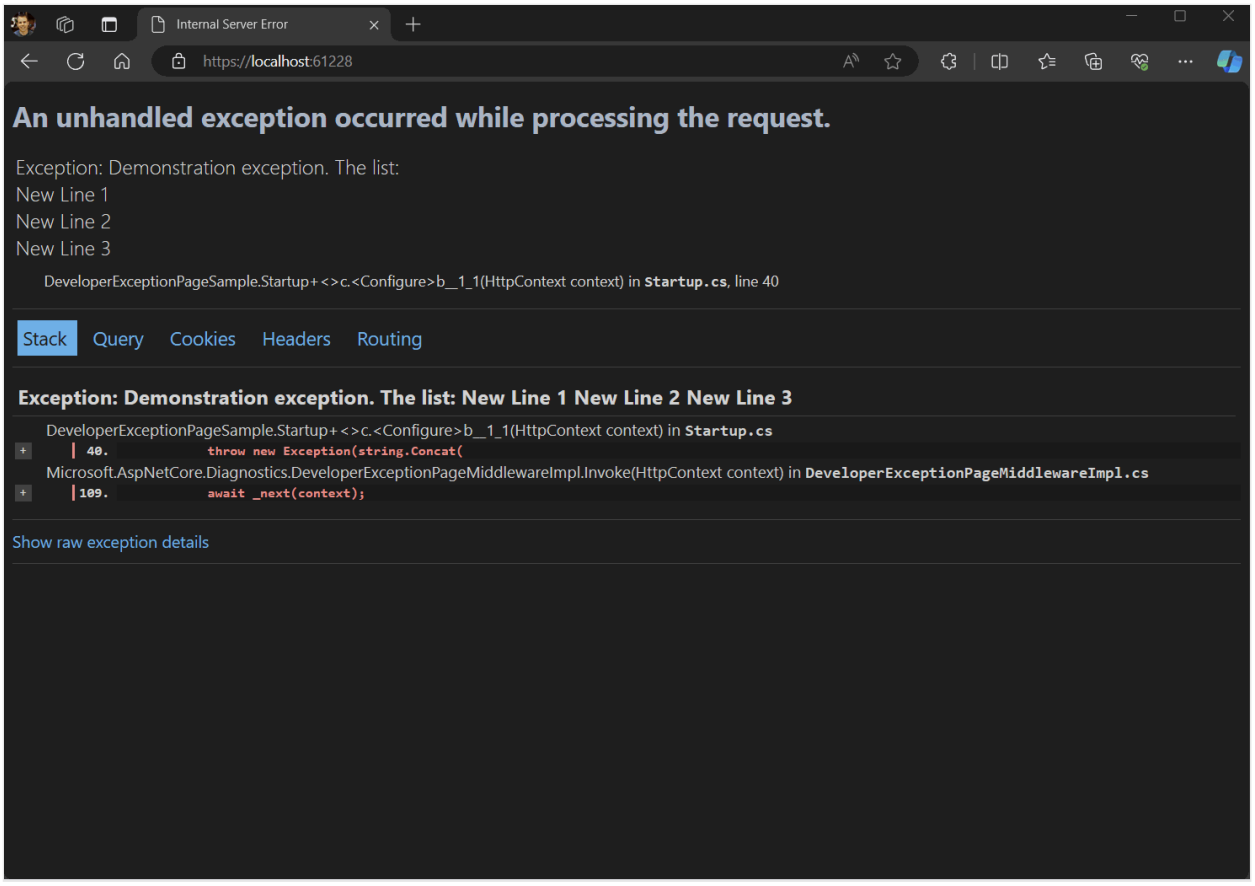
> ⚠ **Warning**
>
> Don't enable the Developer Exception Page **unless the app is running in the Development environment**. Don't share detailed exception information publicly when the app runs in production. For more information on configuring environments, see **Use multiple environments in ASP.NET Core**.

The Developer Exception Page can include the following information about the exception and the request:

- Stack trace
- Query string parameters, if any
- Cookies, if any
- Headers
- Endpoint metadata, if any

The Developer Exception Page isn't guaranteed to provide any information. Use Logging for complete error information.

The following image shows a sample developer exception page with animation to show the tabs and the information displayed:



In response to a request with an `Accept: text/plain` header, the Developer Exception Page returns plain text instead of HTML. For example:

text

```
Status: 500 Internal Server Error
Time: 9.39 msSize: 480 bytes
FormattedRawHeadersRequest
Body
text/plain; charset=utf-8, 480 bytes
System.InvalidOperationException: Sample Exception
   at WebApplicationMinimal.Program.<>c.<Main>b__0_0() in
C:\Source\WebApplicationMinimal\Program.cs:line 12
   at lambda_method1(Closure, Object, HttpContext)
   at
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddlewareImpl.Invoke
(HttpContext context)

HEADERS
=======
Accept: text/plain
```

```
Host: localhost:7267
traceparent: 00-0eab195ea19d07b90a46cd7d6bf2f
```

To see the Developer Exception Page:

- Add the following controller action to a controller-based API. The action throws an exception when the endpoint is requested.

  C#

  ```
  [HttpGet("Throw")]
  public IActionResult Throw() =>
      throw new Exception("Sample exception.");
  ```

- Run the app in the development environment.
- Go to the endpoint defined by the controller action.

Exception handler

In non-development environments, use Exception Handling Middleware to produce an error payload:

1. In `Program.cs`, call UseExceptionHandler to add the Exception Handling Middleware:

   C#

   ```
   var app = builder.Build();

   app.UseHttpsRedirection();

   if (!app.Environment.IsDevelopment())
   {
       app.UseExceptionHandler("/error");
   }

   app.UseAuthorization();

   app.MapControllers();

   app.Run();
   ```

2. Configure a controller action to respond to the `/error` route:

   C#

```
[Route("/error")]
public IActionResult HandleError() =>
    Problem();
```

The preceding `HandleError` action sends an RFC 7807 -compliant payload to the client.

> ⚠ **Warning**
>
> Don't mark the error handler action method with HTTP method attributes, such as `HttpGet`. Explicit verbs prevent some requests from reaching the action method.
>
> For web APIs that use Swagger / OpenAPI, mark the error handler action with the [ApiExplorerSettings] attribute and set its IgnoreApi property to `true`. This attribute configuration excludes the error handler action from the app's OpenAPI specification:
>
> C#
>
> ```
> [ApiExplorerSettings(IgnoreApi = true)]
> ```
>
> Allow anonymous access to the method if unauthenticated users should see the error.

Exception Handling Middleware can also be used in the Development environment to produce a consistent payload format across all environments:

1. In `Program.cs`, register environment-specific Exception Handling Middleware instances:

   C#

   ```
   if (app.Environment.IsDevelopment())
   {
       app.UseExceptionHandler("/error-development");
   }
   else
   {
       app.UseExceptionHandler("/error");
   }
   ```

   In the preceding code, the middleware is registered with:

   - A route of `/error-development` in the Development environment.

- A route of `/error` in non-Development environments.

2. Add controller actions for both the Development and non-Development routes:

```C#
[Route("/error-development")]
public IActionResult HandleErrorDevelopment(
    [FromServices] IHostEnvironment hostEnvironment)
{
    if (!hostEnvironment.IsDevelopment())
    {
        return NotFound();
    }

    var exceptionHandlerFeature =
        HttpContext.Features.Get<IExceptionHandlerFeature>()!;

    return Problem(
        detail: exceptionHandlerFeature.Error.StackTrace,
        title: exceptionHandlerFeature.Error.Message);
}

[Route("/error")]
public IActionResult HandleError() =>
    Problem();
```

Use exceptions to modify the response

The contents of the response can be modified from outside of the controller using a custom exception and an action filter:

1. Create a well-known exception type named `HttpResponseException`:

```C#
public class HttpResponseException : Exception
{
    public HttpResponseException(int statusCode, object? value = null)
=>
        (StatusCode, Value) = (statusCode, value);

    public int StatusCode { get; }

    public object? Value { get; }
}
```

2. Create an action filter named `HttpResponseExceptionFilter`:

C#

```
public class HttpResponseExceptionFilter : IActionFilter,
IOrderedFilter
{
    public int Order => int.MaxValue - 10;

    public void OnActionExecuting(ActionExecutingContext context) { }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        if (context.Exception is HttpResponseException
httpResponseException)
        {
            context.Result = new
ObjectResult(httpResponseException.Value)
            {
                StatusCode = httpResponseException.StatusCode
            };

            context.ExceptionHandled = true;
        }
    }
}
```

The preceding filter specifies an `Order` of the maximum integer value minus 10. This `Order` allows other filters to run at the end of the pipeline.

3. In `Program.cs`, add the action filter to the filters collection:

C#

```
builder.Services.AddControllers(options =>
{
    options.Filters.Add<HttpResponseExceptionFilter>();
});
```

Validation failure error response

For web API controllers, MVC responds with a ValidationProblemDetails response type when model validation fails. MVC uses the results of InvalidModelStateResponseFactory to construct the error response for a validation failure. The following example replaces the default factory with an implementation that also supports formatting responses as XML, in `Program.cs`:

C#

```
builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
            new BadRequestObjectResult(context.ModelState)
            {
                ContentTypes =
                {
                    // using static System.Net.Mime.MediaTypeNames;
                    Application.Json,
                    Application.Xml
                }
            };
    })
    .AddXmlSerializerFormatters();
```

Client error response

An *error result* is defined as a result with an HTTP status code of 400 or higher. For web API controllers, MVC transforms an error result to produce a ProblemDetails.

The automatic creation of a `ProblemDetails` for error status codes is enabled by default, but error responses can be configured in one of the following ways:

1. Use the problem details service
2. Implement ProblemDetailsFactory
3. Use ApiBehaviorOptions.ClientErrorMapping

Default problem details response

The following `Program.cs` file was generated by the web application templates for API controllers:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();
```

```
app.Run();
```

Consider the following controller, which returns BadRequest when the input is invalid:

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class Values2Controller : ControllerBase
{
    // /api/values2/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            return BadRequest();
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values2 /squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
        {
            return BadRequest();
        }

        return Ok(Math.Sqrt(radicand));
    }
}
```

A problem details response is generated with the preceding code when any of the following conditions apply:

- The `/api/values2/divide` endpoint is called with a zero denominator.
- The `/api/values2/squareroot` endpoint is called with a radicand less than zero.

The default problem details response body has the following `type`, `title`, and `status` values:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
  "title": "Bad Request",
```

```
  "status": 400,
  "traceId": "00-84c1fd4063c38d9f3900d06e56542d48-85d1d4-00"
}
```

Problem details service

ASP.NET Core supports creating Problem Details for HTTP APIs using the IProblemDetailsService. For more information, see the Problem details service.

The following code configures the app to generate a problem details response for all HTTP client and server error responses that ***don't have body content yet***:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();

if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}

app.MapControllers();
app.Run();
```

Consider the API controller from the preceding section, which returns BadRequest when the input is invalid:

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class Values2Controller : ControllerBase
{
    // /api/values2/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            return BadRequest();
        }
```

```
        return Ok(Numerator / Denominator);
    }

    // /api/values2 /squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
        {
            return BadRequest();
        }

        return Ok(Math.Sqrt(radicand));
    }
}
```

A problem details response is generated with the preceding code when any of the following conditions apply:

- An invalid input is supplied.
- The URI has no matching endpoint.
- An unhandled exception occurs.

The automatic creation of a `ProblemDetails` for error status codes is disabled when the SuppressMapClientErrors property is set to `true`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.SuppressMapClientErrors = true;
    });

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Using the preceding code, when an API controller returns `BadRequest`, an HTTP 400 response status is returned with no response body. `SuppressMapClientErrors` prevents a

`ProblemDetails` response from being created, even when calling `WriteAsync` for an API Controller endpoint. `WriteAsync` is explained later in this article.

The next section shows how to customize the problem details response body, using CustomizeProblemDetails, to return a more helpful response. For more customization options, see Customizing problem details.

Customize problem details with `CustomizeProblemDetails`

The following code uses ProblemDetailsOptions to set CustomizeProblemDetails:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddProblemDetails(options =>
        options.CustomizeProblemDetails = (context) =>
        {

            var mathErrorFeature = context.HttpContext.Features

.Get<MathErrorFeature>();
            if (mathErrorFeature is not null)
            {
                (string Detail, string Type) details =
mathErrorFeature.MathError switch
                {
                    MathErrorType.DivisionByZeroError =>
                    ("Divison by zero is not defined.",

"https://wikipedia.org/wiki/Division_by_zero"),
                    _ => ("Negative or complex numbers are not valid
input.",

"https://wikipedia.org/wiki/Square_root")
                };

                context.ProblemDetails.Type = details.Type;
                context.ProblemDetails.Title = "Bad Input";
                context.ProblemDetails.Detail = details.Detail;
            }
        }
    );

var app = builder.Build();

app.UseHttpsRedirection();

app.UseStatusCodePages();
```

```
app.UseAuthorization();

app.MapControllers();

app.Run();
```

The updated API controller:

C#

```
[Route("api/[controller]/[action]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // /api/values/divide/1/2
    [HttpGet("{Numerator}/{Denominator}")]
    public IActionResult Divide(double Numerator, double Denominator)
    {
        if (Denominator == 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.DivisionByZeroError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }

        return Ok(Numerator / Denominator);
    }

    // /api/values/squareroot/4
    [HttpGet("{radicand}")]
    public IActionResult Squareroot(double radicand)
    {
        if (radicand < 0)
        {
            var errorType = new MathErrorFeature
            {
                MathError = MathErrorType.NegativeRadicandError
            };
            HttpContext.Features.Set(errorType);
            return BadRequest();
        }
        return Ok(Math.Sqrt(radicand));
    }

}
```

The following code contains the `MathErrorFeature` and `MathErrorType`, which are used with the preceding sample:

C#

```
// Custom Http Request Feature
class MathErrorFeature
{
    public MathErrorType MathError { get; set; }
}

// Custom math errors
enum MathErrorType
{
    DivisionByZeroError,
    NegativeRadicandError
}
```

A problem details response is generated with the preceding code when any of the following conditions apply:

- The `/divide` endpoint is called with a zero denominator.
- The `/squareroot` endpoint is called with a radicand less than zero.
- The URI has no matching endpoint.

The problem details response body contains the following when either `squareroot` endpoint is called with a radicand less than zero:

JSON

```
{
  "type": "https://en.wikipedia.org/wiki/Square_root",
  "title": "Bad Input",
  "status": 400,
  "detail": "Negative or complex numbers are not allowed."
}
```

View or download sample code

Implement `ProblemDetailsFactory`

MVC uses Microsoft.AspNetCore.Mvc.Infrastructure.ProblemDetailsFactory to produce all instances of ProblemDetails and ValidationProblemDetails. This factory is used for:

- Client error responses
- Validation failure error responses
- ControllerBase.Problem and ControllerBase.ValidationProblem

To customize the problem details response, register a custom implementation of ProblemDetailsFactory in `Program.cs`:

```
C#

builder.Services.AddControllers();
builder.Services.AddTransient<ProblemDetailsFactory,
SampleProblemDetailsFactory>();
```

Use `ApiBehaviorOptions.ClientErrorMapping`

Use the ClientErrorMapping property to configure the contents of the `ProblemDetails` response. For example, the following code in `Program.cs` updates the Link property for 404 responses:

```
C#

builder.Services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.ClientErrorMapping[StatusCodes.Status404NotFound].Link =
            "https://httpstatuses.com/404";
    });
```

Additional resources

- How to Use ModelState Validation in ASP.NET Core Web API
- View or download sample code
- Hellang.Middleware.ProblemDetails

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Test web APIs with the HttpRepl

Article • 07/28/2023

The HTTP Read-Eval-Print Loop (REPL) is:

- A lightweight, cross-platform command-line tool that's supported everywhere .NET Core is supported.
- Used for making HTTP requests to test ASP.NET Core web APIs (and non-ASP.NET Core web APIs) and view their results.
- Capable of testing web APIs hosted in any environment, including localhost and Azure App Service.

The following HTTP verbs are supported:

- DELETE
- GET
- HEAD
- OPTIONS
- PATCH
- POST
- PUT

To follow along, view or download the sample ASP.NET Core web API (how to download).

Prerequisites

- .NET Core 3.1 SDK

Installation

To install the HttpRepl, run the following command:

.NET CLI

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

A .NET Core Global Tool is installed from the Microsoft.dotnet-httprepl NuGet package.

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see **dotnet tool install, --arch option**. For more information, see GitHub issue **dotnet/AspNetCore.Docs #29262**.

On macOS, update the path:

```Bash
export PATH="$HOME/.dotnet/tools:$PATH"
```

Usage

After successful installation of the tool, run the following command to start the HttpRepl:

```Console
httprepl
```

To view the available HttpRepl commands, run one of the following commands:

```Console
httprepl -h
```

```Console
httprepl --help
```

The following output is displayed:

```Console
Usage:
  httprepl [<BASE_ADDRESS>] [options]

Arguments:
  <BASE_ADDRESS> - The initial base address for the REPL.

Options:
```

```
  -h|--help - Show help information.

Once the REPL starts, these commands are valid:

Setup Commands:
Use these commands to configure the tool for your API server

connect        Configures the directory structure and base address of the
api server
set header     Sets or clears a header for all requests. e.g. `set header
content-type application/json`

HTTP Commands:
Use these commands to execute requests against your application.

GET            get - Issues a GET request
POST           post - Issues a POST request
PUT            put - Issues a PUT request
DELETE         delete - Issues a DELETE request
PATCH          patch - Issues a PATCH request
HEAD           head - Issues a HEAD request
OPTIONS        options - Issues a OPTIONS request

Navigation Commands:
The REPL allows you to navigate your URL space and focus on specific APIs
that you are working on.

ls             Show all endpoints for the current path
cd             Append the given directory to the currently selected path, or
move up a path when using `cd ..`

Shell Commands:
Use these commands to interact with the REPL shell.

clear          Removes all text from the shell
echo [on/off]  Turns request echoing on or off, show the request that was
made when using request commands
exit           Exit the shell

REPL Customization Commands:
Use these commands to customize the REPL behavior.

pref [get/set] Allows viewing or changing preferences, e.g. 'pref set
editor.command.default 'C:\\Program Files\\Microsoft VS Code\\Code.exe'`
run            Runs the script at the given path. A script is a set of
commands that can be typed with one command per line
ui             Displays the Swagger UI page, if available, in the default
browser

Use `help <COMMAND>` for more detail on an individual command. e.g. `help
get`.
For detailed tool info, see https://aka.ms/http-repl-doc.
```

The HttpRepl offers command completion. Pressing the `Tab` key iterates through the list of commands that complete the characters or API endpoint that you typed. The following sections outline the available CLI commands.

Connect to the web API

Connect to a web API by running the following command:

```
Console

httprepl <ROOT URI>
```

`<ROOT URI>` is the base URI for the web API. For example:

```
Console

httprepl https://localhost:5001
```

Alternatively, run the following command at any time while the HttpRepl is running:

```
Console

connect <ROOT URI>
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001
```

Manually point to the OpenAPI description for the web API

The connect command above will attempt to find the OpenAPI description automatically. If for some reason it's unable to do so, you can specify the URI of the OpenAPI description for the web API by using the `--openapi` option:

```
Console

connect <ROOT URI> --openapi <OPENAPI DESCRIPTION ADDRESS>
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001 --openapi
/swagger/v1/swagger.json
```

Enable verbose output for details on OpenAPI description searching, parsing, and validation

Specifying the `--verbose` option with the `connect` command will produce more details when the tool searches for the OpenAPI description, parses, and validates it.

```
Console

connect <ROOT URI> --verbose
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001 --verbose
Checking https://localhost:5001/swagger.json... 404 NotFound
Checking https://localhost:5001/swagger/v1/swagger.json... 404 NotFound
Checking https://localhost:5001/openapi.json... Found
Parsing... Successful (with warnings)
The field 'info' in 'document' object is REQUIRED [#/info]
The field 'paths' in 'document' object is REQUIRED [#/paths]
```

Navigate the web API

View available endpoints

To list the different endpoints (controllers) at the current path of the web API address, run the `ls` or `dir` command:

```
Console

https://localhost:5001/> ls
```

The following output format is displayed:

Console

```
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/>
```

The preceding output indicates that there are two controllers available: `Fruits` and `People`. Both controllers support parameterless HTTP GET and POST operations.

Navigating into a specific controller reveals more detail. For example, the following command's output shows the `Fruits` controller also supports HTTP GET, PUT, and DELETE operations. Each of these operations expects an `id` parameter in the route:

Console

```
https://localhost:5001/fruits> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/fruits>
```

Alternatively, run the `ui` command to open the web API's Swagger UI page in a browser. For example:

Console

```
https://localhost:5001/> ui
```

Navigate to an endpoint

To navigate to a different endpoint on the web API, run the `cd` command:

Console

```
https://localhost:5001/> cd people
```

The path following the `cd` command is case insensitive. The following output format is displayed:

Console

```
/people    [get|post]

https://localhost:5001/people>
```

Customize the HttpRepl

The HttpRepl's default colors can be customized. Additionally, a default text editor can be defined. The HttpRepl preferences are persisted across the current session and are honored in future sessions. Once modified, the preferences are stored in the following file:

Windows

%USERPROFILE%\.httpreplprefs

The *.httpreplprefs* file is loaded on startup and not monitored for changes at runtime. Manual modifications to the file take effect only after restarting the tool.

View the settings

To view the available settings, run the `pref get` command. For example:

Console

```
https://localhost:5001/> pref get
```

The preceding command displays the available key-value pairs:

Console

```
colors.json=Green
colors.json.arrayBrace=BoldCyan
colors.json.comma=BoldYellow
colors.json.name=BoldMagenta
colors.json.nameSeparator=BoldWhite
colors.json.objectBrace=Cyan
colors.protocol=BoldGreen
colors.status=BoldYellow
```

Set color preferences

Response colorization is currently supported for JSON only. To customize the default HttpRepl tool coloring, locate the key corresponding to the color to be changed. For instructions on how to find the keys, see the View the settings section. For example, change the `colors.json` key value from `Green` to `White` as follows:

Console

```
https://localhost:5001/people> pref set colors.json White
```

Only the allowed colors may be used. Subsequent HTTP requests display output with the new coloring.

When specific color keys aren't set, more generic keys are considered. To demonstrate this fallback behavior, consider the following example:

- If `colors.json.name` doesn't have a value, `colors.json.string` is used.
- If `colors.json.string` doesn't have a value, `colors.json.literal` is used.
- If `colors.json.literal` doesn't have a value, `colors.json` is used.
- If `colors.json` doesn't have a value, the command shell's default text color (`AllowedColors.None`) is used.

Set indentation size

Response indentation size customization is currently supported for JSON only. The default size is two spaces. For example:

JSON

```
[
  {
    "id": 1,
    "name": "Apple"
  },
  {
    "id": 2,
    "name": "Orange"
  },
  {
    "id": 3,
    "name": "Strawberry"
  }
]
```

To change the default size, set the `formatting.json.indentSize` key. For example, to always use four spaces:

Console

```
pref set formatting.json.indentSize 4
```

Subsequent responses honor the setting of four spaces:

JSON

```
[
    {
        "id": 1,
        "name": "Apple"
    },
    {
        "id": 2,
        "name": "Orange"
    },
    {
        "id": 3,
        "name": "Strawberry"
    }
]
```

Set the default text editor

By default, the HttpRepl has no text editor configured for use. To test web API methods requiring an HTTP request body, a default text editor must be set. The HttpRepl tool launches the configured text editor for the sole purpose of composing the request body. Run the following command to set your preferred text editor as the default:

Console

```
pref set editor.command.default "<EXECUTABLE>"
```

In the preceding command, `<EXECUTABLE>` is the full path to the text editor's executable file. For example, run the following command to set Visual Studio Code as the default text editor:

Windows

Console

```
pref set editor.command.default "C:\Program Files\Microsoft VS
Code\Code.exe"
```

To launch the default text editor with specific CLI arguments, set the `editor.command.default.arguments` key. For example, assume Visual Studio Code is the default text editor and that you always want the HttpRepl to open Visual Studio Code in a new session with extensions disabled. Run the following command:

Console

```
pref set editor.command.default.arguments "--disable-extensions --new-
window"
```

> **Tip**
>
> If your default editor is Visual Studio Code, you'll usually want to pass the `-w` or `--wait` argument to force Visual Studio Code to wait for you to close the file before returning.

Set the OpenAPI Description search paths

By default, the HttpRepl has a set of relative paths that it uses to find the OpenAPI description when executing the `connect` command without the `--openapi` option. These relative paths are combined with the root and base paths specified in the `connect` command. The default relative paths are:

- `swagger.json`
- `swagger/v1/swagger.json`
- `/swagger.json`
- `/swagger/v1/swagger.json`
- `openapi.json`
- `/openapi.json`

To use a different set of search paths in your environment, set the `swagger.searchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.searchPaths
"swagger/v2/swagger.json|swagger/v3/swagger.json"
```

Instead of replacing the default list altogether, the list can also be modified by adding or removing paths.

To add one or more search paths to the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.addToSearchPaths
"openapi/v2/openapi.json|openapi/v3/openapi.json"
```

To remove one or more search paths from the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.removeFromSearchPaths "swagger.json|/swagger.json"
```

Test HTTP GET requests

Synopsis

Console

```
get <PARAMETER> [-F|--no-formatting] [-h|--header] [--response:body] [--
response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

The following options are available for the `get` command:

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP GET request:

1. Run the `get` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/people> get
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Fri, 21 Jun 2019 03:38:45 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "name": "Scott Hunter"
     },
     {
       "id": 2,
       "name": "Scott Hanselman"
   ```

```
  },
  {
    "id": 3,
    "name": "Scott Guthrie"
  }
]


https://localhost:5001/people>
```

2. Retrieve a specific record by passing a parameter to the `get` command:

```
Console

https://localhost:5001/people> get 2
```

The preceding command displays the following output format:

```
Console

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 21 Jun 2019 06:17:57 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 2,
    "name": "Scott Hanselman"
  }
]


https://localhost:5001/people>
```

Test HTTP POST requests

Synopsis

```
Console

post <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--
no-formatting] [--response] [--response:body] [--response:headers] [-s|--
streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

Indicates that no HTTP request body is needed.

Example

To issue an HTTP POST request:

1. Run the `post` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/people> post -h Content-Type=application/json
   ```

   In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

   JSON

   ```
   {
     "id": 0,
     "name": ""
   }
   ```

   > **Tip**
   >
   > To set the default text editor, see the **Set the default text editor** section.

2. Modify the JSON template to satisfy model validation requirements:

   JSON

   ```
   {
     "id": 0,
     "name": "Scott Addie"
   }
   ```

3. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

   Console

   ```
   HTTP/1.1 201 Created
   Content-Type: application/json; charset=utf-8
   Date: Thu, 27 Jun 2019 21:24:18 GMT
   ```

```
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Test HTTP PUT requests

Synopsis

Console

```
put <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Example

To issue an HTTP PUT request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
   ```

```
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `put` command on an endpoint that supports it:

```
Console

https://localhost:5001/fruits> put 2 -h Content-Type=application/json
```

In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

```
JSON

{
  "id": 0,
  "name": ""
}
```

> **Tip**
>
> To set the default text editor, see the **Set the default text editor** section.

3. Modify the JSON template to satisfy model validation requirements:

```
JSON

{
  "id": 2,
  "name": "Cherry"
}
```

4. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

```
Console
```

```
[main 2019-06-28T17:27:01.805Z] update#setState idle
HTTP/1.1 204 No Content
Date: Fri, 28 Jun 2019 17:28:21 GMT
Server: Kestrel
```

5. *Optional*: Issue a `get` command to see the modifications. For example, if you typed "Cherry" in the text editor, a `get` returns the following output:

Console

```
https://localhost:5001/fruits> get
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sat, 22 Jun 2019 00:08:20 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Cherry"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]


https://localhost:5001/fruits>
```

Test HTTP DELETE requests

Synopsis

Console

```
delete <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP DELETE request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked
   ```

```
[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `delete` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/fruits> delete 2
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 204 No Content
   Date: Fri, 28 Jun 2019 17:36:42 GMT
   Server: Kestrel
   ```

3. *Optional*: Issue a `get` command to see the modifications. In this example, a `get` returns the following output:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:16:30 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
     {
       "id": 3,
       "data": "Strawberry"
     }
   ```

```
]


https://localhost:5001/fruits>
```

Test HTTP PATCH requests

Synopsis

Console

```
patch <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't

exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Test HTTP HEAD requests

Synopsis

Console

```
head <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Test HTTP OPTIONS requests

Synopsis

Console

```
options <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Set HTTP request headers

To set an HTTP request header, use one of the following approaches:

- Set inline with the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> post -h Content-Type=application/json
  ```

  With the preceding approach, each distinct HTTP request header requires its own `-h` option.

- Set before sending the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> set header Content-Type application/json
  ```

  When setting the header before sending a request, the header remains set for the duration of the command shell session. To clear the header, provide an empty value. For example:

  Console

```
https://localhost:5001/people> set header Content-Type
```

Test secured endpoints

The HttpRepl supports the testing of secured endpoints in the following ways:

- Via the default credentials of the logged in user.
- Through the use of HTTP request headers.

Default credentials

Consider a web API you're testing that's hosted in IIS and secured with Windows authentication. You want the credentials of the user running the tool to flow across to the HTTP endpoints being tested. To pass the default credentials of the logged in user:

1. Set the `httpClient.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

Default proxy credentials

Consider a scenario in which the web API you're testing is behind a proxy secured with Windows authentication. You want the credentials of the user running the tool to flow to the proxy. To pass the default credentials of the logged in user:

1. Set the `httpClient.proxy.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.proxy.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

HTTP request headers

Examples of supported authentication and authorization schemes include:

- basic authentication
- JWT bearer tokens
- digest authentication

For example, you can send a bearer token to an endpoint with the following command:

Console

```
set header Authorization "bearer <TOKEN VALUE>"
```

To access an Azure-hosted endpoint or to use the Azure REST API, you need a bearer token. Use the following steps to obtain a bearer token for your Azure subscription via the Azure CLI. The HttpRepl sets the bearer token in an HTTP request header. A list of Azure App Service Web Apps is retrieved.

1. Sign in to Azure:

   Azure CLI

   ```
   az login
   ```

2. Get your subscription ID with the following command:

   Azure CLI

   ```
   az account show --query id
   ```

3. Copy your subscription ID and run the following command:

   Azure CLI

   ```
   az account set --subscription "<SUBSCRIPTION ID>"
   ```

4. Get your bearer token with the following command:

   Azure CLI

   ```
   az account get-access-token --query accessToken
   ```

5. Connect to the Azure REST API via the HttpRepl:

   Console

```
httprepl https://management.azure.com
```

6. Set the `Authorization` HTTP request header:

```
Console

https://management.azure.com/> set header Authorization "bearer <ACCESS
TOKEN>"
```

7. Navigate to the subscription:

```
Console

https://management.azure.com/> cd subscriptions/<SUBSCRIPTION ID>
```

8. Get a list of your subscription's Azure App Service Web Apps:

```
Console

https://management.azure.com/subscriptions/{SUBSCRIPTION ID}> get
providers/Microsoft.Web/sites?api-version=2016-08-01
```

The following response is displayed:

```
Console

HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Length: 35948
Content-Type: application/json; charset=utf-8
Date: Thu, 19 Sep 2019 23:04:03 GMT
Expires: -1
Pragma: no-cache
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
x-ms-correlation-request-id: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx</em>
x-ms-original-request-ids: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx;xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx</em>
x-ms-ratelimit-remaining-subscription-reads: 11999
x-ms-request-id: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
x-ms-routing-request-id: WESTUS:xxxxxxxxxxxxxxxx:xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxx
{
  "value": [
    <AZURE RESOURCES LIST>
```

```
  ]
}
```

Toggle HTTP request display

By default, display of the HTTP request being sent is suppressed. It's possible to change the corresponding setting for the duration of the command shell session.

Enable request display

View the HTTP request being sent by running the `echo on` command. For example:

Console

```
https://localhost:5001/people> echo on
Request echoing is on
```

Subsequent HTTP requests in the current session display the request headers. For example:

Console

```
https://localhost:5001/people> post

[main 2019-06-28T18:50:11.930Z] update#setState idle
Request to https://localhost:5001...

POST /people HTTP/1.1
Content-Length: 41
Content-Type: application/json
User-Agent: HTTP-REPL

{
  "id": 0,
  "name": "Scott Addie"
}

Response from https://localhost:5001...

HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Fri, 28 Jun 2019 18:50:21 GMT
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
```

```
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Disable request display

Suppress display of the HTTP request being sent by running the `echo off` command. For example:

Console

```
https://localhost:5001/people> echo off
Request echoing is off
```

Run a script

If you frequently execute the same set of HttpRepl commands, consider storing them in a text file. Commands in the file take the same form as commands executed manually on the command line. The commands can be executed in a batched fashion using the `run` command. For example:

1. Create a text file containing a set of newline-delimited commands. To illustrate, consider a *people-script.txt* file containing the following commands:

   text

   ```
   set base https://localhost:5001
   ls
   cd People
   ls
   get 1
   ```

2. Execute the `run` command, passing in the text file's path. For example:

   Console

   ```
   https://localhost:5001/> run C:\http-repl-scripts\people-script.txt
   ```

   The following output appears:

   Console

```
https://localhost:5001/> set base https://localhost:5001
Using OpenAPI description at
https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/> cd People
/People    [get|post]

https://localhost:5001/People> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/People> get 1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 12 Jul 2019 19:20:10 GMT
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "name": "Scott Hunter"
}


https://localhost:5001/People>
```

Clear the output

To remove all output written to the command shell by the HttpRepl tool, run the `clear` or `cls` command. To illustrate, imagine the command shell contains the following output:

Console

```
httprepl https://localhost:5001
(Disconnected)> set base "https://localhost:5001"
Using OpenAPI description at https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]
```

```
https://localhost:5001/>
```

Run the following command to clear the output:

Console

```
https://localhost:5001/> clear
```

After running the preceding command, the command shell contains only the following output:

Console

```
https://localhost:5001/>
```

Additional resources

- REST API requests
- HttpRepl GitHub repository
- Configure Visual Studio to launch HttpRepl
- Configure Visual Studio Code to launch HttpRepl
- Configure Visual Studio for Mac to launch HttpRepl

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

HttpRepl telemetry

Article • 04/11/2023

The HttpRepl includes a telemetry feature that collects usage data. It's important that the HttpRepl team understands how the tool is used so it can be improved.

How to opt out

The HttpRepl telemetry feature is enabled by default. To opt out of the telemetry feature, set the `DOTNET_HTTPREPL_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

Disclosure

The HttpRepl displays text similar to the following when you first run the tool. Text may vary slightly depending on the version of the tool you're running. This "first run" experience is how Microsoft notifies you about data collection.

Console

```
Telemetry
---------
The .NET tools collect usage data in order to help us improve your
experience. It is collected by Microsoft and shared with the community. You
can opt-out of telemetry by setting the DOTNET_HTTPREPL_TELEMETRY_OPTOUT
environment variable to '1' or 'true' using your favorite shell.
```

To suppress the "first run" experience text, set the `DOTNET_HTTPREPL_SKIP_FIRST_TIME_EXPERIENCE` environment variable to `1` or `true`.

Data points

The telemetry feature doesn't:

- Collect personal data, such as usernames, email addresses, or URLs.
- Scan your HTTP requests or responses.

The data is sent securely to Microsoft servers and held under restricted access.

Protecting your privacy is important to us. If you suspect the telemetry feature is collecting sensitive data or the data is being insecurely or inappropriately handled, take one of the following actions:

- File an issue in the dotnet/httprepl repository.
- Send an email to dotnet@microsoft.com for investigation.

The telemetry feature collects the following data.

Expand table

.NET SDK versions	Data
>=5.0	Timestamp of invocation.
>=5.0	Three-octet IP address used to determine the geographical location.
>=5.0	Operating system and version.
>=5.0	Runtime ID (RID) the tool is running on.
>=5.0	Whether the tool is running in a container.
>=5.0	Hashed Media Access Control (MAC) address: a cryptographically (SHA256) hashed and unique ID for a machine.
>=5.0	Kernel version.
>=5.0	HttpRepl version.
>=5.0	Whether the tool was started with `help`, `run`, or `connect` arguments. Actual argument values aren't collected.
>=5.0	Command invoked (for example, `get`) and whether it succeeded.
>=5.0	For the `connect` command, whether the `root`, `base`, or `openapi` arguments were supplied. Actual argument values aren't collected.
>=5.0	For the `pref` command, whether a `get` or `set` was issued and which preference was accessed. If not a well-known preference, the name is hashed. The value isn't collected.
>=5.0	For the `set header` command, the header name being set. If not a well-known header, the name is hashed. The value isn't collected.
>=5.0	For the `connect` command, whether a special case for `dotnet new webapi` was used and, whether it was bypassed via preference.
>=5.0	For all HTTP commands (for example, GET, POST, PUT), whether each of the options was specified. The values of the options aren't collected.

Additional resources

- .NET Core SDK telemetry
- .NET CLI telemetry data

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Minimal APIs overview

Article • 07/26/2024

Minimal APIs are a simplified approach for building fast HTTP APIs with ASP.NET Core. You can build fully functioning REST endpoints with minimal code and configuration. Skip traditional scaffolding and avoid unnecessary controllers by fluently declaring API routes and actions. For example, the following code creates an API at the root of the web app that returns the text, `"Hello World!"`.

C#

```
var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run();
```

Most APIs accept parameters as part of the route.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is
{bookId}");

app.Run();
```

That's all it takes to get started, but it's not all that's available. Minimal APIs support the configuration and customization needed to scale to multiple APIs, handle complex routes, apply authorization rules, and control the content of API responses. A good place to get started is Tutorial: Create a minimal API with ASP.NET Core.

Want to see some code examples?

For a full list of common scenarios with code examples, see Minimal APIs quick reference.

Want to jump straight into your first project?

Build a minimal API app with our tutorial: Tutorial: Create a minimal API with ASP.NET Core.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Tutorial: Create a minimal API with ASP.NET Core

Article • 08/21/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Rick Anderson and Tom Dykstra

Minimal APIs are architected to create HTTP APIs with minimal dependencies. They're ideal for microservices and apps that want to include only the minimum files, features, and dependencies in ASP.NET Core.

This tutorial teaches the basics of building a minimal API with ASP.NET Core. Another approach to creating APIs in ASP.NET Core is to use controllers. For help with choosing between minimal APIs and controller-based APIs, see APIs overview. For a tutorial on creating an API project based on controllers that contains more features, see Create a web API.

Overview

This tutorial creates the following API:

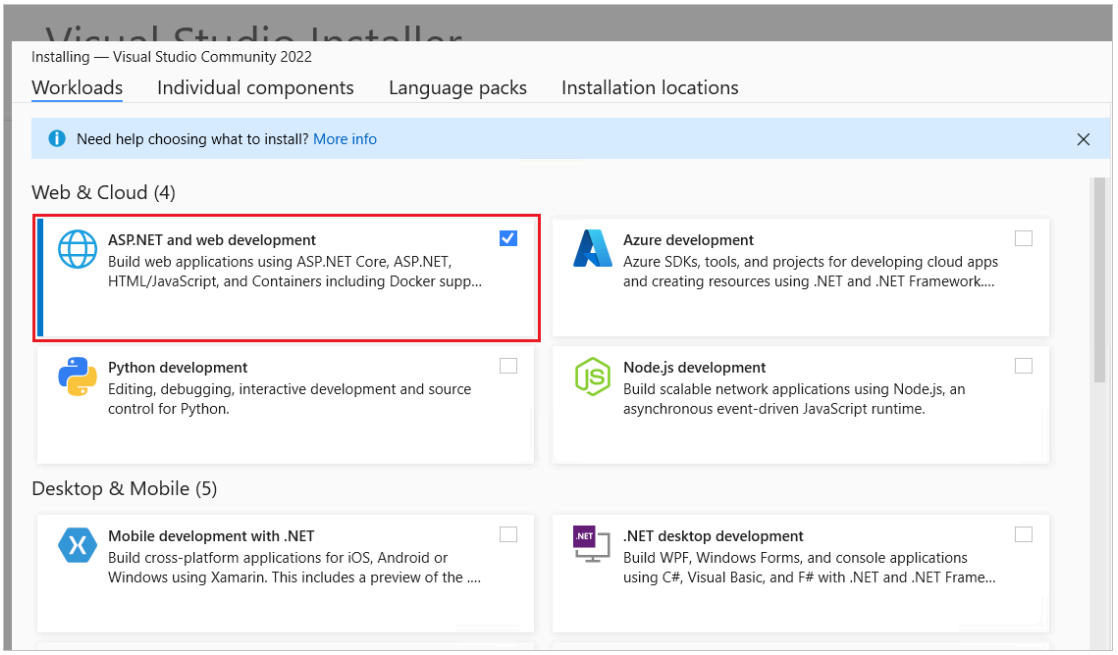
Expand table

API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item
`POST /todoitems`	Add a new item	To-do item	To-do item
`PUT /todoitems/{id}`	Update an existing item	To-do item	None
`DELETE /todoitems/{id}`	Delete an item	None	None

Prerequisites

Visual Studio

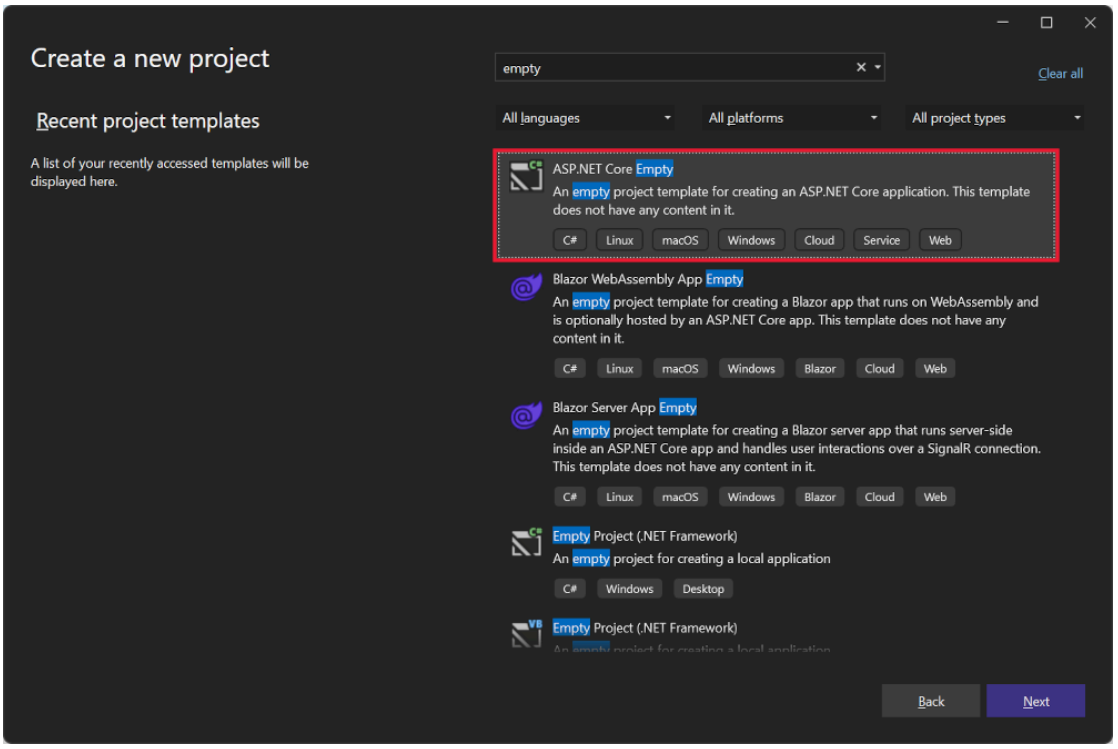
- Visual Studio 2022 with the **ASP.NET and web development** workload.



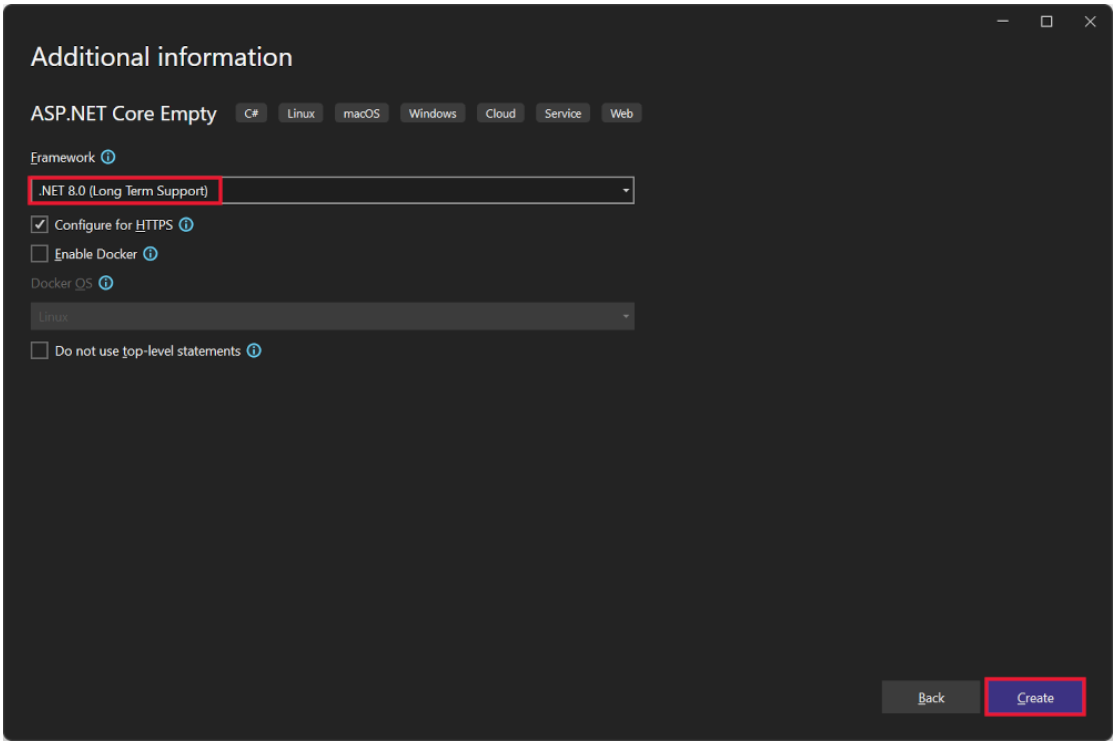
Create an API project

Visual Studio

- Start Visual Studio 2022 and select **Create a new project**.
- In the **Create a new project** dialog:
  - Enter `Empty` in the **Search for templates** search box.
  - Select the **ASP.NET Core Empty** template and select **Next**.



- Name the project *TodoApi* and select **Next**.
- In the **Additional information** dialog:
  - Select **.NET 8.0 (Long Term Support)**
  - Uncheck **Do not use top-level statements**
  - Select **Create**



Examine the code

The `Program.cs` file contains the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding code:

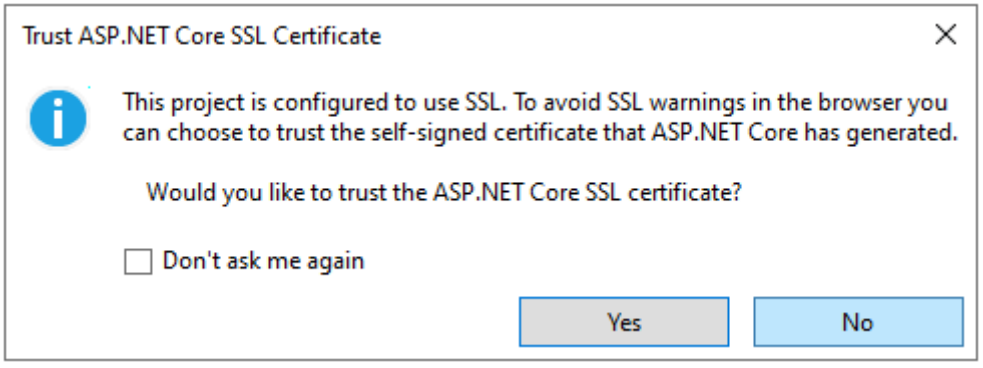
- Creates a WebApplicationBuilder and a WebApplication with preconfigured defaults.
- Creates an HTTP GET endpoint `/` that returns `Hello World!`:

Run the app

Visual Studio

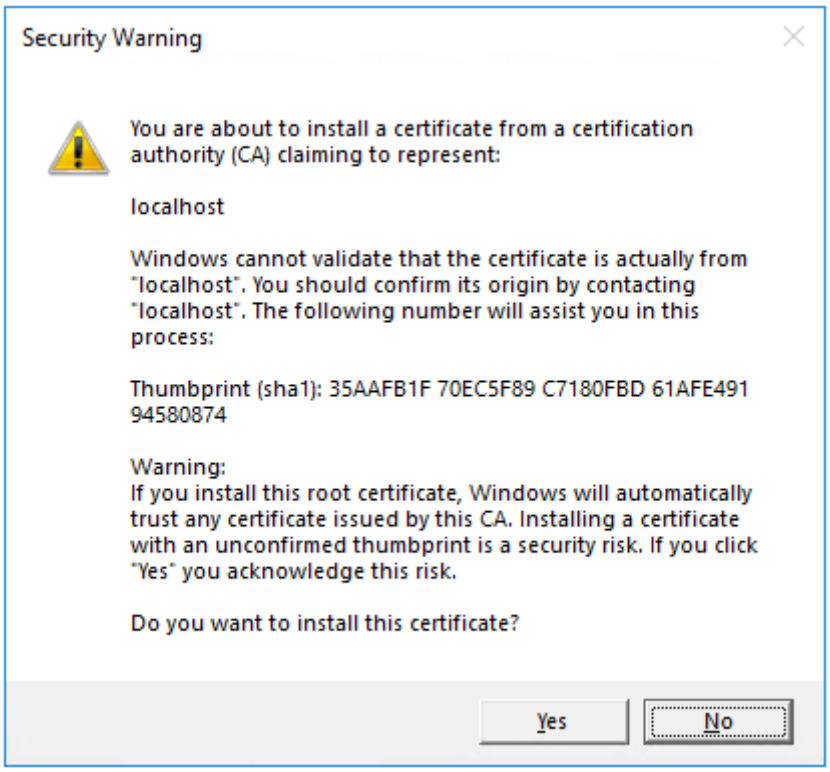
Press Ctrl+F5 to run without the debugger.

Visual Studio displays the following dialog:



Select **Yes** if you trust the IIS Express SSL certificate.

The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio launches the Kestrel web server and opens a browser window.

`Hello World!` is displayed in the browser. The `Program.cs` file contains a minimal but complete app.

Close the browser window.

Add NuGet packages

NuGet packages must be added to support the database and diagnostics used in this tutorial.

Visual Studio

- From the **Tools** menu, select **NuGet Package Manager** > **Manage NuGet Packages for Solution**.
- Select the **Browse** tab.
- Enter **Microsoft.EntityFrameworkCore.InMemory** in the search box, and then select `Microsoft.EntityFrameworkCore.InMemory`.
- Select the **Project** checkbox in the right pane and then select **Install**.

- Follow the preceding instructions to add the `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` package.

The model and database context classes

- In the project folder, create a file named `Todo.cs` with the following code:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
```

The preceding code creates the model for this app. A *model* is a class that represents data that the app manages.

- Create a file named `TodoDb.cs` with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

class TodoDb : DbContext
{
    public TodoDb(DbContextOptions<TodoDb> options)
        : base(options) { }

    public DbSet<Todo> Todos => Set<Todo>();
}
```

The preceding code defines the *database context*, which is the main class that coordinates Entity Framework functionality for a data model. This class derives from the Microsoft.EntityFrameworkCore.DbContext class.

Add the API code

- Replace the contents of the `Program.cs` file with the following code:

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

```
app.Run();
```

The following highlighted code adds the database context to the dependency injection (DI) container and enables displaying database-related exceptions:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();
```

The DI container provides access to the database context and other services.

Visual Studio

This tutorial uses Endpoints Explorer and .http files to test the API.

Test posting data

The following code in `Program.cs` creates an HTTP POST endpoint `/todoitems` that adds data to the in-memory database:

C#

```
app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});
```

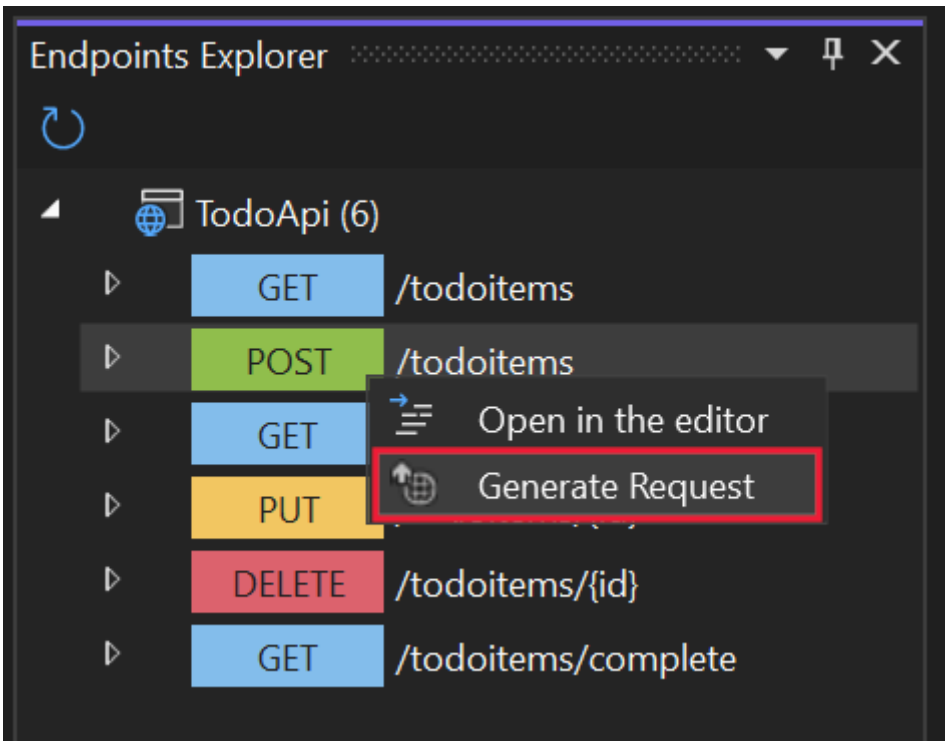
Run the app. The browser displays a 404 error because there's no longer a `/` endpoint.

The POST endpoint will be used to add data to the app.

Visual Studio

- Select **View** > **Other Windows** > **Endpoints Explorer**.

- Right-click the **POST** endpoint and select **Generate request**.



A new file is created in the project folder named `TodoApi.http`, with contents similar to the following example:

```
@TodoApi_HostAddress = https://localhost:7031

Post {{TodoApi_HostAddress}}/todoitems

###
```

  - The first line creates a variable that is used for all of the endpoints.
  - The next line defines a POST request.
  - The triple hashtag (`###`) line is a request delimiter: what comes after it is for a different request.

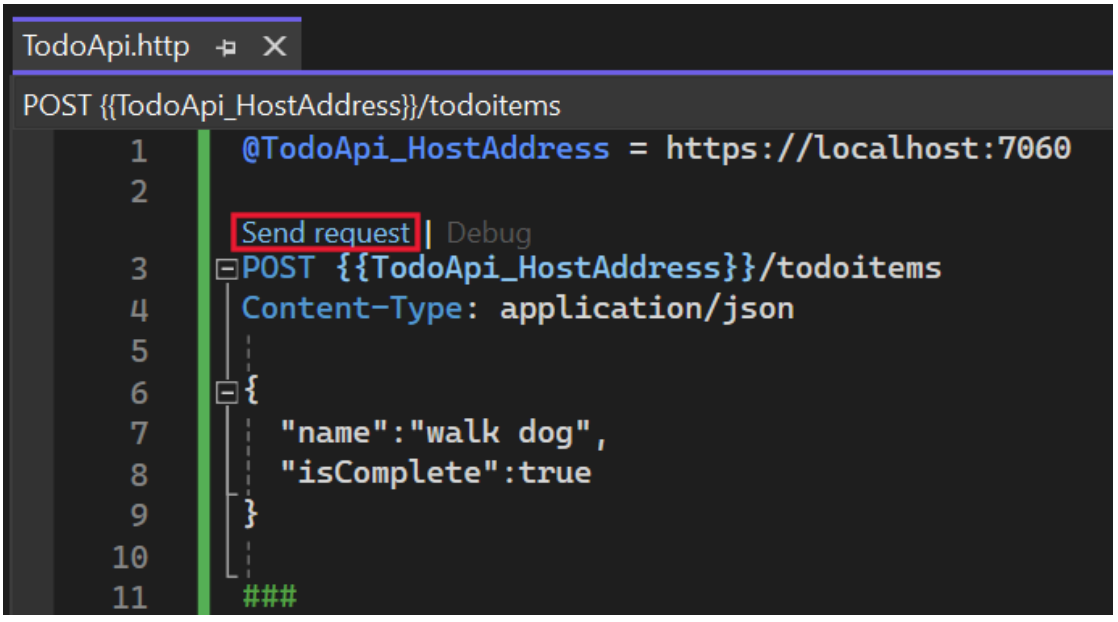
- The POST request needs headers and a body. To define those parts of the request, add the following lines immediately after the POST request line:

```
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}
```

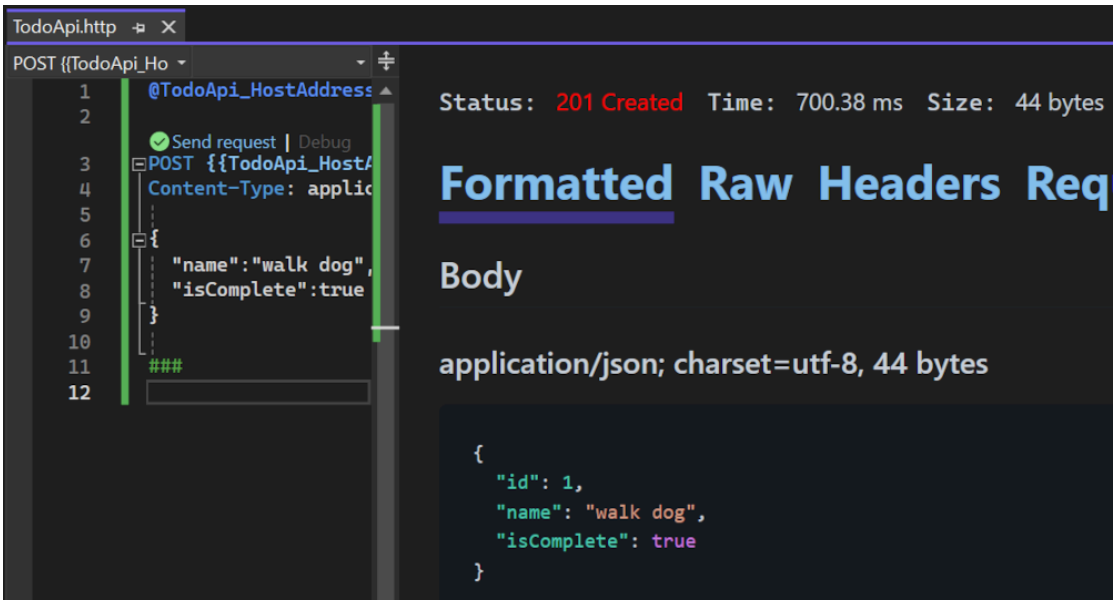
The preceding code adds a Content-Type header and a JSON request body. The TodoApi.http file should now look like the following example, but with your port number:

```
@TodoApi_HostAddress = https://localhost:7057

Post {{TodoApi_HostAddress}}/todoitems
Content-Type: application/json

{
  "name":"walk dog",
  "isComplete":true
}

###
```

- Run the app.
- Select the **Send request** link that is above the `POST` request line.



The POST request is sent to the app and the response is displayed in the **Response** pane.



Examine the GET endpoints

The sample app implements several GET endpoints by calling `MapGet`:

Expand table

API	Description	Request body	Response body
`GET /todoitems`	Get all to-do items	None	Array of to-do items
`GET /todoitems/complete`	Get all completed to-do items	None	Array of to-do items
`GET /todoitems/{id}`	Get an item by ID	None	To-do item

C#

```
app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync());

app.MapGet("/todoitems/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

Test the GET endpoints

Visual Studio

Test the app by calling the `GET` endpoints from a browser or by using **Endpoints Explorer**. The following steps are for **Endpoints Explorer**.

- In **Endpoints Explorer**, right-click the first **GET** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Get {{TodoApi_HostAddress}}/todoitems

  ###
  ```

- Select the **Send request** link that is above the new `GET` request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  JSON

  ```
  [
    {
      "id": 1,
      "name": "walk dog",
      "isComplete": true
    }
  ]
  ```

- In **Endpoints Explorer**, right-click the `/todoitems/{id}` **GET** endpoint and select **Generate request**. The following content is added to the `TodoApi.http` file:

  ```
  GET {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- Replace `{id}` with `1`.

- Select the **Send request** link that is above the new GET request line.

  The GET request is sent to the app and the response is displayed in the **Response** pane.

- The response body is similar to the following JSON:

  ```
  JSON

  {
    "id": 1,
    "name": "walk dog",
    "isComplete": true
  }
  ```

This app uses an in-memory database. If the app is restarted, the GET request doesn't return any data. If no data is returned, POST data to the app and try the GET request again.

Return values

ASP.NET Core automatically serializes the object to JSON and writes the JSON into the body of the response message. The response code for this return type is 200 OK, assuming there are no unhandled exceptions. Unhandled exceptions are translated into 5xx errors.

The return types can represent a wide range of HTTP status codes. For example, `GET /todoitems/{id}` can return two different status values:

- If no item matches the requested ID, the method returns a 404 status NotFound error code.
- Otherwise, the method returns 200 with a JSON response body. Returning `item` results in an HTTP 200 response.

Examine the PUT endpoint

The sample app implements a single PUT endpoint using `MapPut`:

```
C#

app.MapPut("/todoitems/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
```

```
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});
```

This method is similar to the `MapPost` method, except it uses HTTP PUT. A successful response returns 204 (No Content). According to the HTTP specification, a PUT request requires the client to send the entire updated entity, not just the changes. To support partial updates, use HTTP PATCH.

Test the PUT endpoint

This sample uses an in-memory database that must be initialized each time the app is started. There must be an item in the database before you make a PUT call. Call GET to ensure there's an item in the database before making a PUT call.

Update the to-do item that has `Id = 1` and set its name to `"feed fish"`.

Visual Studio

- In **Endpoints Explorer**, right-click the **PUT** endpoint, and select **Generate request**.

  The following content is added to the `TodoApi.http` file:

  ```
  Put {{TodoApi_HostAddress}}/todoitems/{id}

  ###
  ```

- In the PUT request line, replace `{id}` with `1`.
- Add the following lines immediately after the PUT request line:

  ```
  ```

```
Content-Type: application/json

{
  "name": "feed fish",
  "isComplete": false
}
```

The preceding code adds a Content-Type header and a JSON request body.

- Select the **Send request** link that is above the new PUT request line.

  The PUT request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Examine and test the DELETE endpoint

The sample app implements a single DELETE endpoint using `MapDelete`:

C#

```
app.MapDelete("/todoitems/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});
```

Visual Studio

- In **Endpoints Explorer**, right-click the **DELETE** endpoint and select **Generate request**.

  A DELETE request is added to `TodoApi.http`.

- Replace `{id}` in the DELETE request line with `1`. The DELETE request should look like the following example:

  ```
  
  ```

```
DELETE {{TodoApi_HostAddress}}/todoitems/1

###
```

- Select the **Send request** link for the DELETE request.

  The DELETE request is sent to the app and the response is displayed in the **Response** pane. The response body is empty, and the status code is 204.

Use the MapGroup API

The sample app code repeats the `todoitems` URL prefix each time it sets up an endpoint. APIs often have groups of endpoints with a common URL prefix, and the MapGroup method is available to help organize such groups. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata.

Replace the contents of `Program.cs` with the following code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", async (TodoDb db) =>
    await db.Todos.ToListAsync());

todoItems.MapGet("/complete", async (TodoDb db) =>
    await db.Todos.Where(t => t.IsComplete).ToListAsync());

todoItems.MapGet("/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

```
todoItems.MapPost("/", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
});

todoItems.MapPut("/{id}", async (int id, Todo inputTodo, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
});

todoItems.MapDelete("/{id}", async (int id, TodoDb db) =>
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return Results.NoContent();
    }

    return Results.NotFound();
});

app.Run();
```

The preceding code has the following changes:

- Adds `var todoItems = app.MapGroup("/todoitems");` to set up the group using the URL prefix `/todoitems`.
- Changes all the `app.Map<HttpVerb>` methods to `todoItems.Map<HttpVerb>`.
- Removes the URL prefix `/todoitems` from the `Map<HttpVerb>` method calls.

Test the endpoints to verify that they work the same.

Use the TypedResults API

Returning TypedResults rather than Results has several advantages, including testability and automatically returning the response type metadata for OpenAPI to describe the

endpoint. For more information, see TypedResults vs Results.

The `Map<HttpVerb>` methods can call route handler methods instead of using lambdas. To see an example, update *Program.cs* with the following code:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);

app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();
```

```
    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

The `Map<HttpVerb>` code now calls methods instead of lambdas:

C#

```
var todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

These methods return objects that implement IResult and are defined by TypedResults:

C#

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

```
static async Task<IResult> GetCompleteTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(todo)
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(Todo todo, TodoDb db)
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return TypedResults.Created($"/todoitems/{todo.Id}", todo);
}

static async Task<IResult> UpdateTodo(int id, Todo inputTodo, TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}

static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Unit tests can call these methods and test that they return the correct type. For example, if the method is `GetAllTodos`:

```
C#
```

```
static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.ToArrayAsync());
}
```

Unit test code can verify that an object of type Ok<Todo[]> is returned from the handler method. For example:

C#

```
public async Task GetAllTodos_ReturnsOkOfTodosResult()
{
    // Arrange
    var db = CreateDbContext();

    // Act
    var result = await TodosApi.GetAllTodos(db);

    // Assert: Check for the correct returned type
    Assert.IsType<Ok<Todo[]>>(result);
}
```

Prevent over-posting

Currently the sample app exposes the entire `Todo` object. Production apps In production applications, a subset of the model is often used to restrict the data that can be input and returned. There are multiple reasons behind this and security is a major one. The subset of a model is usually referred to as a Data Transfer Object (DTO), input model, or view model. **DTO** is used in this article.

A DTO can be used to:

- Prevent over-posting.
- Hide properties that clients aren't supposed to view.
- Omit some properties to reduce payload size.
- Flatten object graphs that contain nested objects. Flattened object graphs can be more convenient for clients.

To demonstrate the DTO approach, update the `Todo` class to include a secret field:

C#

```
public class Todo
{
    public int Id { get; set; }
```

```
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
    public string? Secret { get; set; }
}
```

The secret field needs to be hidden from this app, but an administrative app could choose to expose it.

Verify you can post and get the secret field.

Create a file named `TodoItemDTO.cs` with the following code:

C#

```
public class TodoItemDTO
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    public TodoItemDTO() { }
    public TodoItemDTO(Todo todoItem) =>
    (Id, Name, IsComplete) = (todoItem.Id, todoItem.Name,
todoItem.IsComplete);
}
```

Replace the contents of the `Program.cs` file with the following code to use this DTO model:

Visual Studio

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
var app = builder.Build();

RouteGroupBuilder todoItems = app.MapGroup("/todoitems");

todoItems.MapGet("/", GetAllTodos);
todoItems.MapGet("/complete", GetCompleteTodos);
todoItems.MapGet("/{id}", GetTodo);
todoItems.MapPost("/", CreateTodo);
todoItems.MapPut("/{id}", UpdateTodo);
todoItems.MapDelete("/{id}", DeleteTodo);
```

```
app.Run();

static async Task<IResult> GetAllTodos(TodoDb db)
{
    return TypedResults.Ok(await db.Todos.Select(x => new
TodoItemDTO(x)).ToArrayAsync());
}

static async Task<IResult> GetCompleteTodos(TodoDb db) {
    return TypedResults.Ok(await db.Todos.Where(t =>
t.IsComplete).Select(x => new TodoItemDTO(x)).ToListAsync());
}

static async Task<IResult> GetTodo(int id, TodoDb db)
{
    return await db.Todos.FindAsync(id)
        is Todo todo
            ? TypedResults.Ok(new TodoItemDTO(todo))
            : TypedResults.NotFound();
}

static async Task<IResult> CreateTodo(TodoItemDTO todoItemDTO, TodoDb
db)
{
    var todoItem = new Todo
    {
        IsComplete = todoItemDTO.IsComplete,
        Name = todoItemDTO.Name
    };

    db.Todos.Add(todoItem);
    await db.SaveChangesAsync();

    todoItemDTO = new TodoItemDTO(todoItem);

    return TypedResults.Created($"/todoitems/{todoItem.Id}",
todoItemDTO);
}

static async Task<IResult> UpdateTodo(int id, TodoItemDTO todoItemDTO,
TodoDb db)
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return TypedResults.NotFound();

    todo.Name = todoItemDTO.Name;
    todo.IsComplete = todoItemDTO.IsComplete;

    await db.SaveChangesAsync();

    return TypedResults.NoContent();
}
```

```
static async Task<IResult> DeleteTodo(int id, TodoDb db)
{
    if (await db.Todos.FindAsync(id) is Todo todo)
    {
        db.Todos.Remove(todo);
        await db.SaveChangesAsync();
        return TypedResults.NoContent();
    }

    return TypedResults.NotFound();
}
```

Verify you can post and get all fields except the secret field.

Troubleshooting with the completed sample

If you run into a problem you can't resolve, compare your code to the completed project. View or download completed project (how to download).

Next steps

- Configure JSON serialization options.
- Handle errors and exceptions: The developer exception page is enabled by default in the development environment for minimal API apps. For information about how to handle errors and exceptions, see Handle errors in ASP.NET Core APIs.
- For an example of testing a minimal API app, see this GitHub sample.
- OpenAPI support in minimal APIs.
- Quickstart: Publish to Azure.
- Organizing ASP.NET Core Minimal APIs.

Learn more

See Minimal APIs quick reference

Minimal APIs quick reference

Article • 11/07/2023

This document:

- Provides a quick reference for minimal APIs.
- Is intended for experienced developers. For an introduction, see Tutorial: Create a minimal API with ASP.NET Core.

The minimal APIs consist of:

- WebApplication and WebApplicationBuilder
- Route Handlers

`WebApplication`

The following code is generated by an ASP.NET Core template:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding code can be created via `dotnet new web` on the command line or selecting the Empty Web template in Visual Studio.

The following code creates a WebApplication (`app`) without explicitly creating a WebApplicationBuilder:

```
C#

var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run();
```

WebApplication.Create initializes a new instance of the WebApplication class with preconfigured defaults.

Working with ports

When a web app is created with Visual Studio or `dotnet new`, a `Properties/launchSettings.json` file is created that specifies the ports the app responds to. In the port setting samples that follow, running the app from Visual Studio returns an error dialog `Unable to connect to web server 'AppName'`. Visual Studio returns an error because it's expecting the port specified in `Properties/launchSettings.json`, but the app is using the port specified by `app.Run("http://localhost:3000")`. Run the following port changing samples from the command line.

The following sections set the port the app responds to.

C#

```
var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run("http://localhost:3000");
```

In the preceding code, the app responds to port `3000`.

Multiple ports

In the following code, the app responds to port `3000` and `4000`.

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://localhost:3000");
app.Urls.Add("http://localhost:4000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Set the port from the command line

The following command makes the app respond to port `7777`:

.NET CLI

```
dotnet run --urls="https://localhost:7777"
```

If the Kestrel endpoint is also configured in the `appsettings.json` file, the `appsettings.json` file specified URL is used. For more information, see Kestrel endpoint configuration

Read the port from environment

The following code reads the port from the environment:

C#

```
var app = WebApplication.Create(args);

var port = Environment.GetEnvironmentVariable("PORT") ?? "3000";

app.MapGet("/", () => "Hello World");

app.Run($"http://localhost:{port}");
```

The preferred way to set the port from the environment is to use the `ASPNETCORE_URLS` environment variable, which is shown in the following section.

Set the ports via the ASPNETCORE_URLS environment variable

The `ASPNETCORE_URLS` environment variable is available to set the port:

```
ASPNETCORE_URLS=http://localhost:3000
```

`ASPNETCORE_URLS` supports multiple URLs:

```
ASPNETCORE_URLS=http://localhost:3000;https://localhost:5000
```

Listen on all interfaces

The following samples demonstrate listening on all interfaces

http://*:3000

```
C#

var app = WebApplication.Create(args);

app.Urls.Add("http://*:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

http://+:3000

```
C#

var app = WebApplication.Create(args);

app.Urls.Add("http://+:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

`http://0.0.0.0:3000`

```
C#

var app = WebApplication.Create(args);

app.Urls.Add("http://0.0.0.0:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Listen on all interfaces using ASPNETCORE_URLS

The preceding samples can use `ASPNETCORE_URLS`

```

ASPNETCORE_URLS=http://*:3000;https://+:5000;http://0.0.0.0:5005
```

Listen on all interfaces using ASPNETCORE_HTTPS_PORTS

The preceding samples can use `ASPNETCORE_HTTPS_PORTS` and `ASPNETCORE_HTTP_PORTS`.

```
ASPNETCORE_HTTP_PORTS=3000;5005
ASPNETCORE_HTTPS_PORTS=5000
```

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server

Specify HTTPS with development certificate

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

For more information on the development certificate, see Trust the ASP.NET Core HTTPS development certificate on Windows and macOS.

Specify HTTPS using a custom certificate

The following sections show how to specify the custom certificate using the `appsettings.json` file and via configuration.

Specify the custom certificate with `appsettings.json`

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
```

```
  "Kestrel": {
    "Certificates": {
      "Default": {
        "Path": "cert.pem",
        "KeyPath": "key.pem"
      }
    }
  }
}
```

Specify the custom certificate via configuration

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure the cert and the key
builder.Configuration["Kestrel:Certificates:Default:Path"] = "cert.pem";
builder.Configuration["Kestrel:Certificates:Default:KeyPath"] = "key.pem";

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Use the certificate APIs

C#

```
using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>
{
    options.ConfigureHttpsDefaults(httpsOptions =>
    {
        var certPath = Path.Combine(builder.Environment.ContentRootPath,
"cert.pem");
        var keyPath = Path.Combine(builder.Environment.ContentRootPath,
"key.pem");

        httpsOptions.ServerCertificate =
X509Certificate2.CreateFromPemFile(certPath,
                                           keyPath);
    });
```

```
});

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Read the environment

C#

```
var app = WebApplication.Create(args);

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/oops");
}

app.MapGet("/", () => "Hello World");
app.MapGet("/oops", () => "Oops! An error happened.");

app.Run();
```

For more information using the environment, see Use multiple environments in ASP.NET Core

Configuration

The following code reads from the configuration system:

C#

```
var app = WebApplication.Create(args);

var message = app.Configuration["HelloKey"] ?? "Config failed!";

app.MapGet("/", () => message);

app.Run();
```

For more information, see Configuration in ASP.NET Core

Logging

The following code writes a message to the log on application startup:

```
C#
```

```
var app = WebApplication.Create(args);

app.Logger.LogInformation("The app started");

app.MapGet("/", () => "Hello World");

app.Run();
```

For more information, see Logging in .NET Core and ASP.NET Core

Access the Dependency Injection (DI) container

The following code shows how to get services from the DI container during application startup:

```
C#
```

```


var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddScoped<SampleService>();

var app = builder.Build();

app.MapControllers();

using (var scope = app.Services.CreateScope())
{
    var sampleService =
scope.ServiceProvider.GetRequiredService<SampleService>();
    sampleService.DoSomething();
}

app.Run();
```

The following code shows how to access keys from the DI container using the [FromKeyedServices] attribute:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
```

```
var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));

app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>
smallCache.Get("date"));

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

For more information on DI, see Dependency injection in ASP.NET Core.

WebApplicationBuilder

This section contains sample code using WebApplicationBuilder.

Change the content root, application name, and environment

The following code sets the content root, application name, and environment:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    ApplicationName = typeof(Program).Assembly.FullName,
    ContentRootPath = Directory.GetCurrentDirectory(),
    EnvironmentName = Environments.Staging,
    WebRootPath = "customwwwroot"
});

Console.WriteLine($"Application Name:
{builder.Environment.ApplicationName}");
```

```
Console.WriteLine($"Environment Name:
{builder.Environment.EnvironmentName}");
Console.WriteLine($"ContentRoot Path:
{builder.Environment.ContentRootPath}");
Console.WriteLine($"WebRootPath: {builder.Environment.WebRootPath}");

var app = builder.Build();
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults.

For more information, see ASP.NET Core fundamentals overview

Change the content root, app name, and environment by using environment variables or command line

The following table shows the environment variable and command-line argument used to change the content root, app name, and environment:

Expand table

feature	Environment variable	Command-line argument
Application name	ASPNETCORE_APPLICATIONNAME	--applicationName
Environment name	ASPNETCORE_ENVIRONMENT	--environment
Content root	ASPNETCORE_CONTENTROOT	--contentRoot

Add configuration providers

The following sample adds the INI configuration provider:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddIniFile("appsettings.ini");

var app = builder.Build();
```

For detailed information, see File configuration providers in Configuration in ASP.NET Core.

Read configuration

By default the WebApplicationBuilder reads configuration from multiple sources, including:

- `appSettings.json` and `appSettings.{environment}.json`
- Environment variables
- The command line

For a complete list of configuration sources read, see Default configuration in Configuration in ASP.NET Core.

The following code reads `HelloKey` from configuration and displays the value at the `/` endpoint. If the configuration value is null, "Hello" is assigned to `message`:

C#

```
var builder = WebApplication.CreateBuilder(args);

var message = builder.Configuration["HelloKey"] ?? "Hello";

var app = builder.Build();

app.MapGet("/", () => message);

app.Run();
```

Read the environment

C#

```
var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsDevelopment())
{
    Console.WriteLine($"Running in development.");
}

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Add logging providers

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure JSON logging to the console.
builder.Logging.AddJsonConsole();

var app = builder.Build();

app.MapGet("/", () => "Hello JSON console!");

app.Run();
```

Add services

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add the memory cache services.
builder.Services.AddMemoryCache();

// Add a custom scoped service.
builder.Services.AddScoped<ITodoRepository, TodoRepository>();
var app = builder.Build();
```

Customize the IHostBuilder

Existing extension methods on IHostBuilder can be accessed using the Host property:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Wait 30 seconds for graceful shutdown.
builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout =
TimeSpan.FromSeconds(30));

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Customize the IWebHostBuilder

Extension methods on IWebHostBuilder can be accessed using the WebApplicationBuilder.WebHost property.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Change the HTTP server implemenation to be HTTP.sys based
builder.WebHost.UseHttpSys();

var app = builder.Build();

app.MapGet("/", () => "Hello HTTP.sys");

app.Run();
```

Change the web root

By default, the web root is relative to the content root in the `wwwroot` folder. Web root is where the static files middleware looks for static files. Web root can be changed with `WebHostOptions`, the command line, or with the UseWebRoot method:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in webroot
    WebRootPath = "webroot"
});

var app = builder.Build();

app.Run();
```

Custom dependency injection (DI) container

The following example uses Autofac:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());

// Register services directly with Autofac here. Don't
// call builder.Populate(), that happens in AutofacServiceProviderFactory.
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
builder.RegisterModule(new MyApplicationModule()));
```

```
var app = builder.Build();
```

Add Middleware

Any existing ASP.NET Core middleware can be configured on the `WebApplication`:

C#

```
var app = WebApplication.Create(args);

// Setup the file server to serve static files.
app.UseFileServer();

app.MapGet("/", () => "Hello World!");

app.Run();
```

For more information, see ASP.NET Core Middleware

Developer exception page

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The developer exception page is enabled in the preconfigured defaults. When the following code is run in the development environment, navigating to `/` renders a friendly page that shows the exception.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () =>
{
    throw new InvalidOperationException("Oops, the '/' route has thrown an
exception.");
});

app.Run();
```

ASP.NET Core Middleware

The following table lists some of the middleware frequently used with minimal APIs.

Expand table

Middleware	Description	API
Authentication	Provides authentication support.	UseAuthentication
Authorization	Provides authorization support.	UseAuthorization
CORS	Configures Cross-Origin Resource Sharing.	UseCors
Exception Handler	Globally handles exceptions thrown by the middleware pipeline.	UseExceptionHandler
Forwarded Headers	Forwards proxied headers onto the current request.	UseForwardedHeaders
HTTPS Redirection	Redirects all HTTP requests to HTTPS.	UseHttpsRedirection
HTTP Strict Transport Security (HSTS)	Security enhancement middleware that adds a special response header.	UseHsts
Request Logging	Provides support for logging HTTP requests and responses.	UseHttpLogging
Request Timeouts	Provides support for configuring request timeouts, global default and per endpoint.	`UseRequestTimeouts`
W3C Request Logging	Provides support for logging HTTP requests and responses in the W3C format.	UseW3CLogging
Response Caching	Provides support for caching responses.	UseResponseCaching
Response Compression	Provides support for compressing responses.	UseResponseCompression
Session	Provides support for managing user sessions.	UseSession
Static Files	Provides support for serving static files and directory browsing.	UseStaticFiles, UseFileServer
WebSockets	Enables the WebSockets protocol.	UseWebSockets

The following sections cover request handling: routing, parameter binding, and responses.

Routing

A configured `WebApplication` supports `Map{Verb}` and MapMethods where `{Verb}` is a camel-cased HTTP method like `Get`, `Post`, `Put` or `Delete`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "This is a GET");
app.MapPost("/", () => "This is a POST");
app.MapPut("/", () => "This is a PUT");
app.MapDelete("/", () => "This is a DELETE");

app.MapMethods("/options-or-head", new[] { "OPTIONS", "HEAD" },
                           () => "This is an options or head request ");

app.Run();
```

The Delegate arguments passed to these methods are called "route handlers".

Route Handlers

Route handlers are methods that execute when the route matches. Route handlers can be a lambda expression, a local function, an instance method or a static method. Route handlers can be synchronous or asynchronous.

Lambda expression

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/inline", () => "This is an inline lambda");

var handler = () => "This is a lambda variable";

app.MapGet("/", handler);

app.Run();
```

Local function

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

string LocalFunction() => "This is local function";

app.MapGet("/", LocalFunction);

app.Run();
```

Instance method

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

var handler = new HelloHandler();

app.MapGet("/", handler.Hello);

app.Run();

class HelloHandler
{
    public string Hello()
    {
        return "Hello Instance method";
    }
}
```

Static method

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", HelloHandler.Hello);

app.Run();

class HelloHandler
{
    public static string Hello()
    {
        return "Hello static method";
    }
}
```

Endpoint defined outside of `Program.cs`

Minimal APIs don't have to be located in `Program.cs`.

`Program.cs`

C#

```
using MinAPISeparateFile;

var builder = WebApplication.CreateSlimBuilder(args);

var app = builder.Build();

TodoEndpoints.Map(app);

app.Run();
```

`TodoEndpoints.cs`

C#

```
namespace MinAPISeparateFile;

public static class TodoEndpoints
{
    public static void Map(WebApplication app)
    {
        app.MapGet("/", async context =>
        {
            // Get all todo items
            await context.Response.WriteAsJsonAsync(new { Message = "All
todo items" });
        });

        app.MapGet("/{id}", async context =>
        {
            // Get one todo item
            await context.Response.WriteAsJsonAsync(new { Message = "One
todo item" });
        });
    }
}
```

See also Route groups later in this article.

Named endpoints and link generation

Endpoints can be given names in order to generate URLs to the endpoint. Using a named endpoint avoids having to hard code paths in an app:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/hello", () => "Hello named route")
   .WithName("hi");

app.MapGet("/", (LinkGenerator linker) =>
        $"The link to the hello route is {linker.GetPathByName("hi", values:
null)}");

app.Run();
```

The preceding code displays `The link to the hello route is /hello` from the `/` endpoint.

NOTE: Endpoint names are case sensitive.

Endpoint names:

- Must be globally unique.
- Are used as the OpenAPI operation id when OpenAPI support is enabled. For more information, see OpenAPI.

Route Parameters

Route parameters can be captured as part of the route pattern definition:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is
{bookId}");

app.Run();
```

The preceding code returns `The user id is 3 and book id is 7` from the URI `/users/3/books/7`.

The route handler can declare the parameters to capture. When a request is made to a route with parameters declared to capture, the parameters are parsed and passed to the handler. This makes it easy to capture the values in a type safe way. In the preceding code, `userId` and `bookId` are both `int`.

In the preceding code, if either route value cannot be converted to an `int`, an exception is thrown. The GET request `/users/hello/books/3` throws the following exception:

```
BadHttpRequestException: Failed to bind parameter "int userId" from "hello".
```

Wildcard and catch all routes

The following catch all route returns `Routing to hello` from the `/posts/hello' endpoint:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/posts/{*rest}", (string rest) => $"Routing to {rest}");

app.Run();
```

Route constraints

Route constraints constrain the matching behavior of a route.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/todos/{id:int}", (int id) => db.Todos.Find(id));
app.MapGet("/todos/{text}", (string text) => db.Todos.Where(t =>
t.Text.Contains(text));
app.MapGet("/posts/{slug:regex(^[a-z0-9_-]+$)}", (string slug) => $"Post
{slug}");

app.Run();
```

The following table demonstrates the preceding route templates and their behavior:

Expand table

Route Template	Example Matching URI
`/todos/{id:int}`	`/todos/1`
`/todos/{text}`	`/todos/something`
`/posts/{slug:regex(^[a-z0-9_-]+$)}`	`/posts/mypost`

For more information, see Route constraint reference in Routing in ASP.NET Core.

Route groups

The MapGroup extension method helps organize groups of endpoints with a common prefix. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata which add endpoint metadata.

For example, the following code creates two similar groups of endpoints:

C#

```
app.MapGroup("/public/todos")
    .MapTodosApi()
    .WithTags("Public");

app.MapGroup("/private/todos")
    .MapTodosApi()
    .WithTags("Private")
    .AddEndpointFilterFactory(QueryPrivateTodos)
    .RequireAuthorization();


EndpointFilterDelegate QueryPrivateTodos(EndpointFilterFactoryContext
factoryContext, EndpointFilterDelegate next)
{
    var dbContextIndex = -1;

    foreach (var argument in factoryContext.MethodInfo.GetParameters())
    {
        if (argument.ParameterType == typeof(TodoDb))
        {
            dbContextIndex = argument.Position;
            break;
        }
    }

    // Skip filter if the method doesn't have a TodoDb parameter.
    if (dbContextIndex < 0)
    {
        return next;
```

```
    }

    return async invocationContext =>
    {
        var dbContext = invocationContext.GetArgument<TodoDb>
(dbContextIndex);
        dbContext.IsPrivate = true;

        try
        {
            return await next(invocationContext);
        }
        finally
        {
            // This should only be relevant if you're pooling or otherwise
reusing the DbContext instance.
            dbContext.IsPrivate = false;
        }
    };
}
```

C#

```
public static RouteGroupBuilder MapTodosApi(this RouteGroupBuilder group)
{
    group.MapGet("/", GetAllTodos);
    group.MapGet("/{id}", GetTodo);
    group.MapPost("/", CreateTodo);
    group.MapPut("/{id}", UpdateTodo);
    group.MapDelete("/{id}", DeleteTodo);

    return group;
}
```

In this scenario, you can use a relative address for the `Location` header in the `201 Created` result:

C#

```
public static async Task<Created<Todo>> CreateTodo(Todo todo, TodoDb
database)
{
    await database.AddAsync(todo);
    await database.SaveChangesAsync();

    return TypedResults.Created($"{todo.Id}", todo);
}
```

The first group of endpoints will only match requests prefixed with `/public/todos` and are accessible without any authentication. The second group of endpoints will only

match requests prefixed with `/private/todos` and require authentication.

The `QueryPrivateTodos` endpoint filter factory is a local function that modifies the route handler's `TodoDb` parameters to allow to access and store private todo data.

Route groups also support nested groups and complex prefix patterns with route parameters and constraints. In the following example, and route handler mapped to the `user` group can capture the `{org}` and `{group}` route parameters defined in the outer group prefixes.

The prefix can also be empty. This can be useful for adding endpoint metadata or filters to a group of endpoints without changing the route pattern.

C#

```
var all = app.MapGroup("").WithOpenApi();
var org = all.MapGroup("{org}");
var user = org.MapGroup("{user}");
user.MapGet("", (string org, string user) => $"{org}/{user}");
```

Adding filters or metadata to a group behaves the same way as adding them individually to each endpoint before adding any extra filters or metadata that may have been added to an inner group or specific endpoint.

C#

```
var outer = app.MapGroup("/outer");
var inner = outer.MapGroup("/inner");

inner.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/inner group filter");
    return next(context);
});

outer.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/outer group filter");
    return next(context);
});

inner.MapGet("/", () => "Hi!").AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("MapGet filter");
    return next(context);
});
```

In the above example, the outer filter will log the incoming request before the inner filter even though it was added second. Because the filters were applied to different groups, the order they were added relative to each other does not matter. The order filters are added does matter if applied to the same group or specific endpoint.

A request to `/outer/inner/` will log the following:

.NET CLI

```
/outer group filter
/inner group filter
MapGet filter
```

Parameter binding

Parameter binding is the process of converting request data into strongly typed parameters that are expressed by route handlers. A binding source determines where parameters are bound from. Binding sources can be explicit or inferred based on HTTP method and parameter type.

Supported binding sources:

- Route values
- Query string
- Header
- Body (as JSON)
- Form values
- Services provided by dependency injection
- Custom

The following `GET` route handler uses some of these parameter binding sources:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Added as service
builder.Services.AddSingleton<Service>();

var app = builder.Build();

app.MapGet("/{id}", (int id,
                     int page,
                     [FromHeader(Name = "X-CUSTOM-HEADER")] string
customHeader,
                     Service service) => { });
```

```
class Service { }
```

The following table shows the relationship between the parameters used in the preceding example and the associated binding sources.

Expand table

Parameter	Binding Source
`id`	route value
`page`	query string
`customHeader`	header
`service`	Provided by dependency injection

The HTTP methods `GET`, `HEAD`, `OPTIONS`, and `DELETE` don't implicitly bind from body. To bind from body (as JSON) for these HTTP methods, bind explicitly with [FromBody] or read from the HttpRequest.

The following example POST route handler uses a binding source of body (as JSON) for the `person` parameter:

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapPost("/", (Person person) => { });

record Person(string Name, int Age);
```

The parameters in the preceding examples are all bound from request data automatically. To demonstrate the convenience that parameter binding provides, the following route handlers show how to read request data directly from the request:

C#

```
app.MapGet("/{id}", (HttpRequest request) =>
{
    var id = request.RouteValues["id"];
    var page = request.Query["page"];
    var customHeader = request.Headers["X-CUSTOM-HEADER"];
```

```
    // ...
});

app.MapPost("/", async (HttpRequest request) =>
{
    var person = await request.ReadFromJsonAsync<Person>();

    // ...
});
```

Explicit Parameter Binding

Attributes can be used to explicitly declare where parameters are bound from.

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

// Added as service
builder.Services.AddSingleton<Service>();

var app = builder.Build();


app.MapGet("/{id}", ([FromRoute] int id,
                     [FromQuery(Name = "p")] int page,
                     [FromServices] Service service,
                     [FromHeader(Name = "Content-Type")] string contentType)
                     => {});

class Service { }

record Person(string Name, int Age);
```

Expand table

Parameter	Binding Source
`id`	route value with the name `id`
`page`	query string with the name `"p"`
`service`	Provided by dependency injection
`contentType`	header with the name `"Content-Type"`

Explicit binding from form values

The [FromForm] attribute binds form values:

C#

```
app.MapPost("/todos", async ([FromForm] string name,
    [FromForm] Visibility visibility, IFormFile? attachment, TodoDb db) =>
{
    var todo = new Todo
    {
        Name = name,
        Visibility = visibility
    };

    if (attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();

        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await attachment.CopyToAsync(stream);
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

// Remaining code removed for brevity.
```

An alternative is to use the [AsParameters] attribute with a custom type that has properties annotated with `[FromForm]`. For example, the following code binds from form values to properties of the `NewTodoRequest` record struct:

C#

```
app.MapPost("/ap/todos", async ([AsParameters] NewTodoRequest request,
TodoDb db) =>
{
    var todo = new Todo
    {
        Name = request.Name,
        Visibility = request.Visibility
    };

    if (request.Attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();
```

```
        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await request.Attachment.CopyToAsync(stream);

        todo.Attachment = attachmentName;
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

// Remaining code removed for brevity.
```

C#

```
public record struct NewTodoRequest([FromForm] string Name,
    [FromForm] Visibility Visibility, IFormFile? Attachment);
```

For more information, see the section on AsParameters later in this article.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Secure binding from IFormFile and IFormFileCollection

Complex form binding is supported using IFormFile and IFormFileCollection using the [FromForm]:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();
app.UseAntiforgery();

// Generate a form with an anti-forgery token and an /upload endpoint.
app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = MyUtils.GenerateHtmlForm(token.FormFieldName,
token.RequestToken!);
    return Results.Content(html, "text/html");
});
```

```
app.MapPost("/upload", async Task<Results<Ok<string>, BadRequest<string>>>
    ([FromForm] FileUploadForm fileUploadForm, HttpContext context,
                                                IAntiforgery antiforgery) =>
{
    await MyUtils.SaveFileWithName(fileUploadForm.FileDocument!,
              fileUploadForm.Name!, app.Environment.ContentRootPath);
    return TypedResults.Ok($"Your file with the description:" +
        $" {fileUploadForm.Description} has been uploaded successfully");
});

app.Run();
```

Parameters bound to the request with `[FromForm]` include an anti-forgery token. The anti-forgery token is validated when the request is processed. For more information, see Antiforgery with Minimal APIs.

For more information, see Form binding in minimal APIs.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Parameter binding with dependency injection

Parameter binding for minimal APIs binds parameters through dependency injection when the type is configured as a service. It's not necessary to explicitly apply the [FromServices] attribute to a parameter. In the following code, both actions return the time:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IDateTime, SystemDateTime>();

var app = builder.Build();

app.MapGet("/",   (               IDateTime dateTime) => dateTime.Now);
app.MapGet("/fs", ([FromServices] IDateTime dateTime) => dateTime.Now);
app.Run();
```

Optional parameters

Parameters declared in route handlers are treated as required:

- If a request matches the route, the route handler only runs if all required parameters are provided in the request.

- Failure to provide all required parameters results in an error.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/products", (int pageNumber) => $"Requesting page
{pageNumber}");

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	`BadHttpRequestException`: Required parameter "int pageNumber" wasn't provided from query string.
`/products/1`	HTTP 404 error, no matching route

To make `pageNumber` optional, define the type as optional or provide a default value:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

string ListProducts(int pageNumber = 1) => $"Requesting page {pageNumber}";

app.MapGet("/products2", ListProducts);

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	1 returned

URI	result
`/products2`	1 returned

The preceding nullable and default value applies to all sources:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapPost("/products", (Product? product) => { });

app.Run();
```

The preceding code calls the method with a null product if no request body is sent.

NOTE: If invalid data is provided and the parameter is nullable, the route handler is ***not*** run.

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	`3` returned
`/products`	`1` returned
`/products?pageNumber=two`	`BadHttpRequestException`: Failed to bind parameter `"Nullable<int> pageNumber"` from "two".
`/products/two`	HTTP 404 error, no matching route

See the Binding Failures section for more information.

Special types

The following types are bound without explicit attributes:

- HttpContext: The context which holds all the information about the current HTTP request or response:

  ```
  C#

  app.MapGet("/", (HttpContext context) =>
  context.Response.WriteAsync("Hello World"));
  ```

- HttpRequest and HttpResponse: The HTTP request and HTTP response:

  ```
  C#

  app.MapGet("/", (HttpRequest request, HttpResponse response) =>
      response.WriteAsync($"Hello World {request.Query["name"]}"));
  ```

- CancellationToken: The cancellation token associated with the current HTTP request:

  ```
  C#

  app.MapGet("/", async (CancellationToken cancellationToken) =>
      await MakeLongRunningRequestAsync(cancellationToken));
  ```

- ClaimsPrincipal: The user associated with the request, bound from HttpContext.User:

  ```
  C#

  app.MapGet("/", (ClaimsPrincipal user) => user.Identity.Name);
  ```

Bind the request body as a `Stream` or `PipeReader`

The request body can bind as a Stream or PipeReader to efficiently support scenarios where the user has to process data and:

- Store the data to blob storage or enqueue the data to a queue provider.
- Process the stored data with a worker process or cloud function.

For example, the data might be enqueued to Azure Queue storage or stored in Azure Blob storage.

The following code implements a background queue:

C#

```
using System.Text.Json;
using System.Threading.Channels;

namespace BackgroundQueueService;

class BackgroundQueue : BackgroundService
{
    private readonly Channel<ReadOnlyMemory<byte>> _queue;
    private readonly ILogger<BackgroundQueue> _logger;

    public BackgroundQueue(Channel<ReadOnlyMemory<byte>> queue,
                           ILogger<BackgroundQueue> logger)
    {
        _queue = queue;
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        await foreach (var dataStream in
_queue.Reader.ReadAllAsync(stoppingToken))
        {
            try
            {
                var person = JsonSerializer.Deserialize<Person>
(dataStream.Span)!;
                _logger.LogInformation($"{person.Name} is {person.Age} " +
                                       $"years and from {person.Country}");
            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }
        }
    }
}

class Person
{
    public string Name { get; set; } = String.Empty;
    public int Age { get; set; }
    public string Country { get; set; } = String.Empty;
}
```

The following code binds the request body to a `Stream`:

C#

```
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
```

```
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});
```

The following code shows the complete `Program.cs` file:

C#

```
using System.Threading.Channels;
using BackgroundQueueService;

var builder = WebApplication.CreateBuilder(args);
// The max memory to use for the upload endpoint on this instance.
var maxMemory = 500 * 1024 * 1024;

// The max size of a single message, staying below the default LOH size of
85K.
var maxMessageSize = 80 * 1024;

// The max size of the queue based on those restrictions
var maxQueueSize = maxMemory / maxMessageSize;
```

```
// Create a channel to send data to the background queue.
builder.Services.AddSingleton<Channel<ReadOnlyMemory<byte>>>((_) =>
                    Channel.CreateBounded<ReadOnlyMemory<byte>>
(maxQueueSize));

// Create a background queue service.
builder.Services.AddHostedService<BackgroundQueue>();
var app = builder.Build();

// curl --request POST 'https://localhost:<port>/register' --header
'Content-Type: application/json' --data-raw '{ "Name":"Samson", "Age": 23,
"Country":"Nigeria" }'
// curl --request POST "https://localhost:<port>/register" --header
"Content-Type: application/json" --data-raw "{ \"Name\":\"Samson\", \"Age\":
23, \"Country\":\"Nigeria\" }"
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});

app.Run();
```

- When reading data, the `Stream` is the same object as `HttpRequest.Body`.
- The request body isn't buffered by default. After the body is read, it's not rewindable. The stream can't be read multiple times.
- The `Stream` and `PipeReader` aren't usable outside of the minimal action handler as the underlying buffers will be disposed or reused.

File uploads using IFormFile and IFormFileCollection

The following code uses IFormFile and IFormFileCollection to upload file:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapPost("/upload", async (IFormFile file) =>
{
    var tempFile = Path.GetTempFileName();
    app.Logger.LogInformation(tempFile);
    using var stream = File.OpenWrite(tempFile);
    await file.CopyToAsync(stream);
});

app.MapPost("/upload_many", async (IFormFileCollection myFiles) =>
{
    foreach (var file in myFiles)
    {
        var tempFile = Path.GetTempFileName();
        app.Logger.LogInformation(tempFile);
        using var stream = File.OpenWrite(tempFile);
        await file.CopyToAsync(stream);
    }
});

app.Run();
```

Authenticated file upload requests are supported using an Authorization header, a client certificate, or a cookie header.

Binding to forms with IFormCollection, IFormFile, and IFormFileCollection

Binding from form-based parameters using IFormCollection, IFormFile, and IFormFileCollection is supported. OpenAPI metadata is inferred for form parameters to support integration with Swagger UI.

The following code uploads files using inferred binding from the `IFormFile` type:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();
app.UseAntiforgery();

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
    """;
```

```
    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});

app.Run();
```

Warning: When implementing forms, the app ***must prevent*** Cross-Site Request Forgery (XSRF/CSRF) attacks. In the preceding code, the IAntiforgery service is used to prevent XSRF attacks by generating and validation an anti-forgery token:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();
app.UseAntiforgery();

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
```

```
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
    """;

    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});

app.Run();
```

For more information on XSRF attacks, see Antiforgery with Minimal APIs

For more information, see Form binding in minimal APIs ↗;

Bind to collections and complex types from forms

Binding is supported for:

- Collections, for example List and Dictionary
- Complex types, for example, `Todo` or `Project`

The following code shows:

- A minimal endpoint that binds a multi-part form input to a complex object.
- How to use the anti-forgery services to support the generation and validation of anti-forgery tokens.

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
        <html><body>
           <form action="/todo" method="POST" enctype="multipart/form-data">
               <input name="{token.FormFieldName}"
                                type="hidden" value="{token.RequestToken}"
/>
               <input type="text" name="name" />
               <input type="date" name="dueDate" />
               <input type="checkbox" name="isCompleted" value="true" />
               <input type="submit" />
               <input name="isCompleted" type="hidden" value="false" />
           </form>
        </body></html>
    """;
    return Results.Content(html, "text/html");
});

app.MapPost("/todo", async Task<Results<Ok<Todo>, BadRequest<string>>>
               ([FromForm] Todo todo, HttpContext context, IAntiforgery
antiforgery) =>
{
    try
    {
        await antiforgery.ValidateRequestAsync(context);
        return TypedResults.Ok(todo);
    }
    catch (AntiforgeryValidationException e)
    {
        return TypedResults.BadRequest("Invalid anti-forgery token");
    }
});

app.Run();

class Todo
{
    public string Name { get; set; } = string.Empty;
    public bool IsCompleted { get; set; } = false;
    public DateTime DueDate { get; set; } =
```

```
DateTime.Now.Add(TimeSpan.FromDays(1));
}
```

In the preceding code:

- The target parameter ***must*** be annotated with the [FromForm] attribute to disambiguate from parameters that should be read from the JSON body.
- Binding from complex or collection types is ***not*** supported for minimal APIs that are compiled with the Request Delegate Generator.
- The markup shows an additional hidden input with a name of `isCompleted` and a value of `false`. If the `isCompleted` checkbox is checked when the form is submitted, both values `true` and `false` are submitted as values. If the checkbox is unchecked, only the hidden input value `false` is submitted. The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

An example of the form data submitted to the preceding endpoint looks as follows:

txt

```
__RequestVerificationToken:
CfDJ8Bveip67DklJm5vI2PF2VOUZ594RC8kcGWpTnVV17zCLZi1yrs-
CSz426ZRRrQnEJ0gybB0AD7hTU-0EGJXDU-OaJaktgAtWLIaaEWMOWCkoxYYm-
9U9eLV7INSUrQ6yBHqdMEE_aJpD4AI72gYiCqc
name: Walk the dog
dueDate: 2024-04-06
isCompleted: true
isCompleted: false
```

Bind arrays and string values from headers and query strings

The following code demonstrates binding query strings to an array of primitive types, string arrays, and StringValues:

C#

```
// Bind query string values to a primitive type array.
// GET  /tags?q=1&q=2&q=3
app.MapGet("/tags", (int[] q) =>
                      $"tag1: {q[0]} , tag2: {q[1]}, tag3: {q[2]}");

// Bind to a string array.
// GET /tags2?names=john&names=jack&names=jane
```

```
app.MapGet("/tags2", (string[] names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");

// Bind to StringValues.
// GET /tags3?names=john&names=jack&names=jane
app.MapGet("/tags3", (StringValues names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");
```

Binding query strings or header values to an array of complex types is supported when the type has `TryParse` implemented. The following code binds to a string array and returns all the items with the specified tags:

C#

```
// GET /todoitems/tags?tags=home&tags=work
app.MapGet("/todoitems/tags", async (Tag[] tags, TodoDb db) =>
{
    return await db.Todos
        .Where(t => tags.Select(i => i.Name).Contains(t.Tag.Name))
        .ToListAsync();
});
```

The following code shows the model and the required `TryParse` implementation:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    // This is an owned entity.
    public Tag Tag { get; set; } = new();
}

[Owned]
public class Tag
{
    public string? Name { get; set; } = "n/a";

    public static bool TryParse(string? name, out Tag tag)
    {
        if (name is null)
        {
            tag = default!;
            return false;
        }

        tag = new Tag { Name = name };
        return true;
```

```
    }
}
```

The following code binds to an `int` array:

C#

```
// GET /todoitems/query-string-ids?ids=1&ids=3
app.MapGet("/todoitems/query-string-ids", async (int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
        .ToListAsync();
});
```

To test the preceding code, add the following endpoint to populate the database with `Todo` items:

C#

```
// POST /todoitems/batch
app.MapPost("/todoitems/batch", async (Todo[] todos, TodoDb db) =>
{
    await db.Todos.AddRangeAsync(todos);
    await db.SaveChangesAsync();

    return Results.Ok(todos);
});
```

Use a tool like HttpRepl to pass the following data to the previous endpoint:

C#

```
[
    {
        "id": 1,
        "name": "Have Breakfast",
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 2,
        "name": "Have Lunch",
        "isComplete": true,
        "tag": {
            "name": "work"
        }
    },
```

```
    {
        "id": 3,
        "name": "Have Supper",
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 4,
        "name": "Have Snacks",
        "isComplete": true,
        "tag": {
            "name": "N/A"
        }
    }
]
```

The following code binds to the header key `X-Todo-Id` and returns the `Todo` items with matching `Id` values:

C#

```
// GET /todoitems/header-ids
// The keys of the headers should all be X-Todo-Id with different values
app.MapGet("/todoitems/header-ids", async ([FromHeader(Name = "X-Todo-Id")]
int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
        .ToListAsync();
});
```

> ⓘ Note
>
> When binding a `string[]` from a query string, the absence of any matching query string value will result in an empty array instead of a null value.

Parameter binding for argument lists with [AsParameters]

AsParametersAttribute enables simple parameter binding to types and not complex or recursive model binding.

Consider the following code:

C#

```
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.Select(x => new TodoItemDTO(x)).ToListAsync());

app.MapGet("/todoitems/{id}",
                            async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
// Remaining code removed for brevity.
```

Consider the following `GET` endpoint:

C#

```
app.MapGet("/todoitems/{id}",
                            async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following `struct` can be used to replace the preceding highlighted parameters:

C#

```
struct TodoItemRequest
{
    public int Id { get; set; }
    public TodoDb Db { get; set; }
}
```

The refactored `GET` endpoint uses the preceding `struct` with the AsParameters attribute:

C#

```
app.MapGet("/ap/todoitems/{id}",
                                async ([AsParameters] TodoItemRequest
request) =>
```

```
    await request.Db.Todos.FindAsync(request.Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following code shows additional endpoints in the app:

C#

```
app.MapPost("/todoitems", async (TodoItemDTO Dto, TodoDb Db) =>
{
    var todoItem = new Todo
    {
        IsComplete = Dto.IsComplete,
        Name = Dto.Name
    };

    Db.Todos.Add(todoItem);
    await Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});

app.MapPut("/todoitems/{id}", async (int Id, TodoItemDTO Dto, TodoDb Db) =>
{
    var todo = await Db.Todos.FindAsync(Id);

    if (todo is null) return Results.NotFound();

    todo.Name = Dto.Name;
    todo.IsComplete = Dto.IsComplete;

    await Db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int Id, TodoDb Db) =>
{
    if (await Db.Todos.FindAsync(Id) is Todo todo)
    {
        Db.Todos.Remove(todo);
        await Db.SaveChangesAsync();
        return Results.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following classes are used to refactor the parameter lists:

C#

```
class CreateTodoItemRequest
{
    public TodoItemDTO Dto { get; set; } = default!;
    public TodoDb Db { get; set; } = default!;
}

class EditTodoItemRequest
{
    public int Id { get; set; }
    public TodoItemDTO Dto { get; set; } = default!;
    public TodoDb Db { get; set; } = default!;
}
```

The following code shows the refactored endpoints using `AsParameters` and the preceding `struct` and classes:

C#

```
app.MapPost("/ap/todoitems", async ([AsParameters] CreateTodoItemRequest
request) =>
{
    var todoItem = new Todo
    {
        IsComplete = request.Dto.IsComplete,
        Name = request.Dto.Name
    };

    request.Db.Todos.Add(todoItem);
    await request.Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});

app.MapPut("/ap/todoitems/{id}", async ([AsParameters] EditTodoItemRequest
request) =>
{
    var todo = await request.Db.Todos.FindAsync(request.Id);

    if (todo is null) return Results.NotFound();

    todo.Name = request.Dto.Name;
    todo.IsComplete = request.Dto.IsComplete;

    await request.Db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/ap/todoitems/{id}", async ([AsParameters] TodoItemRequest
```

```
request) =>
{
    if (await request.Db.Todos.FindAsync(request.Id) is Todo todo)
    {
        request.Db.Todos.Remove(todo);
        await request.Db.SaveChangesAsync();
        return Results.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following record types can be used to replace the preceding parameters:

C#

```
record TodoItemRequest(int Id, TodoDb Db);
record CreateTodoItemRequest(TodoItemDTO Dto, TodoDb Db);
record EditTodoItemRequest(int Id, TodoItemDTO Dto, TodoDb Db);
```

Using a `struct` with `AsParameters` can be more performant than using a `record` type.

The complete sample code in the AspNetCore.Docs.Samples repository.

Custom Binding

There are two ways to customize parameter binding:

1. For route, query, and header binding sources, bind custom types by adding a static `TryParse` method for the type.
2. Control the binding process by implementing a `BindAsync` method on a type.

TryParse

`TryParse` has two APIs:

C#

```
public static bool TryParse(string value, out T result);
public static bool TryParse(string value, IFormatProvider provider, out T
result);
```

The following code displays `Point: 12.3, 10.1` with the URI `/map?Point=12.3,10.1`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// GET /map?Point=12.3,10.1
app.MapGet("/map", (Point point) => $"Point: {point.X}, {point.Y}");

app.Run();

public class Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public static bool TryParse(string? value, IFormatProvider? provider,
                                out Point? point)
    {
        // Format is "(12.3,10.1)"
        var trimmedValue = value?.TrimStart('(').TrimEnd(')');
        var segments = trimmedValue?.Split(',',
                StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries);
        if (segments?.Length == 2
            && double.TryParse(segments[0], out var x)
            && double.TryParse(segments[1], out var y))
        {
            point = new Point { X = x, Y = y };
            return true;
        }

        point = null;
        return false;
    }
}
```

BindAsync

`BindAsync` has the following APIs:

C#

```
public static ValueTask<T?> BindAsync(HttpContext context, ParameterInfo
parameter);
public static ValueTask<T?> BindAsync(HttpContext context);
```

The following code displays `SortBy:xyz, SortDirection:Desc, CurrentPage:99` with the URI `/products?SortBy=xyz&SortDir=Desc&Page=99`:

C#

```
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// GET /products?SortBy=xyz&SortDir=Desc&Page=99
app.MapGet("/products", (PagingData pageData) => $"SortBy:{pageData.SortBy},
" +
       $"SortDirection:{pageData.SortDirection}, CurrentPage:
{pageData.CurrentPage}");

app.Run();

public class PagingData
{
    public string? SortBy { get; init; }
    public SortDirection SortDirection { get; init; }
    public int CurrentPage { get; init; } = 1;

    public static ValueTask<PagingData?> BindAsync(HttpContext context,
                                                   ParameterInfo parameter)
    {
        const string sortByKey = "sortBy";
        const string sortDirectionKey = "sortDir";
        const string currentPageKey = "page";

        Enum.TryParse<SortDirection>
(context.Request.Query[sortDirectionKey],
                                     ignoreCase: true, out var
sortDirection);
        int.TryParse(context.Request.Query[currentPageKey], out var page);
        page = page == 0 ? 1 : page;

        var result = new PagingData
        {
            SortBy = context.Request.Query[sortByKey],
            SortDirection = sortDirection,
            CurrentPage = page
        };

        return ValueTask.FromResult<PagingData?>(result);
    }
}

public enum SortDirection
{
    Default,
    Asc,
    Desc
}
```

Binding failures

When binding fails, the framework logs a debug message and returns various status codes to the client depending on the failure mode.

Expand table

Failure mode	Nullable Parameter Type	Binding Source	Status code
`{ParameterType}.TryParse` returns `false`	yes	route/query/header	400
`{ParameterType}.BindAsync` returns `null`	yes	custom	400
`{ParameterType}.BindAsync` throws	doesn't matter	custom	500
Failure to deserialize JSON body	doesn't matter	body	400
Wrong content type (not `application/json`)	doesn't matter	body	415

Binding Precedence

The rules for determining a binding source from a parameter:

1. Explicit attribute defined on parameter (From* attributes) in the following order:
   a. Route values: [FromRoute]
   b. Query string: [FromQuery]
   c. Header: [FromHeader]
   d. Body: [FromBody]
   e. Form: [FromForm]
   f. Service: [FromServices]
   g. Parameter values: [AsParameters]
2. Special types
   a. HttpContext
   b. HttpRequest (HttpContext.Request)
   c. HttpResponse (HttpContext.Response)
   d. ClaimsPrincipal (HttpContext.User)
   e. CancellationToken (HttpContext.RequestAborted)
   f. IFormCollection (HttpContext.Request.Form)
   g. IFormFileCollection (HttpContext.Request.Form.Files)
   h. IFormFile (HttpContext.Request.Form.Files[paramName])
   i. Stream (HttpContext.Request.Body)
   j. PipeReader (HttpContext.Request.BodyReader)

3. Parameter type has a valid static BindAsync method.
4. Parameter type is a string or has a valid static TryParse method.
   a. If the parameter name exists in the route template for example, `app.Map("/todo/{id}", (int id) => {});`, then it's bound from the route.
   b. Bound from the query string.
5. If the parameter type is a service provided by dependency injection, it uses that service as the source.
6. The parameter is from the body.

Configure JSON deserialization options for body binding

The body binding source uses System.Text.Json for deserialization. It is ***not*** possible to change this default, but JSON serialization and deserialization options can be configured.

Configure JSON deserialization options globally

Options that apply globally for an app can be configured by invoking ConfigureHttpJsonOptions. The following example includes public fields and formats JSON output.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options => {
    options.SerializerOptions.WriteIndented = true;
    options.SerializerOptions.IncludeFields = true;
});

var app = builder.Build();

app.MapPost("/", (Todo todo) => {
    if (todo is not null) {
        todo.Name = todo.NameField;
    }
    return todo;
});

app.Run();

class Todo {
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
```

```
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "nameField":"Walk dog",
//    "isComplete":false
// }
```

Since the sample code configures both serialization and deserialization, it can read `NameField` and include `NameField` in the output JSON.

Configure JSON deserialization options for an endpoint

ReadFromJsonAsync has overloads that accept a JsonSerializerOptions object. The following example includes public fields and formats JSON output.

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web) {
    IncludeFields = true,
    WriteIndented = true
};

app.MapPost("/", async (HttpContext context) => {
    if (context.Request.HasJsonContentType()) {
        var todo = await context.Request.ReadFromJsonAsync<Todo>(options);
        if (todo is not null) {
            todo.Name = todo.NameField;
        }
        return Results.Ok(todo);
    }
    else {
        return Results.BadRequest();
    }
});

app.Run();

class Todo
{
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
```

```
// If the request body contains the following JSON:
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "isComplete":false
// }
```

Since the preceding code applies the customized options only to deserialization, the output JSON excludes `NameField`.

Read the request body

Read the request body directly using a HttpContext or HttpRequest parameter:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapPost("/uploadstream", async (IConfiguration config, HttpRequest
request) =>
{
    var filePath = Path.Combine(config["StoredFilesPath"],
Path.GetRandomFileName());

    await using var writeStream = File.Create(filePath);
    await request.BodyReader.CopyToAsync(writeStream);
});

app.Run();
```

The preceding code:

- Accesses the request body using HttpRequest.BodyReader.
- Copies the request body to a local file.

Responses

Route handlers support the following types of return values:

1. `IResult` based - This includes `Task<IResult>` and `ValueTask<IResult>`
2. `string` - This includes `Task<string>` and `ValueTask<string>`
3. `T` (Any other type) - This includes `Task<T>` and `ValueTask<T>`

Expand table

Return value	Behavior	Content-Type
`IResult`	The framework calls IResult.ExecuteAsync	Decided by the `IResult` implementation
`string`	The framework writes the string directly to the response	`text/plain`
`T` (Any other type)	The framework JSON-serializes the response	`application/json`

For a more in-depth guide to route handler return values see Create responses in Minimal API applications

Example return values

string return values

C#

```
app.MapGet("/hello", () => "Hello World");
```

JSON return values

C#

```
app.MapGet("/hello", () => new { Message = "Hello World" });
```

Return TypedResults

The following code returns a TypedResults:

C#

```
app.MapGet("/hello", () => TypedResults.Ok(new Message() {  Text = "Hello World!" }));
```

Returning `TypedResults` is preferred to returning Results. For more information, see TypedResults vs Results.

IResult return values

```
C#
```

```
app.MapGet("/hello", () => Results.Ok(new { Message = "Hello World" }));
```

The following example uses the built-in result types to customize the response:

```
C#
```

```
app.MapGet("/api/todoitems/{id}", async (int id, TodoDb db) =>
         await db.Todos.FindAsync(id)
         is Todo todo
         ? Results.Ok(todo)
         : Results.NotFound())
   .Produces<Todo>(StatusCodes.Status200OK)
   .Produces(StatusCodes.Status404NotFound);
```

JSON

```
C#
```

```
app.MapGet("/hello", () => Results.Json(new { Message = "Hello World" }));
```

Custom Status Code

```
C#
```

```
app.MapGet("/405", () => Results.StatusCode(405));
```

Text

```
C#
```

```
app.MapGet("/text", () => Results.Text("This is some text"));
```

Stream

```
C#
```

```
var proxyClient = new HttpClient();
app.MapGet("/pokemon", async () =>
```

```
{
    var stream = await
proxyClient.GetStreamAsync("http://contoso/pokedex.json");
    // Proxy the response as JSON
    return Results.Stream(stream, "application/json");
});
```

See Create responses in Minimal API applications for more examples.

Redirect

C#

```
app.MapGet("/old-path", () => Results.Redirect("/new-path"));
```

File

C#

```
app.MapGet("/download", () => Results.File("myfile.text"));
```

Built-in results

Common result helpers exist in the Results and TypedResults static classes. Returning `TypedResults` is preferred to returning `Results`. For more information, see TypedResults vs Results.

Customizing results

Applications can control responses by implementing a custom IResult type. The following code is an example of an HTML result type:

C#

```
using System.Net.Mime;
using System.Text;
static class ResultsExtensions
{
    public static IResult Html(this IResultExtensions resultExtensions,
string html)
    {
        ArgumentNullException.ThrowIfNull(resultExtensions);

        return new HtmlResult(html);
```

```
    }
}

class HtmlResult : IResult
{
    private readonly string _html;

    public HtmlResult(string html)
    {
        _html = html;
    }

    public Task ExecuteAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Html;
        httpContext.Response.ContentLength =
Encoding.UTF8.GetByteCount(_html);
        return httpContext.Response.WriteAsync(_html);
    }
}
```

We recommend adding an extension method to Microsoft.AspNetCore.Http.IResultExtensions to make these custom results more discoverable.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/html", () => Results.Extensions.Html(@$"<!doctype html>
<html>
    <head><title>miniHTML</title></head>
    <body>
        <h1>Hello World</h1>
        <p>The time on the server is {DateTime.Now:O}</p>
    </body>
</html>"));

app.Run();
```

Typed results

The IResult interface can represent values returned from minimal APIs that don't utilize the implicit support for JSON serializing the returned object to the HTTP response. The static Results class is used to create varying `IResult` objects that represent different types of responses. For example, setting the response status code or redirecting to another URL.

The types implementing `IResult` are public, allowing for type assertions when testing. For example:

C#

```
[TestClass()]
public class WeatherApiTests
{
    [TestMethod()]
    public void MapWeatherApiTest()
    {
        var result = WeatherApi.GetAllWeathers();
        Assert.IsInstanceOfType(result, typeof(Ok<WeatherForecast[]>));
    }
}
```

You can look at the return types of the corresponding methods on the static TypedResults class to find the correct public `IResult` type to cast to.

See Create responses in Minimal API applications for more examples.

Filters

For more information, see Filters in Minimal API apps.

Authorization

Routes can be protected using authorization policies. These can be declared via the [Authorize] attribute or by using the RequireAuthorization method:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRPauth.Data;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthorization(o => o.AddPolicy("AdminsOnly",
                                  b => b.RequireClaim("admin", "true")));

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
```

```
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/auth", [Authorize] () => "This endpoint requires
authorization.");
app.MapGet("/", () => "This endpoint doesn't require authorization.");
app.MapGet("/Identity/Account/Login", () => "Sign in page at this
endpoint.");

app.Run();
```

The preceding code can be written with RequireAuthorization:

C#

```
app.MapGet("/auth", () => "This endpoint requires authorization")
   .RequireAuthorization();
```

The following sample uses policy-based authorization:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRPauth.Data;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthorization(o => o.AddPolicy("AdminsOnly",
                                    b => b.RequireClaim("admin", "true")));

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/admin", [Authorize("AdminsOnly")] () =>
                              "The /admin endpoint is for admins only.");
```

```
app.MapGet("/admin2", () => "The /admin2 endpoint is for admins only.")
   .RequireAuthorization("AdminsOnly");

app.MapGet("/", () => "This endpoint doesn't require authorization.");
app.MapGet("/Identity/Account/Login", () => "Sign in page at this
endpoint.");

app.Run();
```

Allow unauthenticated users to access an endpoint

The [AllowAnonymous] allows unauthenticated users to access endpoints:

C#

```
app.MapGet("/login", [AllowAnonymous] () => "This endpoint is for all
roles.");


app.MapGet("/login2", () => "This endpoint also for all roles.")
   .AllowAnonymous();
```

CORS

Routes can be CORS enabled using CORS policies. CORS can be declared via the [EnableCors] attribute or by using the RequireCors method. The following samples enable CORS:

C#

```
const string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      builder =>
                      {
                          builder.WithOrigins("http://example.com",
                                              "http://www.contoso.com");
                      });
});

var app = builder.Build();
app.UseCors();
```

```
app.MapGet("/",() => "Hello CORS!");

app.Run();
```

C#

```
using Microsoft.AspNetCore.Cors;

const string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      builder =>
                      {
                          builder.WithOrigins("http://example.com",
                                              "http://www.contoso.com");
                      });
});

var app = builder.Build();
app.UseCors();

app.MapGet("/cors", [EnableCors(MyAllowSpecificOrigins)] () =>
                           "This endpoint allows cross origin requests!");
app.MapGet("/cors2", () => "This endpoint allows cross origin requests!")
                     .RequireCors(MyAllowSpecificOrigins);

app.Run();
```

For more information, see Enable Cross-Origin Requests (CORS) in ASP.NET Core

ValidateScopes and ValidateOnBuild

ValidateScopes and ValidateOnBuild are enabled by default in the Development environment but disabled in other environments.

When `ValidateOnBuild` is `true`, the DI container validates the service configuration at build time. If the service configuration is invalid, the build fails at app startup, rather than at runtime when the service is requested.

When `ValidateScopes` is `true`, the DI container validates that a scoped service isn't resolved from the root scope. Resolving a scoped service from the root scope can result in a memory leak because the service is retained in memory longer than the scope of the request.

`ValidateScopes` and `ValidateOnBuild` are false by default in non-Development modes for performance reasons.

The following code shows `ValidateScopes` is enabled by default in development mode but disabled in release mode:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<MyScopedService>();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    Console.WriteLine("Development environment");
}
else
{
    Console.WriteLine("Release environment");
}

app.MapGet("/", context =>
{
    // Intentionally getting service provider from app, not from the request
    // This causes an exception from attempting to resolve a scoped service
    // outside of a scope.
    // Throws System.InvalidOperationException:
    // 'Cannot resolve scoped service 'MyScopedService' from root provider.'
    var service = app.Services.GetRequiredService<MyScopedService>();
    return context.Response.WriteAsync("Service resolved");
});

app.Run();

public class MyScopedService { }
```

The following code shows `ValidateOnBuild` is enabled by default in development mode but disabled in release mode:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<MyScopedService>();
builder.Services.AddScoped<AnotherService>();

// System.AggregateException: 'Some services are not able to be constructed
(Error
// while validating the service descriptor 'ServiceType: AnotherService
```

```
Lifetime:
// Scoped ImplementationType: AnotherService': Unable to resolve service for
type
// 'BrokenService' while attempting to activate 'AnotherService'.)'
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    Console.WriteLine("Development environment");
}
else
{
    Console.WriteLine("Release environment");
}

app.MapGet("/", context =>
{
    var service =
context.RequestServices.GetRequiredService<MyScopedService>();
    return context.Response.WriteAsync("Service resolved correctly!");
});

app.Run();

public class MyScopedService { }

public class AnotherService
{
    public AnotherService(BrokenService brokenService) { }
}

public class BrokenService { }
```

The following code disables `ValidateScopes` and `ValidateOnBuild` in `Development`:

C#

```
var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsDevelopment())
{
    Console.WriteLine("Development environment");
    // Doesn't detect the validation problems because ValidateScopes is
false.
    builder.Host.UseDefaultServiceProvider(options =>
    {
        options.ValidateScopes = false;
        options.ValidateOnBuild = false;
    });
}
```

See also

- Minimal APIs quick reference
- Work with OpenAPI documents
- Create responses in Minimal API applications
- Filters in Minimal API apps
- Handle errors in minimal APIs
- Authentication and authorization in minimal APIs
- Test Minimal API apps
- Short-circuit routing
- Identity API endpoints
- Keyed service dependency injection container support
- A look behind the scenes of minimal API endpoints
- Organizing ASP.NET Core Minimal APIs
- Fluent validation discussion on GitHub

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

WebApplication and WebApplicationBuilder in Minimal API apps

Article • 10/25/2023

`WebApplication`

The following code is generated by an ASP.NET Core template:

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding code can be created via `dotnet new web` on the command line or selecting the Empty Web template in Visual Studio.

The following code creates a WebApplication (`app`) without explicitly creating a WebApplicationBuilder:

```
C#

var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run();
```

WebApplication.Create initializes a new instance of the WebApplication class with preconfigured defaults.

Working with ports

When a web app is created with Visual Studio or `dotnet new`, a `Properties/launchSettings.json` file is created that specifies the ports the app responds to. In the port setting samples that follow, running the app from Visual Studio returns an

error dialog `Unable to connect to web server 'AppName'`. Visual Studio returns an error because it's expecting the port specified in `Properties/launchSettings.json`, but the app is using the port specified by `app.Run("http://localhost:3000")`. Run the following port changing samples from the command line.

The following sections set the port the app responds to.

C#

```
var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run("http://localhost:3000");
```

In the preceding code, the app responds to port `3000`.

Multiple ports

In the following code, the app responds to port `3000` and `4000`.

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://localhost:3000");
app.Urls.Add("http://localhost:4000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Set the port from the command line

The following command makes the app respond to port `7777`:

.NET CLI

```
dotnet run --urls="https://localhost:7777"
```

If the Kestrel endpoint is also configured in the `appsettings.json` file, the `appsettings.json` file specified URL is used. For more information, see Kestrel endpoint configuration

Read the port from environment

The following code reads the port from the environment:

```
C#
```

```
var app = WebApplication.Create(args);

var port = Environment.GetEnvironmentVariable("PORT") ?? "3000";

app.MapGet("/", () => "Hello World");

app.Run($"http://localhost:{port}");
```

The preferred way to set the port from the environment is to use the `ASPNETCORE_URLS` environment variable, which is shown in the following section.

Set the ports via the ASPNETCORE_URLS environment variable

The `ASPNETCORE_URLS` environment variable is available to set the port:

```

ASPNETCORE_URLS=http://localhost:3000
```

`ASPNETCORE_URLS` supports multiple URLs:

```

ASPNETCORE_URLS=http://localhost:3000;https://localhost:5000
```

Listen on all interfaces

The following samples demonstrate listening on all interfaces

http://*:3000

```
C#
```

```
var app = WebApplication.Create(args);

app.Urls.Add("http://*:3000");

app.MapGet("/", () => "Hello World");
```

```
app.Run();
```

http://+:3000

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://+:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

`http://0.0.0.0:3000`

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("http://0.0.0.0:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Listen on all interfaces using ASPNETCORE_URLS

The preceding samples can use `ASPNETCORE_URLS`

```
ASPNETCORE_URLS=http://*:3000;https://+:5000;http://0.0.0.0:5005
```

Listen on all interfaces using ASPNETCORE_HTTPS_PORTS

The preceding samples can use `ASPNETCORE_HTTPS_PORTS` and `ASPNETCORE_HTTP_PORTS`.

```
ASPNETCORE_HTTP_PORTS=3000;5005
ASPNETCORE_HTTPS_PORTS=5000
```

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server

Specify HTTPS with development certificate

C#

```
var app = WebApplication.Create(args);

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

For more information on the development certificate, see Trust the ASP.NET Core HTTPS development certificate on Windows and macOS.

Specify HTTPS using a custom certificate

The following sections show how to specify the custom certificate using the `appsettings.json` file and via configuration.

Specify the custom certificate with `appsettings.json`

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "Kestrel": {
    "Certificates": {
      "Default": {
        "Path": "cert.pem",
        "KeyPath": "key.pem"
      }
    }
  }
}
```

Specify the custom certificate via configuration

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure the cert and the key
builder.Configuration["Kestrel:Certificates:Default:Path"] = "cert.pem";
builder.Configuration["Kestrel:Certificates:Default:KeyPath"] = "key.pem";

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Use the certificate APIs

C#

```
using System.Security.Cryptography.X509Certificates;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>
{
    options.ConfigureHttpsDefaults(httpsOptions =>
    {
        var certPath = Path.Combine(builder.Environment.ContentRootPath,
"cert.pem");
        var keyPath = Path.Combine(builder.Environment.ContentRootPath,
"key.pem");

        httpsOptions.ServerCertificate =
X509Certificate2.CreateFromPemFile(certPath,
                                           keyPath);
    });
});

var app = builder.Build();

app.Urls.Add("https://localhost:3000");

app.MapGet("/", () => "Hello World");

app.Run();
```

Read the environment

C#

```
var app = WebApplication.Create(args);

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/oops");
}

app.MapGet("/", () => "Hello World");
app.MapGet("/oops", () => "Oops! An error happened.");

app.Run();
```

For more information using the environment, see Use multiple environments in ASP.NET Core

Configuration

The following code reads from the configuration system:

C#

```
var app = WebApplication.Create(args);

var message = app.Configuration["HelloKey"] ?? "Config failed!";

app.MapGet("/", () => message);

app.Run();
```

For more information, see Configuration in ASP.NET Core

Logging

The following code writes a message to the log on application startup:

C#

```
var app = WebApplication.Create(args);

app.Logger.LogInformation("The app started");

app.MapGet("/", () => "Hello World");

app.Run();
```

For more information, see Logging in .NET Core and ASP.NET Core

Access the Dependency Injection (DI) container

The following code shows how to get services from the DI container during application startup:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddScoped<SampleService>();

var app = builder.Build();

app.MapControllers();

using (var scope = app.Services.CreateScope())
{
    var sampleService =
scope.ServiceProvider.GetRequiredService<SampleService>();
    sampleService.DoSomething();
}

app.Run();
```

The following code shows how to access keys from the DI container using the [FromKeyedServices] attribute:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");

var app = builder.Build();

app.MapGet("/big", ([FromKeyedServices("big")] ICache bigCache) =>
bigCache.Get("date"));

app.MapGet("/small", ([FromKeyedServices("small")] ICache smallCache) =>
smallCache.Get("date"));

app.Run();

public interface ICache
{
```

```
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

For more information on DI, see Dependency injection in ASP.NET Core.

WebApplicationBuilder

This section contains sample code using WebApplicationBuilder.

Change the content root, application name, and environment

The following code sets the content root, application name, and environment:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    ApplicationName = typeof(Program).Assembly.FullName,
    ContentRootPath = Directory.GetCurrentDirectory(),
    EnvironmentName = Environments.Staging,
    WebRootPath = "customwwwroot"
});

Console.WriteLine($"Application Name:
{builder.Environment.ApplicationName}");
Console.WriteLine($"Environment Name:
{builder.Environment.EnvironmentName}");
Console.WriteLine($"ContentRoot Path:
{builder.Environment.ContentRootPath}");
Console.WriteLine($"WebRootPath: {builder.Environment.WebRootPath}");

var app = builder.Build();
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults.

For more information, see ASP.NET Core fundamentals overview

Change the content root, app name, and environment by using environment variables or command line

The following table shows the environment variable and command-line argument used to change the content root, app name, and environment:

Expand table

feature	Environment variable	Command-line argument
Application name	ASPNETCORE_APPLICATIONNAME	--applicationName
Environment name	ASPNETCORE_ENVIRONMENT	--environment
Content root	ASPNETCORE_CONTENTROOT	--contentRoot

Add configuration providers

The following sample adds the INI configuration provider:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddIniFile("appsettings.ini");

var app = builder.Build();
```

For detailed information, see File configuration providers in Configuration in ASP.NET Core.

Read configuration

By default the WebApplicationBuilder reads configuration from multiple sources, including:

- `appSettings.json` and `appSettings.{environment}.json`
- Environment variables
- The command line

For a complete list of configuration sources read, see Default configuration in Configuration in ASP.NET Core.

The following code reads `HelloKey` from configuration and displays the value at the `/` endpoint. If the configuration value is null, "Hello" is assigned to `message`:

C#

```
var builder = WebApplication.CreateBuilder(args);

var message = builder.Configuration["HelloKey"] ?? "Hello";

var app = builder.Build();

app.MapGet("/", () => message);

app.Run();
```

Read the environment

C#

```
var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsDevelopment())
{
    Console.WriteLine($"Running in development.");
}

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Add logging providers

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure JSON logging to the console.
builder.Logging.AddJsonConsole();

var app = builder.Build();

app.MapGet("/", () => "Hello JSON console!");

app.Run();
```

Add services

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add the memory cache services.
builder.Services.AddMemoryCache();

// Add a custom scoped service.
builder.Services.AddScoped<ITodoRepository, TodoRepository>();
var app = builder.Build();
```

Customize the IHostBuilder

Existing extension methods on IHostBuilder can be accessed using the Host property:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Wait 30 seconds for graceful shutdown.
builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout =
TimeSpan.FromSeconds(30));

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Customize the IWebHostBuilder

Extension methods on IWebHostBuilder can be accessed using the WebApplicationBuilder.WebHost property.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Change the HTTP server implemenation to be HTTP.sys based
builder.WebHost.UseHttpSys();

var app = builder.Build();

app.MapGet("/", () => "Hello HTTP.sys");
```

```
app.Run();
```

Change the web root

By default, the web root is relative to the content root in the `wwwroot` folder. Web root is where the static files middleware looks for static files. Web root can be changed with `WebHostOptions`, the command line, or with the UseWebRoot method:

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in webroot
    WebRootPath = "webroot"
});

var app = builder.Build();

app.Run();
```

Custom dependency injection (DI) container

The following example uses Autofac:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());

// Register services directly with Autofac here. Don't
// call builder.Populate(), that happens in AutofacServiceProviderFactory.
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
builder.RegisterModule(new MyApplicationModule()));

var app = builder.Build();
```

Add Middleware

Any existing ASP.NET Core middleware can be configured on the `WebApplication`:

C#

```
var app = WebApplication.Create(args);

// Setup the file server to serve static files.
app.UseFileServer();

app.MapGet("/", () => "Hello World!");

app.Run();
```

For more information, see ASP.NET Core Middleware

Developer exception page

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The developer exception page is enabled in the preconfigured defaults. When the following code is run in the development environment, navigating to `/` renders a friendly page that shows the exception.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () =>
{
    throw new InvalidOperationException("Oops, the '/' route has thrown an exception.");
});

app.Run();
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Route Handlers in Minimal API apps

Article • 07/26/2024

A configured `WebApplication` supports `Map{Verb}` and MapMethods where `{Verb}` is a Pascal-cased HTTP method like `Get`, `Post`, `Put` or `Delete`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "This is a GET");
app.MapPost("/", () => "This is a POST");
app.MapPut("/", () => "This is a PUT");
app.MapDelete("/", () => "This is a DELETE");

app.MapMethods("/options-or-head", new[] { "OPTIONS", "HEAD" },
                           () => "This is an options or head request ");

app.Run();
```

The Delegate arguments passed to these methods are called "route handlers".

Route handlers

Route handlers are methods that execute when the route matches. Route handlers can be a lambda expression, a local function, an instance method or a static method. Route handlers can be synchronous or asynchronous.

Lambda expression

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/inline", () => "This is an inline lambda");

var handler = () => "This is a lambda variable";

app.MapGet("/", handler);

app.Run();
```

Local function

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

string LocalFunction() => "This is local function";

app.MapGet("/", LocalFunction);

app.Run();
```

Instance method

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

var handler = new HelloHandler();

app.MapGet("/", handler.Hello);

app.Run();

class HelloHandler
{
    public string Hello()
    {
        return "Hello Instance method";
    }
}
```

Static method

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", HelloHandler.Hello);

app.Run();

class HelloHandler
{
    public static string Hello()
```

```
    {
        return "Hello static method";
    }
}
```

Endpoint defined outside of `Program.cs`

Minimal APIs don't have to be located in `Program.cs`.

`Program.cs`

C#

```
using MinAPISeparateFile;

var builder = WebApplication.CreateSlimBuilder(args);

var app = builder.Build();

TodoEndpoints.Map(app);

app.Run();
```

`TodoEndpoints.cs`

C#

```
namespace MinAPISeparateFile;

public static class TodoEndpoints
{
    public static void Map(WebApplication app)
    {
        app.MapGet("/", async context =>
        {
            // Get all todo items
            await context.Response.WriteAsJsonAsync(new { Message = "All
todo items" });
        });

        app.MapGet("/{id}", async context =>
        {
            // Get one todo item
            await context.Response.WriteAsJsonAsync(new { Message = "One
todo item" });
        });
    }
}
```

See also Route groups later in this article.

Named endpoints and link generation

Endpoints can be given names in order to generate URLs to the endpoint. Using a named endpoint avoids having to hard code paths in an app:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/hello", () => "Hello named route")
   .WithName("hi");

app.MapGet("/", (LinkGenerator linker) =>
        $"The link to the hello route is {linker.GetPathByName("hi", values:
null)}");

app.Run();
```

The preceding code displays `The link to the hello route is /hello` from the `/` endpoint.

NOTE: Endpoint names are case sensitive.

Endpoint names:

- Must be globally unique.
- Are used as the OpenAPI operation id when OpenAPI support is enabled. For more information, see OpenAPI.

Route Parameters

Route parameters can be captured as part of the route pattern definition:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/users/{userId}/books/{bookId}",
    (int userId, int bookId) => $"The user id is {userId} and book id is
{bookId}");

app.Run();
```

The preceding code returns `The user id is 3 and book id is 7` from the URI `/users/3/books/7`.

The route handler can declare the parameters to capture. When a request is made to a route with parameters declared to capture, the parameters are parsed and passed to the handler. This makes it easy to capture the values in a type safe way. In the preceding code, `userId` and `bookId` are both `int`.

In the preceding code, if either route value cannot be converted to an `int`, an exception is thrown. The GET request `/users/hello/books/3` throws the following exception:

```
BadHttpRequestException: Failed to bind parameter "int userId" from "hello".
```

Wildcard and catch all routes

The following catch all route returns `Routing to hello` from the `/posts/hello' endpoint:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/posts/{*rest}", (string rest) => $"Routing to {rest}");

app.Run();
```

Route constraints

Route constraints constrain the matching behavior of a route.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/todos/{id:int}", (int id) => db.Todos.Find(id));
app.MapGet("/todos/{text}", (string text) => db.Todos.Where(t =>
t.Text.Contains(text));
app.MapGet("/posts/{slug:regex(^[a-z0-9_-]+$)}", (string slug) => $"Post
{slug}");

app.Run();
```

The following table demonstrates the preceding route templates and their behavior:

Expand table

Route Template	Example Matching URI
`/todos/{id:int}`	`/todos/1`
`/todos/{text}`	`/todos/something`
`/posts/{slug:regex(^[a-z0-9_-]+$)}`	`/posts/mypost`

For more information, see Route constraint reference in Routing in ASP.NET Core.

Route groups

The MapGroup extension method helps organize groups of endpoints with a common prefix. It reduces repetitive code and allows for customizing entire groups of endpoints with a single call to methods like RequireAuthorization and WithMetadata which add endpoint metadata.

For example, the following code creates two similar groups of endpoints:

C#

```
app.MapGroup("/public/todos")
    .MapTodosApi()
    .WithTags("Public");

app.MapGroup("/private/todos")
    .MapTodosApi()
    .WithTags("Private")
    .AddEndpointFilterFactory(QueryPrivateTodos)
    .RequireAuthorization();


EndpointFilterDelegate QueryPrivateTodos(EndpointFilterFactoryContext
factoryContext, EndpointFilterDelegate next)
{
    var dbContextIndex = -1;

    foreach (var argument in factoryContext.MethodInfo.GetParameters())
    {
        if (argument.ParameterType == typeof(TodoDb))
        {
            dbContextIndex = argument.Position;
            break;
        }
    }

    // Skip filter if the method doesn't have a TodoDb parameter.
    if (dbContextIndex < 0)
```

```
    {
        return next;
    }

    return async invocationContext =>
    {
        var dbContext = invocationContext.GetArgument<TodoDb>
(dbContextIndex);
        dbContext.IsPrivate = true;

        try
        {
            return await next(invocationContext);
        }
        finally
        {
            // This should only be relevant if you're pooling or otherwise
reusing the DbContext instance.
            dbContext.IsPrivate = false;
        }
    };
}
```

C#

```
public static RouteGroupBuilder MapTodosApi(this RouteGroupBuilder group)
{
    group.MapGet("/", GetAllTodos);
    group.MapGet("/{id}", GetTodo);
    group.MapPost("/", CreateTodo);
    group.MapPut("/{id}", UpdateTodo);
    group.MapDelete("/{id}", DeleteTodo);

    return group;
}
```

In this scenario, you can use a relative address for the `Location` header in the `201 Created` result:

C#

```
public static async Task<Created<Todo>> CreateTodo(Todo todo, TodoDb
database)
{
    await database.AddAsync(todo);
    await database.SaveChangesAsync();

    return TypedResults.Created($"{todo.Id}", todo);
}
```

The first group of endpoints will only match requests prefixed with `/public/todos` and are accessible without any authentication. The second group of endpoints will only match requests prefixed with `/private/todos` and require authentication.

The `QueryPrivateTodos` endpoint filter factory is a local function that modifies the route handler's `TodoDb` parameters to allow to access and store private todo data.

Route groups also support nested groups and complex prefix patterns with route parameters and constraints. In the following example, and route handler mapped to the `user` group can capture the `{org}` and `{group}` route parameters defined in the outer group prefixes.

The prefix can also be empty. This can be useful for adding endpoint metadata or filters to a group of endpoints without changing the route pattern.

C#

```
var all = app.MapGroup("").WithOpenApi();
var org = all.MapGroup("{org}");
var user = org.MapGroup("{user}");
user.MapGet("", (string org, string user) => $"{org}/{user}");
```

Adding filters or metadata to a group behaves the same way as adding them individually to each endpoint before adding any extra filters or metadata that may have been added to an inner group or specific endpoint.

C#

```
var outer = app.MapGroup("/outer");
var inner = outer.MapGroup("/inner");

inner.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/inner group filter");
    return next(context);
});

outer.AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("/outer group filter");
    return next(context);
});

inner.MapGet("/", () => "Hi!").AddEndpointFilter((context, next) =>
{
    app.Logger.LogInformation("MapGet filter");
```

```
    return next(context);
});
```

In the above example, the outer filter will log the incoming request before the inner filter even though it was added second. Because the filters were applied to different groups, the order they were added relative to each other does not matter. The order filters are added does matter if applied to the same group or specific endpoint.

A request to `/outer/inner/` will log the following:

.NET CLI

```
/outer group filter
/inner group filter
MapGet filter
```

Parameter binding

Parameter binding in Minimal API applications describes the rules in detail for how route handler parameters are populated.

Responses

Create responses in Minimal API applications describes in detail how values returned from route handlers are converted into responses.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Parameter Binding in Minimal API apps

Article • 07/26/2024

Parameter binding is the process of converting request data into strongly typed parameters that are expressed by route handlers. A binding source determines where parameters are bound from. Binding sources can be explicit or inferred based on HTTP method and parameter type.

Supported binding sources:

- Route values
- Query string
- Header
- Body (as JSON)
- Form values
- Services provided by dependency injection
- Custom

The following `GET` route handler uses some of these parameter binding sources:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Added as service
builder.Services.AddSingleton<Service>();

var app = builder.Build();

app.MapGet("/{id}", (int id,
                     int page,
                     [FromHeader(Name = "X-CUSTOM-HEADER")] string
customHeader,
                     Service service) => { });

class Service { }
```

The following table shows the relationship between the parameters used in the preceding example and the associated binding sources.

Expand table

Parameter	Binding Source
`id`	route value

Parameter	Binding Source
`page`	query string
`customHeader`	header
`service`	Provided by dependency injection

The HTTP methods `GET`, `HEAD`, `OPTIONS`, and `DELETE` don't implicitly bind from body. To bind from body (as JSON) for these HTTP methods, bind explicitly with [FromBody] or read from the HttpRequest.

The following example POST route handler uses a binding source of body (as JSON) for the `person` parameter:

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapPost("/", (Person person) => { });

record Person(string Name, int Age);
```

The parameters in the preceding examples are all bound from request data automatically. To demonstrate the convenience that parameter binding provides, the following route handlers show how to read request data directly from the request:

C#

```
app.MapGet("/{id}", (HttpRequest request) =>
{
    var id = request.RouteValues["id"];
    var page = request.Query["page"];
    var customHeader = request.Headers["X-CUSTOM-HEADER"];

    // ...
});

app.MapPost("/", async (HttpRequest request) =>
{
    var person = await request.ReadFromJsonAsync<Person>();

    // ...
});
```

Explicit Parameter Binding

Attributes can be used to explicitly declare where parameters are bound from.

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

// Added as service
builder.Services.AddSingleton<Service>();

var app = builder.Build();


app.MapGet("/{id}", ([FromRoute] int id,
                     [FromQuery(Name = "p")] int page,
                     [FromServices] Service service,
                     [FromHeader(Name = "Content-Type")] string contentType)
                     => {});

class Service { }

record Person(string Name, int Age);
```

Expand table

Parameter	Binding Source
`id`	route value with the name `id`
`page`	query string with the name `"p"`
`service`	Provided by dependency injection
`contentType`	header with the name `"Content-Type"`

Explicit binding from form values

The [FromForm] attribute binds form values:

C#

```
app.MapPost("/todos", async ([FromForm] string name,
    [FromForm] Visibility visibility, IFormFile? attachment, TodoDb db) =>
{
    var todo = new Todo
    {
```

```
        Name = name,
        Visibility = visibility
    };

    if (attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();

        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await attachment.CopyToAsync(stream);
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

// Remaining code removed for brevity.
```

An alternative is to use the [AsParameters] attribute with a custom type that has properties annotated with `[FromForm]`. For example, the following code binds from form values to properties of the `NewTodoRequest` record struct:

C#

```
app.MapPost("/ap/todos", async ([AsParameters] NewTodoRequest request,
TodoDb db) =>
{
    var todo = new Todo
    {
        Name = request.Name,
        Visibility = request.Visibility
    };

    if (request.Attachment is not null)
    {
        var attachmentName = Path.GetRandomFileName();

        using var stream = File.Create(Path.Combine("wwwroot",
attachmentName));
        await request.Attachment.CopyToAsync(stream);

        todo.Attachment = attachmentName;
    }

    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});
```

```
// Remaining code removed for brevity.
```

C#

```
public record struct NewTodoRequest([FromForm] string Name,
    [FromForm] Visibility Visibility, IFormFile? Attachment);
```

For more information, see the section on AsParameters later in this article.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Secure binding from IFormFile and IFormFileCollection

Complex form binding is supported using IFormFile and IFormFileCollection using the [FromForm]:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();
app.UseAntiforgery();

// Generate a form with an anti-forgery token and an /upload endpoint.
app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = MyUtils.GenerateHtmlForm(token.FormFieldName,
token.RequestToken!);
    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>, BadRequest<string>>>
    ([FromForm] FileUploadForm fileUploadForm, HttpContext context,
                                                IAntiforgery antiforgery) =>
{
    await MyUtils.SaveFileWithName(fileUploadForm.FileDocument!,
              fileUploadForm.Name!, app.Environment.ContentRootPath);
    return TypedResults.Ok($"Your file with the description:" +
        $" {fileUploadForm.Description} has been uploaded successfully");
});
```

```
app.Run();
```

Parameters bound to the request with `[FromForm]` include an anti-forgery token. The anti-forgery token is validated when the request is processed. For more information, see Antiforgery with Minimal APIs.

For more information, see Form binding in minimal APIs.

The complete sample code is in the AspNetCore.Docs.Samples repository.

Parameter binding with dependency injection

Parameter binding for minimal APIs binds parameters through dependency injection when the type is configured as a service. It's not necessary to explicitly apply the [FromServices] attribute to a parameter. In the following code, both actions return the time:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IDateTime, SystemDateTime>();

var app = builder.Build();

app.MapGet("/",   (               IDateTime dateTime) => dateTime.Now);
app.MapGet("/fs", ([FromServices] IDateTime dateTime) => dateTime.Now);
app.Run();
```

Optional parameters

Parameters declared in route handlers are treated as required:

- If a request matches the route, the route handler only runs if all required parameters are provided in the request.
- Failure to provide all required parameters results in an error.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/products", (int pageNumber) => $"Requesting page
{pageNumber}");
```

```
app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	`BadHttpRequestException`: Required parameter "int pageNumber" wasn't provided from query string.
`/products/1`	HTTP 404 error, no matching route

To make `pageNumber` optional, define the type as optional or provide a default value:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

string ListProducts(int pageNumber = 1) => $"Requesting page {pageNumber}";

app.MapGet("/products2", ListProducts);

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	3 returned
`/products`	1 returned
`/products2`	1 returned

The preceding nullable and default value applies to all sources:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
```

```
app.MapPost("/products", (Product? product) => { });

app.Run();
```

The preceding code calls the method with a null product if no request body is sent.

NOTE: If invalid data is provided and the parameter is nullable, the route handler is ***not*** run.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/products", (int? pageNumber) => $"Requesting page {pageNumber
?? 1}");

app.Run();
```

Expand table

URI	result
`/products?pageNumber=3`	`3` returned
`/products`	`1` returned
`/products?pageNumber=two`	`BadHttpRequestException`: Failed to bind parameter `"Nullable<int> pageNumber"` from "two".
`/products/two`	HTTP 404 error, no matching route

See the Binding Failures section for more information.

Special types

The following types are bound without explicit attributes:

- HttpContext: The context which holds all the information about the current HTTP request or response:

  C#

  ```
  app.MapGet("/", (HttpContext context) =>
  context.Response.WriteAsync("Hello World"));
  ```

- HttpRequest and HttpResponse: The HTTP request and HTTP response:

```
C#

app.MapGet("/", (HttpRequest request, HttpResponse response) =>
    response.WriteAsync($"Hello World {request.Query["name"]}"));
```

- CancellationToken: The cancellation token associated with the current HTTP request:

```
C#

app.MapGet("/", async (CancellationToken cancellationToken) =>
    await MakeLongRunningRequestAsync(cancellationToken));
```

- ClaimsPrincipal: The user associated with the request, bound from HttpContext.User:

```
C#

app.MapGet("/", (ClaimsPrincipal user) => user.Identity.Name);
```

Bind the request body as a `Stream` or `PipeReader`

The request body can bind as a Stream or PipeReader to efficiently support scenarios where the user has to process data and:

- Store the data to blob storage or enqueue the data to a queue provider.
- Process the stored data with a worker process or cloud function.

For example, the data might be enqueued to Azure Queue storage or stored in Azure Blob storage.

The following code implements a background queue:

```
C#

using System.Text.Json;
using System.Threading.Channels;

namespace BackgroundQueueService;

class BackgroundQueue : BackgroundService
{
    private readonly Channel<ReadOnlyMemory<byte>> _queue;
    private readonly ILogger<BackgroundQueue> _logger;
```

```
    public BackgroundQueue(Channel<ReadOnlyMemory<byte>> queue,
                           ILogger<BackgroundQueue> logger)
    {
        _queue = queue;
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        await foreach (var dataStream in
_queue.Reader.ReadAllAsync(stoppingToken))
        {
            try
            {
                var person = JsonSerializer.Deserialize<Person>
(dataStream.Span)!;
                _logger.LogInformation($"{person.Name} is {person.Age} " +
                                       $"years and from {person.Country}");
            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }
        }
    }
}

class Person
{
    public string Name { get; set; } = String.Empty;
    public int Age { get; set; }
    public string Country { get; set; } = String.Empty;
}
```

The following code binds the request body to a `Stream`:

C#

```
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
```

```
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});
```

The following code shows the complete `Program.cs` file:

C#

```
using System.Threading.Channels;
using BackgroundQueueService;

var builder = WebApplication.CreateBuilder(args);
// The max memory to use for the upload endpoint on this instance.
var maxMemory = 500 * 1024 * 1024;

// The max size of a single message, staying below the default LOH size of
85K.
var maxMessageSize = 80 * 1024;

// The max size of the queue based on those restrictions
var maxQueueSize = maxMemory / maxMessageSize;

// Create a channel to send data to the background queue.
builder.Services.AddSingleton<Channel<ReadOnlyMemory<byte>>>((_) =>
                    Channel.CreateBounded<ReadOnlyMemory<byte>>
(maxQueueSize));

// Create a background queue service.
builder.Services.AddHostedService<BackgroundQueue>();
var app = builder.Build();

// curl --request POST 'https://localhost:<port>/register' --header
```

```
'Content-Type: application/json' --data-raw '{ "Name":"Samson", "Age": 23,
"Country":"Nigeria" }'
// curl --request POST "https://localhost:<port>/register" --header
"Content-Type: application/json" --data-raw "{ \"Name\":\"Samson\", \"Age\":
23, \"Country\":\"Nigeria\" }"
app.MapPost("/register", async (HttpRequest req, Stream body,
                                Channel<ReadOnlyMemory<byte>> queue) =>
{
    if (req.ContentLength is not null && req.ContentLength > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // We're not above the message size and we have a content length, or
    // we're a chunked request and we're going to read up to the
maxMessageSize + 1.
    // We add one to the message size so that we can detect when a chunked
request body
    // is bigger than our configured max.
    var readSize = (int?)req.ContentLength ?? (maxMessageSize + 1);

    var buffer = new byte[readSize];

    // Read at least that many bytes from the body.
    var read = await body.ReadAtLeastAsync(buffer, readSize,
throwOnEndOfStream: false);

    // We read more than the max, so this is a bad request.
    if (read > maxMessageSize)
    {
        return Results.BadRequest();
    }

    // Attempt to send the buffer to the background queue.
    if (queue.Writer.TryWrite(buffer.AsMemory(0..read)))
    {
        return Results.Accepted();
    }

    // We couldn't accept the message since we're overloaded.
    return Results.StatusCode(StatusCodes.Status429TooManyRequests);
});

app.Run();
```

- When reading data, the `Stream` is the same object as `HttpRequest.Body`.
- The request body isn't buffered by default. After the body is read, it's not rewindable. The stream can't be read multiple times.
- The `Stream` and `PipeReader` aren't usable outside of the minimal action handler as the underlying buffers will be disposed or reused.

File uploads using IFormFile and IFormFileCollection

The following code uses IFormFile and IFormFileCollection to upload file:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.MapPost("/upload", async (IFormFile file) =>
{
    var tempFile = Path.GetTempFileName();
    app.Logger.LogInformation(tempFile);
    using var stream = File.OpenWrite(tempFile);
    await file.CopyToAsync(stream);
});

app.MapPost("/upload_many", async (IFormFileCollection myFiles) =>
{
    foreach (var file in myFiles)
    {
        var tempFile = Path.GetTempFileName();
        app.Logger.LogInformation(tempFile);
        using var stream = File.OpenWrite(tempFile);
        await file.CopyToAsync(stream);
    }
});

app.Run();
```

Authenticated file upload requests are supported using an Authorization header, a client certificate, or a cookie header.

Binding to forms with IFormCollection, IFormFile, and IFormFileCollection

Binding from form-based parameters using IFormCollection, IFormFile, and IFormFileCollection is supported. OpenAPI metadata is inferred for form parameters to support integration with Swagger UI.

The following code uploads files using inferred binding from the `IFormFile` type:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
```

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();
app.UseAntiforgery();

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
    """;

    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});
```

```
app.Run();
```

Warning: When implementing forms, the app ***must prevent*** Cross-Site Request Forgery (XSRF/CSRF) attacks. In the preceding code, the IAntiforgery service is used to prevent XSRF attacks by generating and validation an anti-forgery token:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();
app.UseAntiforgery();

string GetOrCreateFilePath(string fileName, string filesDirectory =
"uploadFiles")
{
    var directoryPath = Path.Combine(app.Environment.ContentRootPath,
filesDirectory);
    Directory.CreateDirectory(directoryPath);
    return Path.Combine(directoryPath, fileName);
}

async Task UploadFileWithName(IFormFile file, string fileSaveName)
{
    var filePath = GetOrCreateFilePath(fileSaveName);
    await using var fileStream = new FileStream(filePath, FileMode.Create);
    await file.CopyToAsync(fileStream);
}

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
      <html>
        <body>
          <form action="/upload" method="POST" enctype="multipart/form-
data">
            <input name="{token.FormFieldName}" type="hidden" value="
{token.RequestToken}"/>
            <input type="file" name="file" placeholder="Upload an image..."
accept=".jpg,

.jpeg, .png" />
            <input type="submit" />
          </form>
        </body>
      </html>
```

```
    """;

    return Results.Content(html, "text/html");
});

app.MapPost("/upload", async Task<Results<Ok<string>,
   BadRequest<string>>> (IFormFile file, HttpContext context, IAntiforgery
antiforgery) =>
{
    var fileSaveName = Guid.NewGuid().ToString("N") +
Path.GetExtension(file.FileName);
    await UploadFileWithName(file, fileSaveName);
    return TypedResults.Ok("File uploaded successfully!");
});

app.Run();
```

For more information on XSRF attacks, see Antiforgery with Minimal APIs

For more information, see Form binding in minimal APIs ;

Bind to collections and complex types from forms

Binding is supported for:

- Collections, for example List and Dictionary
- Complex types, for example, `Todo` or `Project`

The following code shows:

- A minimal endpoint that binds a multi-part form input to a complex object.
- How to use the anti-forgery services to support the generation and validation of anti-forgery tokens.

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Http.HttpResults;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
```

```
    var token = antiforgery.GetAndStoreTokens(context);
    var html = $"""
        <html><body>
           <form action="/todo" method="POST" enctype="multipart/form-data">
               <input name="{token.FormFieldName}"
                                type="hidden" value="{token.RequestToken}"
/>
               <input type="text" name="name" />
               <input type="date" name="dueDate" />
               <input type="checkbox" name="isCompleted" value="true" />
               <input type="submit" />
               <input name="isCompleted" type="hidden" value="false" />
           </form>
        </body></html>
    """;
    return Results.Content(html, "text/html");
});

app.MapPost("/todo", async Task<Results<Ok<Todo>, BadRequest<string>>>
               ([FromForm] Todo todo, HttpContext context, IAntiforgery
antiforgery) =>
{
    try
    {
        await antiforgery.ValidateRequestAsync(context);
        return TypedResults.Ok(todo);
    }
    catch (AntiforgeryValidationException e)
    {
        return TypedResults.BadRequest("Invalid anti-forgery token");
    }
});

app.Run();

class Todo
{
    public string Name { get; set; } = string.Empty;
    public bool IsCompleted { get; set; } = false;
    public DateTime DueDate { get; set; } =
DateTime.Now.Add(TimeSpan.FromDays(1));
}
```

In the preceding code:

- The target parameter ***must*** be annotated with the [FromForm] attribute to disambiguate from parameters that should be read from the JSON body.
- Binding from complex or collection types is ***not*** supported for minimal APIs that are compiled with the Request Delegate Generator.
- The markup shows an additional hidden input with a name of `isCompleted` and a value of `false`. If the `isCompleted` checkbox is checked when the form is

submitted, both values `true` and `false` are submitted as values. If the checkbox is unchecked, only the hidden input value `false` is submitted. The ASP.NET Core model-binding process reads only the first value when binding to a `bool` value, which results in `true` for checked checkboxes and `false` for unchecked checkboxes.

An example of the form data submitted to the preceding endpoint looks as follows:

```txt
__RequestVerificationToken:
CfDJ8Bveip67DklJm5vI2PF2VOUZ594RC8kcGWpTnVV17zCLZi1yrs-
CSz426ZRRrQnEJ0gybB0AD7hTU-0EGJXDU-OaJaktgAtWLIaaEWMOWCkoxYYm-
9U9eLV7INSUrQ6yBHqdMEE_aJpD4AI72gYiCqc
name: Walk the dog
dueDate: 2024-04-06
isCompleted: true
isCompleted: false
```

Bind arrays and string values from headers and query strings

The following code demonstrates binding query strings to an array of primitive types, string arrays, and StringValues:

```C#
// Bind query string values to a primitive type array.
// GET  /tags?q=1&q=2&q=3
app.MapGet("/tags", (int[] q) =>
                      $"tag1: {q[0]} , tag2: {q[1]}, tag3: {q[2]}");

// Bind to a string array.
// GET /tags2?names=john&names=jack&names=jane
app.MapGet("/tags2", (string[] names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");

// Bind to StringValues.
// GET /tags3?names=john&names=jack&names=jane
app.MapGet("/tags3", (StringValues names) =>
            $"tag1: {names[0]} , tag2: {names[1]}, tag3: {names[2]}");
```

Binding query strings or header values to an array of complex types is supported when the type has `TryParse` implemented. The following code binds to a string array and returns all the items with the specified tags:

C#

```
// GET /todoitems/tags?tags=home&tags=work
app.MapGet("/todoitems/tags", async (Tag[] tags, TodoDb db) =>
{
    return await db.Todos
        .Where(t => tags.Select(i => i.Name).Contains(t.Tag.Name))
        .ToListAsync();
});
```

The following code shows the model and the required `TryParse` implementation:

C#

```
public class Todo
{
    public int Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }

    // This is an owned entity.
    public Tag Tag { get; set; } = new();
}

[Owned]
public class Tag
{
    public string? Name { get; set; } = "n/a";

    public static bool TryParse(string? name, out Tag tag)
    {
        if (name is null)
        {
            tag = default!;
            return false;
        }

        tag = new Tag { Name = name };
        return true;
    }
}
```

The following code binds to an `int` array:

C#

```
// GET /todoitems/query-string-ids?ids=1&ids=3
app.MapGet("/todoitems/query-string-ids", async (int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
```

```
        .ToListAsync();
});
```

To test the preceding code, add the following endpoint to populate the database with `Todo` items:

C#

```
// POST /todoitems/batch
app.MapPost("/todoitems/batch", async (Todo[] todos, TodoDb db) =>
{
    await db.Todos.AddRangeAsync(todos);
    await db.SaveChangesAsync();

    return Results.Ok(todos);
});
```

Use a tool like HttpRepl to pass the following data to the previous endpoint:

C#

```
[
    {
        "id": 1,
        "name": "Have Breakfast",
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 2,
        "name": "Have Lunch",
        "isComplete": true,
        "tag": {
            "name": "work"
        }
    },
    {
        "id": 3,
        "name": "Have Supper",
        "isComplete": true,
        "tag": {
            "name": "home"
        }
    },
    {
        "id": 4,
        "name": "Have Snacks",
        "isComplete": true,
        "tag": {
```

```
            "name": "N/A"
        }
    }
]
```

The following code binds to the header key `X-Todo-Id` and returns the `Todo` items with matching `Id` values:

C#

```
// GET /todoitems/header-ids
// The keys of the headers should all be X-Todo-Id with different values
app.MapGet("/todoitems/header-ids", async ([FromHeader(Name = "X-Todo-Id")]
int[] ids, TodoDb db) =>
{
    return await db.Todos
        .Where(t => ids.Contains(t.Id))
        .ToListAsync();
});
```

> **Note**
>
> When binding a `string[]` from a query string, the absence of any matching query string value will result in an empty array instead of a null value.

Parameter binding for argument lists with [AsParameters]

AsParametersAttribute enables simple parameter binding to types and not complex or recursive model binding.

Consider the following code:

C#

```
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
builder.Services.AddDbContext<TodoDb>(opt =>
opt.UseInMemoryDatabase("TodoList"));
var app = builder.Build();

app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.Select(x => new TodoItemDTO(x)).ToListAsync());

app.MapGet("/todoitems/{id}",
```

```
                                async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
// Remaining code removed for brevity.
```

Consider the following `GET` endpoint:

C#

```
app.MapGet("/todoitems/{id}",
                                async (int Id, TodoDb Db) =>
    await Db.Todos.FindAsync(Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following `struct` can be used to replace the preceding highlighted parameters:

C#

```
struct TodoItemRequest
{
    public int Id { get; set; }
    public TodoDb Db { get; set; }
}
```

The refactored `GET` endpoint uses the preceding `struct` with the AsParameters attribute:

C#

```
app.MapGet("/ap/todoitems/{id}",
                                    async ([AsParameters] TodoItemRequest
request) =>
    await request.Db.Todos.FindAsync(request.Id)
        is Todo todo
            ? Results.Ok(new TodoItemDTO(todo))
            : Results.NotFound());
```

The following code shows additional endpoints in the app:

C#

```
app.MapPost("/todoitems", async (TodoItemDTO Dto, TodoDb Db) =>
{
    var todoItem = new Todo
```

```
    {
        IsComplete = Dto.IsComplete,
        Name = Dto.Name
    };

    Db.Todos.Add(todoItem);
    await Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});

app.MapPut("/todoitems/{id}", async (int Id, TodoItemDTO Dto, TodoDb Db) =>
{
    var todo = await Db.Todos.FindAsync(Id);

    if (todo is null) return Results.NotFound();

    todo.Name = Dto.Name;
    todo.IsComplete = Dto.IsComplete;

    await Db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/todoitems/{id}", async (int Id, TodoDb Db) =>
{
    if (await Db.Todos.FindAsync(Id) is Todo todo)
    {
        Db.Todos.Remove(todo);
        await Db.SaveChangesAsync();
        return Results.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following classes are used to refactor the parameter lists:

C#

```
class CreateTodoItemRequest
{
    public TodoItemDTO Dto { get; set; } = default!;
    public TodoDb Db { get; set; } = default!;
}

class EditTodoItemRequest
{
    public int Id { get; set; }
    public TodoItemDTO Dto { get; set; } = default!;
```

```
    public TodoDb Db { get; set; } = default!;
}
```

The following code shows the refactored endpoints using `AsParameters` and the preceding `struct` and classes:

C#

```
app.MapPost("/ap/todoitems", async ([AsParameters] CreateTodoItemRequest
request) =>
{
    var todoItem = new Todo
    {
        IsComplete = request.Dto.IsComplete,
        Name = request.Dto.Name
    };

    request.Db.Todos.Add(todoItem);
    await request.Db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todoItem.Id}", new
TodoItemDTO(todoItem));
});

app.MapPut("/ap/todoitems/{id}", async ([AsParameters] EditTodoItemRequest
request) =>
{
    var todo = await request.Db.Todos.FindAsync(request.Id);

    if (todo is null) return Results.NotFound();

    todo.Name = request.Dto.Name;
    todo.IsComplete = request.Dto.IsComplete;

    await request.Db.SaveChangesAsync();

    return Results.NoContent();
});

app.MapDelete("/ap/todoitems/{id}", async ([AsParameters] TodoItemRequest
request) =>
{
    if (await request.Db.Todos.FindAsync(request.Id) is Todo todo)
    {
        request.Db.Todos.Remove(todo);
        await request.Db.SaveChangesAsync();
        return Results.Ok(new TodoItemDTO(todo));
    }

    return Results.NotFound();
});
```

The following record types can be used to replace the preceding parameters:

```
C#
```

```
record TodoItemRequest(int Id, TodoDb Db);
record CreateTodoItemRequest(TodoItemDTO Dto, TodoDb Db);
record EditTodoItemRequest(int Id, TodoItemDTO Dto, TodoDb Db);
```

Using a `struct` with `AsParameters` can be more performant than using a `record` type.

The complete sample code in the AspNetCore.Docs.Samples repository.

Custom Binding

There are two ways to customize parameter binding:

1. For route, query, and header binding sources, bind custom types by adding a static `TryParse` method for the type.
2. Control the binding process by implementing a `BindAsync` method on a type.

TryParse

`TryParse` has two APIs:

```
C#
```

```
public static bool TryParse(string value, out T result);
public static bool TryParse(string value, IFormatProvider provider, out T
result);
```

The following code displays `Point: 12.3, 10.1` with the URI `/map?Point=12.3,10.1`:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// GET /map?Point=12.3,10.1
app.MapGet("/map", (Point point) => $"Point: {point.X}, {point.Y}");

app.Run();

public class Point
{
    public double X { get; set; }
    public double Y { get; set; }
```

```
    public static bool TryParse(string? value, IFormatProvider? provider,
                                out Point? point)
    {
        // Format is "(12.3,10.1)"
        var trimmedValue = value?.TrimStart('(').TrimEnd(')');
        var segments = trimmedValue?.Split(',',
                StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries);
        if (segments?.Length == 2
            && double.TryParse(segments[0], out var x)
            && double.TryParse(segments[1], out var y))
        {
            point = new Point { X = x, Y = y };
            return true;
        }

        point = null;
        return false;
    }
}
```

BindAsync

`BindAsync` has the following APIs:

C#

```
public static ValueTask<T?> BindAsync(HttpContext context, ParameterInfo
parameter);
public static ValueTask<T?> BindAsync(HttpContext context);
```

The following code displays `SortBy:xyz, SortDirection:Desc, CurrentPage:99` with the URI `/products?SortBy=xyz&SortDir=Desc&Page=99`:

C#

```
using System.Reflection;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// GET /products?SortBy=xyz&SortDir=Desc&Page=99
app.MapGet("/products", (PagingData pageData) => $"SortBy:{pageData.SortBy},
" +
       $"SortDirection:{pageData.SortDirection}, CurrentPage:
{pageData.CurrentPage}");

app.Run();
```

```
public class PagingData
{
    public string? SortBy { get; init; }
    public SortDirection SortDirection { get; init; }
    public int CurrentPage { get; init; } = 1;

    public static ValueTask<PagingData?> BindAsync(HttpContext context,
                                                   ParameterInfo parameter)
    {
        const string sortByKey = "sortBy";
        const string sortDirectionKey = "sortDir";
        const string currentPageKey = "page";

        Enum.TryParse<SortDirection>
(context.Request.Query[sortDirectionKey],
                                      ignoreCase: true, out var
sortDirection);
        int.TryParse(context.Request.Query[currentPageKey], out var page);
        page = page == 0 ? 1 : page;

        var result = new PagingData
        {
            SortBy = context.Request.Query[sortByKey],
            SortDirection = sortDirection,
            CurrentPage = page
        };

        return ValueTask.FromResult<PagingData?>(result);
    }
}

public enum SortDirection
{
    Default,
    Asc,
    Desc
}
```

Binding failures

When binding fails, the framework logs a debug message and returns various status codes to the client depending on the failure mode.

Expand table

Failure mode	Nullable Parameter Type	Binding Source	Status code
`{ParameterType}.TryParse` returns `false`	yes	route/query/header	400

Failure mode	Nullable Parameter Type	Binding Source	Status code
`{ParameterType}.BindAsync` returns `null`	yes	custom	400
`{ParameterType}.BindAsync` throws	doesn't matter	custom	500
Failure to deserialize JSON body	doesn't matter	body	400
Wrong content type (not `application/json`)	doesn't matter	body	415

Binding Precedence

The rules for determining a binding source from a parameter:

1. Explicit attribute defined on parameter (From* attributes) in the following order:
   a. Route values: [FromRoute]
   b. Query string: [FromQuery]
   c. Header: [FromHeader]
   d. Body: [FromBody]
   e. Form: [FromForm]
   f. Service: [FromServices]
   g. Parameter values: [AsParameters]
2. Special types
   a. HttpContext
   b. HttpRequest (HttpContext.Request)
   c. HttpResponse (HttpContext.Response)
   d. ClaimsPrincipal (HttpContext.User)
   e. CancellationToken (HttpContext.RequestAborted)
   f. IFormCollection (HttpContext.Request.Form)
   g. IFormFileCollection (HttpContext.Request.Form.Files)
   h. IFormFile (HttpContext.Request.Form.Files[paramName])
   i. Stream (HttpContext.Request.Body)
   j. PipeReader (HttpContext.Request.BodyReader)
3. Parameter type has a valid static BindAsync method.
4. Parameter type is a string or has a valid static TryParse method.
   a. If the parameter name exists in the route template for example, `app.Map("/todo/{id}", (int id) => {});`, then it's bound from the route.
   b. Bound from the query string.
5. If the parameter type is a service provided by dependency injection, it uses that service as the source.

6. The parameter is from the body.

Configure JSON deserialization options for body binding

The body binding source uses System.Text.Json for deserialization. It is ***not*** possible to change this default, but JSON serialization and deserialization options can be configured.

Configure JSON deserialization options globally

Options that apply globally for an app can be configured by invoking ConfigureHttpJsonOptions. The following example includes public fields and formats JSON output.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options => {
    options.SerializerOptions.WriteIndented = true;
    options.SerializerOptions.IncludeFields = true;
});

var app = builder.Build();

app.MapPost("/", (Todo todo) => {
    if (todo is not null) {
        todo.Name = todo.NameField;
    }
    return todo;
});

app.Run();

class Todo {
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "nameField":"Walk dog",
```

```
//    "isComplete":false
// }
```

Since the sample code configures both serialization and deserialization, it can read `NameField` and include `NameField` in the output JSON.

Configure JSON deserialization options for an endpoint

ReadFromJsonAsync has overloads that accept a JsonSerializerOptions object. The following example includes public fields and formats JSON output.

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web) {
    IncludeFields = true,
    WriteIndented = true
};

app.MapPost("/", async (HttpContext context) => {
    if (context.Request.HasJsonContentType()) {
        var todo = await context.Request.ReadFromJsonAsync<Todo>(options);
        if (todo is not null) {
            todo.Name = todo.NameField;
        }
        return Results.Ok(todo);
    }
    else {
        return Results.BadRequest();
    }
});

app.Run();

class Todo
{
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
```

```
//    "isComplete":false
// }
```

Since the preceding code applies the customized options only to deserialization, the output JSON excludes `NameField`.

Read the request body

Read the request body directly using a HttpContext or HttpRequest parameter:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapPost("/uploadstream", async (IConfiguration config, HttpRequest
request) =>
{
    var filePath = Path.Combine(config["StoredFilesPath"],
Path.GetRandomFileName());

    await using var writeStream = File.Create(filePath);
    await request.BodyReader.CopyToAsync(writeStream);
});

app.Run();
```

The preceding code:

- Accesses the request body using HttpRequest.BodyReader.
- Copies the request body to a local file.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

How to create responses in Minimal API apps

Article • 06/04/2024

Minimal endpoints support the following types of return values:

1. `string` - This includes `Task<string>` and `ValueTask<string>`.
2. `T` (Any other type) - This includes `Task<T>` and `ValueTask<T>`.
3. `IResult` based - This includes `Task<IResult>` and `ValueTask<IResult>`.

`string` return values

Expand table

Behavior	Content-Type
The framework writes the string directly to the response.	`text/plain`

Consider the following route handler, which returns a `Hello world` text.

C#

```
app.MapGet("/hello", () => "Hello World");
```

The `200` status code is returned with `text/plain` Content-Type header and the following content.

text

```
Hello World
```

`T` (Any other type) return values

Expand table

Behavior	Content-Type
The framework JSON-serializes the response.	`application/json`

Consider the following route handler, which returns an anonymous type containing a `Message` string property.

C#

```
app.MapGet("/hello", () => new { Message = "Hello World" });
```

The `200` status code is returned with `application/json` Content-Type header and the following content.

JSON

```
{"message":"Hello World"}
```

`IResult` return values

Expand table

Behavior	Content-Type
The framework calls IResult.ExecuteAsync.	Decided by the `IResult` implementation.

The `IResult` interface defines a contract that represents the result of an HTTP endpoint. The static Results class and the static TypedResults are used to create various `IResult` objects that represent different types of responses.

TypedResults vs Results

The Results and TypedResults static classes provide similar sets of results helpers. The `TypedResults` class is the *typed* equivalent of the `Results` class. However, the `Results` helpers' return type is IResult, while each `TypedResults` helper's return type is one of the `IResult` implementation types. The difference means that for `Results` helpers a conversion is needed when the concrete type is needed, for example, for unit testing. The implementation types are defined in the Microsoft.AspNetCore.Http.HttpResults namespace.

Returning `TypedResults` rather than `Results` has the following advantages:

- `TypedResults` helpers return strongly typed objects, which can improve code readability, unit testing, and reduce the chance of runtime errors.

- The implementation type automatically provides the response type metadata for OpenAPI to describe the endpoint.

Consider the following endpoint, for which a `200 OK` status code with the expected JSON response is produced.

C#

```
app.MapGet("/hello", () => Results.Ok(new Message() { Text = "Hello World!"
}))
    .Produces<Message>();
```

In order to document this endpoint correctly the extensions method `Produces` is called. However, it's not necessary to call `Produces` if `TypedResults` is used instead of `Results`, as shown in the following code. `TypedResults` automatically provides the metadata for the endpoint.

C#

```
app.MapGet("/hello2", () => TypedResults.Ok(new Message() { Text = "Hello
World!" }));
```

For more information about describing a response type, see OpenAPI support in minimal APIs.

As mentioned previously, when using `TypedResults`, a conversion is not needed. Consider the following minimal API which returns a `TypedResults` class

C#

```
public static async Task<Ok<Todo[]>> GetAllTodos(TodoGroupDbContext
database)
{
    var todos = await database.Todos.ToArrayAsync();
    return TypedResults.Ok(todos);
}
```

The following test checks for the full concrete type:

C#

```
[Fact]
public async Task GetAllReturnsTodosFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();
```

```
    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title 1",
        Description = "Test description 1",
        IsDone = false
    });

    context.Todos.Add(new Todo
    {
        Id = 2,
        Title = "Test title 2",
        Description = "Test description 2",
        IsDone = true
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetAllTodos(context);

    //Assert
    Assert.IsType<Ok<Todo[]>>(result);

    Assert.NotNull(result.Value);
    Assert.NotEmpty(result.Value);
    Assert.Collection(result.Value, todo1 =>
    {
        Assert.Equal(1, todo1.Id);
        Assert.Equal("Test title 1", todo1.Title);
        Assert.False(todo1.IsDone);
    }, todo2 =>
    {
        Assert.Equal(2, todo2.Id);
        Assert.Equal("Test title 2", todo2.Title);
        Assert.True(todo2.IsDone);
    });
}
```

Because all methods on `Results` return `IResult` in their signature, the compiler automatically infers that as the request delegate return type when returning different results from a single endpoint. `TypedResults` requires the use of `Results<T1, TN>` from such delegates.

The following method compiles because both Results.Ok and Results.NotFound are declared as returning `IResult`, even though the actual concrete types of the objects returned are different:

```
C#
```

```
app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
    await db.Todos.FindAsync(id)
        is Todo todo
            ? Results.Ok(todo)
            : Results.NotFound());
```

The following method does not compile, because `TypedResults.Ok` and `TypedResults.NotFound` are declared as returning different types and the compiler won't attempt to infer the best matching type:

C#

```
app.MapGet("/todoitems/{id}", async (int id, TodoDb db) =>
     await db.Todos.FindAsync(id)
     is Todo todo
        ? TypedResults.Ok(todo)
        : TypedResults.NotFound());
```

To use `TypedResults`, the return type must be fully declared, which when asynchronous requires the `Task<>` wrapper. Using `TypedResults` is more verbose, but that's the trade-off for having the type information be statically available and thus capable of self-describing to OpenAPI:

C#

```
app.MapGet("/todoitems/{id}", async Task<Results<Ok<Todo>, NotFound>> (int
id, TodoDb db) =>
   await db.Todos.FindAsync(id)
    is Todo todo
       ? TypedResults.Ok(todo)
       : TypedResults.NotFound());
```

Results<TResult1, TResultN>

Use Results<TResult1, TResultN> as the endpoint handler return type instead of `IResult` when:

- Multiple `IResult` implementation types are returned from the endpoint handler.
- The static `TypedResult` class is used to create the `IResult` objects.

This alternative is better than returning `IResult` because the generic union types automatically retain the endpoint metadata. And since the `Results<TResult1, TResultN>`

union types implement implicit cast operators, the compiler can automatically convert the types specified in the generic arguments to an instance of the union type.

This has the added benefit of providing compile-time checking that a route handler actually only returns the results that it declares it does. Attempting to return a type that isn't declared as one of the generic arguments to `Results<>` results in a compilation error.

Consider the following endpoint, for which a `400 BadRequest` status code is returned when the `orderId` is greater than `999`. Otherwise, it produces a `200 OK` with the expected content.

C#

```
app.MapGet("/orders/{orderId}", IResult (int orderId)
    => orderId > 999 ? TypedResults.BadRequest() : TypedResults.Ok(new
Order(orderId)))
    .Produces(400)
    .Produces<Order>();
```

In order to document this endpoint correctly the extension method `Produces` is called. However, since the `TypedResults` helper automatically includes the metadata for the endpoint, you can return the `Results<T1, Tn>` union type instead, as shown in the following code.

C#

```
app.MapGet("/orders/{orderId}", Results<BadRequest, Ok<Order>> (int orderId)
    => orderId > 999 ? TypedResults.BadRequest() : TypedResults.Ok(new
Order(orderId)));
```

Built-in results

Common result helpers exist in the Results and TypedResults static classes. Returning `TypedResults` is preferred to returning `Results`. For more information, see TypedResults vs Results.

The following sections demonstrate the usage of the common result helpers.

JSON

C#

```
app.MapGet("/hello", () => Results.Json(new { Message = "Hello World" }));
```

WriteAsJsonAsync is an alternative way to return JSON:

```
C#

app.MapGet("/", (HttpContext context) => context.Response.WriteAsJsonAsync
    (new { Message = "Hello World" }));
```

Custom Status Code

```
C#

app.MapGet("/405", () => Results.StatusCode(405));
```

Text

```
C#

app.MapGet("/text", () => Results.Text("This is some text"));
```

Stream

```
C#

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

var proxyClient = new HttpClient();
app.MapGet("/pokemon", async () =>
{
    var stream = await
proxyClient.GetStreamAsync("http://contoso/pokedex.json");
    // Proxy the response as JSON
    return Results.Stream(stream, "application/json");
});

app.Run();
```

Results.Stream overloads allow access to the underlying HTTP response stream without buffering. The following example uses ImageSharp to return a reduced size of the specified image:

C#

```
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Formats.Jpeg;
using SixLabors.ImageSharp.Processing;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/process-image/{strImage}", (string strImage, HttpContext http,
CancellationToken token) =>
{
    http.Response.Headers.CacheControl = $"public,max-age=
{TimeSpan.FromHours(24).TotalSeconds}";
    return Results.Stream(stream => ResizeImageAsync(strImage, stream,
token), "image/jpeg");
});

async Task ResizeImageAsync(string strImage, Stream stream,
CancellationToken token)
{
    var strPath = $"wwwroot/img/{strImage}";
    using var image = await Image.LoadAsync(strPath, token);
    int width = image.Width / 2;
    int height = image.Height / 2;
    image.Mutate(x =>x.Resize(width, height));
    await image.SaveAsync(stream, JpegFormat.Instance, cancellationToken:
token);
}
```

The following example streams an image from Azure Blob storage:

C#

```
app.MapGet("/stream-image/{containerName}/{blobName}",
    async (string blobName, string containerName, CancellationToken token)
=>
{
    var conStr = builder.Configuration["blogConStr"];
    BlobContainerClient blobContainerClient = new
BlobContainerClient(conStr, containerName);
    BlobClient blobClient = blobContainerClient.GetBlobClient(blobName);
    return Results.Stream(await blobClient.OpenReadAsync(cancellationToken:
token), "image/jpeg");
});
```

The following example streams a video from an Azure Blob:

C#

```
// GET /stream-video/videos/earth.mp4
app.MapGet("/stream-video/{containerName}/{blobName}",
     async (HttpContext http, CancellationToken token, string blobName,
string containerName) =>
{
    var conStr = builder.Configuration["blogConStr"];
    BlobContainerClient blobContainerClient = new
BlobContainerClient(conStr, containerName);
    BlobClient blobClient = blobContainerClient.GetBlobClient(blobName);

    var properties = await blobClient.GetPropertiesAsync(cancellationToken:
token);

    DateTimeOffset lastModified = properties.Value.LastModified;
    long length = properties.Value.ContentLength;

    long etagHash = lastModified.ToFileTime() ^ length;
    var entityTag = new EntityTagHeaderValue('\"' +
Convert.ToString(etagHash, 16) + '\"');

    http.Response.Headers.CacheControl = $"public,max-age=
{TimeSpan.FromHours(24).TotalSeconds}";

    return Results.Stream(await blobClient.OpenReadAsync(cancellationToken:
token),
        contentType: "video/mp4",
        lastModified: lastModified,
        entityTag: entityTag,
        enableRangeProcessing: true);
});
```

Redirect

C#

```
app.MapGet("/old-path", () => Results.Redirect("/new-path"));
```

File

C#

```
app.MapGet("/download", () => Results.File("myfile.text"));
```

HttpResult interfaces

The following interfaces in the Microsoft.AspNetCore.Http namespace provide a way to detect the `IResult` type at runtime, which is a common pattern in filter implementations:

- IContentTypeHttpResult
- IFileHttpResult
- INestedHttpResult
- IStatusCodeHttpResult
- IValueHttpResult
- IValueHttpResult<TValue>

Here's an example of a filter that uses one of these interfaces:

C#

```
app.MapGet("/weatherforecast", (int days) =>
{
    if (days <= 0)
    {
        return Results.BadRequest();
    }

    var forecast = Enumerable.Range(1, days).Select(index =>
       new WeatherForecast(DateTime.Now.AddDays(index),
Random.Shared.Next(-20, 55), "Cool"))
        .ToArray();
    return Results.Ok(forecast);
}).
AddEndpointFilter(async (context, next) =>
{
    var result = await next(context);

    return result switch
    {
        IValueHttpResult<WeatherForecast[]> weatherForecastResult => new
WeatherHttpResult(weatherForecastResult.Value),
        _ => result
    };
});
```

For more information, see Filters in Minimal API apps and IResult implementation types.

Customizing responses

Applications can control responses by implementing a custom IResult type. The following code is an example of an HTML result type:

C#

```
using System.Net.Mime;
using System.Text;
static class ResultsExtensions
{
    public static IResult Html(this IResultExtensions resultExtensions,
string html)
    {
        ArgumentNullException.ThrowIfNull(resultExtensions);

        return new HtmlResult(html);
    }
}

class HtmlResult : IResult
{
    private readonly string _html;

    public HtmlResult(string html)
    {
        _html = html;
    }

    public Task ExecuteAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Html;
        httpContext.Response.ContentLength =
Encoding.UTF8.GetByteCount(_html);
        return httpContext.Response.WriteAsync(_html);
    }
}
```

We recommend adding an extension method to Microsoft.AspNetCore.Http.IResultExtensions to make these custom results more discoverable.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/html", () => Results.Extensions.Html(@$"<!doctype html>
<html>
    <head><title>miniHTML</title></head>
    <body>
        <h1>Hello World</h1>
        <p>The time on the server is {DateTime.Now:O}</p>
    </body>
</html>"));
```

```
app.Run();
```

Also, a custom `IResult` type can provide its own annotation by implementing the IEndpointMetadataProvider interface. For example, the following code adds an annotation to the preceding `HtmlResult` type that describes the response produced by the endpoint.

C#

```
class HtmlResult : IResult, IEndpointMetadataProvider
{
    private readonly string _html;

    public HtmlResult(string html)
    {
        _html = html;
    }

    public Task ExecuteAsync(HttpContext httpContext)
    {
        httpContext.Response.ContentType = MediaTypeNames.Text.Html;
        httpContext.Response.ContentLength =
Encoding.UTF8.GetByteCount(_html);
        return httpContext.Response.WriteAsync(_html);
    }

    public static void PopulateMetadata(MethodInfo method, EndpointBuilder
builder)
    {
        builder.Metadata.Add(new ProducesHtmlMetadata());
    }
}
```

The `ProducesHtmlMetadata` is an implementation of IProducesResponseTypeMetadata that defines the produced response content type `text/html` and the status code `200 OK`.

C#

```
internal sealed class ProducesHtmlMetadata : IProducesResponseTypeMetadata
{
    public Type? Type => null;

    public int StatusCode => 200;

    public IEnumerable<string> ContentTypes { get; } = new[] {
MediaTypeNames.Text.Html };
}
```

An alternative approach is using the Microsoft.AspNetCore.Mvc.ProducesAttribute to describe the produced response. The following code changes the `PopulateMetadata` method to use `ProducesAttribute`.

C#

```
public static void PopulateMetadata(MethodInfo method, EndpointBuilder
builder)
{
    builder.Metadata.Add(new ProducesAttribute(MediaTypeNames.Text.Html));
}
```

Configure JSON serialization options

By default, minimal API apps use Web defaults options during JSON serialization and deserialization.

Configure JSON serialization options globally

Options can be configured globally for an app by invoking ConfigureHttpJsonOptions. The following example includes public fields and formats JSON output.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options => {
    options.SerializerOptions.WriteIndented = true;
    options.SerializerOptions.IncludeFields = true;
});

var app = builder.Build();

app.MapPost("/", (Todo todo) => {
    if (todo is not null) {
        todo.Name = todo.NameField;
    }
    return todo;
});

app.Run();

class Todo {
    public string? Name { get; set; }
    public string? NameField;
    public bool IsComplete { get; set; }
}
// If the request body contains the following JSON:
```

```
//
// {"nameField":"Walk dog", "isComplete":false}
//
// The endpoint returns the following JSON:
//
// {
//    "name":"Walk dog",
//    "nameField":"Walk dog",
//    "isComplete":false
// }
```

Since fields are included, the preceding code reads `NameField` and includes it in the output JSON.

Configure JSON serialization options for an endpoint

To configure serialization options for an endpoint, invoke Results.Json and pass to it a JsonSerializerOptions object, as shown in the following example:

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web)
    { WriteIndented = true };

app.MapGet("/", () =>
    Results.Json(new Todo { Name = "Walk dog", IsComplete = false },
options));

app.Run();

class Todo
{
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
// The endpoint returns the following JSON:
//
// {
//   "name":"Walk dog",
//   "isComplete":false
// }
```

As an alternative, use an overload of WriteAsJsonAsync that accepts a JsonSerializerOptions object. The following example uses this overload to format the output JSON:

C#

```
using System.Text.Json;

var app = WebApplication.Create();

var options = new JsonSerializerOptions(JsonSerializerDefaults.Web) {
    WriteIndented = true };

app.MapGet("/", (HttpContext context) =>
    context.Response.WriteAsJsonAsync<Todo>(
        new Todo { Name = "Walk dog", IsComplete = false }, options));

app.Run();

class Todo
{
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}
// The endpoint returns the following JSON:
//
// {
//   "name":"Walk dog",
//   "isComplete":false
// }
```

Additional Resources

- Authentication and authorization in minimal APIs

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Filters in Minimal API apps

Article • 07/26/2024

By Fiyaz Bin Hasan, Martin Costello, and Rick Anderson

Minimal API filters allow developers to implement business logic that supports:

- Running code before and after the endpoint handler.
- Inspecting and modifying parameters provided during an endpoint handler invocation.
- Intercepting the response behavior of an endpoint handler.

Filters can be helpful in the following scenarios:

- Validating the request parameters and body that are sent to an endpoint.
- Logging information about the request and response.
- Validating that a request is targeting a supported API version.

Filters can be registered by providing a Delegate that takes a EndpointFilterInvocationContext and returns a EndpointFilterDelegate. The `EndpointFilterInvocationContext` provides access to the `HttpContext` of the request and an `Arguments` list indicating the arguments passed to the handler in the order in which they appear in the declaration of the handler.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

string ColorName(string color) => $"Color specified: {color}!";

app.MapGet("/colorSelector/{color}", ColorName)
    .AddEndpointFilter(async (invocationContext, next) =>
    {
        var color = invocationContext.GetArgument<string>(0);

        if (color == "Red")
        {
            return Results.Problem("Red not allowed!");
        }
        return await next(invocationContext);
    });

app.Run();
```

The preceding code:

- Calls the `AddEndpointFilter` extension method to add a filter to the `/colorSelector/{color}` endpoint.
- Returns the color specified except for the value `"Red"`.
- Returns Results.Problem when the `/colorSelector/Red` is requested.
- Uses `next` as the `EndpointFilterDelegate` and `invocationContext` as the `EndpointFilterInvocationContext` to invoke the next filter in the pipeline or the request delegate if the last filter has been invoked.

The filter is run before the endpoint handler. When multiple `AddEndpointFilter` invocations are made on a handler:

- Filter code called before the `EndpointFilterDelegate` (`next`) is called are executed in order of First In, First Out (FIFO) order.
- Filter code called after the `EndpointFilterDelegate` (`next`) is called are executed in order of First In, Last Out (FILO) order.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () =>
    {
        app.Logger.LogInformation("             Endpoint");
        return "Test of multiple filters";
    })
    .AddEndpointFilter(async (efiContext, next) =>
    {
        app.Logger.LogInformation("Before first filter");
        var result = await next(efiContext);
        app.Logger.LogInformation("After first filter");
        return result;
    })
    .AddEndpointFilter(async (efiContext, next) =>
    {
        app.Logger.LogInformation(" Before 2nd filter");
        var result = await next(efiContext);
        app.Logger.LogInformation(" After 2nd filter");
        return result;
    })
    .AddEndpointFilter(async (efiContext, next) =>
    {
        app.Logger.LogInformation("     Before 3rd filter");
        var result = await next(efiContext);
        app.Logger.LogInformation("     After 3rd filter");
        return result;
```

```
    });

app.Run();
```

In the preceding code, the filters and endpoint log the following output:

.NET CLI

```
Before first filter
    Before 2nd filter
        Before 3rd filter
            Endpoint
        After 3rd filter
    After 2nd filter
After first filter
```

The following code uses filters that implement the `IEndpointFilter` interface:

C#

```
using Filters.EndpointFilters;

var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapGet("/", () =>
    {
        app.Logger.LogInformation("Endpoint");
        return "Test of multiple filters";
    })
    .AddEndpointFilter<AEndpointFilter>()
    .AddEndpointFilter<BEndpointFilter>()
    .AddEndpointFilter<CEndpointFilter>();

app.Run();
```

In the preceding code, the filters and handlers logs show the order they are run:

.NET CLI

```
AEndpointFilter Before next
BEndpointFilter Before next
CEndpointFilter Before next
      Endpoint
CEndpointFilter After next
BEndpointFilter After next
AEndpointFilter After next
```

Filters implementing the `IEndpointFilter` interface are shown in the following example:

C#

```
namespace Filters.EndpointFilters;

public abstract class ABCEndpointFilters : IEndpointFilter
{
    protected readonly ILogger Logger;
    private readonly string _methodName;

    protected ABCEndpointFilters(ILoggerFactory loggerFactory)
    {
        Logger = loggerFactory.CreateLogger<ABCEndpointFilters>();
        _methodName = GetType().Name;
    }

    public virtual async ValueTask<object?>
InvokeAsync(EndpointFilterInvocationContext context,
        EndpointFilterDelegate next)
    {
        Logger.LogInformation("{MethodName} Before next", _methodName);
        var result = await next(context);
        Logger.LogInformation("{MethodName} After next", _methodName);
        return result;
    }
}

class AEndpointFilter : ABCEndpointFilters
{
    public AEndpointFilter(ILoggerFactory loggerFactory) :
base(loggerFactory) { }
}

class BEndpointFilter : ABCEndpointFilters
{
    public BEndpointFilter(ILoggerFactory loggerFactory) :
base(loggerFactory) { }
}

class CEndpointFilter : ABCEndpointFilters
{
    public CEndpointFilter(ILoggerFactory loggerFactory) :
base(loggerFactory) { }
}
```

Validate an object with a filter

Consider a filter that validates a `Todo` object:

C#

```
app.MapPut("/todoitems/{id}", async (Todo inputTodo, int id, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
}).AddEndpointFilter(async (efiContext, next) =>
{
    var tdparam = efiContext.GetArgument<Todo>(0);

    var validationError = Utilities.IsValid(tdparam);

    if (!string.IsNullOrEmpty(validationError))
    {
        return Results.Problem(validationError);
    }
    return await next(efiContext);
});
```

In the preceding code:

- The `EndpointFilterInvocationContext` object provides access to the parameters associated with a particular request issued to the endpoint via the `GetArguments` method.
- The filter is registered using a `delegate` that takes a `EndpointFilterInvocationContext` and returns a `EndpointFilterDelegate`.

In addition to being passed as delegates, filters can be registered by implementing the `IEndpointFilter` interface. The following code shows the preceding filter encapsulated in a class which implements `IEndpointFilter`:

C#

```
public class TodoIsValidFilter : IEndpointFilter
{
    private ILogger _logger;

    public TodoIsValidFilter(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<TodoIsValidFilter>();
    }
```

```
    public async ValueTask<object?>
InvokeAsync(EndpointFilterInvocationContext efiContext,
        EndpointFilterDelegate next)
    {
        var todo = efiContext.GetArgument<Todo>(0);

        var validationError = Utilities.IsValid(todo!);

        if (!string.IsNullOrEmpty(validationError))
        {
            _logger.LogWarning(validationError);
            return Results.Problem(validationError);
        }
        return await next(efiContext);
    }
}
```

Filters that implement the `IEndpointFilter` interface can resolve dependencies from Dependency Injection(DI), as shown in the previous code. Although filters can resolve dependencies from DI, filters themselves can ***not*** be resolved from DI.

The `ToDoIsValidFilter` is applied to the following endpoints:

C#

```
app.MapPut("/todoitems2/{id}", async (Todo inputTodo, int id, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
}).AddEndpointFilter<TodoIsValidFilter>();

app.MapPost("/todoitems", async (Todo todo, TodoDb db) =>
{
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
}).AddEndpointFilter<TodoIsValidFilter>();
```

The following filter validates the `Todo` object and modifies the `Name` property:

C#

```
public class TodoIsValidUcFilter : IEndpointFilter
{
    public async ValueTask<object?>
InvokeAsync(EndpointFilterInvocationContext efiContext,
        EndpointFilterDelegate next)
    {
        var todo = efiContext.GetArgument<Todo>(0);
        todo.Name = todo.Name!.ToUpper();

        var validationError = Utilities.IsValid(todo!);

        if (!string.IsNullOrEmpty(validationError))
        {
            return Results.Problem(validationError);
        }
        return await next(efiContext);
    }
}
```

Register a filter using an endpoint filter factory

In some scenarios, it might be necessary to cache some of the information provided in the MethodInfo in a filter. For example, let's assume that we wanted to verify that the handler an endpoint filter is attached to has a first parameter that evaluates to a `Todo` type.

C#

```
app.MapPut("/todoitems/{id}", async (Todo inputTodo, int id, TodoDb db) =>
{
    var todo = await db.Todos.FindAsync(id);

    if (todo is null) return Results.NotFound();

    todo.Name = inputTodo.Name;
    todo.IsComplete = inputTodo.IsComplete;

    await db.SaveChangesAsync();

    return Results.NoContent();
}).AddEndpointFilterFactory((filterFactoryContext, next) =>
{
    var parameters = filterFactoryContext.MethodInfo.GetParameters();
    if (parameters.Length >= 1 && parameters[0].ParameterType ==
typeof(Todo))
    {
        return async invocationContext =>
        {
            var todoParam = invocationContext.GetArgument<Todo>(0);
```

```
            var validationError = Utilities.IsValid(todoParam);

            if (!string.IsNullOrEmpty(validationError))
            {
                return Results.Problem(validationError);
            }
            return await next(invocationContext);
        };
    }
    return invocationContext => next(invocationContext);
});
```

In the preceding code:

- The `EndpointFilterFactoryContext` object provides access to the MethodInfo associated with the endpoint's handler.
- The signature of the handler is examined by inspecting `MethodInfo` for the expected type signature. If the expected signature is found, the validation filter is registered onto the endpoint. This factory pattern is useful to register a filter that depends on the signature of the target endpoint handler.
- If a matching signature isn't found, then a pass-through filter is registered.

Register a filter on controller actions

In some scenarios, it might be necessary to apply the same filter logic for both route-handler based endpoints and controller actions. For this scenario, it is possible to invoke `AddEndpointFilter` on `ControllerActionEndpointConventionBuilder` to support executing the same filter logic on actions and endpoints.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.MapController()
    .AddEndpointFilter(async (efiContext, next) =>
    {
        efiContext.HttpContext.Items["endpointFilterCalled"] = true;
        var result = await next(efiContext);
        return result;
    });

app.Run();
```

Additional Resources

- View or download sample code (how to download)
- ValidationFilterRouteHandlerBuilderExtensions Validation extension methods.
- Tutorial: Create a minimal API with ASP.NET Core
- Authentication and authorization in minimal APIs

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Unit and integration tests in Minimal API apps

Article • 07/26/2024

By Fiyaz Bin Hasan, and Rick Anderson

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test, choose the unit test.

In discussions of integration tests, the tested project is frequently called the ***System Under Test***, or "**SUT**" for short. "SUT" is used throughout this article to refer to the ASP.NET Core app being tested.

Don't write integration tests for every permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with

databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes or mocks result in faster test execution.

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (`.deps`) from the SUT into the test project's `bin` directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.

- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

Separate unit tests from integration tests into different projects. Separating the tests:

- Helps ensure that infrastructure testing components aren't accidentally included in the unit tests.
- Allows control over which set of tests are run.

The sample code on GitHub provides an example of unit and integration tests on a Minimal API app.

IResult implementation types

Public IResult implementation types in the Microsoft.AspNetCore.Http.HttpResults namespace can be used to unit test minimal route handlers when using named methods instead of lambdas.

The following code uses the NotFound<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsNotFoundIfNotExists()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var notFoundResult = (NotFound) result.Result;

    Assert.NotNull(notFoundResult);
}
```

The following code uses the Ok<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsTodoFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title",
        Description = "Test description",
        IsDone = false
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var okResult = (Ok<Todo>)result.Result;

    Assert.NotNull(okResult.Value);
    Assert.Equal(1, okResult.Value.Id);
}
```

Additional Resources

- Basic authentication tests is not a .NET repository but was written by a member of the .NET team. It provides examples of basic authentication testing.
- View or download sample code
- Authentication and authorization in minimal APIs
- Use port tunneling Visual Studio to debug web APIs
- Test controller logic in ASP.NET Core
- Razor Pages unit tests in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Middleware in Minimal API apps

Article • 07/26/2024

WebApplication automatically adds the following middleware in Minimal API applications depending on certain conditions:

- UseDeveloperExceptionPage is added first when the HostingEnvironment is `"Development"`.
- UseRouting is added second if user code didn't already call `UseRouting` and if there are endpoints configured, for example `app.MapGet`.
- UseEndpoints is added at the end of the middleware pipeline if any endpoints are configured.
- UseAuthentication is added immediately after `UseRouting` if user code didn't already call `UseAuthentication` and if IAuthenticationSchemeProvider can be detected in the service provider. `IAuthenticationSchemeProvider` is added by default when using AddAuthentication, and services are detected using IServiceProviderIsService.
- UseAuthorization is added next if user code didn't already call `UseAuthorization` and if IAuthorizationHandlerProvider can be detected in the service provider. `IAuthorizationHandlerProvider` is added by default when using AddAuthorization, and services are detected using `IServiceProviderIsService`.
- User configured middleware and endpoints are added between `UseRouting` and `UseEndpoints`.

The following code is effectively what the automatic middleware being added to the app produces:

C#

```
if (isDevelopment)
{
    app.UseDeveloperExceptionPage();
}

app.UseRouting();

if (isAuthenticationConfigured)
{
    app.UseAuthentication();
}

if (isAuthorizationConfigured)
{
    app.UseAuthorization();
```

```
}

// user middleware/endpoints
app.CustomMiddleware(...);
app.MapGet("/", () => "hello world");
// end user middleware/endpoints

app.UseEndpoints(e => {});
```

In some cases, the default middleware configuration isn't correct for the app and requires modification. For example, UseCors should be called before UseAuthentication and UseAuthorization. The app needs to call `UseAuthentication` and `UseAuthorization` if `UseCors` is called:

C#

```
app.UseCors();
app.UseAuthentication();
app.UseAuthorization();
```

If middleware should be run before route matching occurs, UseRouting should be called and the middleware should be placed before the call to `UseRouting`. UseEndpoints isn't required in this case as it is automatically added as described previously:

C#

```
app.Use((context, next) =>
{
    return next(context);
});

app.UseRouting();

// other middleware and endpoints
```

When adding a terminal middleware:

- The middleware must be added after `UseEndpoints`.
- The app needs to call `UseRouting` and `UseEndpoints` so that the terminal middleware can be placed at the correct location.

C#

```
app.UseRouting();

app.MapGet("/", () => "hello world");
```

```
app.UseEndpoints(e => {});

app.Run(context =>
{
    context.Response.StatusCode = 404;
    return Task.CompletedTask;
});
```

Terminal middleware is middleware that runs if no endpoint handles the request.

For information on antiforgery middleware in Minimal APIs, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

For more information about middleware see ASP.NET Core Middleware, and the list of built-in middleware that can be added to applications.

For more information about Minimal APIs see Minimal APIs overview.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

How to handle errors in Minimal API apps

Article • 06/01/2024

With contributions by David Acker

This article describes how to handle errors in Minimal API apps. For information about error handling in controller-based APIs, see Handle errors in ASP.NET Core and Handle errors in ASP.NET Core controller-based web APIs.

Exceptions

In a Minimal API app, there are two different built-in centralized mechanisms to handle unhandled exceptions:

- Developer Exception Page middleware (For use in the **Development environment only**.)
- Exception handler middleware

This section refers to the following sample app to demonstrate ways to handle exceptions in a Minimal API. It throws an exception when the endpoint `/exception` is requested:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/exception", () =>
{
    throw new InvalidOperationException("Sample Exception");
});

app.MapGet("/", () => "Test by calling /exception");

app.Run();
```

Developer Exception Page

The *Developer Exception Page* displays detailed information about unhandled request exceptions. It uses DeveloperExceptionPageMiddleware to capture synchronous and asynchronous exceptions from the HTTP pipeline and to generate error responses. The

developer exception page runs early in the middleware pipeline, so that it can catch unhandled exceptions thrown in middleware that follows.

ASP.NET Core apps enable the developer exception page by default when both:

- Running in the Development environment.
- The app was created with the current templates, that is, by using WebApplication.CreateBuilder.

Apps created using earlier templates, that is, by using WebHost.CreateDefaultBuilder, can enable the developer exception page by calling app.UseDeveloperExceptionPage.

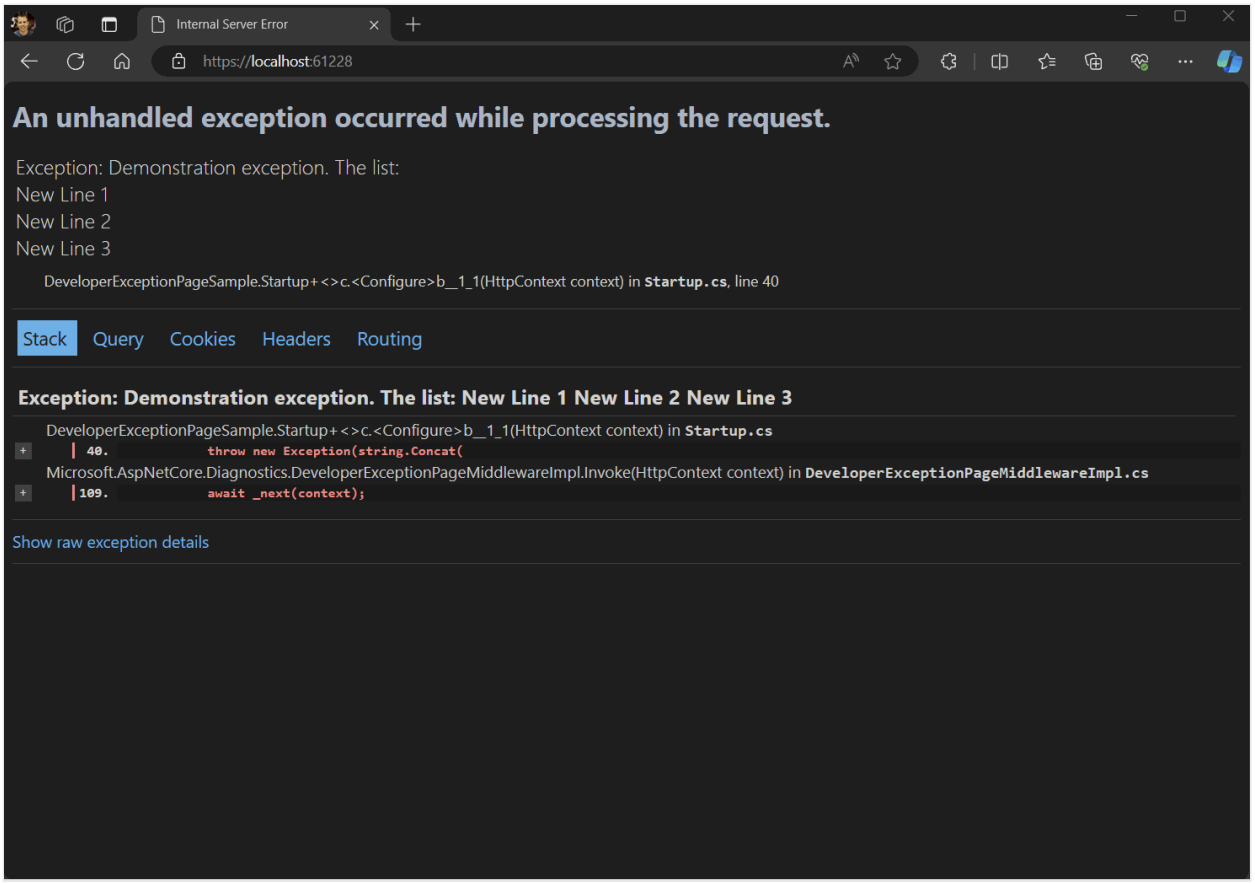
> ⚠ **Warning**
>
> Don't enable the Developer Exception Page **unless the app is running in the Development environment**. Don't share detailed exception information publicly when the app runs in production. For more information on configuring environments, see **Use multiple environments in ASP.NET Core**.

The Developer Exception Page can include the following information about the exception and the request:

- Stack trace
- Query string parameters, if any
- Cookies, if any
- Headers
- Endpoint metadata, if any

The Developer Exception Page isn't guaranteed to provide any information. Use Logging for complete error information.

The following image shows a sample developer exception page with animation to show the tabs and the information displayed:



In response to a request with an `Accept: text/plain` header, the Developer Exception Page returns plain text instead of HTML. For example:

text

```
Status: 500 Internal Server Error
Time: 9.39 msSize: 480 bytes
FormattedRawHeadersRequest
Body
text/plain; charset=utf-8, 480 bytes
System.InvalidOperationException: Sample Exception
   at WebApplicationMinimal.Program.<>c.<Main>b__0_0() in
C:\Source\WebApplicationMinimal\Program.cs:line 12
   at lambda_method1(Closure, Object, HttpContext)
   at
Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddlewareImpl.Invoke
(HttpContext context)

HEADERS
=======
Accept: text/plain
Host: localhost:7267
traceparent: 00-0eab195ea19d07b90a46cd7d6bf2f
```

To see the Developer Exception Page:

- Run the sample app in the Development environment.
- Go to the `/exception` endpoint.

Exception handler

In non-development environments, use the Exception Handler Middleware to produce an error payload. To configure the `Exception Handler Middleware`, call UseExceptionHandler.

For example, the following code changes the app to respond with an RFC 7807 - compliant payload to the client. For more information, see the Problem Details section later in this article.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseExceptionHandler(exceptionHandlerApp
    => exceptionHandlerApp.Run(async context
        => await Results.Problem()
                      .ExecuteAsync(context)));

app.MapGet("/exception", () =>
{
    throw new InvalidOperationException("Sample Exception");
});

app.MapGet("/", () => "Test by calling /exception");

app.Run();
```

Client and Server error responses

Consider the following Minimal API app.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/users/{id:int}", (int id)
    => id <= 0 ? Results.BadRequest() : Results.Ok(new User(id)));

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

The `/users` endpoint produces `200 OK` with a `json` representation of `User` when `id` is greater than `0`, otherwise a `400 BAD REQUEST` status code without a response body. For more information about creating a response, see Create responses in Minimal API apps.

The Status Code Pages middleware can be configured to produce a common body content, **when empty**, for all HTTP client (`400`-`499`) or server (`500` -`599`) responses. The middleware is configured by calling the UseStatusCodePages extension method.

For example, the following example changes the app to respond with an RFC 7807 - compliant payload to the client for all client and server responses, including routing errors (for example, `404 NOT FOUND`). For more information, see the Problem Details section.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseStatusCodePages(async statusCodeContext
    => await Results.Problem(statusCode:
statusCodeContext.HttpContext.Response.StatusCode)
                  .ExecuteAsync(statusCodeContext.HttpContext));
app.MapGet("/users/{id:int}", (int id)
    => id <= 0 ? Results.BadRequest() : Results.Ok(new User(id)) );

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

Problem details

Problem Details are not the only response format to describe an HTTP API error, however, they are commonly used to report errors for HTTP APIs.

The problem details service implements the IProblemDetailsService interface, which supports creating problem details in ASP.NET Core. The AddProblemDetails(IServiceCollection) extension method on IServiceCollection registers the default `IProblemDetailsService` implementation.

In ASP.NET Core apps, the following middleware generates problem details HTTP responses when `AddProblemDetails` is called, except when the Accept request HTTP header doesn't include one of the content types supported by the registered IProblemDetailsWriter (default: `application/json`):

- ExceptionHandlerMiddleware: Generates a problem details response when a custom handler is not defined.
- StatusCodePagesMiddleware: Generates a problem details response by default.
- DeveloperExceptionPageMiddleware: Generates a problem details response in development when the `Accept` request HTTP header doesn't include `text/html`.

Minimal API apps can be configured to generate problem details response for all HTTP client and server error responses that ***don't have body content yet*** by using the `AddProblemDetails` extension method.

The following code configures the app to generate problem details:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler();
app.UseStatusCodePages();

app.MapGet("/users/{id:int}", (int id)
    => id <= 0 ? Results.BadRequest() : Results.Ok(new User(id)));

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

For more information on using `AddProblemDetails`, see Problem Details

IProblemDetailsService fallback

In the following code, `httpContext.Response.WriteAsync("Fallback: An error occurred.")` returns an error if the IProblemDetailsService implementation isn't able to generate a ProblemDetails:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseExceptionHandler(exceptionHandlerApp =>
```

```
{
    exceptionHandlerApp.Run(async httpContext =>
    {
        var pds =
httpContext.RequestServices.GetService<IProblemDetailsService>();
        if (pds == null
            || !await pds.TryWriteAsync(new() { HttpContext = httpContext
}))
        {
            // Fallback behavior
            await httpContext.Response.WriteAsync("Fallback: An error
occurred.");
        }
    });
});

app.MapGet("/exception", () =>
{
    throw new InvalidOperationException("Sample Exception");
});

app.MapGet("/", () => "Test by calling /exception");

app.Run();
```

The preceding code:

- Writes an error message with the fallback code if the `problemDetailsService` is unable to write a `ProblemDetails`. For example, an endpoint where the Accept request header specifies a media type that the `DefaulProblemDetailsWriter` does not support.
- Uses the Exception Handler Middleware.

The following sample is similar to the preceding except that it calls the Status Code Pages middleware.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddProblemDetails();

var app = builder.Build();

app.UseStatusCodePages(statusCodeHandlerApp =>
{
    statusCodeHandlerApp.Run(async httpContext =>
    {
        var pds =
httpContext.RequestServices.GetService<IProblemDetailsService>();
        if (pds == null
            || !await pds.TryWriteAsync(new() { HttpContext = httpContext
```

```
}))
        {
            // Fallback behavior
            await httpContext.Response.WriteAsync("Fallback: An error
occurred.");
        }
    });
});

app.MapGet("/users/{id:int}", (int id) =>
{
    return id <= 0 ? Results.BadRequest() : Results.Ok(new User(id));
});

app.MapGet("/", () => "Test by calling /users/{id:int}");

app.Run();

public record User(int Id);
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Authentication and authorization in minimal APIs

Article • 07/26/2024

Minimal APIs support all the authentication and authorization options available in ASP.NET Core and provide some additional functionality to improve the experience working with authentication.

Key concepts in authentication and authorization

Authentication is the process of determining a user's identity. Authorization is the process of determining whether a user has access to a resource. Both authentication and authorization scenarios share similar implementation semantics in ASP.NET Core. Authentication is handled by the authentication service, IAuthenticationService, which is used by authentication middleware. Authorization is handled by the authorization service, IAuthorizationService, which is used by the authorization middleware.

The authentication service uses registered authentication handlers to complete authentication-related actions. For example, an authentication-related action is authenticating a user or signing out a user. Authentication schemes are names that are used to uniquely identify an authentication handler and its configuration options. Authentication handlers are responsible for implementing the strategies for authentication and generating a user's claims given a particular authentication strategy, such as OAuth or OIDC. The configuration options are unique to the strategy as well and provide the handler with configuration that affects authentication behavior, such as redirect URIs.

There are two strategies for determining user access to resources in the authorization layer:

- Role-based strategies determine a user's access based on the role they are assigned, such as `Administrator` or `User`. For more information on role-based authorization, see role-based authorization documentation.
- Claim-based strategies determine a user's access based on claims that are issued by a central authority. For more information on claim-based authorization, see claim-based authorization documentation.

In ASP.NET Core, both strategies are captured into an authorization requirement. The authorization service leverages authorization handlers to determine whether or not a particular user meets the authorization requirements applied onto a resource.

Enabling authentication in minimal apps

To enable authentication, call AddAuthentication to register the required authentication services on the app's service provider.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication();
var app = builder.Build();

app.MapGet("/", () => "Hello World!");
app.Run();
```

Typically, a specific authentication strategy is used. In the following sample, the app is configured with support for JWT bearer-based authentication. This example makes use of the APIs available in the Microsoft.AspNetCore.Authentication.JwtBearer NuGet package.

C#

```
var builder = WebApplication.CreateBuilder(args);
// Requires Microsoft.AspNetCore.Authentication.JwtBearer
builder.Services.AddAuthentication().AddJwtBearer();
var app = builder.Build();

app.MapGet("/", () => "Hello World!");
app.Run();
```

By default, the WebApplication automatically registers the authentication and authorization middlewares if certain authentication and authorization services are enabled. In the following sample, it's not necessary to invoke UseAuthentication or UseAuthorization to register the middlewares because WebApplication does this automatically after `AddAuthentication` or `AddAuthorization` are called.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
var app = builder.Build();
```

```
app.MapGet("/", () => "Hello World!");
app.Run();
```

In some cases, such as controlling middleware order, it's necessary to explicitly register authentication and authorization. In the following sample, the authentication middleware runs *after* the CORS middleware has run. For more information on middlewares and this automatic behavior, see Middleware in Minimal API apps.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors();
builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();

var app = builder.Build();

app.UseCors();
app.UseAuthentication();
app.UseAuthorization();

app.MapGet("/", () => "Hello World!");
app.Run();
```

Configuring authentication strategy

Authentication strategies typically support a variety of configurations that are loaded via options. Minimal apps support loading options from configuration for the following authentication strategies:

- JWT bearer-based
- OpenID Connection-based

The ASP.NET Core framework expects to find these options under the `Authentication:Schemes:{SchemeName}` section in configuration. In the following sample, two different schemes, `Bearer` and `LocalAuthIssuer`, are defined with their respective options. The `Authentication:DefaultScheme` option can be used to configure the default authentication strategy that's used.

JSON

```
{
  "Authentication": {
    "DefaultScheme":  "LocalAuthIssuer",
```

```
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "https://localhost:7259",
          "http://localhost:5259"
        ],
        "ValidIssuer": "dotnet-user-jwts"
      },
      "LocalAuthIssuer": {
        "ValidAudiences": [
          "https://localhost:7259",
          "http://localhost:5259"
        ],
        "ValidIssuer": "local-auth"
      }
    }
  }
}
```

In `Program.cs`, two JWT bearer-based authentication strategies are registered, with the:

- "Bearer" scheme name.
- "LocalAuthIssuer" scheme name.

"Bearer" is the typical default scheme in JWT-bearer based enabled apps, but the default scheme can be overridden by setting the `DefaultScheme` property as in the preceding example.

The scheme name is used to uniquely identify an authentication strategy and is used as the lookup key when resolving authentication options from config, as shown in the following example:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication()
  .AddJwtBearer()
  .AddJwtBearer("LocalAuthIssuer");

var app = builder.Build();

app.MapGet("/", () => "Hello World!");
app.Run();
```

Configuring authorization policies in minimal apps

Authentication is used to identify and validate the identity of users against an API. Authorization is used to validate and verify access to resources in an API and is facilitated by the `IAuthorizationService` registered by the `AddAuthorization` extension method. In the following scenario, a `/hello` resource is added that requires a user to present an `admin` role claim with a `greetings_api` scope claim.

Configuring authorization requirements on a resource is a two-step process that requires:

1. Configuring the authorization requirements in a policy globally.
2. Applying individual policies to resources.

In the following code, AddAuthorizationBuilder is invoked which:

- Adds authorization-related services to the DI container.
- Returns an AuthorizationBuilder that can be used to directly register authorization policies.

The code creates a new authorization policy, named `admin_greetings`, that encapsulates two authorization requirements:

- A role-based requirement via RequireRole for users with an `admin` role.
- A claim-based requirement via RequireClaim that the user must provide a `greetings_api` scope claim.

The `admin_greetings` policy is provided as a required policy to the `/hello` endpoint.

C#

```
using Microsoft.Identity.Web;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorizationBuilder()
  .AddPolicy("admin_greetings", policy =>
        policy
            .RequireRole("admin")
            .RequireClaim("scope", "greetings_api"));

var app = builder.Build();

app.MapGet("/hello", () => "Hello world!")
  .RequireAuthorization("admin_greetings");

app.Run();
```

Use `dotnet user-jwts` for development testing

Throughout this article, an app configured with JWT-bearer based authentication is used. JWT bearer-based authentication requires that clients present a token in the request header to validate their identity and claims. Typically, these tokens are issued by a central authority, such as an identity server.

When developing on the local machine, the dotnet user-jwts tool can be used to create bearer tokens.

.NET CLI

```
dotnet user-jwts create
```

> **Note**
>
> When invoked on a project, the tool automatically adds the authentication options matching the generated token to `appsettings.json`.

Tokens can be configured with a variety of customizations. For example, to create a token for the `admin` role and `greetings_api` scope expected by the authorization policy in the preceding code:

.NET CLI

```
dotnet user-jwts create --scope "greetings_api" --role "admin"
```

The generated token can then be sent as part of the header in the testing tool of choice. For example, with curl:

.NET CLI

```
curl -i -H "Authorization: Bearer {token}" https://localhost:{port}/hello
```

For more information on the `dotnet user-jwts` tool, read the complete documentation.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Collaborate with us on

.NET ASP.NET Core feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

OpenAPI support in ASP.NET Core API apps

Article • 10/28/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

The OpenAPI specification is a programming language-agnostic standard for documenting HTTP APIs. This standard is supported in minimal APIs through a combination of built-in APIs and open-source libraries. There are three key aspects to OpenAPI integration in an application:

- Generating information about the endpoints in the app.
- Gathering the information into a format that matches the OpenAPI schema.
- Exposing the generated OpenAPI schema via a visual UI or a serialized file.

Minimal APIs provide built-in support for generating information about endpoints in an app via the `Microsoft.AspNetCore.OpenApi` package. Exposing the generated OpenAPI definition via a visual UI requires a third-party package.

For information about support for OpenAPI in controller-based APIs, see the .NET 9 version of this article.

Generate OpenAPI documents

Article • 01/17/2025

Trimming and Native AOT

OpenAPI in ASP.NET Core supports trimming and native AOT. The following steps create and publish an OpenAPI app with trimming and native AOT:

Create a new ASP.NET Core Web API (Native AOT) project.

Console

```
dotnet new webapiaot
```

Add the Microsoft.AspNetCore.OpenAPI package.

Console

```
dotnet add package Microsoft.AspNetCore.OpenApi --prerelease
```

Update `Program.cs` to enable generating OpenAPI documents.

diff

```
+ builder.Services.AddOpenApi();

var app = builder.Build();

+ app.MapOpenApi();
```

Publish the app.

Console

```
dotnet publish
```

Minimal APIs provide built-in support for generating information about endpoints in an app via the `Microsoft.AspNetCore.OpenApi` package. Exposing the generated OpenAPI definition via a visual UI requires a third-party package. For information about support for OpenAPI in controller-based APIs, see the .NET 9 version of this article.

The following code is generated by the ASP.NET Core minimal web API template and uses OpenAPI:

```C#
using Microsoft.AspNetCore.OpenApi;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot",
"Sweltering", "Scorching"
};

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateTime.Now.AddDays(index),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast")
.WithOpenApi();

app.Run();

internal record WeatherForecast(DateTime Date, int TemperatureC, string?
Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

In the preceding highlighted code:

- `Microsoft.AspNetCore.OpenApi` is explained in the next section.
- AddEndpointsApiExplorer : Configures the app to use the API Explorer to discover and describe endpoints with default annotations. `WithOpenApi` overrides matching, default annotations generated by the API Explorer with those produced from the `Microsoft.AspNetCore.OpenApi` package.
- `UseSwagger` adds the Swagger middleware.
- `UseSwaggerUI` enables an embedded version of the Swagger UI tool.
- WithName: The IEndpointNameMetadata on the endpoint is used for link generation and is treated as the operation ID in the given endpoint's OpenAPI specification.
- WithOpenApi is explained later in this article.

`Microsoft.AspNetCore.OpenApi` NuGet package

ASP.NET Core provides the Microsoft.AspNetCore.OpenApi package to interact with OpenAPI specifications for endpoints. The package acts as a link between the OpenAPI models that are defined in the `Microsoft.AspNetCore.OpenApi` package and the endpoints that are defined in Minimal APIs. The package provides an API that examines an endpoint's parameters, responses, and metadata to construct an OpenAPI annotation type that is used to describe an endpoint.

`Microsoft.AspNetCore.OpenApi` is added as a PackageReference to a project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="7.0.*-
*" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.4.0" />
  </ItemGroup>

</Project>
```

When using Swashbuckle.AspNetCore with `Microsoft.AspNetCore.OpenApi`, `Swashbuckle.AspNetCore` 6.4.0 or later must be used. Microsoft.OpenApi 1.4.3 or later must be used to leverage copy constructors in `WithOpenApi` invocations.

Add OpenAPI annotations to endpoints via `WithOpenApi`

Calling WithOpenApi on the endpoint adds to the endpoint's metadata. This metadata can be:

- Consumed in third-party packages like Swashbuckle.AspNetCore.
- Displayed in the Swagger user interface or in YAML or JSON generated to define the API.

C#

```
app.MapPost("/todoitems/{id}", async (int id, Todo todo, TodoDb db) =>
{
    todo.Id = id;
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
})
.WithOpenApi();
```

Modify the OpenAPI annotation in `WithOpenApi`

The WithOpenApi method accepts a function that can be used to modify the OpenAPI annotation. For example, in the following code, a description is added to the first parameter of the endpoint:

C#

```
app.MapPost("/todo2/{id}", async (int id, Todo todo, TodoDb db) =>
{
    todo.Id = id;
    db.Todos.Add(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todoitems/{todo.Id}", todo);
})
.WithOpenApi(generatedOperation =>
{
    var parameter = generatedOperation.Parameters[0];
    parameter.Description = "The ID associated with the created Todo";
    return generatedOperation;
});
```

Add operation IDs to OpenAPI

Operation IDs are used to uniquely identify a given endpoint in OpenAPI. The WithName extension method can be used to set the operation ID used for a method.

C#

```
app.MapGet("/todoitems2", async (TodoDb db) =>
    await db.Todos.ToListAsync())
    .WithName("GetToDoItems");
```

Alternatively, the `OperationId` property can be set directly on the OpenAPI annotation.

C#

```
app.MapGet("/todos", async (TodoDb db) => await db.Todos.ToListAsync())
    .WithOpenApi(operation => new(operation)
    {
        OperationId = "GetTodos"
    });
```

Add tags to the OpenAPI description

OpenAPI supports using tag objects to categorize operations. These tags are typically used to group operations in the Swagger UI. These tags can be added to an operation by invoking the WithTags extension method on the endpoint with the desired tags.

C#

```
app.MapGet("/todoitems", async (TodoDb db) =>
    await db.Todos.ToListAsync())
    .WithTags("TodoGroup");
```

Alternatively, the list of `OpenApiTags` can be set on the OpenAPI annotation via the `WithOpenApi` extension method.

C#

```
app.MapGet("/todos", async (TodoDb db) => await db.Todos.ToListAsync())
    .WithOpenApi(operation => new(operation)
    {
        Tags = new List<OpenApiTag> { new() { Name = "Todos" } }
    });
```

Add endpoint summary or description

The endpoint summary and description can be added by invoking the `WithOpenApi` extension method. In the following code, the summaries are set directly on the OpenAPI annotation.

C#

```
app.MapGet("/todoitems2", async (TodoDb db) => await db.Todos.ToListAsync())
    .WithOpenApi(operation => new(operation)
    {
        Summary = "This is a summary",
        Description = "This is a description"
    });
```

Exclude OpenAPI description

In the following sample, the `/skipme` endpoint is excluded from generating an OpenAPI description:

C#

```
using Microsoft.AspNetCore.OpenApi;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.MapGet("/swag", () => "Hello Swagger!")
    .WithOpenApi();
app.MapGet("/skipme", () => "Skipping Swagger.")
                    .ExcludeFromDescription();

app.Run();
```

Mark an API as obsolete

To mark an endpoint as obsolete, set the `Deprecated` property on the OpenAPI annotation.

```
C#
```

```
app.MapGet("/todos", async (TodoDb db) => await db.Todos.ToListAsync())
    .WithOpenApi(operation => new(operation)
    {
        Deprecated = true
    });
```

Describe response types

OpenAPI supports providing a description of the responses returned from an API. Minimal APIs support three strategies for setting the response type of an endpoint:

- Via the Produces extension method on the endpoint
- Via the ProducesResponseType attribute on the route handler
- By returning TypedResults from the route handler

The `Produces` extension method can be used to add `Produces` metadata to an endpoint. When no parameters are provided, the extension method populates metadata for the targeted type under a `200` status code and an `application/json` content type.

```
C#
```

```
app
    .MapGet("/todos", async (TodoDb db) => await db.Todos.ToListAsync())
    .Produces<IList<Todo>>();
```

Using TypedResults in the implementation of an endpoint's route handler automatically includes the response type metadata for the endpoint. For example, the following code automatically annotates the endpoint with a response under the `200` status code with an `application/json` content type.

```
C#
```

```
app.MapGet("/todos", async (TodoDb db) =>
{
    var todos = await db.Todos.ToListAsync());
```

```
    return TypedResults.Ok(todos);
});
```

Set responses for `ProblemDetails`

When setting the response type for endpoints that may return a ProblemDetails response, the ProducesProblem extension method, ProducesValidationProblem, or TypedResults.Problem can be used to add the appropriate annotation to the endpoint's metadata. Note that the `ProducesProblem` and `ProducesValidationProblem` extension methods can't be used with route groups in .NET 8 and earlier.

When there are no explicit annotations provided by one of the strategies above, the framework attempts to determine a default response type by examining the signature of the response. This default response is populated under the `200` status code in the OpenAPI definition.

Multiple response types

If an endpoint can return different response types in different scenarios, you can provide metadata in the following ways:

- Call the Produces extension method multiple times, as shown in the following example:

  C#

  ```
  app.MapGet("/api/todoitems/{id}", async (int id, TodoDb db) =>
          await db.Todos.FindAsync(id)
          is Todo todo
          ? Results.Ok(todo)
          : Results.NotFound())
     .Produces<Todo>(StatusCodes.Status200OK)
     .Produces(StatusCodes.Status404NotFound);
  ```

- Use Results<TResult1,TResult2,TResultN> in the signature and TypedResults in the body of the handler, as shown in the following example:

  C#

  ```
  app.MapGet("/book/{id}", Results<Ok<Book>, NotFound> (int id,
  List<Book> bookList) =>
  {
      return bookList.FirstOrDefault((i) => i.Id == id) is Book book
       ? TypedResults.Ok(book)
  ```

```
        : TypedResults.NotFound();
});
```

The `Results<TResult1,TResult2,TResultN>` union types declare that a route handler returns multiple `IResult`-implementing concrete types, and any of those types that implement `IEndpointMetadataProvider` will contribute to the endpoint's metadata.

The union types implement implicit cast operators. These operators enable the compiler to automatically convert the types specified in the generic arguments to an instance of the union type. This capability has the added benefit of providing compile-time checking that a route handler only returns the results that it declares it does. Attempting to return a type that isn't declared as one of the generic arguments to `Results<TResult1,TResult2,TResultN>` results in a compilation error.

Describe request body and parameters

In addition to describing the types that are returned by an endpoint, OpenAPI also supports annotating the inputs that are consumed by an API. These inputs fall into two categories:

- Parameters that appear in the path, query string, headers, or cookies
- Data transmitted as part of the request body

The framework infers the types for request parameters in the path, query, and header string automatically based on the signature of the route handler.

To define the type of inputs transmitted as the request body, configure the properties by using the Accepts extension method to define the object type and content type that are expected by the request handler. In the following example, the endpoint accepts a `Todo` object in the request body with an expected content-type of `application/xml`.

C#

```
app.MapPost("/todos/{id}", (int id, Todo todo) => ...)
  .Accepts<Todo>("application/xml");
```

In addition to the Accepts extension method, A parameter type can describe its own annotation by implementing the IEndpointParameterMetadataProvider interface. For example, the following `Todo` type adds an annotation that requires a request body with an `application/xml` content-type.

C#

```
public class Todo : IEndpointParameterMetadataProvider
{
    public static void PopulateMetadata(ParameterInfo parameter,
EndpointBuilder builder)
    {
        builder.Metadata.Add(new ConsumesAttribute(typeof(Todo), isOptional:
false, "application/xml"));
    }
}
```

When no explicit annotation is provided, the framework attempts to determine the default request type if there's a request body parameter in the endpoint handler. The inference uses the following heuristics to produce the annotation:

- Request body parameters that are read from a form via the [FromForm] attribute are described with the `multipart/form-data` content-type.
- All other request body parameters are described with the `application/json` content-type.
- The request body is treated as optional if it's nullable or if the AllowEmpty property is set on the FromBody attribute.

Support API versioning

Minimal APIs support API versioning via the Asp.Versioning.Http package. Examples of configuring versioning with minimal APIs can be found in the API versioning repo.

ASP.NET Core OpenAPI source code on GitHub

- WithOpenApi
- OpenApiGenerator

Additional Resources

- Authentication and authorization in minimal APIs

Use openAPI documents

Article • 12/11/2024

To learn how to use the generated OpenAPI document in a minimal API in .NET 6, 7, or 8, see the Swagger and NSwag overview.

.NET OpenAPI tool command reference and installation

Article • 08/06/2024

Microsoft.dotnet-openapi is a .NET Core Global Tool for managing OpenAPI references within a project.

Installation

To install `Microsoft.dotnet-openapi`, run the following command:

.NET CLI

```
dotnet tool install -g Microsoft.dotnet-openapi
```

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

Add

Adding an OpenAPI reference using any of the commands on this page adds an `<OpenApiReference />` element similar to the following to the `.csproj` file:

XML

```
<OpenApiReference Include="openapi.json" />
```

The preceding reference is required for the app to call the generated client code.

Add File

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi add file *--updateProject .\Ref.csproj* .\OpenAPI.json
-c	--code-generator	The code generator to apply to the reference. Options are `NSwagCSharp` and `NSwagTypeScript`. If `--code-generator` is not specified the tooling defaults to `NSwagCSharp`.	dotnet openapi add file .\OpenApi.json --code-generator
-h	--help	Show help information	dotnet openapi add file --help

Arguments

Expand table

Argument	Description	Example
source-file	The source to create a reference from. Must be an OpenAPI file.	dotnet openapi add file *.\OpenAPI.json*

Add URL

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi add url *--updateProject .\Ref.csproj* `https://contoso.com/openapi.json`
-o	--output-file	Where to place the local copy of the OpenAPI file.	dotnet openapi add url `https://contoso.com/openapi.json` *--output-file myclient.json*

Short option	Long option	Description	Example
-c	--code-generator	The code generator to apply to the reference. Options are `NSwagCSharp` and `NSwagTypeScript`.	dotnet openapi add url `https://contoso.com/openapi.json` --code-generator
-h	--help	Show help information	dotnet openapi add url --help

Arguments

Expand table

Argument	Description	Example
source-URL	The source to create a reference from. Must be a URL.	dotnet openapi add url `https://contoso.com/openapi.json`

Remove

Removes the OpenAPI reference matching the given filename from the `.csproj` file. When the OpenAPI reference is removed, clients won't be generated. Local `.json` and `.yaml` files are deleted.

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi remove *--updateProject .\Ref.csproj* .\OpenAPI.json
-h	--help	Show help information	dotnet openapi remove --help

Arguments

Expand table

Argument	Description	Example
source-file	The source to remove the reference to.	dotnet openapi remove .*OpenAPI.json*

Refresh

Refreshes the local version of a file that was downloaded using the latest content from the download URL.

Options

Expand table

Short option	Long option	Description	Example
-p	--updateProject	The project to operate on.	dotnet openapi refresh *--updateProject .\Ref.csproj* `https://contoso.com/openapi.json`
-h	--help	Show help information	dotnet openapi refresh --help

Arguments

Expand table

Argument	Description	Example
source-URL	The URL to refresh the reference from.	dotnet openapi refresh `https://contoso.com/openapi.json`

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core web API documentation with Swagger / OpenAPI

Article • 08/26/2024

By Christoph Nienaber and Rico Suter

Swagger (OpenAPI) is a language-agnostic specification for describing REST APIs. It allows both computers and humans to understand the capabilities of a REST API without direct access to the source code. Its main goals are to:

- Minimize the amount of work needed to connect decoupled services.
- Reduce the amount of time needed to accurately document a service.

The two main OpenAPI implementations for .NET are Swashbuckle and NSwag, see:

- Getting Started with Swashbuckle
- Getting Started with NSwag

OpenAPI vs. Swagger

The Swagger project was donated to the OpenAPI Initiative in 2015 and has since been referred to as OpenAPI. Both names are used interchangeably. However, "OpenAPI" refers to the specification. "Swagger" refers to the family of open-source and commercial products from SmartBear that work with the OpenAPI Specification. Subsequent open-source products, such as OpenAPIGenerator, also fall under the Swagger family name, despite not being released by SmartBear.

In short:

- OpenAPI is a specification.
- Swagger is tooling that uses the OpenAPI specification. For example, OpenAPIGenerator and SwaggerUI.

OpenAPI specification (`openapi.json`)

The OpenAPI specification is a document that describes the capabilities of your API. The document is based on the XML and attribute annotations within the controllers and models. It's the core part of the OpenAPI flow and is used to drive tooling such as SwaggerUI. By default, it's named `openapi.json`. Here's an example of an OpenAPI specification, reduced for brevity:

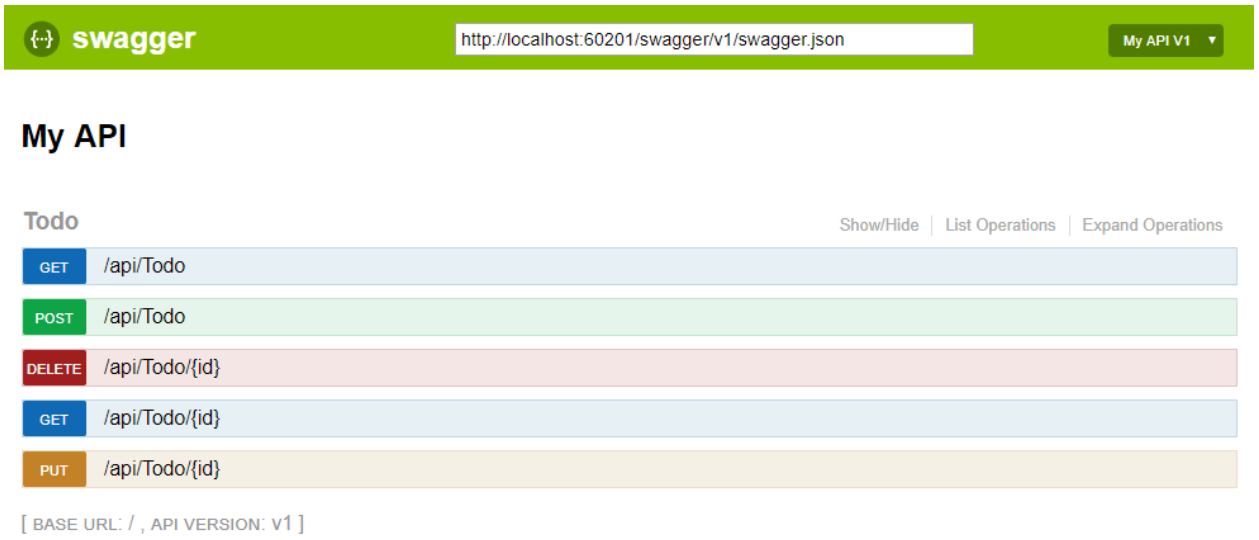
JSON


```
{
  "openapi": "3.0.1",
  "info": {
    "title": "API V1",
    "version": "v1"
  },
  "paths": {
    "/api/Todo": {
      "get": {
        "tags": [
          "Todo"
        ],
        "operationId": "ApiTodoGet",
        "responses": {
          "200": {
            "description": "Success",
            "content": {
              "text/plain": {
                "schema": {
                  "type": "array",
                  "items": {
                    "$ref": "#/components/schemas/ToDoItem"
                  }
                }
              },
              "application/json": {
                "schema": {
                  "type": "array",
                  "items": {
                    "$ref": "#/components/schemas/ToDoItem"
                  }
                }
              },
              "text/json": {
                "schema": {
                  "type": "array",
                  "items": {
                    "$ref": "#/components/schemas/ToDoItem"
                  }
                }
              }
            }
          }
        }
      },
      "post": {
        …
      }
    },
    "/api/Todo/{id}": {
      "get": {
        …
```

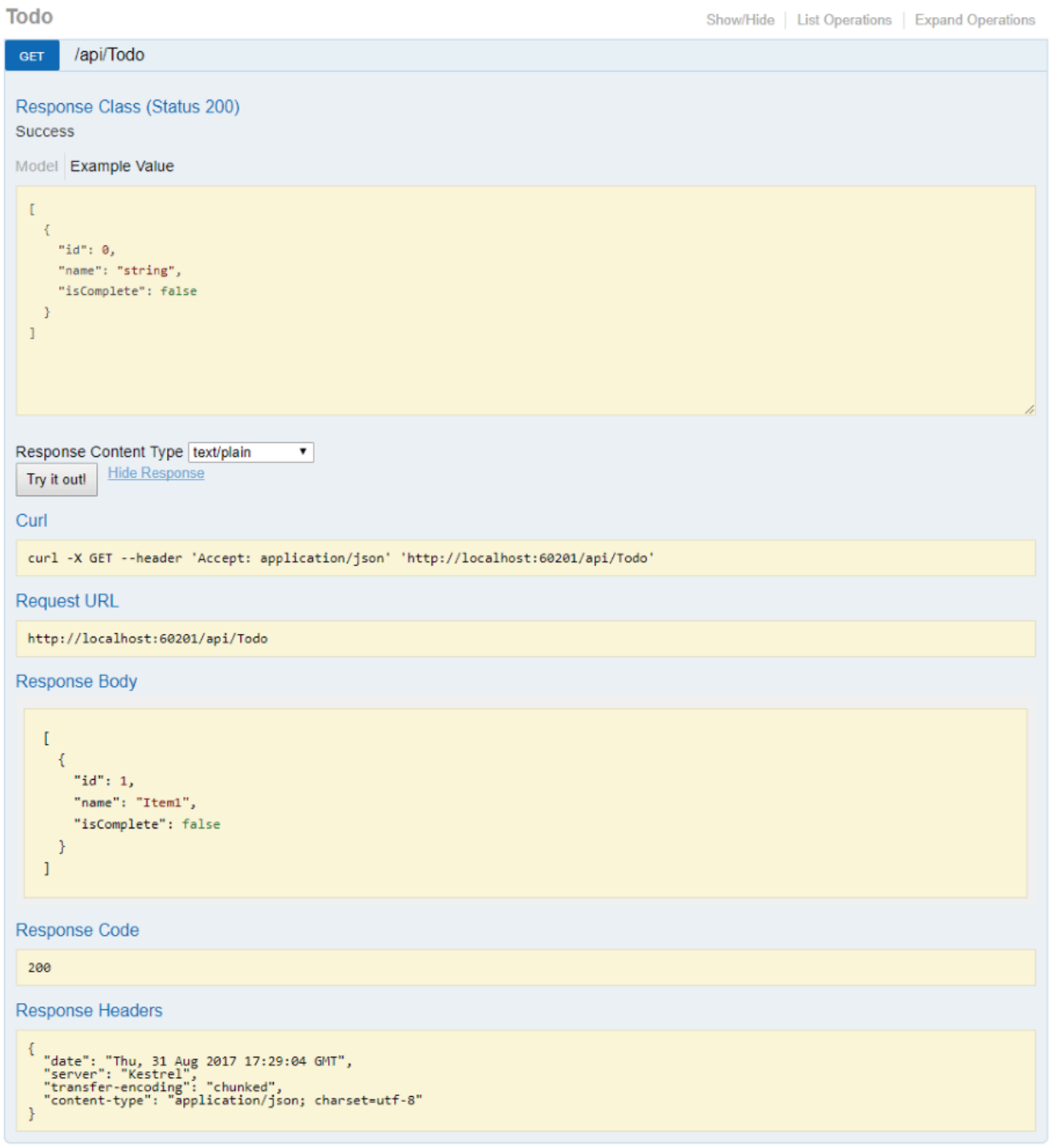
```
      },
      "put": {
        …
      },
      "delete": {
        …
      }
    }
  },
  "components": {
    "schemas": {
      "ToDoItem": {
        "type": "object",
        "properties": {
          "id": {
            "type": "integer",
            "format": "int32"
          },
          "name": {
            "type": "string",
            "nullable": true
          },
          "isCompleted": {
            "type": "boolean"
          }
        },
        "additionalProperties": false
      }
    }
  }
}
```

Swagger UI

Swagger UI offers a web-based UI that provides information about the service, using the generated OpenAPI specification. Both Swashbuckle and NSwag include an embedded version of Swagger UI, so that it can be hosted in your ASP.NET Core app using a middleware registration call. The web UI looks like this:



Each public action method in your controllers can be tested from the UI. Select a method name to expand the section. Add any necessary parameters, and select **Try it out!**.



> **Note**
>
> The Swagger UI version used for the screenshots is version 2. For a version 3 example, see Petstore example.

Securing Swagger UI endpoints

Call MapSwagger().RequireAuthorization to secure the Swagger UI endpoints. The following example secures the swagger endpoints:

```
C#
```

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

  if (app.Environment.IsDevelopment())
  {
    app.UseSwagger();
    app.UseSwaggerUI();
  }

app.UseHttpsRedirection();

var summaries = new[]
{
    "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot",
"Sweltering", "Scorching"
};

app.MapSwagger().RequireAuthorization();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast")
.WithOpenApi();

app.Run();

internal record WeatherForecast(DateOnly Date, int TemperatureC, string?
Summary)
{
```

```
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

In the preceding code, the `/weatherforecast` endpoint doesn't need authorization, but the Swagger endpoints do.

The following Curl passes a JWT token to test the Swagger UI endpoint:

Bash

```
curl -i -H "Authorization: Bearer {token}" https://localhost:
{port}/swagger/v1/swagger.json
```

For more information on testing with JWT tokens, see Generate tokens with dotnet user-jwts.

Generate an XML documentation file at compile time.

See GenerateDocumentationFile for more information.

Next steps

- Get started with Swashbuckle
- Get started with NSwag

Get started with Swashbuckle and ASP.NET Core

Article • 08/27/2024

> **Note**
>
> Swashbuckle is not available in .NET 9 and later. For an alternative, see Overview of OpenAPI support in ASP.NET Core API apps.

There are three main components to Swashbuckle:

- Swashbuckle.AspNetCore.Swagger: a Swagger object model and middleware to expose `SwaggerDocument` objects as JSON endpoints.
- Swashbuckle.AspNetCore.SwaggerGen: a Swagger generator that builds `SwaggerDocument` objects directly from your routes, controllers, and models. It's typically combined with the Swagger endpoint middleware to automatically expose Swagger JSON.
- Swashbuckle.AspNetCore.SwaggerUI: an embedded version of the Swagger UI tool. It interprets Swagger JSON to build a rich, customizable experience for describing the web API functionality. It includes built-in test harnesses for the public methods.

Package installation

Swashbuckle can be added with the following approaches:

Visual Studio

- From the **Package Manager Console** window:
  - Go to **View** > **Other Windows** > **Package Manager Console**
  - Navigate to the directory in which the `.csproj` file exists
  - Execute the following command:

    PowerShell

```
Install-Package Swashbuckle.AspNetCore -Version 6.6.2
```

- From the **Manage NuGet Packages** dialog:
  - Right-click the project in **Solution Explorer** > **Manage NuGet Packages**
  - Set the **Package source** to "nuget.org"
  - Ensure the "Include prerelease" option is enabled
  - Enter "Swashbuckle.AspNetCore" in the search box
  - Select the latest "Swashbuckle.AspNetCore" package from the **Browse** tab and click **Install**

Add and configure Swagger middleware

Add the Swagger generator to the services collection in `Program.cs`:

C#

```
builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
```

The call to AddEndpointsApiExplorer shown in the preceding example is required only for minimal APIs. For more information, see this StackOverflow post.

Enable the middleware for serving the generated JSON document and the Swagger UI, also in `Program.cs`:

C#

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

The preceding code adds the Swagger middleware only if the current environment is set to Development. The `UseSwaggerUI` method call enables an embedded version of the Swagger UI tool.

Launch the app and navigate to `https://localhost:<port>/swagger/v1/swagger.json`. The generated document describing the endpoints appears as shown in OpenAPI

specification (openapi.json).

The Swagger UI can be found at `https://localhost:<port>/swagger`. Explore the API via Swagger UI and incorporate it in other programs.

> **Tip**
>
> To serve the Swagger UI at the app's root (`https://localhost:<port>/`), set the `RoutePrefix` property to an empty string:

```
C#

if (builder.Environment.IsDevelopment())
{
    app.UseSwaggerUI(options => // UseSwaggerUI is called only in
Development.
    {
        options.SwaggerEndpoint("/swagger/v1/swagger.json", "v1");
        options.RoutePrefix = string.Empty;
    });
}
```

If using directories with IIS or a reverse proxy, set the Swagger endpoint to a relative path using the `./` prefix. For example, `./swagger/v1/swagger.json`. Using `/swagger/v1/swagger.json` instructs the app to look for the JSON file at the true root of the URL (plus the route prefix, if used). For example, use `https://localhost:<port>/<route_prefix>/swagger/v1/swagger.json` instead of `https://localhost:<port>/<virtual_directory>/<route_prefix>/swagger/v1/swagger.json`.

> **Note**
>
> By default, Swashbuckle generates and exposes Swagger JSON in version 3.0 of the specification—officially called the OpenAPI Specification. To support backwards compatibility, you can opt into exposing JSON in the 2.0 format instead. This 2.0 format is important for integrations such as Microsoft Power Apps and Microsoft Flow that currently support OpenAPI version 2.0. To opt into the 2.0 format, set the `SerializeAsV2` property in `Program.cs`:

```
C#

app.UseSwagger(options =>
{
```

```
    options.SerializeAsV2 = true;
});
```

Customize and extend

Swagger provides options for documenting the object model and customizing the UI to match your theme.

API info and description

The configuration action passed to the `AddSwaggerGen` method adds information such as the author, license, and description.

In `Program.cs`, import the following namespace to use the `OpenApiInfo` class: H source="~/tutorials/web-api-help-pages-using-swagger

C#

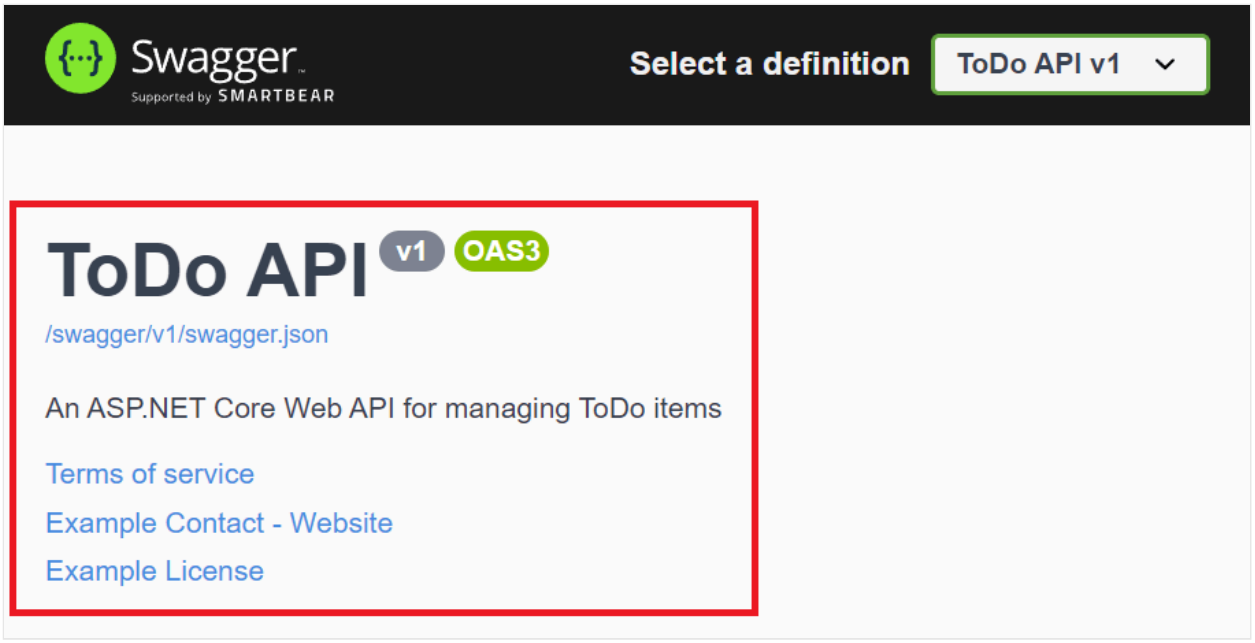
```
using Microsoft.OpenApi.Models;
```

Using the `OpenApiInfo` class, modify the information displayed in the UI:

C#

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "An ASP.NET Core Web API for managing ToDo items",
        TermsOfService = new Uri("https://example.com/terms"),
        Contact = new OpenApiContact
        {
            Name = "Example Contact",
            Url = new Uri("https://example.com/contact")
        },
        License = new OpenApiLicense
        {
            Name = "Example License",
            Url = new Uri("https://example.com/license")
        }
    });
});
```

The Swagger UI displays the version's information:



XML comments

XML comments can be enabled with the following approaches:

Visual Studio

- Right-click the project in **Solution Explorer** and select `Edit <project_name>.csproj`.
- Add GenerateDocumentationFile to the `.csproj` file:

XML

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

Enabling XML comments provides debug information for undocumented public types and members. Undocumented types and members are indicated by the warning message. For example, the following message indicates a violation of warning code 1591:

text

```
warning CS1591: Missing XML comment for publicly visible type or member
'TodoController'
```

To suppress warnings project-wide, define a semicolon-delimited list of warning codes to ignore in the project file. Appending the warning codes to `$(NoWarn);` applies the C# default values too.

XML

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

To suppress warnings only for specific members, enclose the code in #pragma warning preprocessor directives. This approach is useful for code that shouldn't be exposed via the API docs. In the following example, warning code CS1591 is ignored for the entire `TodoContext` class. Enforcement of the warning code is restored at the close of the class definition. Specify multiple warning codes with a comma-delimited list.

C#

```
namespace SwashbuckleSample.Models;

#pragma warning disable CS1591
public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options) :
base(options) { }

    public DbSet<TodoItem> TodoItems => Set<TodoItem>();
}
#pragma warning restore CS1591
```

Configure Swagger to use the XML file that's generated with the preceding instructions. For Linux or non-Windows operating systems, file names and paths can be case-sensitive. For example, a `TodoApi.XML` file is valid on Windows but not Ubuntu.

C#

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "An ASP.NET Core Web API for managing ToDo items",
        TermsOfService = new Uri("https://example.com/terms"),
        Contact = new OpenApiContact
        {
            Name = "Example Contact",
```

```
            Url = new Uri("https://example.com/contact")
        },
        License = new OpenApiLicense
        {
            Name = "Example License",
            Url = new Uri("https://example.com/license")
        }
    });

    // using System.Reflection;
    var xmlFilename = $"
{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,
xmlFilename));
});
```

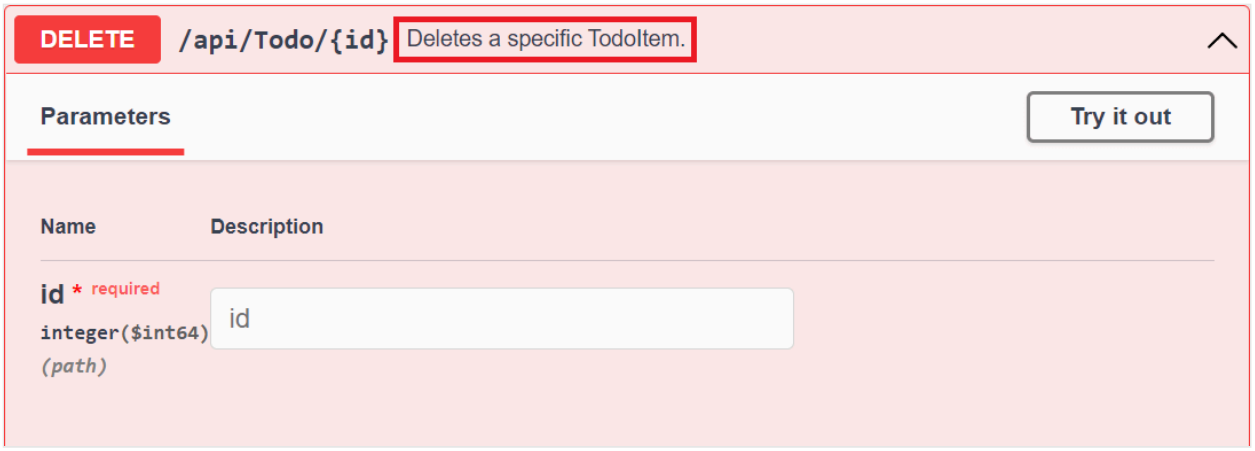
In the preceding code, Reflection is used to build an XML file name matching that of the web API project. The AppContext.BaseDirectory property is used to construct a path to the XML file. Some Swagger features (for example, schemata of input parameters or HTTP methods and response codes from the respective attributes) work without the use of an XML documentation file. For most features, namely method summaries and the descriptions of parameters and response codes, the use of an XML file is mandatory.

Adding triple-slash comments to an action enhances the Swagger UI by adding the description to the section header. Add a <summary> element above the `Delete` action:

C#

```
/// <summary>
/// Deletes a specific TodoItem.
/// </summary>
/// <param name="id"></param>
/// <returns></returns>
[HttpDelete("{id}")]
public async Task<IActionResult> Delete(long id)
{
    var item = await _context.TodoItems.FindAsync(id);

    if (item is null)
    {
        return NotFound();
    }

    _context.TodoItems.Remove(item);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

The Swagger UI displays the inner text of the preceding code's `<summary>` element:



The UI is driven by the generated JSON schema:

JSON

```
"delete": {
    "tags": [
        "Todo"
    ],
    "summary": "Deletes a specific TodoItem.",
    "parameters": [
        {
            "name": "id",
            "in": "path",
            "description": "",
            "required": true,
            "schema": {
                "type": "integer",
                "format": "int64"
            }
        }
    ],
    "responses": {
        "200": {
            "description": "Success"
        }
    }
},
```

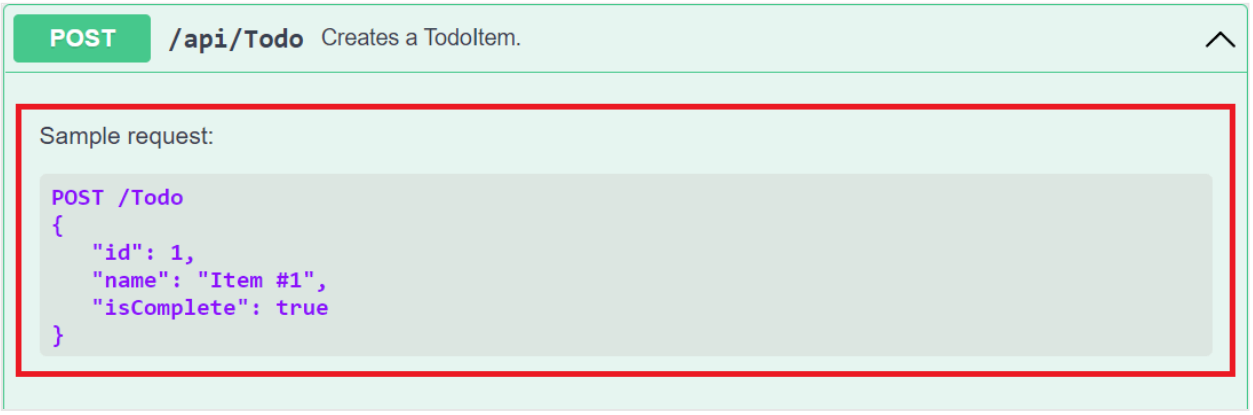
Add a <remarks> element to the `Create` action method documentation. It supplements information specified in the `<summary>` element and provides a more robust Swagger UI. The `<remarks>` element content can consist of text, JSON, or XML.

C#

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
```

```
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item #1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(Get), new { id = item.Id }, item);
}
```

Notice the UI enhancements with these additional comments:



Data annotations

Mark the model with attributes, found in the System.ComponentModel.DataAnnotations namespace, to help drive the Swagger UI components.

Add the `[Required]` attribute to the `Name` property of the `TodoItem` class:

C#

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace SwashbuckleSample.Models;
```

```
public class TodoItem
{
    public long Id { get; set; }

    [Required]
    public string Name { get; set; } = null!;

    [DefaultValue(false)]
    public bool IsComplete { get; set; }
}
```

The presence of this attribute changes the UI behavior and alters the underlying JSON schema:

JSON

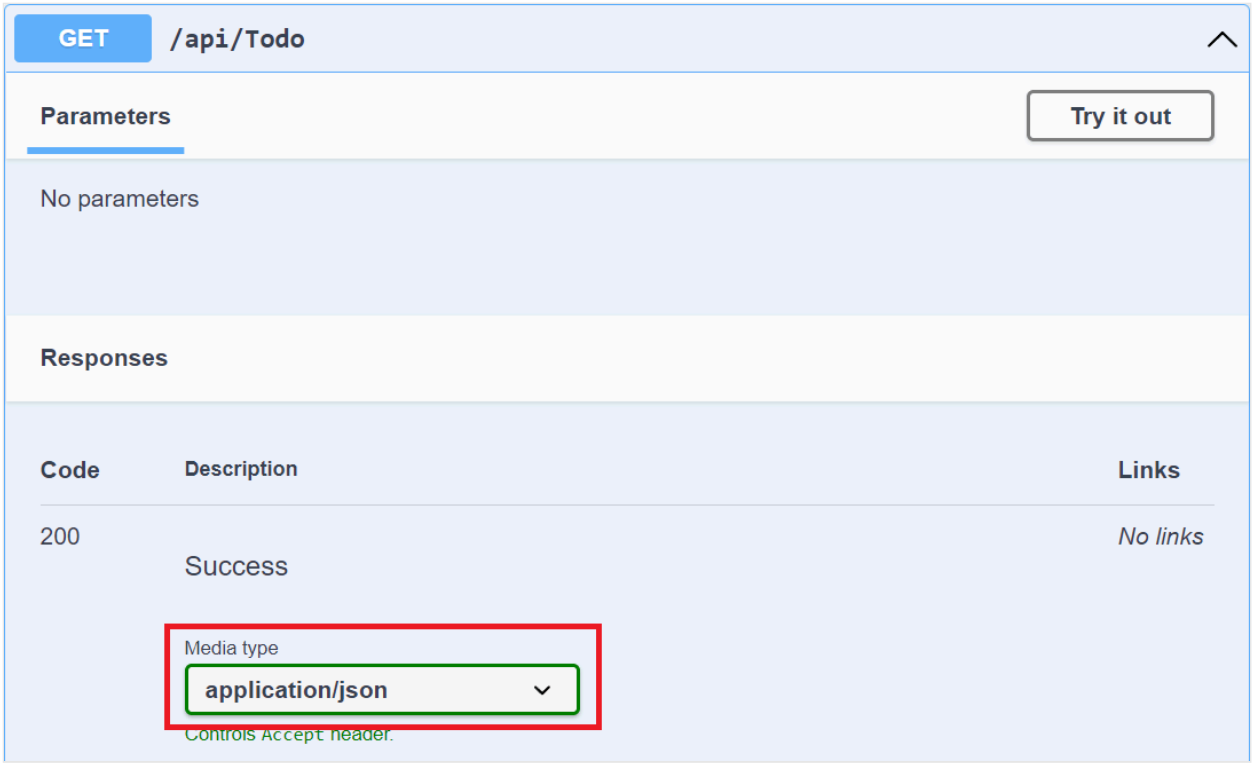
```
"schemas": {
    "TodoItem": {
        "required": [
            "name"
        ],
        "type": "object",
        "properties": {
            "id": {
                "type": "integer",
                "format": "int64"
            },
            "name": {
                "type": "string"
            },
            "isComplete": {
                "type": "boolean",
                "default": false
            }
        },
        "additionalProperties": false
    }
},
```

Add the `[Produces("application/json")]` attribute to the API controller. Its purpose is to declare that the controller's actions support a response content type of *application/json*:

C#

```
[ApiController]
[Route("api/[controller]")]
[Produces("application/json")]
public class TodoController : ControllerBase
{
```

The **Media type** drop-down selects this content type as the default for the controller's GET actions:



As the usage of data annotations in the web API increases, the UI and API help pages become more descriptive and useful.

Describe response types

Developers consuming a web API are most concerned with what's returned—specifically response types and error codes (if not standard). The response types and error codes are denoted in the XML comments and data annotations.

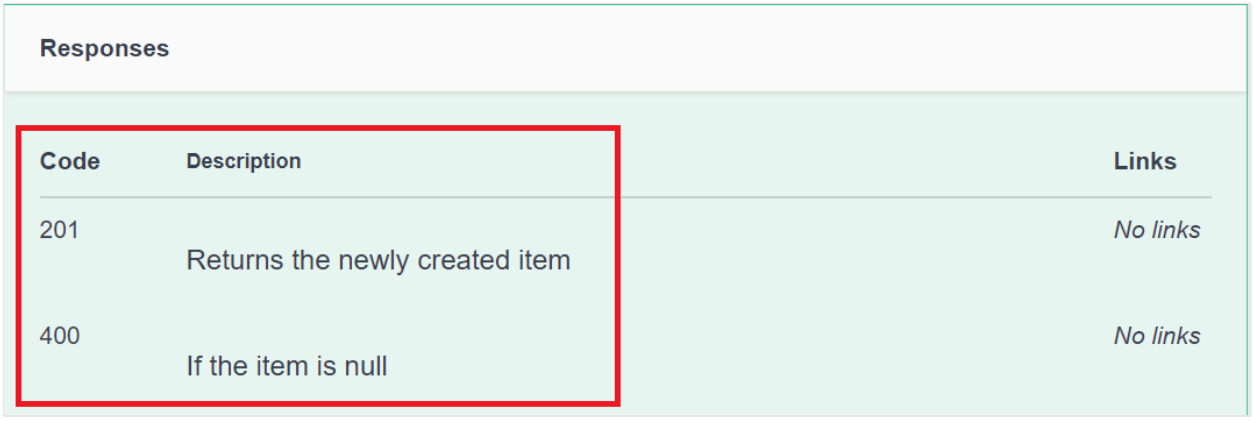
The `Create` action returns an HTTP 201 status code on success. An HTTP 400 status code is returned when the posted request body is null. Without proper documentation in the Swagger UI, the consumer lacks knowledge of these expected outcomes. Fix that problem by adding the highlighted lines in the following example:

C#

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <remarks>
/// Sample request:
///
///     POST /Todo
```

```
///     {
///        "id": 1,
///        "name": "Item #1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(Get), new { id = item.Id }, item);
}
```

The Swagger UI now clearly documents the expected HTTP response codes:



Conventions can be used as an alternative to explicitly decorating individual actions with `[ProducesResponseType]`. For more information, see Use web API conventions.

To support the `[ProducesResponseType]` decoration, the Swashbuckle.AspNetCore.Annotations package offers extensions to enable and enrich the response, schema, and parameter metadata.

Customize the UI

The default UI is both functional and presentable. However, API documentation pages should represent your brand or theme. Branding the Swashbuckle components requires adding the resources to serve static files and building the folder structure to host those files.

Enable Static File Middleware:

C#

```
app.UseHttpsRedirection();
app.UseStaticFiles();
app.MapControllers();
```

To inject additional CSS stylesheets, add them to the project's *wwwroot* folder and specify the relative path in the middleware options:

C#

```
if (app.Environment.IsDevelopment())
{
    app.UseSwaggerUI(options => // UseSwaggerUI is called only in
Development.
    {
        options.InjectStylesheet("/swagger-ui/custom.css");
    });
}
```

Additional resources

- View or download sample code (how to download)
- Improve the developer experience of an API with Swagger documentation

Get started with NSwag and ASP.NET Core

Article • 08/26/2024

By Christoph Nienaber, Rico Suter, and Dave Brock

View or download sample code (how to download)

NSwag offers the following capabilities:

- The ability to utilize the Swagger UI and Swagger generator.
- Flexible code generation capabilities.

With NSwag, you don't need an existing API—you can use third-party APIs that incorporate Swagger and generate a client implementation. NSwag allows you to expedite the development cycle and easily adapt to API changes.

Package installation

Install NSwag to:

- Generate the Swagger specification for the implemented web API.
- Serve the Swagger UI to browse and test the web API.
- Serve the Redoc to add API documentation for the Web API.

To use the NSwag ASP.NET Core middleware, install the NSwag.AspNetCore NuGet package. This package contains the middleware to generate and serve the Swagger specification, Swagger UI (v2 and v3), and ReDoc UI. NSwag 14 supports only v3 of the Swagger UI spec.

Use one of the following approaches to install the NSwag NuGet package:

Visual Studio

- From the **Package Manager Console** window:
  - Go to **View** > **Other Windows** > **Package Manager Console**
  - Navigate to the directory in which the `NSwagSample.csproj` file exists
  - Execute the following command:

PowerShell

```
Install-Package NSwag.AspNetCore
```

- From the **Manage NuGet Packages** dialog:
  - Right-click the project in **Solution Explorer** > **Manage NuGet Packages**
  - Set the **Package source** to "nuget.org"
  - Enter "NSwag.AspNetCore" in the search box
  - Select the "NSwag.AspNetCore" package from the **Browse** tab and click **Install**

Add and configure Swagger middleware

Add and configure Swagger in your ASP.NET Core app by performing the following steps:

- Add the OpenApi generator to the services collection in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddOpenApiDocument();
```

- Enable the middleware for serving the generated OpenApi specification, the Swagger UI, and the Redoc UI, also in `Program.cs`:

C#

```
if (app.Environment.IsDevelopment())
{
    // Add OpenAPI 3.0 document serving middleware
    // Available at: http://localhost:<port>/swagger/v1/swagger.json
    app.UseOpenApi();

    // Add web UIs to interact with the document
    // Available at: http://localhost:<port>/swagger
    app.UseSwaggerUi(); // UseSwaggerUI Protected by if
(env.IsDevelopment())
}
```

- Launch the app. Navigate to:

- `http://localhost:<port>/swagger` to view the Swagger UI.
- `http://localhost:<port>/swagger/v1/swagger.json` to view the Swagger specification.

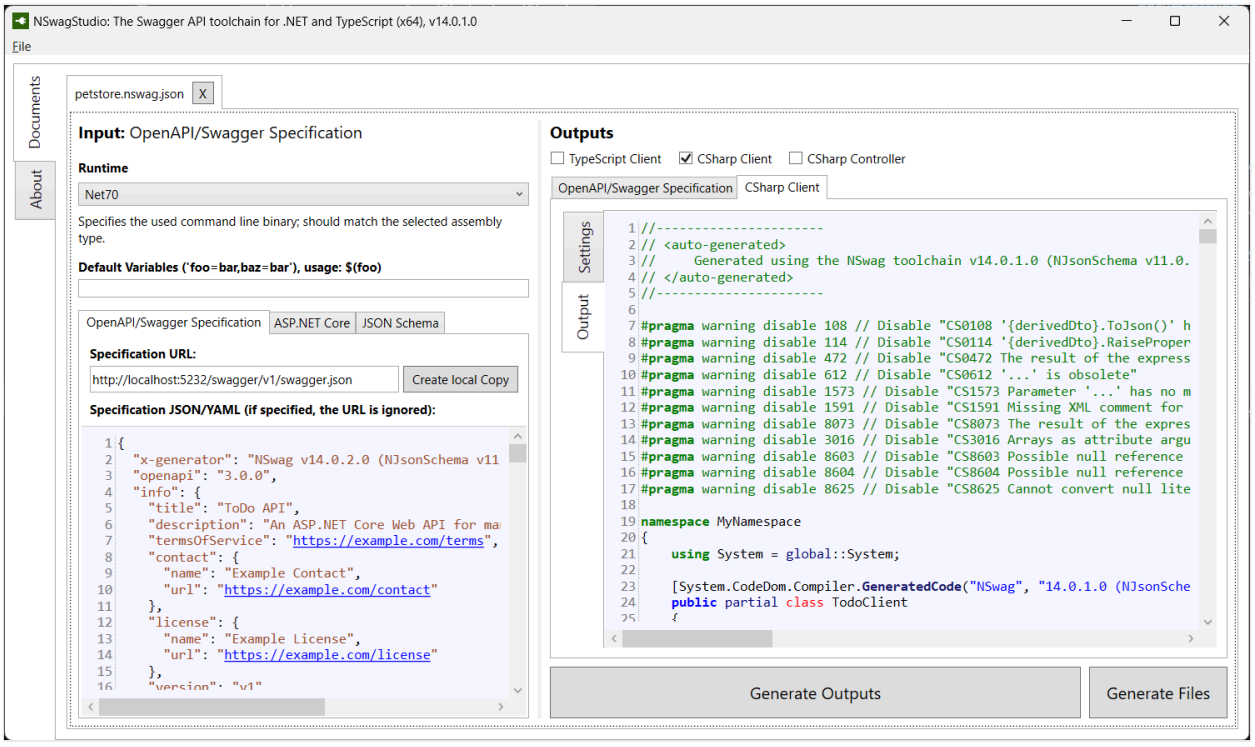
Code generation

You can take advantage of NSwag's code generation capabilities by choosing one of the following options:

- NSwagStudio: A Windows desktop app for generating API client code in C# or TypeScript.
- The NSwag.CodeGeneration.CSharp or NSwag.CodeGeneration.TypeScript NuGet packages for code generation inside your project.
- NSwag from the command line.
- The NSwag.MSBuild NuGet package.
- The Unchase OpenAPI (Swagger) Connected Service: A Visual Studio Connected Service for generating API client code in C# or TypeScript. Also generates C# controllers for OpenAPI services with NSwag.

Generate code with NSwagStudio

- Install NSwagStudio by following the instructions at the NSwagStudio GitHub repository. On the NSwag release page, you can download an xcopy version which can be started without installation and admin privileges.
- Launch NSwagStudio and enter the `swagger.json` file URL in the **Swagger Specification URL** text box. For example, `http://localhost:5232/swagger/v1/swagger.json`.
- Click the **Create local Copy** button to generate a JSON representation of your Swagger specification.



- In the **Outputs** area, click the **CSharp Client** checkbox. Depending on your project, you can also choose **TypeScript Client** or **CSharp Web API Controller**. If you select **CSharp Web API Controller**, a service specification rebuilds the service, serving as a reverse generation.
- Click **Generate Outputs** to produce a complete C# client implementation of the *TodoApi.NSwag* project. To see the generated client code, click the **CSharp Client** tab:

C#

```
namespace MyNamespace
{
    using System = global::System;

    [System.CodeDom.Compiler.GeneratedCode("NSwag", "14.0.1.0 (NJsonSchema
v11.0.0.0 (Newtonsoft.Json v13.0.0.0))")]
    public partial class TodoClient
    {
    #pragma warning disable 8618 // Set by constructor via BaseUrl property
        private string _baseUrl;
    #pragma warning restore 8618 // Set by constructor via BaseUrl property
        private System.Net.Http.HttpClient _httpClient;
        private static System.Lazy<Newtonsoft.Json.JsonSerializerSettings>
_settings = new System.Lazy<Newtonsoft.Json.JsonSerializerSettings>
(CreateSerializerSettings, true);

        public TodoClient(System.Net.Http.HttpClient httpClient)
        {
            BaseUrl = "http://localhost:5232";
            _httpClient = httpClient;
        }
```

```
        private static Newtonsoft.Json.JsonSerializerSettings
CreateSerializerSettings()
        {
            var settings = new Newtonsoft.Json.JsonSerializerSettings();
            UpdateJsonSerializerSettings(settings);
            return settings;
        }

        public string BaseUrl
        {
            get { return _baseUrl; }
            set
            {
                _baseUrl = value;
                if (!string.IsNullOrEmpty(_baseUrl) &&
!_baseUrl.EndsWith("/"))
                    _baseUrl += '/';
            }
        }
        // code omitted for brevity
```

> **Tip**
>
> The C# client code is generated based on selections in the **Settings** tab. Modify the settings to perform tasks such as default namespace renaming and synchronous method generation.

- Copy the generated C# code into a file in the client project that will consume the API.
- Start consuming the web API:

C#

```
var todoClient = new TodoClient(new HttpClient());

// Gets all to-dos from the API
var allTodos = await todoClient.GetAsync();

// Create a new TodoItem, and save it via the API.
await todoClient.CreateAsync(new TodoItem());

// Get a single to-do by ID
var foundTodo = await todoClient.GetByIdAsync(1);
```

Customize API documentation

OpenApi provides options for documenting the object model to ease the consumption of the web API.

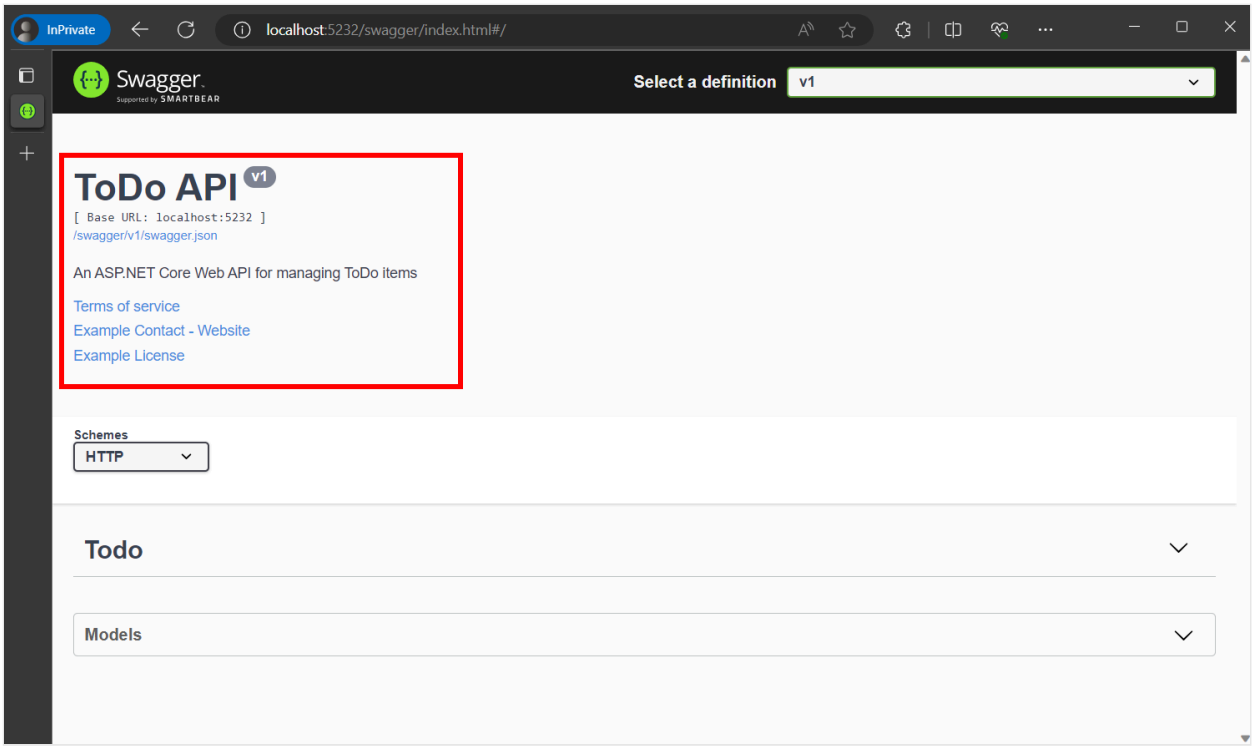
API info and description

In `Program.cs`, update `AddOpenApiDocument` to configure the document info of the Web API and include more information such as the author, license, and description. Import the `NSwag` namespace first to use the `OpenApi` classes.

C#

```
using NSwag;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddOpenApiDocument(options => {
     options.PostProcess = document =>
     {
          document.Info = new OpenApiInfo
          {
              Version = "v1",
              Title = "ToDo API",
              Description = "An ASP.NET Core Web API for managing ToDo
items",
              TermsOfService = "https://example.com/terms",
              Contact = new OpenApiContact
              {
                  Name = "Example Contact",
                  Url = "https://example.com/contact"
              },
              License = new OpenApiLicense
              {
                  Name = "Example License",
                  Url = "https://example.com/license"
              }
          };
     };
});
```

The Swagger UI displays the version's information:



XML comments

To enable XML comments, perform the following steps:

Visual Studio

- Right-click the project in **Solution Explorer** and select `Edit <project_name>.csproj`.
- Manually add the highlighted lines to the `.csproj` file:

XML

```xml
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

Enabling XML comments provides debug information for undocumented public types and members. Undocumented types and members are indicated by the warning message. For example, the following message indicates a violation of warning code 1591:

text

```
warning CS1591: Missing XML comment for publicly visible type or member
'TodoContext'
```

To suppress warnings project-wide, define a semicolon-delimited list of warning codes to ignore in the project file. Appending the warning codes to `$(NoWarn);` applies the C# default values too.

```xml
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <NoWarn>$(NoWarn);1591</NoWarn>
</PropertyGroup>
```

To suppress warnings only for specific members, enclose the code in #pragma warning preprocessor directives. This approach is useful for code that shouldn't be exposed via the API docs. In the following example, warning code CS1591 is ignored for the entire `TodoContext` class. Enforcement of the warning code is restored at the close of the class definition. Specify multiple warning codes with a comma-delimited list.

```csharp
namespace NSwagSample.Models;

#pragma warning disable CS1591
public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options) :
base(options) { }

    public DbSet<TodoItem> TodoItems => Set<TodoItem>();
}
#pragma warning restore CS1591
```

Data annotations

Mark the model with attributes, found in the System.ComponentModel.DataAnnotations namespace, to help drive the Swagger UI components.

Add the `[Required]` attribute to the `Name` property of the `TodoItem` class:

```csharp
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace NSwagSample.Models;

public class TodoItem
{
```

```
    public long Id { get; set; }

    [Required]
    public string Name { get; set; } = null!;

    [DefaultValue(false)]
    public bool IsComplete { get; set; }
}
```

The presence of this attribute changes the UI behavior and alters the underlying JSON schema:

JSON

```
"TodoItem": {
  "type": "object",
  "additionalProperties": false,
  "required": [
    "name"
  ],
  "properties": {
    "id": {
      "type": "integer",
      "format": "int64"
    },
    "name": {
      "type": "string",
      "minLength": 1
    },
    "isComplete": {
      "type": "boolean",
      "default": false
    }
  }
}
```

As the usage of data annotations in the web API increases, the UI and API help pages become more descriptive and useful.

Describe response types

Developers consuming a web API are most concerned with what's returned—specifically response types and error codes (if not standard). The response types and error codes are denoted in the XML comments and data annotations.

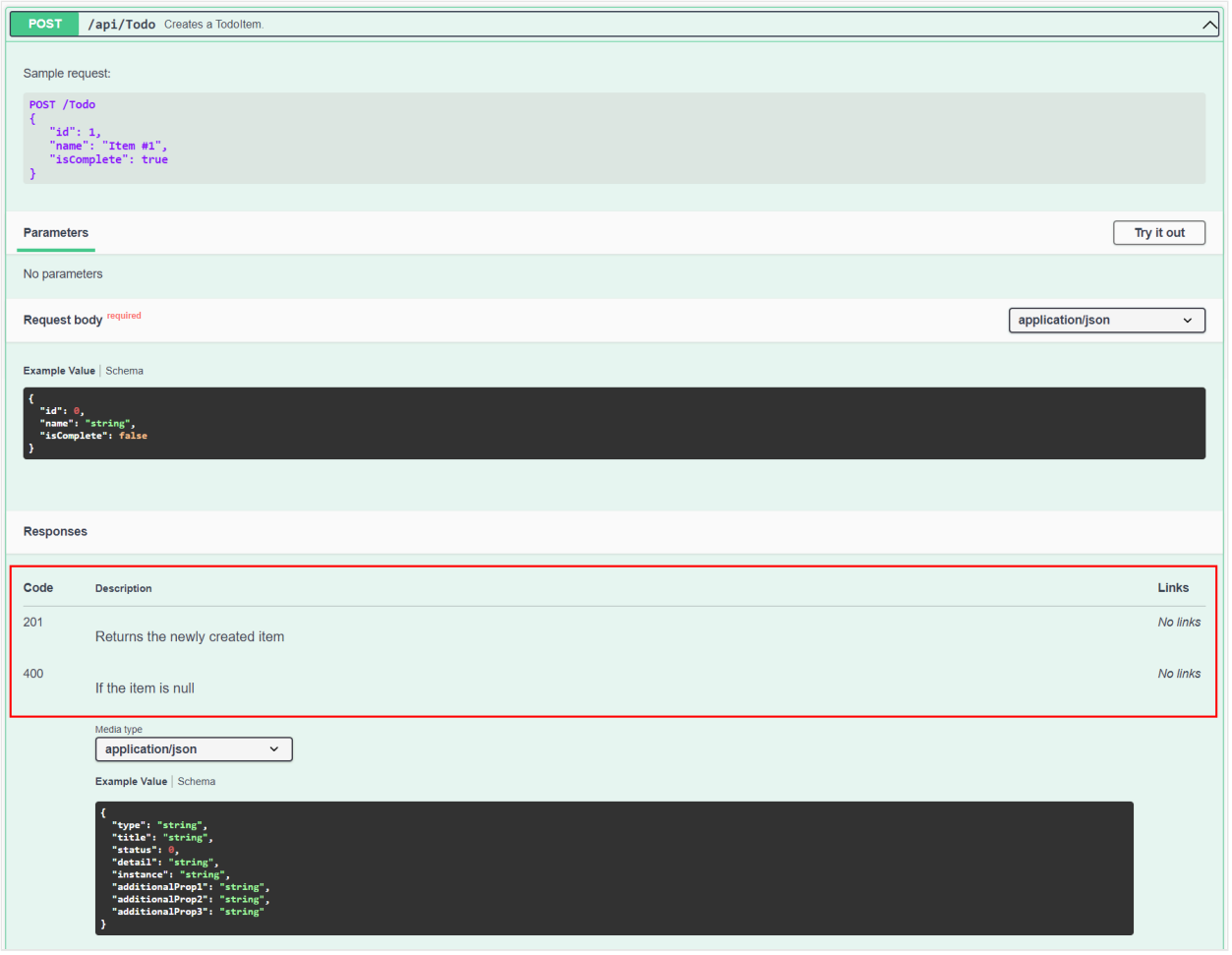
The `Create` action returns an HTTP 201 status code on success. An HTTP 400 status code is returned when the posted request body is `null`. Without proper documentation

in the Swagger UI, the consumer lacks knowledge of these expected outcomes. Fix that problem by adding the highlighted lines in the following example:

C#

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item #1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(Get), new { id = item.Id }, item);
}
```

The Swagger UI now clearly documents the expected HTTP response codes (and the XML comments are also displayed):



Conventions can be used as an alternative to explicitly decorating individual actions with `[ProducesResponseType]`. For more information, see Use web API conventions.

Redoc

Redoc is an alternative to the Swagger UI. It's similar because it also provides a documentation page for the Web API using the OpenAPI specification. The difference is that Redoc UI is more focused on the documentation, and doesn't provide an interactive UI to test the API.

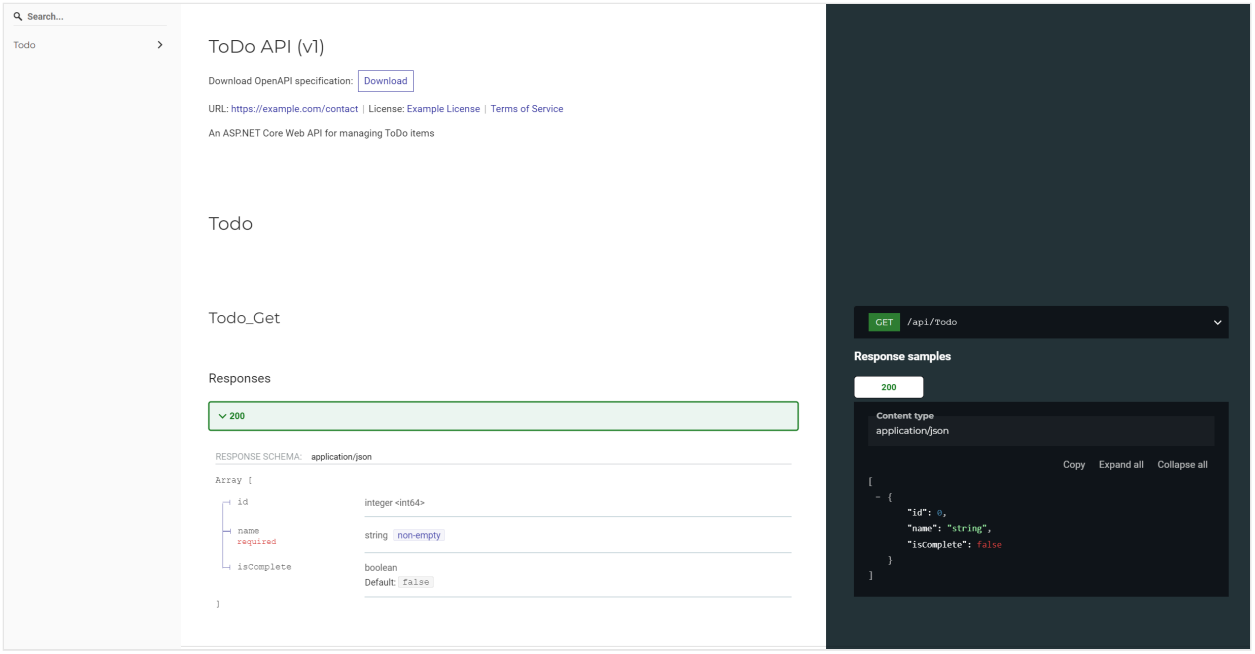
To enable Redoc, add its middleware to `Program.cs`:

C#

```
if (app.Environment.IsDevelopment())
{
    // Add OpenAPI 3.0 document serving middleware
    // Available at: http://localhost:<port>/swagger/v1/swagger.json
    app.UseOpenApi();

    // Add web UIs to interact with the document
    // Available at: http://localhost:<port>/swagger
    app.UseSwaggerUi(); // UseSwaggerUI is called only in Development.
```

```
    // Add ReDoc UI to interact with the document
    // Available at: http://localhost:<port>/redoc
    app.UseReDoc(options =>
    {
        options.Path = "/redoc";
    });
}
```

Run the application and navigate to `http://localhost:<port>/redoc` to view the Redoc UI:



Overview of ASP.NET Core SignalR

Article • 12/02/2024

What is SignalR?

ASP.NET Core SignalR is an open-source library that simplifies adding real-time web functionality to apps. Real-time web functionality enables server-side code to push content to clients instantly.

Good candidates for SignalR:

- Apps that require high frequency updates from the server. Examples are gaming, social networks, voting, auction, maps, and GPS apps.
- Dashboards and monitoring apps. Examples include company dashboards, instant sales updates, or travel alerts.
- Collaborative apps. Whiteboard apps and team meeting software are examples of collaborative apps.
- Apps that require notifications. Social networks, email, chat, games, travel alerts, and many other apps use notifications.

SignalR provides an API for creating server-to-client remote procedure calls (RPC). The RPCs invoke functions on clients from server-side .NET Core code. There are several supported platforms, each with their respective client SDK. Because of this, the programming language being invoked by the RPC call varies.

Here are some features of SignalR for ASP.NET Core:

- Handles connection management automatically.
- Sends messages to all connected clients simultaneously. For example, a chat room.
- Sends messages to specific clients or groups of clients.
- Scales to handle increasing traffic.
- SignalR Hub Protocol

The source is hosted in a SignalR repository on GitHub.

Transports

SignalR supports the following techniques for handling real-time communication (in order of graceful fallback):

- WebSockets

- Server-Sent Events
- Long Polling

SignalR automatically chooses the best transport method that is within the capabilities of the server and client.

Hubs

SignalR uses *hubs* to communicate between clients and servers.

A hub is a high-level pipeline that allows a client and server to call methods on each other. SignalR handles the dispatching across machine boundaries automatically, allowing clients to call methods on the server and vice versa. You can pass strongly-typed parameters to methods, which enables model binding. SignalR provides two built-in hub protocols: a text protocol based on JSON and a binary protocol based on MessagePack. MessagePack generally creates smaller messages compared to JSON. Older browsers must support XHR level 2 to provide MessagePack protocol support.

Hubs call client-side code by sending messages that contain the name and parameters of the client-side method. Objects sent as method parameters are deserialized using the configured protocol. The client tries to match the name to a method in the client-side code. When the client finds a match, it calls the method and passes to it the deserialized parameter data.

Browsers that don't support ECMAScript 6 (ES6)

SignalR targets ES6. For browsers that don't support ES6, transpile the library to ES5. For more information, see Getting Started with ES6 – Transpiling ES6 to ES5 with Traceur and Babel.

Additional resources

- Introduction to ASP.NET Core SignalR
- Get started with SignalR for ASP.NET Core
- Supported Platforms
- Hubs
- JavaScript client
- Browsers that don't support ECMAScript 6 (ES6)
- ASP.NET Core Blazor SignalR guidance

ASP.NET Core SignalR supported platforms

Article • 01/16/2025

Server system requirements

SignalR for ASP.NET Core supports any server platform that ASP.NET Core supports.

JavaScript client

The JavaScript client runs on the current Node.js long-term support (LTS) release and the following browsers:

Expand table

Browser	Version
Apple Safari, including iOS	Current†
Google Chrome, including Android	Current†
Microsoft Edge	Current†
Mozilla Firefox	Current†

†*Current* refers to the latest version of the browser.

The JavaScript client doesn't support Internet Explorer and other older browsers. The client might have unexpected behavior and errors on unsupported browsers.

.NET client

The .NET client runs on any platform supported by ASP.NET Core. For example, .NET Multi-platform App UI (.NET MAUI) developers can use SignalR for building Android and iOS apps.

If the server runs IIS, the WebSockets transport requires IIS 8.0 or later on Windows Server 2012 or later. Other transports are supported on all platforms.

Java client

The Java client supports Java 8 and later versions.

Unsupported clients

The following clients are available but are experimental or unofficial. The following clients aren't currently supported and may never be supported:

- C++ client
- Swift client

Browsers that don't support ECMAScript 6 (ES6)

SignalR targets ES6. For browsers that don't support ES6, transpile the library to ES5. For more information, see Getting Started with ES6 – Transpiling ES6 to ES5 with Traceur and Babel.

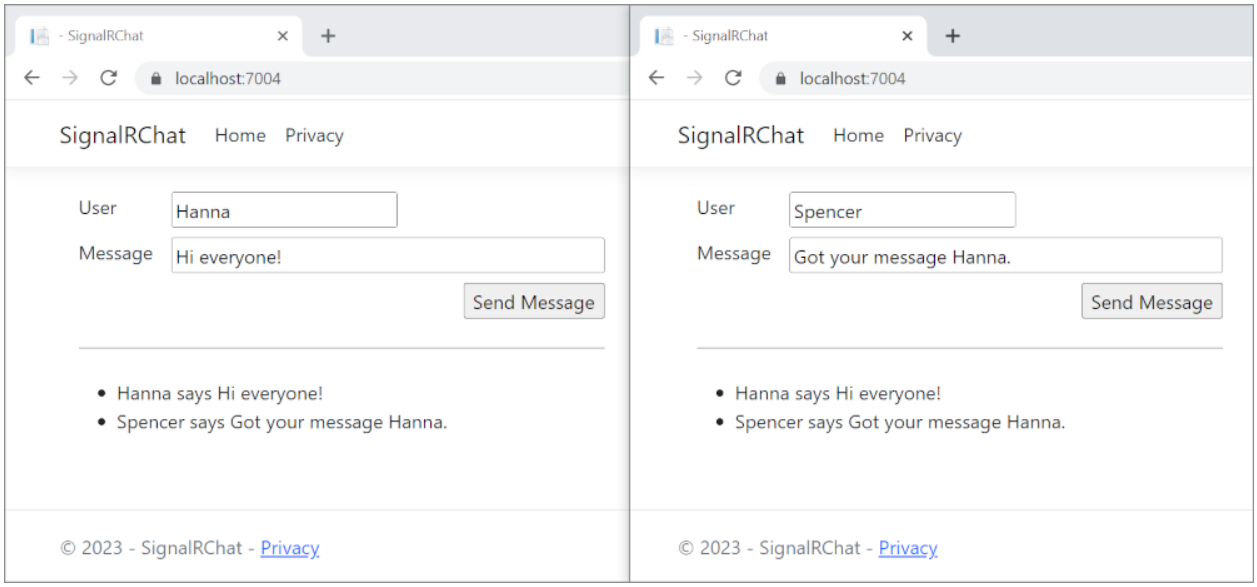
Tutorial: Get started with ASP.NET Core SignalR

Article • 11/16/2023

This tutorial teaches the basics of building a real-time app using SignalR. You learn how to:

- ✓ Create a web project.
- ✓ Add the SignalR client library.
- ✓ Create a SignalR hub.
- ✓ Configure the project to use SignalR.
- ✓ Add code that sends messages from any client to all connected clients.

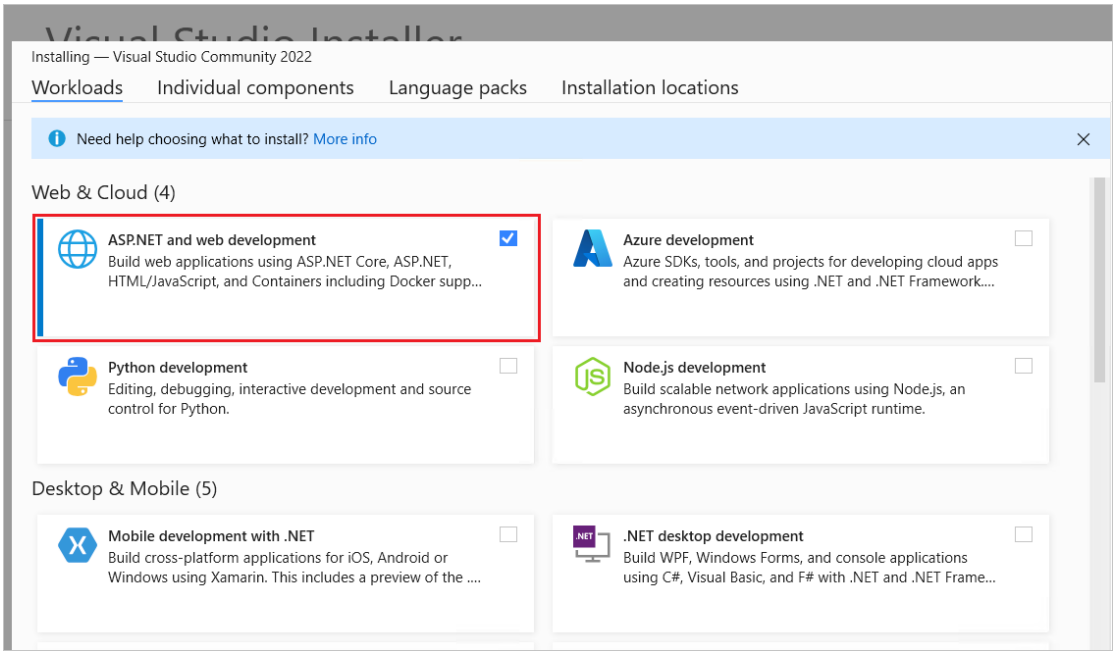
At the end, you'll have a working chat app:



Prerequisites

Visual Studio

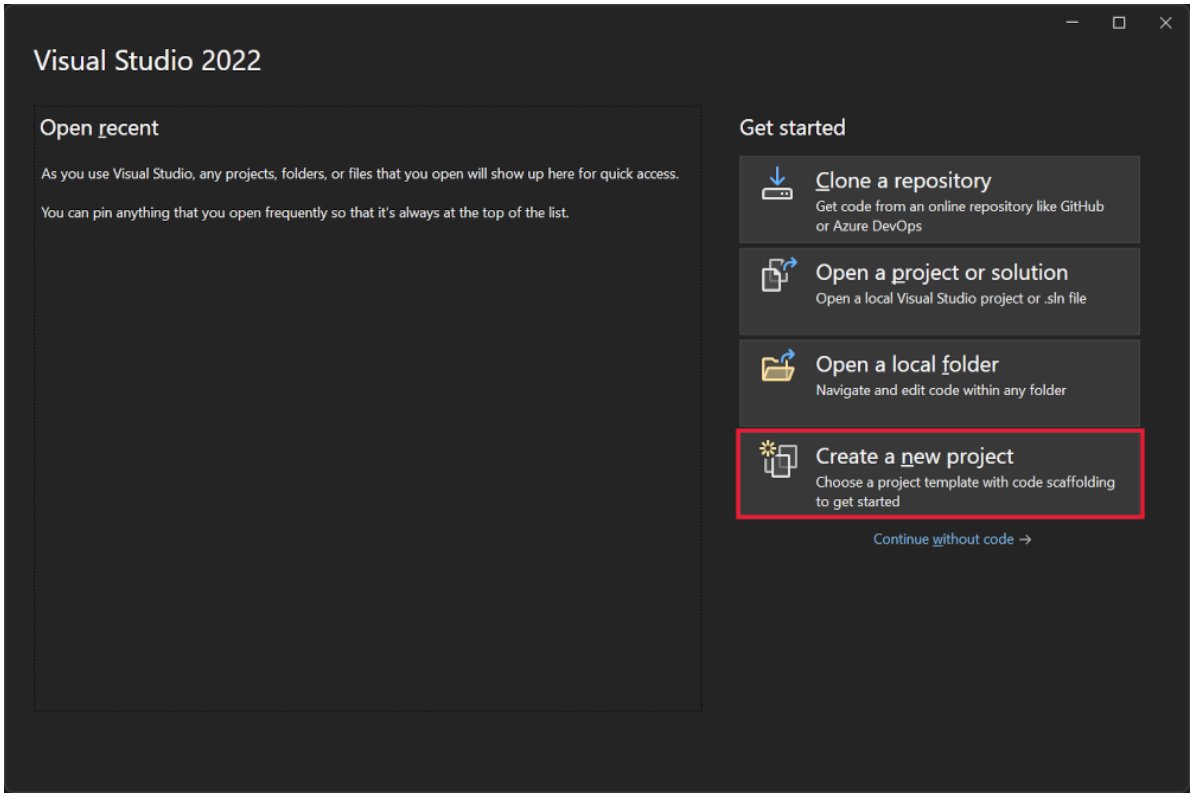
- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



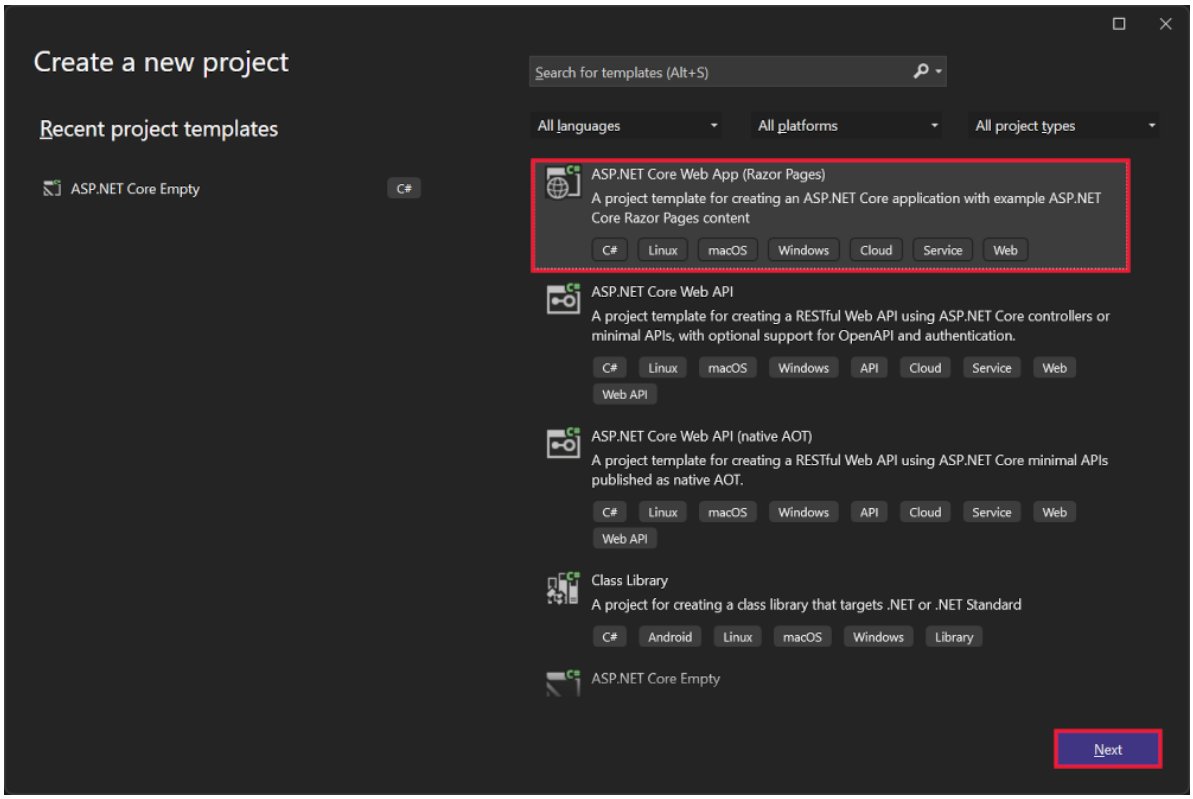
Create a web app project

Visual Studio

Start Visual Studio 2022 and select **Create a new project**.



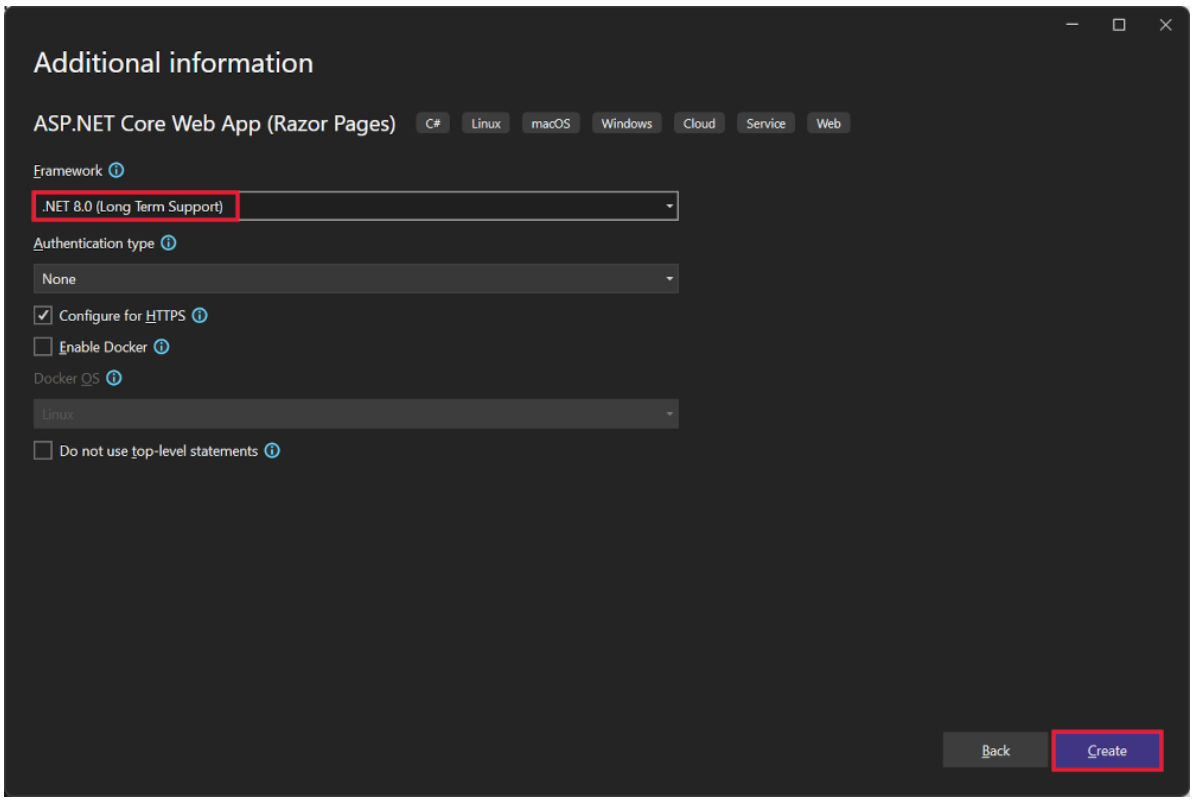
In the **Create a new project** dialog, select **ASP.NET Core Web App (Razor Pages)**, and then select **Next**.



In the **Configure your new project** dialog, enter `SignalRChat` for **Project name**. It's important to name the project `SignalRChat`, including matching the capitalization, so the namespaces match the code in the tutorial.

Select **Next**.

In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.



Add the SignalR client library

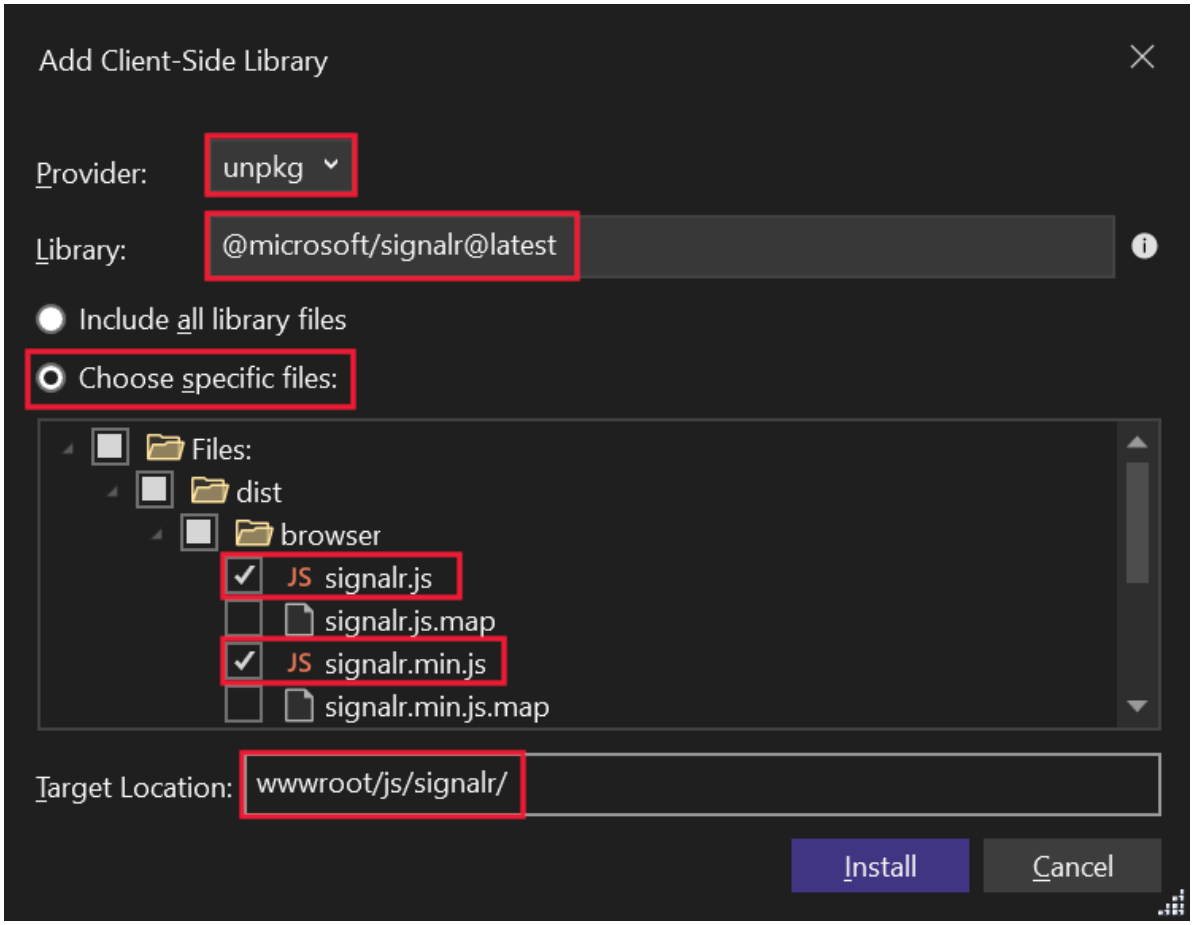
The SignalR server library is included in the ASP.NET Core shared framework. The JavaScript client library isn't automatically included in the project. For this tutorial, use Library Manager (LibMan) to get the client library from unpkg. `unpkg` is a fast, global content delivery network for everything on npm.

Visual Studio

In **Solution Explorer**, right-click the project, and select **Add** > **Client-Side Library**.

In the **Add Client-Side Library** dialog:

- Select **unpkg** for **Provider**
- Enter `@microsoft/signalr@latest` for **Library**.
- Select **Choose specific files**, expand the *dist/browser* folder, and select `signalr.js` and `signalr.min.js`.
- Set **Target Location** to `wwwroot/js/signalr/`.
- Select **Install**.



LibMan creates a `wwwroot/js/signalr` folder and copies the selected files to it.

Create a SignalR hub

A *hub* is a class that serves as a high-level pipeline that handles client-server communication.

In the SignalRChat project folder, create a `Hubs` folder.

In the `Hubs` folder, create the `ChatHub` class with the following code:

C#

```
using Microsoft.AspNetCore.SignalR;

namespace SignalRChat.Hubs
{
    public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
}
```

The `ChatHub` class inherits from the SignalR Hub class. The `Hub` class manages connections, groups, and messaging.

The `SendMessage` method can be called by a connected client to send a message to all clients. JavaScript client code that calls the method is shown later in the tutorial. SignalR code is asynchronous to provide maximum scalability.

Configure SignalR

The SignalR server must be configured to pass SignalR requests to SignalR. Add the following highlighted code to the `Program.cs` file.

C#

```
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();
```

```
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");

app.Run();
```

The preceding highlighted code adds SignalR to the ASP.NET Core dependency injection and routing systems.

Add SignalR client code

Replace the content in `Pages/Index.cshtml` with the following code:

CSHTML

```
@page
<div class="container">
    <div class="row p-1">
        <div class="col-1">User</div>
        <div class="col-5"><input type="text" id="userInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-1">Message</div>
        <div class="col-5"><input type="text" class="w-100"
id="messageInput" /></div>
    </div>
    <div class="row p-1">
        <div class="col-6 text-end">
            <input type="button" id="sendButton" value="Send Message" />
        </div>
    </div>
    <div class="row p-1">
        <div class="col-6">
            <hr />
        </div>
```

```
    </div>
    <div class="row p-1">
        <div class="col-6">
            <ul id="messagesList"></ul>
        </div>
    </div>
</div>
<script src="~/js/signalr/dist/browser/signalr.js"></script>
<script src="~/js/chat.js"></script>
```

The preceding markup:

- Creates text boxes and a submit button.
- Creates a list with `id="messagesList"` for displaying messages that are received from the SignalR hub.
- Includes script references to SignalR and the `chat.js` app code is created in the next step.

In the `wwwroot/js` folder, create a `chat.js` file with the following code:

JavaScript

```
"use strict";

var connection = new
signalR.HubConnectionBuilder().withUrl("/chatHub").build();

//Disable the send button until connection is established.
document.getElementById("sendButton").disabled = true;

connection.on("ReceiveMessage", function (user, message) {
    var li = document.createElement("li");
    document.getElementById("messagesList").appendChild(li);
    // We can assign user-supplied strings to an element's textContent
because it
    // is not interpreted as markup. If you're assigning in any other way,
you
    // should be aware of possible script injection concerns.
    li.textContent = `${user} says ${message}`;
});

connection.start().then(function () {
    document.getElementById("sendButton").disabled = false;
}).catch(function (err) {
    return console.error(err.toString());
});

document.getElementById("sendButton").addEventListener("click", function
(event) {
    var user = document.getElementById("userInput").value;
    var message = document.getElementById("messageInput").value;
```

```
    connection.invoke("SendMessage", user, message).catch(function (err) {
        return console.error(err.toString());
    });
    event.preventDefault();
});
```

The preceding JavaScript:

- Creates and starts a connection.
- Adds to the submit button a handler that sends messages to the hub.
- Adds to the connection object a handler that receives messages from the hub and adds them to the list.

Run the app

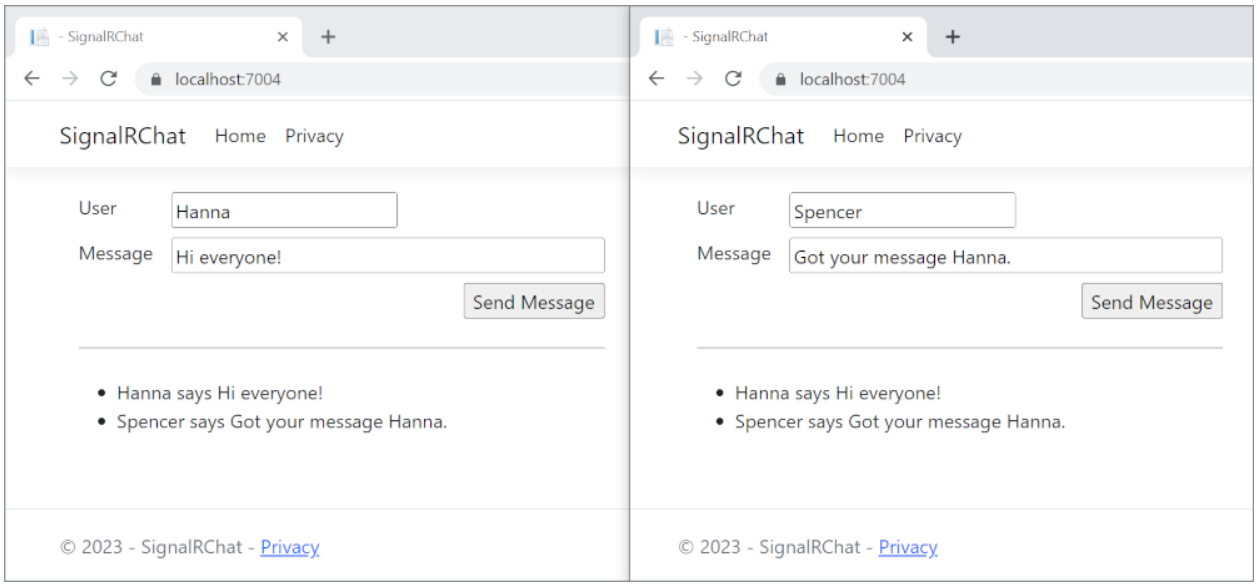
Visual Studio

Select `Ctrl`+`F5` to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

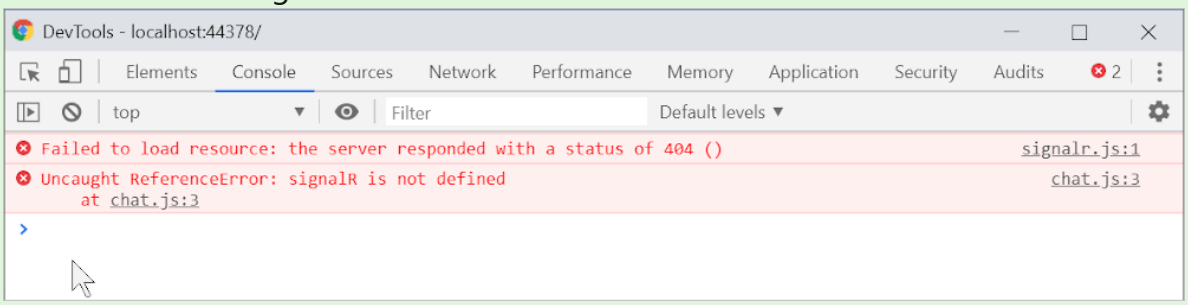
Choose either browser, enter a name and message, and select the **Send Message** button.

The name and message are displayed on both pages instantly.



Tip

If the app doesn't work, open the browser developer tools (F12) and go to the console. Look for possible errors related to HTML and JavaScript code. For example, if `signalr.js` was put in a different folder than directed, the reference to that file won't work resulting in a 404 error in the console.



If an `ERR_SPDY_INADEQUATE_TRANSPORT_SECURITY` error has occurred in Chrome, run the following commands to update the development certificate:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Publish to Azure

For information on deploying to Azure, see Quickstart: Deploy an ASP.NET web app. For more information on Azure SignalR Service, see What is Azure SignalR Service?.

Next steps

- Use hubs
- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- View or download sample code (how to download)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Tutorial: Get started with ASP.NET Core SignalR using TypeScript and Webpack

Article • 11/16/2023

By Sébastien Sougnez

This tutorial demonstrates using Webpack in an ASP.NET Core SignalR web app to bundle and build a client written in TypeScript. Webpack enables developers to bundle and build the client-side resources of a web app.

In this tutorial, you learn how to:

- ✓ Create an ASP.NET Core SignalR app
- ✓ Configure the SignalR server
- ✓ Configure a build pipeline using Webpack
- ✓ Configure the SignalR TypeScript client
- ✓ Enable communication between the client and the server

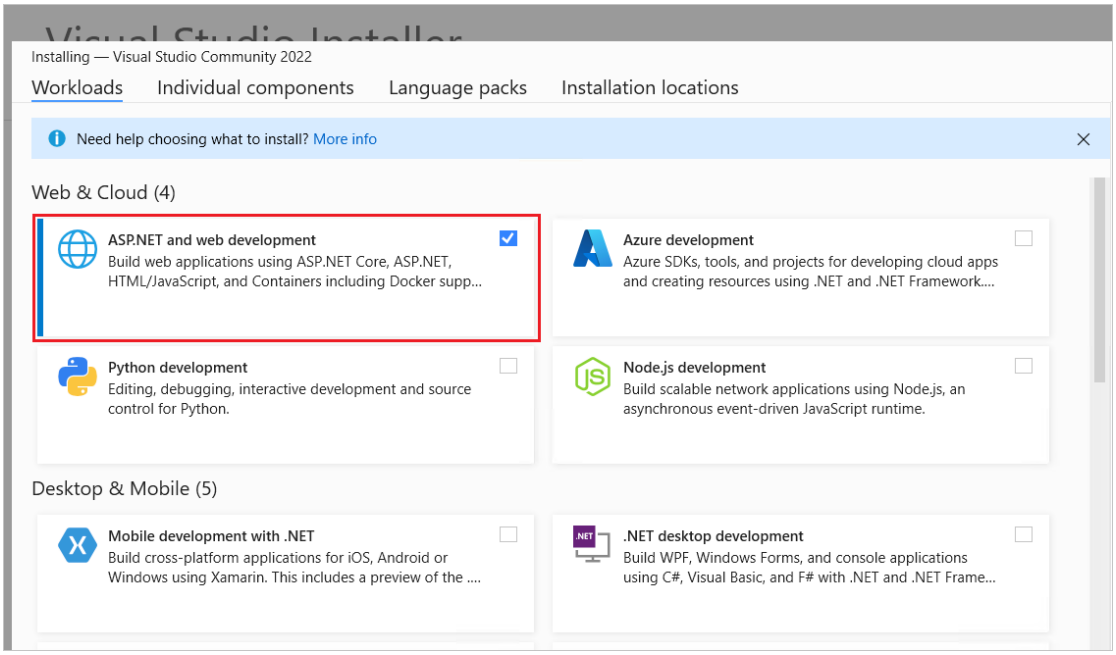
View or download sample code (how to download)

Prerequisites

- Node.js with npm

Visual Studio

- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



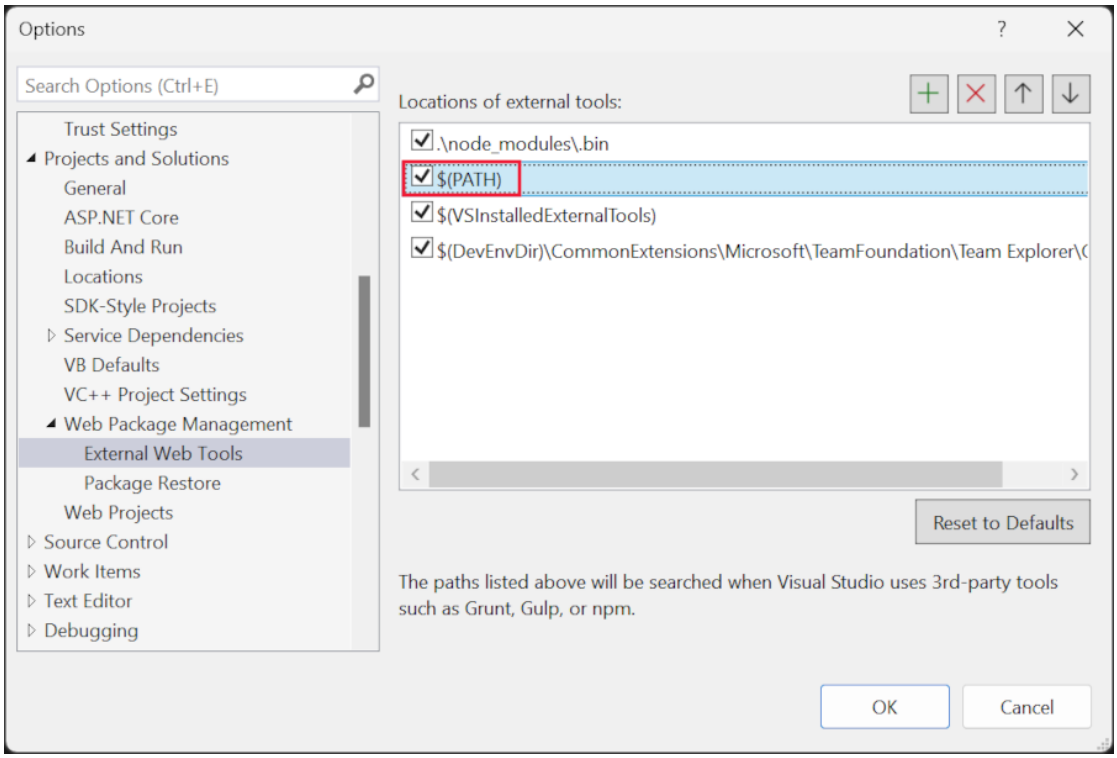
Create the ASP.NET Core web app

Visual Studio

By default, Visual Studio uses the version of npm found in its installation directory. To configure Visual Studio to look for npm in the `PATH` environment variable:

Launch Visual Studio. At the start window, select **Continue without code**.

1. Navigate to **Tools** > **Options** > **Projects and Solutions** > **Web Package Management** > **External Web Tools**.
2. Select the `$(PATH)` entry from the list. Select the up arrow to move the entry to the second position in the list, and select **OK**:



.

To create a new ASP.NET Core web app:

1. Use the **File** > **New** > **Project** menu option and choose the **ASP.NET Core Empty** template. Select **Next**.
2. Name the project `SignalRWebpack`, and select **Create**.
3. Select **.NET 8.0 (Long Term Support)** from the **Framework** drop-down. Select **Create**.

Add the Microsoft.TypeScript.MSBuild NuGet package to the project:

1. In **Solution Explorer**, right-click the project node and select **Manage NuGet Packages**. In the **Browse** tab, search for `Microsoft.TypeScript.MSBuild` and then select **Install** on the right to install the package.

Visual Studio adds the NuGet package under the **Dependencies** node in **Solution Explorer**, enabling TypeScript compilation in the project.

Configure the server

In this section, you configure the ASP.NET Core web app to send and receive SignalR messages.

1. In `Program.cs`, call AddSignalR:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();
```

2. Again, in `Program.cs`, call UseDefaultFiles and UseStaticFiles:

```
C#

var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();
```

   The preceding code allows the server to locate and serve the `index.html` file. The file is served whether the user enters its full URL or the root URL of the web app.

3. Create a new directory named `Hubs` in the project root, `SignalRWebpack/`, for the SignalR hub class.

4. Create a new file, `Hubs/ChatHub.cs`, with the following code:

```
C#

using Microsoft.AspNetCore.SignalR;

namespace SignalRWebpack.Hubs;

public class ChatHub : Hub
{
    public async Task NewMessage(long username, string message) =>
        await Clients.All.SendAsync("messageReceived", username,
message);
}
```

   The preceding code broadcasts received messages to all connected users once the server receives them. It's unnecessary to have a generic `on` method to receive all the messages. A method named after the message name is enough.

   In this example:

   - The TypeScript client sends a message identified as `newMessage`.
   - The C# `NewMessage` method expects the data sent by the client.
   - A call is made to SendAsync on Clients.All.
   - The received messages are sent to all clients connected to the hub.

5. Add the following `using` statement at the top of `Program.cs` to resolve the `ChatHub` reference:

```C#
using SignalRWebpack.Hubs;
```

6. In `Program.cs`, map the `/hub` route to the `ChatHub` hub. Replace the code that displays `Hello World!` with the following code:

```C#
app.MapHub<ChatHub>("/hub");
```

Configure the client

In this section, you create a Node.js project to convert TypeScript to JavaScript and bundle client-side resources, including HTML and CSS, using Webpack.

1. Run the following command in the project root to create a `package.json` file:

```Console
npm init -y
```

2. Add the highlighted property to the `package.json` file and save the file changes:

```JSON
{
  "name": "SignalRWebpack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Setting the `private` property to `true` prevents package installation warnings in the next step.

3. Install the required npm packages. Run the following command from the project root:

```
Console

npm i -D -E clean-webpack-plugin css-loader html-webpack-plugin mini-
css-extract-plugin ts-loader typescript webpack webpack-cli
```

The `-E` option disables npm's default behavior of writing semantic versioning range operators to `package.json`. For example, `"webpack": "5.76.1"` is used instead of `"webpack": "^5.76.1"`. This option prevents unintended upgrades to newer package versions.

For more information, see the npm-install documentation.

4. Replace the `scripts` property of `package.json` file with the following code:

```
JSON

"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

The following scripts are defined:

- `build`: Bundles the client-side resources in development mode and watches for file changes. The file watcher causes the bundle to regenerate each time a project file changes. The `mode` option disables production optimizations, such as tree shaking and minification. use `build` in development only.
- `release`: Bundles the client-side resources in production mode.
- `publish`: Runs the `release` script to bundle the client-side resources in production mode. It calls the .NET CLI's publish command to publish the app.

5. Create a file named `webpack.config.js` in the project root, with the following code:

```
JavaScript

const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
```

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/",
    },
    resolve: {
        extensions: [".js", ".ts"],
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader",
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"],
            },
        ],
    },
    plugins: [
        new CleanWebpackPlugin(),
        new HtmlWebpackPlugin({
            template: "./src/index.html",
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css",
        }),
    ],
};
```

The preceding file configures the Webpack compilation process:

- The `output` property overrides the default value of `dist`. The bundle is instead emitted in the `wwwroot` directory.
- The `resolve.extensions` array includes `.js` to import the SignalR client JavaScript.

6. Create a new directory named `src` in the project root, `SignalRWebpack/`, for the client code.

7. Copy the `src` directory and its contents from the sample project into the project root. The `src` directory contains the following files:

    - `index.html`, which defines the homepage's boilerplate markup:

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>ASP.NET Core SignalR with TypeScript and
Webpack</title>
  </head>
  <body>
    <div id="divMessages" class="messages"></div>
    <div class="input-zone">
      <label id="lblMessage" for="tbMessage">Message:</label>
      <input id="tbMessage" class="input-zone-input" type="text"
/>
      <button id="btnSend">Send</button>
    </div>
  </body>
</html>
```

- `css/main.css`, which provides CSS styles for the homepage:

css

```
*,
*::before,
*::after {
  box-sizing: border-box;
}

html,
body {
  margin: 0;
  padding: 0;
}

.input-zone {
  align-items: center;
  display: flex;
  flex-direction: row;
  margin: 10px;
}

.input-zone-input {
  flex: 1;
  margin-right: 10px;
}

.message-author {
  font-weight: bold;
}

.messages {
```

```
  border: 1px solid #000;
  margin: 10px;
  max-height: 300px;
  min-height: 300px;
  overflow-y: auto;
  padding: 5px;
}
```

- `tsconfig.json`, which configures the TypeScript compiler to produce ECMAScript 5-compatible JavaScript:

JSON

```
{
  "compilerOptions": {
    "target": "es5"
  }
}
```

- `index.ts`:

TypeScript

```
import * as signalR from "@microsoft/signalr";
import "./css/main.css";

const divMessages: HTMLDivElement =
document.querySelector("#divMessages");
const tbMessage: HTMLInputElement =
document.querySelector("#tbMessage");
const btnSend: HTMLButtonElement =
document.querySelector("#btnSend");
const username = new Date().getTime();

const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();

connection.on("messageReceived", (username: string, message:
string) => {
  const m = document.createElement("div");

  m.innerHTML = `<div class="message-author">${username}</div>
<div>${message}</div>`;

  divMessages.appendChild(m);
  divMessages.scrollTop = divMessages.scrollHeight;
});

connection.start().catch((err) => document.write(err));
```

```
tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
  if (e.key === "Enter") {
    send();
  }
});

btnSend.addEventListener("click", send);

function send() {
  connection.send("newMessage", username, tbMessage.value)
    .then(() => (tbMessage.value = ""));
}
```

The preceding code retrieves references to DOM elements and attaches two event handlers:

- `keyup`: Fires when the user types in the `tbMessage` textbox and calls the `send` function when the user presses the **Enter** key.
- `click`: Fires when the user selects the **Send** button and calls `send` function is called.

The `HubConnectionBuilder` class creates a new builder for configuring the server connection. The `withUrl` function configures the hub URL.

SignalR enables the exchange of messages between a client and a server. Each message has a specific name. For example, messages with the name `messageReceived` can run the logic responsible for displaying the new message in the messages zone. Listening to a specific message can be done via the `on` function. Any number of message names can be listened to. It's also possible to pass parameters to the message, such as the author's name and the content of the message received. Once the client receives a message, a new `div` element is created with the author's name and the message content in its `innerHTML` attribute. It's added to the main `div` element displaying the messages.

Sending a message through the WebSockets connection requires calling the `send` method. The method's first parameter is the message name. The message data inhabits the other parameters. In this example, a message identified as `newMessage` is sent to the server. The message consists of the username and the user input from a text box. If the send works, the text box value is cleared.

8. Run the following command at the project root:

```
Console
```

```
npm i @microsoft/signalr @types/node
```

The preceding command installs:

- The SignalR TypeScript client, which allows the client to send messages to the server.
- The TypeScript type definitions for Node.js, which enables compile-time checking of Node.js types.

Test the app

Confirm that the app works with the following steps:

Visual Studio

1. Run Webpack in `release` mode. Using the **Package Manager Console** window, run the following command in the project root.

   Console

   ```
   npm run release
   ```

   This command generates the client-side assets to be served when running the app. The assets are placed in the `wwwroot` folder.

   Webpack completed the following tasks:

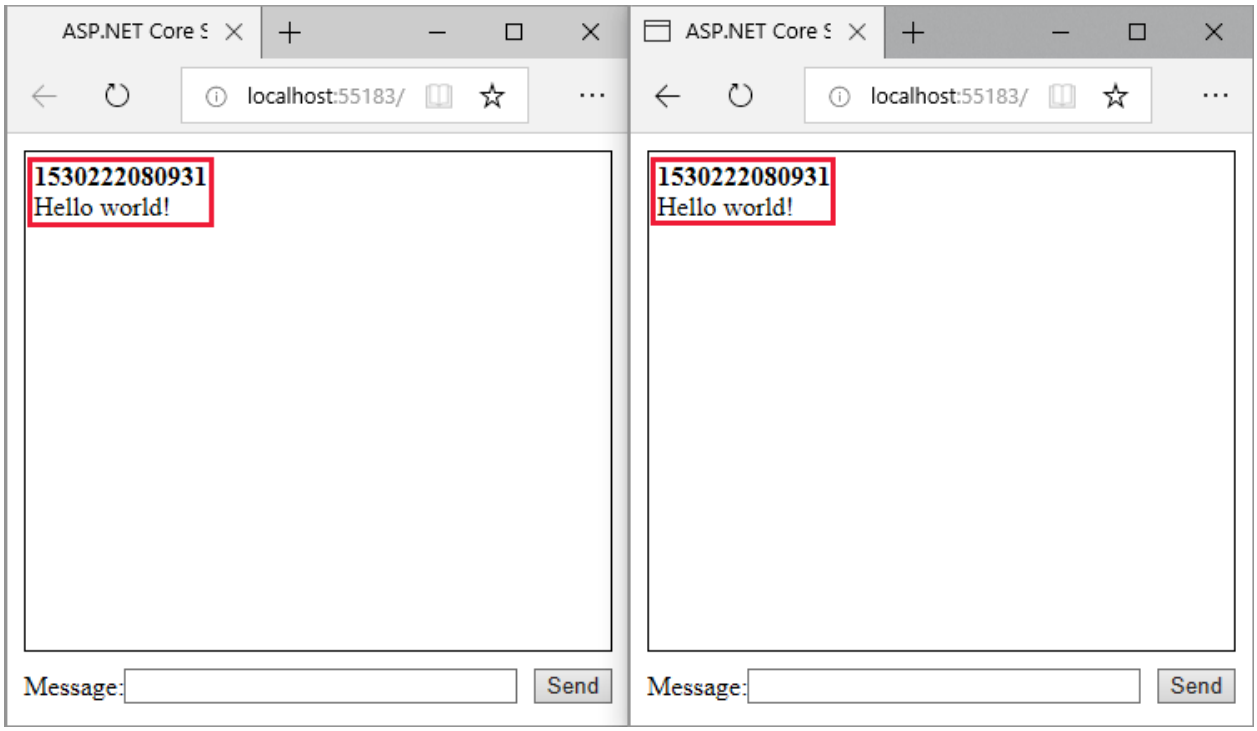
   - Purged the contents of the `wwwroot` directory.
   - Converted the TypeScript to JavaScript in a process known as *transpilation*.
   - Mangled the generated JavaScript to reduce file size in a process known as *minification*.
   - Copied the processed JavaScript, CSS, and HTML files from `src` to the `wwwroot` directory.
   - Injected the following elements into the `wwwroot/index.html` file:
     - A `<link>` tag, referencing the `wwwroot/main.<hash>.css` file. This tag is placed immediately before the closing `</head>` tag.
     - A `<script>` tag, referencing the minified `wwwroot/main.<hash>.js` file. This tag is placed immediately after the closing `</title>` tag.

2. Select **Debug** > **Start without debugging** to launch the app in a browser without attaching the debugger. The `wwwroot/index.html` file is served at `https://localhost:<port>`.

   If there are compile errors, try closing and reopening the solution.

3. Open another browser instance (any browser) and paste the URL in the address bar.

4. Choose either browser, type something in the **Message** text box, and select the **Send** button. The unique user name and message are displayed on both pages instantly.



Next steps

- Strongly typed hubs
- Authentication and authorization in ASP.NET Core SignalR
- MessagePack Hub Protocol in SignalR for ASP.NET Core

Additional resources

- ASP.NET Core SignalR JavaScript client
- Use hubs in ASP.NET Core SignalR

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Use ASP.NET Core SignalR with Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This tutorial provides a basic working experience for building a real-time app using SignalR with Blazor. This article is useful for developers who are already familiar with SignalR and are seeking to understand how to use SignalR in a Blazor app. For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets and the API documentation:

- Overview of ASP.NET Core SignalR
- ASP.NET Core Blazor
- .NET API browser

Learn how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

At the end of this tutorial, you'll have a working chat app.

Prerequisites

Visual Studio

Visual Studio (latest release) with the **ASP.NET and web development** workload

Sample app

Downloading the tutorial's sample chat app isn't required for this tutorial. The sample app is the final, working app produced by following the steps of this tutorial. When you

open the samples repository, open the version folder that you plan to target and find the sample named `BlazorSignalRApp`.

View or download sample code (how to download)

Create a Blazor Web App

Follow the guidance for your choice of tooling:

Visual Studio

> **Note**
>
> Visual Studio 2022 or later and .NET Core SDK 8.0.0 or later are required.

In Visual Studio:

- Select **Create a new project** from the **Start Window** or select **File** > **New** > **Project** from the menu bar.
- In the **Create a new project** dialog, select **Blazor Web App** from the list of project templates. Select the **Next** button.
- In the **Configure your new project** dialog, name the project `BlazorSignalRApp` in the **Project name** field, including matching the capitalization. Using this exact project name is important to ensure that the namespaces match for code that you copy from the tutorial into the app that you're building.
- Confirm that the **Location** for the app is suitable. Leave the **Place solution and project in the same directory** checkbox selected. Select the **Next** button.
- In the **Additional information** dialog, use the following settings:
  - **Framework**: Confirm that the latest framework is selected. If Visual Studio's **Framework** dropdown list doesn't include the latest available .NET framework, update Visual Studio and restart the tutorial.
  - **Authentication type**: **None**
  - **Configure for HTTPS**: Selected
  - **Interactive render mode**: **WebAssembly**
  - **Interactivity location**: **Per page/component**
  - **Include sample pages**: Selected
  - **Do not use top-level statements**: Not selected
  - Select **Create**.

The guidance in this article uses a WebAssembly component for the SignalR client because it doesn't make sense to use SignalR to connect to a hub from an Interactive Server component in the same app, as that can lead to server port exhaustion.

Add the SignalR client library

Visual Studio

In **Solution Explorer**, right-click the `BlazorSignalRApp.Client` project and select **Manage NuGet Packages**.

In the **Manage NuGet Packages** dialog, confirm that the **Package source** is set to `nuget.org`.

With **Browse** selected, type `Microsoft.AspNetCore.SignalR.Client` in the search box.

In the search results, select the latest release of the Microsoft.AspNetCore.SignalR.Client package. Select **Install**.

If the **Preview Changes** dialog appears, select **OK**.

If the **License Acceptance** dialog appears, select **I Accept** if you agree with the license terms.

Add a SignalR hub

In the server `BlazorSignalRApp` project, create a `Hubs` (plural) folder and add the following `ChatHub` class (`Hubs/ChatHub.cs`):

C#

```
using Microsoft.AspNetCore.SignalR;

namespace BlazorSignalRApp.Hubs;

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

Add services and an endpoint for the SignalR hub

Open the `Program` file of the server `BlazorSignalRApp` project.

Add the namespaces for Microsoft.AspNetCore.ResponseCompression and the `ChatHub` class to the top of the file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using BlazorSignalRApp.Hubs;
```

Add SignalR and Response Compression Middleware services:

C#

```
builder.Services.AddSignalR();

builder.Services.AddResponseCompression(opts =>
{
   opts.MimeTypes = ResponseCompressionDefaults.MimeTypes.Concat(
       ["application/octet-stream"]);
});
```

Use Response Compression Middleware at the top of the processing pipeline's configuration. Place the following line of code immediately after the line that builds the app (`var app = builder.Build();`):

C#

```
app.UseResponseCompression();
```

Add an endpoint for the hub immediately before the line that runs the app (`app.Run();`):

C#

```
app.MapHub<ChatHub>("/chathub");
```

Add Razor component code for chat

Add the following `Pages/Chat.razor` file to the `BlazorSignalRApp.Client` project:

razor

```razor
@page "/chat"
@rendermode InteractiveWebAssembly
@using Microsoft.AspNetCore.SignalR.Client
@inject NavigationManager Navigation
@implements IAsyncDisposable

<PageTitle>Chat</PageTitle>

<div class="form-group">
    <label>
        User:
        <input @bind="userInput" />
    </label>
</div>
<div class="form-group">
    <label>
        Message:
        <input @bind="messageInput" size="50" />
    </label>
</div>
<button @onclick="Send" disabled="@(!IsConnected)">Send</button>

<hr>

<ul id="messagesList">
    @foreach (var message in messages)
    {
        <li>@message</li>
    }
</ul>

@code {
    private HubConnection? hubConnection;
    private List<string> messages = [];
    private string? userInput;
    private string? messageInput;

    protected override async Task OnInitializedAsync()
    {
        hubConnection = new HubConnectionBuilder()
            .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
            .Build();

        hubConnection.On<string, string>("ReceiveMessage", (user, message)
=>
        {
            var encodedMsg = $"{user}: {message}";
            messages.Add(encodedMsg);
            InvokeAsync(StateHasChanged);
        });

        await hubConnection.StartAsync();
```

```
    }

    private async Task Send()
    {
        if (hubConnection is not null)
        {
            await hubConnection.SendAsync("SendMessage", userInput,
messageInput);
        }
    }

    public bool IsConnected =>
        hubConnection?.State == HubConnectionState.Connected;

    public async ValueTask DisposeAsync()
    {
        if (hubConnection is not null)
        {
            await hubConnection.DisposeAsync();
        }
    }
}
```

Add an entry to the `NavMenu` component to reach the chat page. In `Components/Layout/NavMenu.razor` immediately after the `<div>` block for the `Weather` component, add the following `<div>` block:

razor

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="chat">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
Chat
    </NavLink>
</div>
```

> **Note**
>
> Disable Response Compression Middleware in the `Development` environment when using Hot Reload. For more information, see ASP.NET Core Blazor SignalR guidance.

Run the app

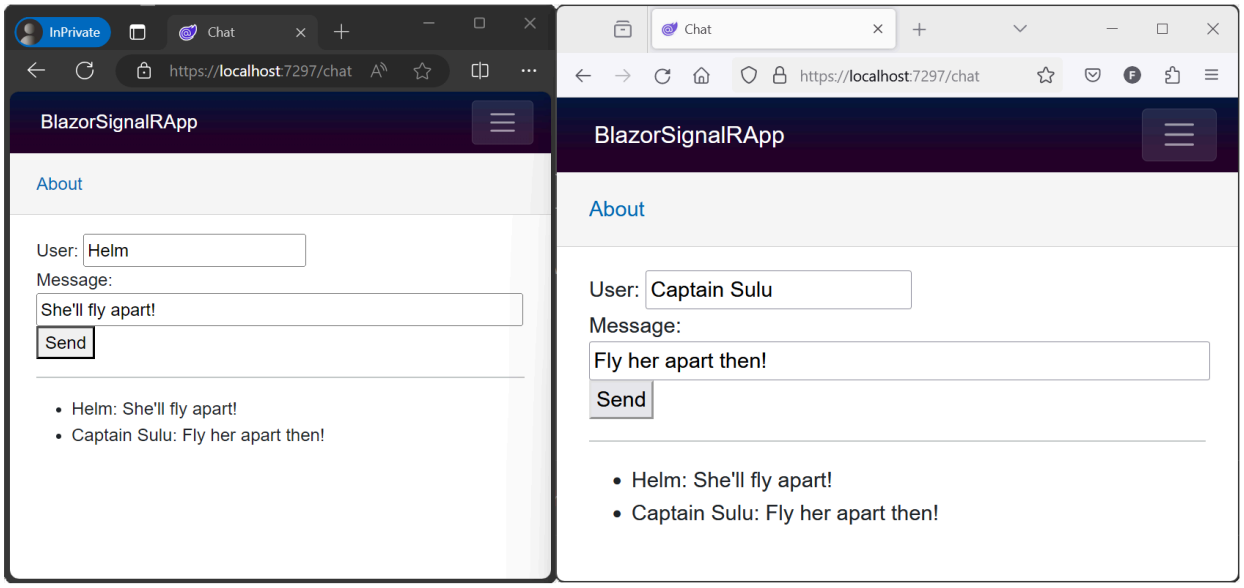
Follow the guidance for your tooling:

Visual Studio

With the server `BlazorSignalRApp` project selected in **Solution Explorer**, press `F5` to run the app with debugging or `Ctrl`+`F5` (Windows)/`⌘`+`F5` (macOS) to run the app without debugging.

Copy the URL from the address bar, open another browser instance or tab, and paste the URL in the address bar.

Choose either browser, enter a name and message, and select the button to send the message. The name and message are displayed on both pages instantly:



Quotes: *Star Trek VI: The Undiscovered Country* ©1991 Paramount

Next steps

In this tutorial, you learned how to:

- ✓ Create a Blazor app
- ✓ Add the SignalR client library
- ✓ Add a SignalR hub
- ✓ Add SignalR services and an endpoint for the SignalR hub
- ✓ Add a Razor component code for chat

For detailed guidance on the SignalR and Blazor frameworks, see the following reference documentation sets:

Overview of ASP.NET Core SignalR | ASP.NET Core Blazor

Additional resources

- Bearer token authentication with Identity Server, WebSockets, and Server-Sent Events
- Secure a SignalR hub in Blazor WebAssembly apps
- SignalR cross-origin negotiation for authentication
- SignalR configuration
- Debug ASP.NET Core Blazor apps
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

Use hubs in SignalR for ASP.NET Core

Article • 05/03/2024

By Rachel Appel and Kevin Griffin

The SignalR Hubs API enables connected clients to call methods on the server, facilitating real-time communication. The server defines methods that are called by the client, and the client defines methods that are called by the server. SignalR also enables indirect client-to-client communication, always mediated by the SignalR Hub, allowing messages to be sent between individual clients, groups, or to all connected clients. SignalR takes care of everything required to make real-time client-to-server and server-to-client communication possible.

Configure SignalR hubs

To register the services required by SignalR hubs, call AddSignalR in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddSignalR();
```

To configure SignalR endpoints, call MapHub, also in `Program.cs`:

C#

```
app.MapRazorPages();
app.MapHub<ChatHub>("/Chat");

app.Run();
```

> **Note**
>
> ASP.NET Core SignalR server-side assemblies are now installed with the .NET Core SDK. See SignalR assemblies in shared framework for more information.

Create and use hubs

Create a hub by declaring a class that inherits from Hub. Add `public` methods to the class to make them callable from clients:

C#

```
public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
        => await Clients.All.SendAsync("ReceiveMessage", user, message);
}
```

> **Note**
>
> Hubs are transient:
>
> - Don't store state in a property of the hub class. Each hub method call is executed on a new hub instance.
> - Don't instantiate a hub directly via dependency injection. To send messages to a client from elsewhere in your application use an IHubContext.
> - Use `await` when calling asynchronous methods that depend on the hub staying alive. For example, a method such as `Clients.All.SendAsync(...)` can fail if it's called without `await` and the hub method completes before `SendAsync` finishes.

The Context object

The Hub class includes a Context property that contains the following properties with information about the connection:

Expand table

Property	Description
ConnectionId	Gets the unique ID for the connection, assigned by SignalR. There's one connection ID for each connection.
UserIdentifier	Gets the user identifier. By default, SignalR uses the ClaimTypes.NameIdentifier from the ClaimsPrincipal associated with the connection as the user identifier.
User	Gets the ClaimsPrincipal associated with the current user.

Property	Description
Items	Gets a key/value collection that can be used to share data within the scope of this connection. Data can be stored in this collection and it will persist for the connection across different hub method invocations.
Features	Gets the collection of features available on the connection. For now, this collection isn't needed in most scenarios, so it isn't documented in detail yet.
ConnectionAborted	Gets a CancellationToken that notifies when the connection is aborted.

Hub.Context also contains the following methods:

Expand table

Method	Description
GetHttpContext	Returns the HttpContext for the connection, or `null` if the connection isn't associated with an HTTP request. For HTTP connections, use this method to get information such as HTTP headers and query strings.
Abort	Aborts the connection.

The Clients object

The Hub class includes a Clients property that contains the following properties for communication between server and client:

Expand table

Property	Description
All	Calls a method on all connected clients
Caller	Calls a method on the client that invoked the hub method
Others	Calls a method on all connected clients except the client that invoked the method

Hub.Clients also contains the following methods:

Expand table

Method	Description
AllExcept	Calls a method on all connected clients except for the specified connections

Method	Description
Client	Calls a method on a specific connected client
Clients	Calls a method on specific connected clients
Group	Calls a method on all connections in the specified group
GroupExcept	Calls a method on all connections in the specified group, except the specified connections
Groups	Calls a method on multiple groups of connections
OthersInGroup	Calls a method on a group of connections, excluding the client that invoked the hub method
User	Calls a method on all connections associated with a specific user
Users	Calls a method on all connections associated with the specified users

Each property or method in the preceding tables returns an object with a `SendAsync` method. The `SendAsync` method receives the name of the client method to call and any parameters.

The object returned by the `Client` and `Caller` methods also contain an `InvokeAsync` method, which can be used to wait for a result from the client.

Send messages to clients

To make calls to specific clients, use the properties of the `Clients` object. In the following example, there are three hub methods:

- `SendMessage` sends a message to all connected clients, using `Clients.All`.
- `SendMessageToCaller` sends a message back to the caller, using `Clients.Caller`.
- `SendMessageToGroup` sends a message to all clients in the `SignalR Users` group.

C#

```
public async Task SendMessage(string user, string message)
    => await Clients.All.SendAsync("ReceiveMessage", user, message);

public async Task SendMessageToCaller(string user, string message)
    => await Clients.Caller.SendAsync("ReceiveMessage", user, message);

public async Task SendMessageToGroup(string user, string message)
    => await Clients.Group("SignalR Users").SendAsync("ReceiveMessage",
user, message);
```

Strongly typed hubs

A drawback of using `SendAsync` is that it relies on a string to specify the client method to be called. This leaves code open to runtime errors if the method name is misspelled or missing from the client.

An alternative to using `SendAsync` is to strongly type the Hub class with Hub<T>. In the following example, the `ChatHub` client method has been extracted out into an interface called `IChatClient`:

C#

```
public interface IChatClient
{
    Task ReceiveMessage(string user, string message);
}
```

This interface can be used to refactor the preceding `ChatHub` example to be strongly typed:

C#

```
public class StronglyTypedChatHub : Hub<IChatClient>
{
    public async Task SendMessage(string user, string message)
        => await Clients.All.ReceiveMessage(user, message);

    public async Task SendMessageToCaller(string user, string message)
        => await Clients.Caller.ReceiveMessage(user, message);

    public async Task SendMessageToGroup(string user, string message)
        => await Clients.Group("SignalR Users").ReceiveMessage(user,
message);
}
```

Using `Hub<IChatClient>` enables compile-time checking of the client methods. This prevents issues caused by using strings, since `Hub<T>` can only provide access to the methods defined in the interface. Using a strongly typed `Hub<T>` disables the ability to use `SendAsync`.

> **Note**
>
> The `Async` suffix isn't stripped from method names. Unless a client method is defined with `.on('MyMethodAsync')`, don't use `MyMethodAsync` as the name.

Client results

In addition to making calls to clients, the server can request a result from a client. This requires the server to use `ISingleClientProxy.InvokeAsync` and the client to return a result from its `.On` handler.

There are two ways to use the API on the server, the first is to call `Client(...)` or `Caller` on the `Clients` property in a Hub method:

C#

```
public class ChatHub : Hub
{
    public async Task<string> WaitForMessage(string connectionId)
    {
        var message = await Clients.Client(connectionId).InvokeAsync<string>
(
            "GetMessage");
        return message;
    }
}
```

The second way is to call `Client(...)` on an instance of IHubContext<T>:

C#

```
async Task SomeMethod(IHubContext<MyHub> context)
{
    string result = await
context.Clients.Client(connectionID).InvokeAsync<string>(
        "GetMessage");
}
```

Strongly-typed hubs can also return values from interface methods:

C#

```
public interface IClient
{
    Task<string> GetMessage();
}

public class ChatHub : Hub<IClient>
{
    public async Task<string> WaitForMessage(string connectionId)
    {
        string message = await Clients.Client(connectionId).GetMessage();
        return message;
```

```
    }
}
```

Clients return results in their `.On(...)` handlers, as shown below:

.NET client

C#

```
hubConnection.On("GetMessage", async () =>
{
    Console.WriteLine("Enter message:");
    var message = await Console.In.ReadLineAsync();
    return message;
});
```

Typescript client

TypeScript

```
hubConnection.on("GetMessage", async () => {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("message");
        }, 100);
    });
    return promise;
});
```

Java client

Java

```
hubConnection.onWithResult("GetMessage", () -> {
    return Single.just("message");
});
```

Change the name of a hub method

By default, a server hub method name is the name of the .NET method. To change this default behavior for a specific method, use the HubMethodName attribute. The client should use this name instead of the .NET method name when invoking the method:

C#

```
[HubMethodName("SendMessageToUser")]
public async Task DirectMessage(string user, string message)
    => await Clients.User(user).SendAsync("ReceiveMessage", user, message);
```

Inject services into a hub

Hub constructors can accept services from DI as parameters, which can be stored in properties on the class for use in a hub method.

When injecting multiple services for different hub methods or as an alternative way of writing code, hub methods can also accept services from DI. By default, hub method parameters are inspected and resolved from DI if possible.

C#

```
services.AddSingleton<IDatabaseService, DatabaseServiceImpl>();

// ...

public class ChatHub : Hub
{
    public Task SendMessage(string user, string message, IDatabaseService
dbService)
    {
        var userName = dbService.GetUserName(user);
        return Clients.All.SendAsync("ReceiveMessage", userName, message);
    }
}
```

If implicit resolution of parameters from services isn't desired, disable it with DisableImplicitFromServicesParameters. To explicitly specify which parameters are resolved from DI in hub methods, use the DisableImplicitFromServicesParameters option and use the `[FromServices]` attribute or a custom attribute that implements `IFromServiceMetadata` on the hub method parameters that should be resolved from DI.

C#

```
services.AddSingleton<IDatabaseService, DatabaseServiceImpl>();
services.AddSignalR(options =>
{
    options.DisableImplicitFromServicesParameters = true;
});

// ...
```

```
public class ChatHub : Hub
{
    public Task SendMessage(string user, string message,
        [FromServices] IDatabaseService dbService)
    {
        var userName = dbService.GetUserName(user);
        return Clients.All.SendAsync("ReceiveMessage", userName, message);
    }
}
```

> **Note**
>
> This feature makes use of IServiceProviderIsService, which is optionally implemented by DI implementations. If the app's DI container doesn't support this feature, injecting services into hub methods isn't supported.

Keyed services support in Dependency Injection

Keyed services refers to a mechanism for registering and retrieving Dependency Injection (DI) services using keys. A service is associated with a key by calling AddKeyedSingleton (or `AddKeyedScoped` or `AddKeyedTransient`) to register it. Access a registered service by specifying the key with the [FromKeyedServices] attribute. The following code shows how to use keyed services:

C#

```
using Microsoft.AspNetCore.SignalR;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");

builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

app.MapRazorPages();
app.MapHub<MyHub>("/myHub");

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
```

```
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

public class MyHub : Hub
{
    public void SmallCacheMethod([FromKeyedServices("small")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }

    public void BigCacheMethod([FromKeyedServices("big")] ICache cache)
    {
        Console.WriteLine(cache.Get("signalr"));
    }
}
```

Handle events for a connection

The SignalR Hubs API provides the OnConnectedAsync and OnDisconnectedAsync virtual methods to manage and track connections. Override the `OnConnectedAsync` virtual method to perform actions when a client connects to the hub, such as adding it to a group:

C#

```
public override async Task OnConnectedAsync()
{
    await Groups.AddToGroupAsync(Context.ConnectionId, "SignalR Users");
    await base.OnConnectedAsync();
}
```

Override the `OnDisconnectedAsync` virtual method to perform actions when a client disconnects. If the client disconnects intentionally, such as by calling `connection.stop()`, the `exception` parameter is set to `null`. However, if the client disconnects due to an error, such as a network failure, the `exception` parameter contains an exception that describes the failure:

C#

```
public override async Task OnDisconnectedAsync(Exception? exception)
{
```

```
    await base.OnDisconnectedAsync(exception);
}
```

RemoveFromGroupAsync doesn't need to be called in OnDisconnectedAsync, it's automatically handled for you.

Handle errors

Exceptions thrown in hub methods are sent to the client that invoked the method. On the JavaScript client, the `invoke` method returns a JavaScript Promise. Clients can attach a `catch` handler to the returned promise or use `try`/`catch` with `async`/`await` to handle exceptions:

JavaScript

```
try {
  await connection.invoke("SendMessage", user, message);
} catch (err) {
  console.error(err);
}
```

Connections aren't closed when a hub throws an exception. By default, SignalR returns a generic error message to the client, as shown in the following example:

Output

```
Microsoft.AspNetCore.SignalR.HubException: An unexpected error occurred
invoking 'SendMessage' on the server.
```

Unexpected exceptions often contain sensitive information, such as the name of a database server in an exception triggered when the database connection fails. SignalR doesn't expose these detailed error messages by default as a security measure. For more information on why exception details are suppressed, see Security considerations in ASP.NET Core SignalR.

If an exceptional condition must be propagated to the client, use the HubException class. If a `HubException` is thrown in a hub method, SignalR **sends the entire exception message to the client**, unmodified:

C#

```
public Task ThrowException()
    => throw new HubException("This error will be sent to the client!");
```

> **Note**
>
> SignalR only sends the `Message` property of the exception to the client. The stack trace and other properties on the exception aren't available to the client.

Additional resources

- View or download sample code (how to download)
- Overview of ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- Publish an ASP.NET Core SignalR app to Azure App Service

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Send messages from outside a hub

Article • 09/15/2023

The SignalR hub is the core abstraction for sending messages to clients connected to the SignalR server. It's also possible to send messages from other places in your app using the `IHubContext` service. This article explains how to access a SignalR `IHubContext` to send notifications to clients from outside a hub.

> **Note**
>
> The `IHubContext` is for sending notifications to clients, it is not used to call methods on the `Hub`.

View or download sample code (how to download)

Get an instance of `IHubContext`

In ASP.NET Core SignalR, you can access an instance of `IHubContext` via dependency injection. You can inject an instance of `IHubContext` into a controller, middleware, or other DI service. Use the instance to send messages to clients.

Inject an instance of `IHubContext` in a controller

You can inject an instance of `IHubContext` into a controller by adding it to your constructor:

C#

```
public class HomeController : Controller
{
    private readonly IHubContext<NotificationHub> _hubContext;

    public HomeController(IHubContext<NotificationHub> hubContext)
    {
        _hubContext = hubContext;
    }
}
```

With access to an instance of `IHubContext`, call client methods as if you were in the hub itself:

C#

```
public async Task<IActionResult> Index()
{
    await _hubContext.Clients.All.SendAsync("Notify", $"Home page loaded at:
{DateTime.Now}");
    return View();
}
```

Get an instance of `IHubContext` in middleware

Access the `IHubContext` within the middleware pipeline like so:

C#

```
app.Use(async (context, next) =>
{
    var hubContext = context.RequestServices
                            .GetRequiredService<IHubContext<ChatHub>>();
    //...

    if (next != null)
    {
        await next.Invoke();
    }
});
```

> **Note**
>
> When client methods are called from outside of the `Hub` class, there's no caller associated with the invocation. Therefore, there's no access to the `ConnectionId`, `Caller`, and `Others` properties.
>
> Apps that need to map a user to the connection ID and persist that mapping can do one of the following:
>
> - Persist mapping of single or multiple connections as groups. See Groups in SignalR for more information.
> - Retain connection and user information through a singleton service. See Inject services into a hub for more information. The singleton service can use any storage method, such as:
>   - In-memory storage in a dictionary.
>   - Permanent external storage. For example, a database or Azure Table storage using the Azure.Data.Tables NuGet package.

- Pass the connection ID between clients.

Get an instance of `IHubContext` from IHost

Accessing an `IHubContext` from the web host is useful for integrating with areas outside of ASP.NET Core, for example, using third-party dependency injection frameworks:

C#

```
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();
            var hubContext =
host.Services.GetService(typeof(IHubContext<ChatHub>));
            host.Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder => {
                    webBuilder.UseStartup<Startup>();
                });
    }
```

Inject a strongly-typed HubContext

To inject a strongly-typed HubContext, ensure your Hub inherits from `Hub<T>`. Inject it using the `IHubContext<THub, T>` interface rather than `IHubContext<THub>`.

C#

```
public class ChatController : Controller
{
    public IHubContext<ChatHub, IChatClient> _strongChatHubContext { get; }

    public ChatController(IHubContext<ChatHub, IChatClient> chatHubContext)
    {
        _strongChatHubContext = chatHubContext;
    }

    public async Task SendMessage(string user, string message)
    {
        await _strongChatHubContext.Clients.All.ReceiveMessage(user,
message);
```

```
    }
}
```

See Strongly typed hubs for more information.

Use `IHubContext` in generic code

An injected `IHubContext<THub>` instance can be cast to `IHubContext` without a generic `Hub` type specified.

C#

```
class MyHub : Hub
{ }

class MyOtherHub : Hub
{ }

app.Use(async (context, next) =>
{
    var myHubContext = context.RequestServices
                              .GetRequiredService<IHubContext<MyHub>>();
    var myOtherHubContext = context.RequestServices
                              .GetRequiredService<IHubContext<MyOtherHub>>();
    await CommonHubContextMethod((IHubContext)myHubContext);
    await CommonHubContextMethod((IHubContext)myOtherHubContext);

    await next.Invoke();
}

async Task CommonHubContextMethod(IHubContext context)
{
    await context.Clients.All.SendAsync("clientMethod", new Args());
}
```

This is useful when:

- Writing libraries that don't have a reference to the specific `Hub` type the app is using.
- Writing code that is generic and can apply to multiple different `Hub` implementations

Additional resources

- SignalR assemblies in shared framework
- Get started with ASP.NET Core SignalR

- Use hubs in ASP.NET Core SignalR
- Publish an ASP.NET Core SignalR app to Azure App Service

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Manage users and groups in SignalR

Article • 04/05/2024

By Brennan Conroy

SignalR allows messages to be sent to all connections associated with a specific user and to named groups of connections.

View or download sample code (how to download)

Users in SignalR

A single user in SignalR can have multiple connections to an app. For example, a user could be connected on their desktop as well as their phone. Each device has a separate SignalR connection, but they're all associated with the same user. If a message is sent to the user, all of the connections associated with that user receive the message. The user identifier for a connection can be accessed by the `Context.UserIdentifier` property in the hub.

By default, SignalR uses the `ClaimTypes.NameIdentifier` from the `ClaimsPrincipal` associated with the connection as the user identifier. To customize this behavior, see Use claims to customize identity handling.

Send a message to a specific user by passing the user identifier to the `User` function in a hub method, as shown in the following example:

> **Note**
>
> The user identifier is case-sensitive.

C#

```
public Task SendPrivateMessage(string user, string message)
{
    return Clients.User(user).SendAsync("ReceiveMessage", message);
}
```

Groups in SignalR

A group is a collection of connections associated with a name. Messages can be sent to all connections in a group. Groups are the recommended way to send to a connection or multiple connections because the groups are managed by the application. A connection can be a member of multiple groups. Groups are ideal for something like a chat application, where each room can be represented as a group.

Add or remove connections from a group

Connections are added to or removed from groups via the `AddToGroupAsync` and `RemoveFromGroupAsync` methods:

C#

```
public async Task AddToGroup(string groupName)
{
    await Groups.AddToGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"
{Context.ConnectionId} has joined the group {groupName}.");
}

public async Task RemoveFromGroup(string groupName)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, groupName);

    await Clients.Group(groupName).SendAsync("Send", $"
{Context.ConnectionId} has left the group {groupName}.");
}
```

It's safe to add a user to a group multiple times, no exception is thrown in the case that the user already exists in the group.

Group membership isn't preserved when a connection reconnects. The connection needs to rejoin the group when it's re-established. It's not possible to count the members of a group, since this information isn't available if the application is scaled to multiple servers.

Groups are kept in memory, so they won't persist through a server restart. Consider the Azure SignalR service for scenarios requiring group membership to be persisted. For more information, see Azure SignalR

To protect access to resources while using groups, use authentication and authorization functionality in ASP.NET Core. If a user is added to a group only when the credentials are valid for that group, messages sent to that group will only go to authorized users. However, groups are not a security feature. Authentication claims have features that

groups don't, such as expiry and revocation. If a user's permission to access the group is revoked, the app must remove the user from the group explicitly.

> **Note**
>
> Group names are case-sensitive.

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- Publish an ASP.NET Core SignalR app to Azure App Service

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

SignalR API design considerations

Article • 02/24/2023

By Andrew Stanton-Nurse

This article provides guidance for building SignalR-based APIs.

Use custom object parameters to ensure backwards-compatibility

Adding parameters to a SignalR hub method (on either the client or the server) is a *breaking change*. This means older clients/servers will get errors when they try to invoke the method without the appropriate number of parameters. However, adding properties to a custom object parameter is **not** a breaking change. This can be used to design compatible APIs that are resilient to changes on the client or the server.

For example, consider a server-side API like the following:

C#

```
public int GetTotalLength(string param1)
{
    return param1.Length;
}
```

The JavaScript client calls this method using `invoke` as follows:

TypeScript

```
connection.invoke("GetTotalLength", "value1");
```

If you later add a second parameter to the server method, older clients won't provide this parameter value. For example:

C#

```
public int GetTotalLength(string param1, string param2)
{
    return param1.Length + param2.Length;
}
```

When the old client tries to invoke this method, it will get an error like this:

```
Microsoft.AspNetCore.SignalR.HubException: Failed to invoke 'GetTotalLength'
due to an error on the server.
```

On the server, you'll see a log message like this:

```
System.IO.InvalidDataException: Invocation provides 1 argument(s) but target
expects 2.
```

The old client only sent one parameter, but the newer server API required two parameters. Using custom objects as parameters gives you more flexibility. Let's redesign the original API to use a custom object:

C#

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
}

public int GetTotalLength(TotalLengthRequest req)
{
    return req.Param1.Length;
}
```

Now, the client uses an object to call the method:

TypeScript

```
connection.invoke("GetTotalLength", { param1: "value1" });
```

Instead of adding a parameter, add a property to the `TotalLengthRequest` object:

C#

```
public class TotalLengthRequest
{
    public string Param1 { get; set; }
    public string Param2 { get; set; }
}

public int GetTotalLength(TotalLengthRequest req)
{
```

```
    var length = req.Param1.Length;
    if (req.Param2 != null)
    {
        length += req.Param2.Length;
    }
    return length;
}
```

When the old client sends a single parameter, the extra `Param2` property will be left `null`. You can detect a message sent by an older client by checking the `Param2` for `null` and apply a default value. A new client can send both parameters.

TypeScript

```
connection.invoke("GetTotalLength", { param1: "value1", param2: "value2" });
```

The same technique works for methods defined on the client. You can send a custom object from the server side:

C#

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Message = message
    });
}
```

On the client side, you access the `Message` property rather than using a parameter:

TypeScript

```
connection.on("ReceiveMessage", (req) => {
    appendMessageToChatWindow(req.message);
});
```

If you later decide to add the sender of the message to the payload, add a property to the object:

C#

```
public async Task Broadcast(string message)
{
    await Clients.All.SendAsync("ReceiveMessage", new
    {
        Sender = Context.User.Identity.Name,
```

```
        Message = message
    });
}
```

The older clients won't be expecting the `Sender` value, so they'll ignore it. A new client can accept it by updating to read the new property:

TypeScript

```
connection.on("ReceiveMessage", (req) => {
    let message = req.message;
    if (req.sender) {
        message = req.sender + ": " + message;
    }
    appendMessageToChatWindow(message);
});
```

In this case, the new client is also tolerant of an old server that doesn't provide the `Sender` value. Since the old server won't provide the `Sender` value, the client checks to see if it exists before accessing it.

Additional resources

- SignalR assemblies in shared framework

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use hub filters in ASP.NET Core SignalR

Article • 07/03/2022

Hub filters:

- Are available in ASP.NET Core 5.0 or later.
- Allow logic to run before and after hub methods are invoked by clients.

This article provides guidance for writing and using hub filters.

Configure hub filters

Hub filters can be applied globally or per hub type. The order in which filters are added is the order in which the filters run. Global hub filters run before local hub filters.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(options =>
    {
        // Global filters will run first
        options.AddFilter<CustomFilter>();
    }).AddHubOptions<ChatHub>(options =>
    {
        // Local filters will run second
        options.AddFilter<CustomFilter2>();
    });
}
```

A hub filter can be added in one of the following ways:

- Add a filter by concrete type:

  C#

  ```
  hubOptions.AddFilter<TFilter>();
  ```

  This will be resolved from dependency injection (DI) or type activated.

- Add a filter by runtime type:

  C#

```
hubOptions.AddFilter(typeof(TFilter));
```

This will be resolved from DI or type activated.

- Add a filter by instance:

```
C#
hubOptions.AddFilter(new MyFilter());
```

This instance will be used like a singleton. All hub method invocations will use the same instance.

Hub filters are created and disposed per hub invocation. If you want to store global state in the filter, or no state, add the hub filter type to DI as a singleton for better performance. Alternatively, add the filter as an instance if you can.

Create hub filters

Create a filter by declaring a class that inherits from `IHubFilter`, and add the `InvokeMethodAsync` method. There is also `OnConnectedAsync` and `OnDisconnectedAsync` that can optionally be implemented to wrap the `OnConnectedAsync` and `OnDisconnectedAsync` hub methods respectively.

```
C#
public class CustomFilter : IHubFilter
{
    public async ValueTask<object> InvokeMethodAsync(
        HubInvocationContext invocationContext, Func<HubInvocationContext,
ValueTask<object>> next)
    {
        Console.WriteLine($"Calling hub method
'{invocationContext.HubMethodName}'");
        try
        {
            return await next(invocationContext);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception calling
'{invocationContext.HubMethodName}': {ex}");
            throw;
        }
    }
```

```
    // Optional method
    public Task OnConnectedAsync(HubLifetimeContext context,
Func<HubLifetimeContext, Task> next)
    {
        return next(context);
    }

    // Optional method
    public Task OnDisconnectedAsync(
        HubLifetimeContext context, Exception exception,
Func<HubLifetimeContext, Exception, Task> next)
    {
        return next(context, exception);
    }
}
```

Filters are very similar to middleware. The `next` method invokes the next filter. The final filter will invoke the hub method. Filters can also store the result from awaiting `next` and run logic after the hub method has been called before returning the result from `next`.

To skip a hub method invocation in a filter, throw an exception of type `HubException` instead of calling `next`. The client will receive an error if it was expecting a result.

Use hub filters

When writing the filter logic, try to make it generic by using attributes on hub methods instead of checking for hub method names.

Consider a filter that will check a hub method argument for banned phrases and replace any phrases it finds with `***`. For this example, assume a `LanguageFilterAttribute` class is defined. The class has a property named `FilterArgument` that can be set when using the attribute.

1. Place the attribute on the hub method that has a string argument to be cleaned:

   C#

   ```
   public class ChatHub
   {
       [LanguageFilter(filterArgument = 0)]
       public async Task SendMessage(string message, string username)
       {
           await Clients.All.SendAsync("SendMessage", $"{username} says:
   {message}");
       }
   }
   ```

2. Define a hub filter to check for the attribute and replace banned phrases in a hub method argument with `***`:

```
C#
```

```
public class LanguageFilter : IHubFilter
{
    // populated from a file or inline
    private List<string> bannedPhrases = new List<string> { "async
void", ".Result" };

    public async ValueTask<object>
InvokeMethodAsync(HubInvocationContext invocationContext,
        Func<HubInvocationContext, ValueTask<object>> next)
    {
        var languageFilter =
(LanguageFilterAttribute)Attribute.GetCustomAttribute(
            invocationContext.HubMethod,
typeof(LanguageFilterAttribute));
        if (languageFilter != null &&
            invocationContext.HubMethodArguments.Count >
languageFilter.FilterArgument &&

invocationContext.HubMethodArguments[languageFilter.FilterArgument] is
string str)
        {
            foreach (var bannedPhrase in bannedPhrases)
            {
                str = str.Replace(bannedPhrase, "***");
            }

            var arguments =
invocationContext.HubMethodArguments.ToArray();
            arguments[languageFilter.FilterArgument] = str;
            invocationContext = new
HubInvocationContext(invocationContext.Context,
                invocationContext.ServiceProvider,
                invocationContext.Hub,
                invocationContext.HubMethod,
                arguments);
        }

        return await next(invocationContext);
    }
}
```

3. Register the hub filter in the `Startup.ConfigureServices` method. To avoid reinitializing the banned phrases list for every invocation, the hub filter is registered as a singleton:

```
C#
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSignalR(hubOptions =>
    {
        hubOptions.AddFilter<LanguageFilter>();
    });

    services.AddSingleton<LanguageFilter>();
}
```

The HubInvocationContext object

The `HubInvocationContext` contains information for the current hub method invocation.

Expand table

Property	Description	Type
`Context`	The `HubCallerContext` contains information about the connection.	`HubCallerContext`
`Hub`	The instance of the Hub being used for this hub method invocation.	`Hub`
`HubMethodName`	The name of the hub method being invoked.	`string`
`HubMethodArguments`	The list of arguments being passed to the hub method.	`IReadOnlyList<string>`
`ServiceProvider`	The scoped service provider for this hub method invocation.	`IServiceProvider`
`HubMethod`	The hub method information.	`MethodInfo`

The HubLifetimeContext object

The `HubLifetimeContext` contains information for the `OnConnectedAsync` and `OnDisconnectedAsync` hub methods.

Expand table

Property	Description	Type
`Context`	The `HubCallerContext` contains information about the connection.	`HubCallerContext`

Property	Description	Type
`Hub`	The instance of the Hub being used for this hub method invocation.	`Hub`
`ServiceProvider`	The scoped service provider for this hub method invocation.	`IServiceProvider`

Authorization and filters

Authorize attributes on hub methods run before hub filters.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core SignalR clients

Article • 02/14/2023

Versioning, support, and compatibility

The SignalR clients ship alongside the server components and are versioned to match. Any supported client can safely connect to any supported server, and any compatibility issues would be considered bugs to be fixed. SignalR clients are supported in the same support lifecycle as the rest of .NET Core. See the .NET Core Support Policy for details.

Many features require a compatible client **and** server. See below for a table showing the minimum versions for various features.

The 1.x versions of SignalR map to the 2.1 and 2.2 .NET Core releases and have the same lifetime. For version 3.x and above, the SignalR version exactly matches the rest of .NET and has the same support lifecycle.

Expand table

SignalR version	.NET Core version	Support level	End of support
1.0.x	2.1.x	Long Term Support	August 21, 2021
1.1.x	2.2.x	End Of Life	December 23, 2019
3.x or higher	*same as SignalR version*	See the the .NET Core Support Policy	

NOTE: In ASP.NET Core 3.0, the JavaScript client *moved* to the `@microsoft/signalr` npm package.

Feature distribution

The table below shows the features and support for the clients that offer real-time support. For each feature, the *minimum* version supporting this feature is listed. If no version is listed, the feature isn't supported.

Expand table

Feature	Server	.NET client	JavaScript client	Java client
Azure SignalR Service Support	2.1.0	1.0.0	1.0.0	1.0.0
Server-to-client Streaming	2.1.0	1.0.0	1.0.0	1.0.0
Client-to-server Streaming	3.0.0	3.0.0	3.0.0	3.0.0
Automatic Reconnection (.NET, JavaScript)	3.0.0	3.0.0	3.0.0	✗
WebSockets Transport	2.1.0	1.0.0	1.0.0	1.0.0
Server-Sent Events Transport	2.1.0	1.0.0	1.0.0	✗
Long Polling Transport	2.1.0	1.0.0	1.0.0	3.0.0
JSON Hub Protocol	2.1.0	1.0.0	1.0.0	1.0.0
MessagePack Hub Protocol	2.1.0	1.0.0	1.0.0	5.0.0
Client Results	7.0.0	7.0.0	7.0.0	7.0.0

Support for enabling additional client features is tracked in our issue tracker.

Browsers that don't support ECMAScript 6 (ES6)

SignalR targets ES6. For browsers that don't support ES6, transpile the library to ES5. For more information, see Getting Started with ES6 – Transpiling ES6 to ES5 with Traceur and Babel.

Additional resources

- Get started with SignalR for ASP.NET Core
- Supported platforms
- Hubs
- JavaScript client

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core SignalR .NET Client

Article • 06/18/2024

The ASP.NET Core SignalR .NET client library lets you communicate with SignalR hubs from .NET apps.

View or download sample code (how to download)

The code sample in this article is a WPF app that uses the ASP.NET Core SignalR .NET client.

Install the SignalR .NET client package

The Microsoft.AspNetCore.SignalR.Client package is required for .NET clients to connect to SignalR hubs.

Visual Studio

To install the client library, run the following command in the **Package Manager Console** window:

PowerShell

```
Install-Package Microsoft.AspNetCore.SignalR.Client
```

Connect to a hub

To establish a connection, create a `HubConnectionBuilder` and call `Build`. The hub URL, protocol, transport type, log level, headers, and other options can be configured while building a connection. Configure any required options by inserting any of the `HubConnectionBuilder` methods into `Build`. Start the connection with `StartAsync`.

C#

```
using System;
using System.Threading.Tasks;
using System.Windows;
using Microsoft.AspNetCore.SignalR.Client;

namespace SignalRChatClient
{
    public partial class MainWindow : Window
```

```
    {
        HubConnection connection;
        public MainWindow()
        {
            InitializeComponent();

            connection = new HubConnectionBuilder()
                .WithUrl("http://localhost:53353/ChatHub")
                .Build();

            connection.Closed += async (error) =>
            {
                await Task.Delay(new Random().Next(0,5) * 1000);
                await connection.StartAsync();
            };
        }

        private async void connectButton_Click(object sender,
RoutedEventArgs e)
        {
            connection.On<string, string>("ReceiveMessage", (user, message)
=>
            {
                this.Dispatcher.Invoke(() =>
                {
                   var newMessage = $"{user}: {message}";
                   messagesList.Items.Add(newMessage);
                });
            });

            try
            {
                await connection.StartAsync();
                messagesList.Items.Add("Connection started");
                connectButton.IsEnabled = false;
                sendButton.IsEnabled = true;
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
        }

        private async void sendButton_Click(object sender, RoutedEventArgs
e)
        {
            try
            {
                await connection.InvokeAsync("SendMessage",
                    userTextBox.Text, messageTextBox.Text);
            }
            catch (Exception ex)
            {
                messagesList.Items.Add(ex.Message);
            }
```

```
        }
    }
}
```

Handle lost connection

Automatically reconnect

The HubConnection can be configured to automatically reconnect using the `WithAutomaticReconnect` method on the HubConnectionBuilder. It won't automatically reconnect by default.

C#

```
HubConnection connection= new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect()
    .Build();
```

Without any parameters, `WithAutomaticReconnect()` configures the client to wait 0, 2, 10, and 30 seconds respectively before trying each reconnect attempt, stopping after four failed attempts.

Before starting any reconnect attempts, the `HubConnection` will transition to the `HubConnectionState.Reconnecting` state and fire the `Reconnecting` event. This provides an opportunity to warn users that the connection has been lost and to disable UI elements. Non-interactive apps can start queuing or dropping messages.

C#

```
connection.Reconnecting += error =>
{
    Debug.Assert(connection.State == HubConnectionState.Reconnecting);

    // Notify users the connection was lost and the client is reconnecting.
    // Start queuing or dropping messages.

    return Task.CompletedTask;
};
```

If the client successfully reconnects within its first four attempts, the `HubConnection` will transition back to the `Connected` state and fire the `Reconnected` event. This provides an

opportunity to inform users the connection has been reestablished and dequeue any queued messages.

Since the connection looks entirely new to the server, a new `ConnectionId` will be provided to the `Reconnected` event handlers.

> ⚠ **Warning**
>
> The `Reconnected` event handler's `connectionId` parameter will be null if the `HubConnection` was configured to skip negotiation.

C#

```
connection.Reconnected += connectionId =>
{
    Debug.Assert(connection.State == HubConnectionState.Connected);

    // Notify users the connection was reestablished.
    // Start dequeuing messages queued while reconnecting if any.

    return Task.CompletedTask;
};
```

`WithAutomaticReconnect()` won't configure the `HubConnection` to retry initial start failures, so start failures need to be handled manually:

C#

```
public static async Task<bool> ConnectWithRetryAsync(HubConnection
connection, CancellationToken token)
{
    // Keep trying to until we can start or the token is canceled.
    while (true)
    {
        try
        {
            await connection.StartAsync(token);
            Debug.Assert(connection.State == HubConnectionState.Connected);
            return true;
        }
        catch when (token.IsCancellationRequested)
        {
            return false;
        }
        catch
        {
            // Failed to connect, trying again in 5000 ms.
            Debug.Assert(connection.State ==
```

```
HubConnectionState.Disconnected);
            await Task.Delay(5000);
        }
    }
}
```

If the client doesn't successfully reconnect within its first four attempts, the `HubConnection` will transition to the `Disconnected` state and fire the Closed event. This provides an opportunity to attempt to restart the connection manually or inform users the connection has been permanently lost.

C#

```
connection.Closed += error =>
{
    Debug.Assert(connection.State == HubConnectionState.Disconnected);

    // Notify users the connection has been closed or manually try to
restart the connection.

    return Task.CompletedTask;
};
```

In order to configure a custom number of reconnect attempts before disconnecting or change the reconnect timing, `WithAutomaticReconnect` accepts an array of numbers representing the delay in milliseconds to wait before starting each reconnect attempt.

C#

```
HubConnection connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect(new[] { TimeSpan.Zero, TimeSpan.Zero,
TimeSpan.FromSeconds(10) })
    .Build();

    // .WithAutomaticReconnect(new[] { TimeSpan.Zero,
TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(10), TimeSpan.FromSeconds(30)
}) yields the default behavior.
```

The preceding example configures the `HubConnection` to start attempting reconnects immediately after the connection is lost. This is also true for the default configuration.

If the first reconnect attempt fails, the second reconnect attempt will also start immediately instead of waiting 2 seconds like it would in the default configuration.

If the second reconnect attempt fails, the third reconnect attempt will start in 10 seconds which is again like the default configuration.

The custom behavior then diverges again from the default behavior by stopping after the third reconnect attempt failure. In the default configuration there would be one more reconnect attempt in another 30 seconds.

If you want even more control over the timing and number of automatic reconnect attempts, `WithAutomaticReconnect` accepts an object implementing the `IRetryPolicy` interface, which has a single method named `NextRetryDelay`.

`NextRetryDelay` takes a single argument with the type `RetryContext`. The `RetryContext` has three properties: `PreviousRetryCount`, `ElapsedTime` and `RetryReason`, which are a `long`, a `TimeSpan` and an `Exception` respectively. Before the first reconnect attempt, both `PreviousRetryCount` and `ElapsedTime` will be zero, and the `RetryReason` will be the Exception that caused the connection to be lost. After each failed retry attempt, `PreviousRetryCount` will be incremented by one, `ElapsedTime` will be updated to reflect the amount of time spent reconnecting so far, and the `RetryReason` will be the Exception that caused the last reconnect attempt to fail.

`NextRetryDelay` must return either a TimeSpan representing the time to wait before the next reconnect attempt or `null` if the `HubConnection` should stop reconnecting.

C#

```
public class RandomRetryPolicy : IRetryPolicy
{
    private readonly Random _random = new Random();

    public TimeSpan? NextRetryDelay(RetryContext retryContext)
    {
        // If we've been reconnecting for less than 60 seconds so far,
        // wait between 0 and 10 seconds before the next reconnect attempt.
        if (retryContext.ElapsedTime < TimeSpan.FromSeconds(60))
        {
            return TimeSpan.FromSeconds(_random.NextDouble() * 10);
        }
        else
        {
            // If we've been reconnecting for more than 60 seconds so far,
stop reconnecting.
            return null;
        }
    }
}
```

C#

```
HubConnection connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
```

```
    .WithAutomaticReconnect(new RandomRetryPolicy())
    .Build();
```

Alternatively, you can write code that will reconnect your client manually as demonstrated in Manually reconnect.

Manually reconnect

Use the Closed event to respond to a lost connection. For example, you might want to automate reconnection.

The `Closed` event requires a delegate that returns a `Task`, which allows async code to run without using `async void`. To satisfy the delegate signature in a `Closed` event handler that runs synchronously, return `Task.CompletedTask`:

C#

```
connection.Closed += (error) => {
    // Do your close logic.
    return Task.CompletedTask;
};
```

The main reason for the async support is so you can restart the connection. Starting a connection is an async action.

In a `Closed` handler that restarts the connection, consider waiting for some random delay to prevent overloading the server, as shown in the following example:

C#

```
connection.Closed += async (error) =>
{
    await Task.Delay(new Random().Next(0,5) * 1000);
    await connection.StartAsync();
};
```

Call hub methods from client

`InvokeAsync` calls methods on the hub. Pass the hub method name and any arguments defined in the hub method to `InvokeAsync`. SignalR is asynchronous, so use `async` and `await` when making the calls.

C#

```
await connection.InvokeAsync("SendMessage",
    userTextBox.Text, messageTextBox.Text);
```

The `InvokeAsync` method returns a `Task` which completes when the server method returns. The return value, if any, is provided as the result of the `Task`. Any exceptions thrown by the method on the server produce a faulted `Task`. Use `await` syntax to wait for the server method to complete and `try...catch` syntax to handle errors.

The `SendAsync` method returns a `Task` which completes when the message has been sent to the server. No return value is provided since this `Task` doesn't wait until the server method completes. Any exceptions thrown on the client while sending the message produce a faulted `Task`. Use `await` and `try...catch` syntax to handle send errors.

> **Note**
>
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions.

Call client methods from hub

Define methods the hub calls using `connection.On` after building, but before starting the connection.

C#

```
connection.On<string, string>("ReceiveMessage", (user, message) =>
{
    this.Dispatcher.Invoke(() =>
    {
        var newMessage = $"{user}: {message}";
        messagesList.Items.Add(newMessage);
    });
});
```

The preceding code in `connection.On` runs when server-side code calls it using the `SendAsync` method.

C#

```
public async Task SendMessage(string user, string message)
{
```

```
    await Clients.All.SendAsync("ReceiveMessage", user, message);
}
```

> **Note**
>
> While the hub side of the connection supports strongly-typed messaging, the client must register using the generic method HubConnection.On with the method name. For an example, see Host ASP.NET Core SignalR in background services.

Error handling and logging

Handle errors with a try-catch statement. Inspect the `Exception` object to determine the proper action to take after an error occurs.

C#

```
try
{
    await connection.InvokeAsync("SendMessage",
        userTextBox.Text, messageTextBox.Text);
}
catch (Exception ex)
{
    messagesList.Items.Add(ex.Message);
}
```

Additional resources

- Hubs
- JavaScript client
- Publish to Azure
- Azure SignalR Service serverless documentation

Microsoft.AspNetCore.SignalR.Client Namespace

Reference

Contains types that are used for communicating with a SignalR server.

Classes

Expand table

HubConnection	A connection used to invoke hub methods on a SignalR Server.
HubConnectionBuilder	A builder for configuring HubConnection instances.
HubConnectionBuilderExtensions	Extension methods for IHubConnectionBuilder.
HubConnectionBuilderHttpExtensions	Extension methods for IHubConnectionBuilder.
HubConnectionExtensions	Extension methods for HubConnectionExtensions.
HubConnectionOptions	Configures options for the HubConnection.
RetryContext	The context passed to NextRetryDelay(RetryContext) to help the policy determine how long to wait before the next retry and whether there should be another retry at all.

Interfaces

Expand table

IHubConnectionBuilder	A builder abstraction for configuring HubConnection instances.
IRetryPolicy	An abstraction that controls when the client attempts to reconnect and how many times it does so.

Enums

Expand table

HubConnectionState	Describes the current state of the HubConnection to the server.

Remarks

For more information about the SignalR client, see ASP.NET Core SignalR .NET Client.

Feedback

Was this page helpful? Yes No

ASP.NET Core SignalR Java client

Article • 01/14/2023

By Mikael Mengistu

The Java client enables connecting to an ASP.NET Core SignalR server from Java code, including Android apps. Like the JavaScript client and the .NET client, the Java client enables you to receive and send messages to a hub in real time. The Java client is available in ASP.NET Core 2.2 and later.

The sample Java console app referenced in this article uses the SignalR Java client.

View or download sample code (how to download)

Install the SignalR Java client package

The *signalr-7.0.0* JAR file allows clients to connect to SignalR hubs. To find the latest JAR file version number, see the Maven search results.

If using Gradle, add the following line to the `dependencies` section of your *build.gradle* file:

Gradle

```
implementation 'com.microsoft.signalr:signalr:7.0.0'
```

If using Maven, add the following lines inside the `<dependencies>` element of your `pom.xml` file:

XML

```
<dependency>
    <groupId>com.microsoft.signalr</groupId>
    <artifactId>signalr</artifactId>
    <version>1.0.0</version>
</dependency>
```

Connect to a hub

To establish a `HubConnection`, the `HubConnectionBuilder` should be used. The hub URL and log level can be configured while building a connection. Configure any required

options by calling any of the `HubConnectionBuilder` methods before `build`. Start the connection with `start`.

```
Java

HubConnection hubConnection = HubConnectionBuilder.create(input)
        .build();
```

Call hub methods from client

A call to `send` invokes a hub method. Pass the hub method name and any arguments defined in the hub method to `send`.

```
Java

hubConnection.send("Send", input);
```

> **Note**
>
> Calling hub methods from a client is only supported when using the Azure SignalR Service in *Default* mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

Call client methods from hub

Use `hubConnection.on` to define methods on the client that the hub can call. Define the methods after building but before starting the connection.

```
Java

hubConnection.on("Send", (message) -> {
    System.out.println("New Message: " + message);
}, String.class);
```

Add logging

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to choose their own specific logging implementation by

bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

Gradle

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

This can safely be ignored.

Android development notes

With regards to Android SDK compatibility for the SignalR client features, consider the following items when specifying your target Android SDK version:

- The SignalR Java Client will run on Android API Level 16 and later.
- Connecting through the Azure SignalR Service will require Android API Level 20 and later because the Azure SignalR Service requires TLS 1.2 and doesn't support SHA-1-based cipher suites. Android added support for SHA-256 (and above) cipher suites in API Level 20.

Configure bearer token authentication

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an "access token factory" to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

Java

```
HubConnection hubConnection = HubConnectionBuilder.create("YOUR HUB URL
HERE")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
```

```
        return Single.just("An Access Token");
    })).build();
```

Passing Class information in Java

When calling the `on`, `invoke`, or `stream` methods of `HubConnection` in the Java client, users should pass a `Type` object rather than a `Class<?>` object to describe any generic `Object` passed to the method. A `Type` can be acquired using the provided `TypeReference` class. For example, using a custom generic class named `Foo<T>`, the following code gets the `Type`:

Java

```
Type fooType = new TypeReference<Foo<String>>() { }).getType();
```

For non-generics, such as primitives or other non-parameterized types like `String`, you can simply use the built-in `.class`.

When calling one of these methods with one or more object types, use the generics syntax when invoking the method. For example, when registering an `on` handler for a method named `func`, which takes as arguments a String and a `Foo<String>` object, use the following code to set an action to print the arguments:

Java

```
hubConnection.<String, Foo<String>>on("func", (param1, param2) ->{
    System.out.println(param1);
    System.out.println(param2);
}, String.class, fooType);
```

This convention is necessary because we can not retrieve complete information about complex types with the `Object.getClass` method due to type erasure in Java. For example, calling `getClass` on an `ArrayList<String>` would not return `Class<ArrayList<String>>`, but rather `Class<ArrayList>`, which does not give the deserializer enough information to correctly deserialize an incoming message. The same is true for custom objects.

Known limitations

- Transport fallback and the Server Sent Events transport aren't supported.

Additional resources

- Java API reference
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- Publish an ASP.NET Core SignalR app to Azure App Service
- Azure SignalR Service serverless documentation

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

com.microsoft.signalr

Reference

Package: com.microsoft.signalr

Maven Artifact: com.microsoft.signalr:signalr:5.0.10

This package contains the classes for SignalR Java client.

Classes

Expand table

CancelInvocationMessage	
CloseMessage	
CompletionMessage	
HttpHubConnectionBuilder	A builder for configuring HubConnection instances.
HubConnection	A connection used to invoke hub methods on a SignalR Server.
HubConnectionBuilder	A builder for configuring HubConnection instances.
HubException	An exception thrown when the server fails to invoke a Hub method.
HubMessage	A base class for hub messages.
InvocationBindingFailureMessage	
InvocationMessage	
PingMessage	
StreamBindingFailureMessage	
StreamInvocationMessage	
StreamItem	
Subscription	Represents the registration of a handler for a client method.
TypeReference<T>	A utility for getting a Java Type from a literal generic Class.
UserAgentHelper	

Interfaces

Expand table

Action	A callback that takes no parameters.
Action1<T1>	A callback that takes one parameter.
Action2<T1,T2>	A callback that takes two parameters.
Action3<T1,T2,T3>	A callback that takes three parameters.
Action4<T1,T2,T3,T4>	A callback that takes four parameters.
Action5<T1,T2,T3,T4,T5>	A callback that takes five parameters.
Action6<T1,T2,T3,T4,T5,T6>	A callback that takes six parameters.
Action7<T1,T2,T3,T4,T5,T6,T7>	A callback that takes seven parameters.
Action8<T1,T2,T3,T4,T5,T6,T7,T8>	A callback that takes eight parameters.
HubProtocol	A protocol abstraction for communicating with SignalR hubs.
InvocationBinder	An abstraction for passing around information about method signatures.
OnClosedCallback	A callback to create and register on a HubConnections On Closed method.

Enums

Expand table

HubConnectionState	Indicates the state of the HubConnection.
HubMessageType	
TransportEnum	Used to specify the transport the client will use.

Feedback

Was this page helpful? Yes No

ASP.NET Core SignalR JavaScript client

Article • 06/18/2024

By Rachel Appel

The ASP.NET Core SignalR JavaScript client library enables developers to call server-side SignalR hub code.

Install the SignalR client package

The SignalR JavaScript client library is delivered as an npm package. The following sections outline different ways to install the client library.

Install with npm

Visual Studio

Run the following commands from **Package Manager Console**:

Bash

```
npm init -y
npm install @microsoft/signalr
```

npm installs the package contents in the *node_modules\@microsoft\signalr\dist\browser* folder. Create the *wwwroot/lib/signalr* folder. Copy the `signalr.js` file to the *wwwroot/lib/signalr* folder.

Reference the SignalR JavaScript client in the `<script>` element. For example:

HTML

```
<script src="~/lib/signalr/signalr.js"></script>
```

Use a Content Delivery Network (CDN)

To use the client library without the npm prerequisite, reference a CDN-hosted copy of the client library. For example:

HTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/microsoft-
signalr/6.0.1/signalr.js"></script>
```

In the preceding markup, version 6.0.1 is specified. Choose from one of the following CDNs to get the latest client library version:

- cdnjs
- jsDelivr
- unpkg

Install with LibMan

LibMan can be used to install specific client library files from the CDN-hosted client library. For example, only add the minified JavaScript file to the project. For details on that approach, see Add the SignalR client library.

Connect to a hub

The following code creates and starts a connection. The hub's name is case insensitive:

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();

async function start() {
    try {
        await connection.start();
        console.log("SignalR Connected.");
    } catch (err) {
        console.log(err);
        setTimeout(start, 5000);
    }
};

connection.onclose(async () => {
    await start();
});

// Start the connection.
start();
```

Cross-origin connections (CORS)

Typically, browsers load connections from the same domain as the requested page. However, there are occasions when a connection to another domain is required.

When making cross domain requests, the client code ***must*** use an absolute URL instead of a relative URL. For cross domain requests, change `.withUrl("/chathub")` to `.withUrl("https://{App domain name}/chathub")`.

To prevent a malicious site from reading sensitive data from another site, cross-origin connections are disabled by default. To allow a cross-origin request, enable CORS:

C#

```
using SignalRChat.Hubs;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddSignalR();

builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(
        builder =>
        {
            builder.WithOrigins("https://example.com")
                .AllowAnyHeader()
                .WithMethods("GET", "POST")
                .AllowCredentials();
        });
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

// UseCors must be called before MapHub.
app.UseCors();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");
```

```
app.Run();
```

UseCors must be called before calling MapHub.

Call hub methods from the client

JavaScript clients call public methods on hubs via the invoke method of the HubConnection. The `invoke` method accepts:

- The name of the hub method.
- Any arguments defined in the hub method.

In the following highlighted code, the method name on the hub is `SendMessage`. The second and third arguments passed to `invoke` map to the hub method's `user` and `message` arguments:

JavaScript

```
try {
    await connection.invoke("SendMessage", user, message);
} catch (err) {
    console.error(err);
}
```

Calling hub methods from a client is only supported when using the ***Azure SignalR Service in Default*** mode. For more information, see Frequently Asked Questions (azure-signalr GitHub repository).

The `invoke` method returns a JavaScript Promise. The `Promise` is resolved with the return value (if any) when the method on the server returns. If the method on the server throws an error, the `Promise` is rejected with the error message. Use `async` and `await` or the `Promise`'s `then` and `catch` methods to handle these cases.

JavaScript clients can also call public methods on hubs via the send method of the `HubConnection`. Unlike the `invoke` method, the `send` method doesn't wait for a response from the server. The `send` method returns a JavaScript `Promise`. The `Promise` is resolved when the message has been sent to the server. If there is an error sending the message, the `Promise` is rejected with the error message. Use `async` and `await` or the `Promise`'s `then` and `catch` methods to handle these cases.

Using `send` ***doesn't*** wait until the server has received the message. Consequently, it's not possible to return data or errors from the server.

Call client methods from the hub

To receive messages from the hub, define a method using the on method of the `HubConnection`.

- The name of the JavaScript client method.
- Arguments the hub passes to the method.

In the following example, the method name is `ReceiveMessage`. The argument names are `user` and `message`:

JavaScript

```
connection.on("ReceiveMessage", (user, message) => {
    const li = document.createElement("li");
    li.textContent = `${user}: ${message}`;
    document.getElementById("messageList").appendChild(li);
});
```

The preceding code in `connection.on` runs when server-side code calls it using the SendAsync method:

C#

```
using Microsoft.AspNetCore.SignalR;
namespace SignalRChat.Hubs;

public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message);
    }
}
```

SignalR determines which client method to call by matching the method name and arguments defined in `SendAsync` and `connection.on`.

A **best practice** is to call the start method on the `HubConnection` after `on`. Doing so ensures the handlers are registered before any messages are received.

Error handling and logging

Use `console.error` to output errors to the browser's console when the client can't connect or send a message:

JavaScript

```
try {
    await connection.invoke("SendMessage", user, message);
} catch (err) {
    console.error(err);
}
```

Set up client-side log tracing by passing a logger and type of event to log when the connection is made. Messages are logged with the specified log level and higher. Available log levels are as follows:

- `signalR.LogLevel.Error`: Error messages. Logs `Error` messages only.
- `signalR.LogLevel.Warning`: Warning messages about potential errors. Logs `Warning`, and `Error` messages.
- `signalR.LogLevel.Information`: Status messages without errors. Logs `Information`, `Warning`, and `Error` messages.
- `signalR.LogLevel.Trace`: Trace messages. Logs everything, including data transported between hub and client.

Use the configureLogging method on HubConnectionBuilder to configure the log level. Messages are logged to the browser console:

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Reconnect clients

Automatically reconnect

The JavaScript client for SignalR can be configured to automatically reconnect using the WithAutomaticReconnect method on HubConnectionBuilder. It won't automatically reconnect by default.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
```

```
    .withAutomaticReconnect()
    .build();
```

Without any parameters, WithAutomaticReconnect configures the client to wait 0, 2, 10, and 30 seconds respectively before trying each reconnect attempt. After four failed attempts, it stops trying to reconnect.

Before starting any reconnect attempts, the `HubConnection`:

- Transitions to the HubConnectionState.Reconnecting state and fires its `onreconnecting` callbacks.
- Doesn't transition to the `Disconnected` state and trigger its `onclose` callbacks like a `HubConnection` without automatic reconnect configured.

The reconnect approach provides an opportunity to:

- Warn users that the connection has been lost.
- Disable UI elements.

JavaScript

```
connection.onreconnecting(error => {
    console.assert(connection.state ===
signalR.HubConnectionState.Reconnecting);

    document.getElementById("messageInput").disabled = true;

    const li = document.createElement("li");
    li.textContent = `Connection lost due to error "${error}".
Reconnecting.`;
    document.getElementById("messageList").appendChild(li);
});
```

If the client successfully reconnects within its first four attempts, the `HubConnection` transitions back to the `Connected` state and fire its `onreconnected` callbacks. This provides an opportunity to inform users the connection has been reestablished.

Since the connection looks entirely new to the server, a new `connectionId` is provided to the `onreconnected` callback.

The `onreconnected` callback's `connectionId` parameter is ***undefined*** if the `HubConnection` is configured to skip negotiation.

JavaScript

```
connection.onreconnected(connectionId => {
    console.assert(connection.state ===
signalR.HubConnectionState.Connected);

    document.getElementById("messageInput").disabled = false;

    const li = document.createElement("li");
    li.textContent = `Connection reestablished. Connected with connectionId
"${connectionId}".`;
    document.getElementById("messageList").appendChild(li);
});
```

`withAutomaticReconnect` won't configure the `HubConnection` to retry initial start failures, so start failures need to be handled manually:

JavaScript

```
async function start() {
    try {
        await connection.start();
        console.assert(connection.state ===
signalR.HubConnectionState.Connected);
        console.log("SignalR Connected.");
    } catch (err) {
        console.assert(connection.state ===
signalR.HubConnectionState.Disconnected);
        console.log(err);
        setTimeout(() => start(), 5000);
    }
};
```

If the client doesn't successfully reconnect within its first four attempts, the `HubConnection` transitions to the `Disconnected` state and fires its onclose callbacks. This provides an opportunity to inform users:

- The connection has been permanently lost.
- Try refreshing the page:

JavaScript

```
connection.onclose(error => {
    console.assert(connection.state ===
signalR.HubConnectionState.Disconnected);

    document.getElementById("messageInput").disabled = true;

    const li = document.createElement("li");
    li.textContent = `Connection closed due to error "${error}". Try
refreshing this page to restart the connection.`;
```

```
    document.getElementById("messageList").appendChild(li);
});
```

In order to configure a custom number of reconnect attempts before disconnecting or change the reconnect timing, `withAutomaticReconnect` accepts an array of numbers representing the delay in milliseconds to wait before starting each reconnect attempt.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect([0, 0, 10000])
    .build();

    // .withAutomaticReconnect([0, 2000, 10000, 30000]) yields the default
behavior
```

The preceding example configures the `HubConnection` to start attempting reconnects immediately after the connection is lost. The default configuration also waits zero seconds to attempt reconnecting.

If the first reconnect attempt fails, the second reconnect attempt also starts immediately instead of waiting 2 seconds using the default configuration.

If the second reconnect attempt fails, the third reconnect attempt start in 10 seconds which is the same as the default configuration.

The configured reconnection timing differs from the default behavior by stopping after the third reconnect attempt failure instead of trying one more reconnect attempt in another 30 seconds.

For more control over the timing and number of automatic reconnect attempts, `withAutomaticReconnect` accepts an object implementing the `IRetryPolicy` interface, which has a single method named `nextRetryDelayInMilliseconds`.

`nextRetryDelayInMilliseconds` takes a single argument with the type `RetryContext`. The `RetryContext` has three properties: `previousRetryCount`, `elapsedMilliseconds` and `retryReason` which are a `number`, a `number` and an `Error` respectively. Before the first reconnect attempt, both `previousRetryCount` and `elapsedMilliseconds` will be zero, and the `retryReason` will be the Error that caused the connection to be lost. After each failed retry attempt, `previousRetryCount` will be incremented by one, `elapsedMilliseconds` will be updated to reflect the amount of time spent reconnecting so far in milliseconds, and the `retryReason` will be the Error that caused the last reconnect attempt to fail.

`nextRetryDelayInMilliseconds` must return either a number representing the number of milliseconds to wait before the next reconnect attempt or `null` if the `HubConnection` should stop reconnecting.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect({
        nextRetryDelayInMilliseconds: retryContext => {
            if (retryContext.elapsedMilliseconds < 60000) {
                // If we've been reconnecting for less than 60 seconds so
far,
                // wait between 0 and 10 seconds before the next reconnect
attempt.
                return Math.random() * 10000;
            } else {
                // If we've been reconnecting for more than 60 seconds so
far, stop reconnecting.
                return null;
            }
        }
    })
    .build();
```

Alternatively, code can be written that reconnects the client manually as demonstrated in the following section.

Manually reconnect

The following code demonstrates a typical manual reconnection approach:

1. A function (in this case, the `start` function) is created to start the connection.
2. Call the `start` function in the connection's `onclose` event handler.

JavaScript

```
async function start() {
    try {
        await connection.start();
        console.log("SignalR Connected.");
    } catch (err) {
        console.log(err);
        setTimeout(start, 5000);
    }
};

connection.onclose(async () => {
```

```
    await start();
});
```

Production implementations typically use an exponential back-off or retry a specified number of times.

Browser sleeping tab

Some browsers have a tab freezing or sleeping feature to reduce computer resource usage for inactive tabs. This can cause SignalR connections to close and may result in an unwanted user experience. Browsers use heuristics to figure out if a tab should be put to sleep, such as:

- Playing audio
- Holding a web lock
- Holding an `IndexedDB` lock
- Being connected to a USB device
- Capturing video or audio
- Being mirrored
- Capturing a window or display

Browser heuristics may change over time and can differ between browsers. Check the support matrix and figure out what method works best for your scenarios.

To avoid putting an app to sleep, the app should trigger one of the heuristics that the browser uses.

The following code example shows how to use a Web Lock to keep a tab awake and avoid an unexpected connection closure.

JavaScript

```
var lockResolver;
if (navigator && navigator.locks && navigator.locks.request) {
    const promise = new Promise((res) => {
        lockResolver = res;
    });

    navigator.locks.request('unique_lock_name', { mode: "shared" }, () => {
        return promise;
    });
}
```

For the preceding code example:

- Web Locks are experimental. The conditional check confirms that the browser supports Web Locks.
- The promise resolver, `lockResolver`, is stored so that the lock can be released when it's acceptable for the tab to sleep.
- When closing the connection, the lock is released by calling `lockResolver()`. When the lock is released, the tab is allowed to sleep.

Additional resources

- View or download sample code (how to download)
- JavaScript API reference
- JavaScript tutorial
- WebPack and TypeScript tutorial
- Hubs
- .NET client
- Publish to Azure
- Cross-Origin Requests (CORS)
- Azure SignalR Service serverless documentation
- Troubleshoot connection errors

@microsoft/signalr package

Reference

Classes

Expand table

DefaultHttpClient	Default implementation of HttpClient.
AbortError	Error thrown when an action is aborted.
HttpError	Error thrown when an HTTP request fails.
TimeoutError	Error thrown when a timeout elapses.
FetchHttpClient	
HeaderNames	
HttpClient	Abstraction over an HTTP client. This class provides an abstraction over an HTTP client so that a different implementation can be provided on different platforms.
HttpResponse	Represents an HTTP response.
HubConnection	Represents a connection to a SignalR Hub.
HubConnectionBuilder	A builder for configuring HubConnection instances.
JsonHubProtocol	Implements the JSON Hub Protocol.
NullLogger	A logger that does nothing when log messages are sent to it.
Subject	Stream implementation to stream items to the server.
XhrHttpClient	

Interfaces

Expand table

AbortSignal	Represents a signal that can be monitored to determine if a request has been aborted.

HttpRequest	Represents an HTTP request.
IHttpConnection Options	Options provided to the 'withUrl' method on HubConnectionBuilder to configure options for the HTTP-based transports.
CancelInvocation Message	A hub message sent to request that a streaming invocation be canceled.
CloseMessage	A hub message indicating that the sender is closing the connection. If error is defined, the sender is closing the connection due to an error.
Completion Message	A hub message representing the result of an invocation.
HubInvocation Message	Defines properties common to all Hub messages relating to a specific invocation.
HubMessageBase	Defines properties common to all Hub messages.
IHubProtocol	A protocol abstraction for communicating with SignalR Hubs.
Invocation Message	A hub message representing a non-streaming invocation.
MessageHeaders	Defines a dictionary of string keys and string values representing headers attached to a Hub message.
PingMessage	A hub message indicating that the sender is still active.
StreamInvocation Message	A hub message representing a streaming invocation.
StreamItem Message	A hub message representing a single item produced as part of a result stream.
ILogger	An abstraction that provides a sink for diagnostic messages.
IRetryPolicy	An abstraction that controls when the client attempts to reconnect and how many times it does so.
RetryContext	
ITransport	An abstraction over the behavior of transports. This is designed to support the framework and not intended for use by applications.
IStreamResult	Defines the result of a streaming hub method.
IStreamSubscriber	Defines the expected type for a receiver of results streamed by the server.
ISubscription	An interface that allows an IStreamSubscriber to be disconnected from a stream.

Type Aliases

Expand table

HubMessage	Union type of all known Hub messages.

Enums

Expand table

HubConnectionState	Describes the current state of the HubConnection to the server.
MessageType	Defines the type of a Hub Message.
LogLevel	Indicates the severity of a log message. Log Levels are ordered in increasing severity. So `Debug` is more severe than `Trace`, etc.
HttpTransportType	Specifies a specific HTTP transport type.
TransferFormat	Specifies the transfer format for a connection.

ASP.NET Core SignalR hosting and scaling

Article • 06/03/2022

By Andrew Stanton-Nurse, Brady Gaster, and Tom Dykstra

This article explains hosting and scaling considerations for high-traffic apps that use ASP.NET Core SignalR.

Sticky Sessions

SignalR requires that all HTTP requests for a specific connection be handled by the same server process. When SignalR is running on a server farm (multiple servers), "sticky sessions" must be used. "Sticky sessions" are also called session affinity. Azure App Service uses Application Request Routing (ARR) to route requests. Enabling the "Session affinity" (ARR Affinity) setting in your Azure App Service enables "sticky sessions." The only circumstances in which sticky sessions aren't required for the app are:

1. When hosting on a single server in a single process.
2. When using the Azure SignalR Service (sticky sessions are enabled for the service, not the app).
3. When all clients are configured to **only** use WebSockets, **and** the SkipNegotiation setting is enabled in the client configuration.

In all other circumstances (including when the Redis backplane is used), the server environment must be configured for sticky sessions.

For guidance on configuring Azure App Service for SignalR, see Publish an ASP.NET Core SignalR app to Azure App Service. For guidance on configuring sticky sessions for Blazor apps that use the Azure SignalR Service, see Host and deploy ASP.NET Core server-side Blazor apps.

TCP connection resources

The number of concurrent TCP connections that a web server can support is limited. Standard HTTP clients use *ephemeral* connections. These connections can be closed when the client goes idle and reopened later. On the other hand, a SignalR connection is *persistent*. SignalR connections stay open even when the client goes idle. In a high-traffic app that serves many clients, these persistent connections can cause servers to hit their maximum number of connections.

Persistent connections also consume some additional memory, to track each connection.

The heavy use of connection-related resources by SignalR can affect other web apps that are hosted on the same server. When SignalR opens and holds the last available TCP connections, other web apps on the same server also have no more connections available to them.

If a server runs out of connections, you'll see random socket errors and connection reset errors. For example:

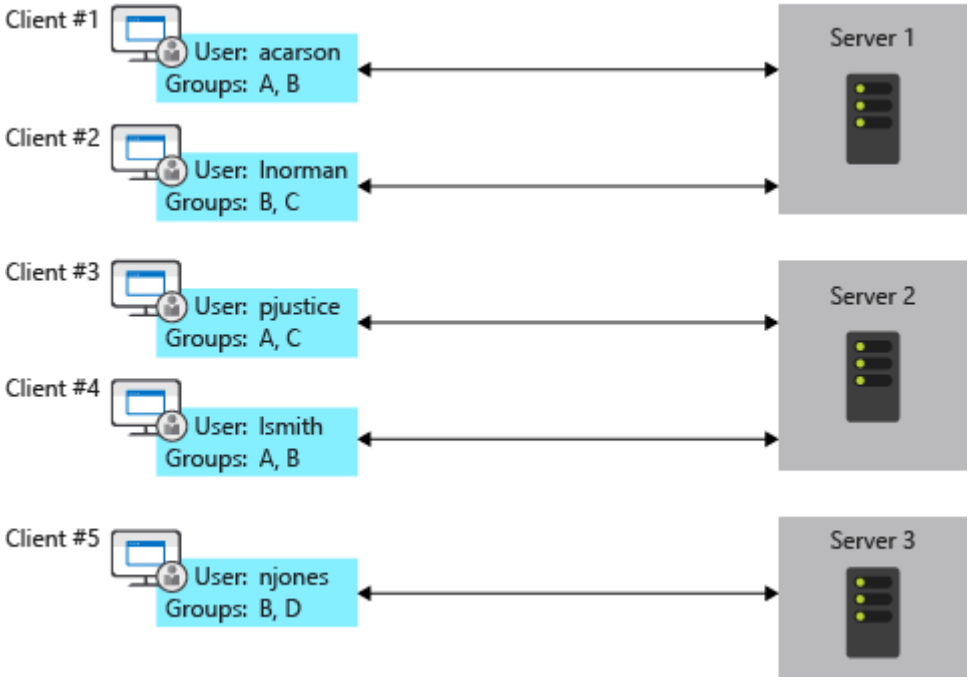
```
An attempt was made to access a socket in a way forbidden by its access permissions...
```

To keep SignalR resource usage from causing errors in other web apps, run SignalR on different servers than your other web apps.

To keep SignalR resource usage from causing errors in a SignalR app, scale out to limit the number of connections a server has to handle.

Scale out

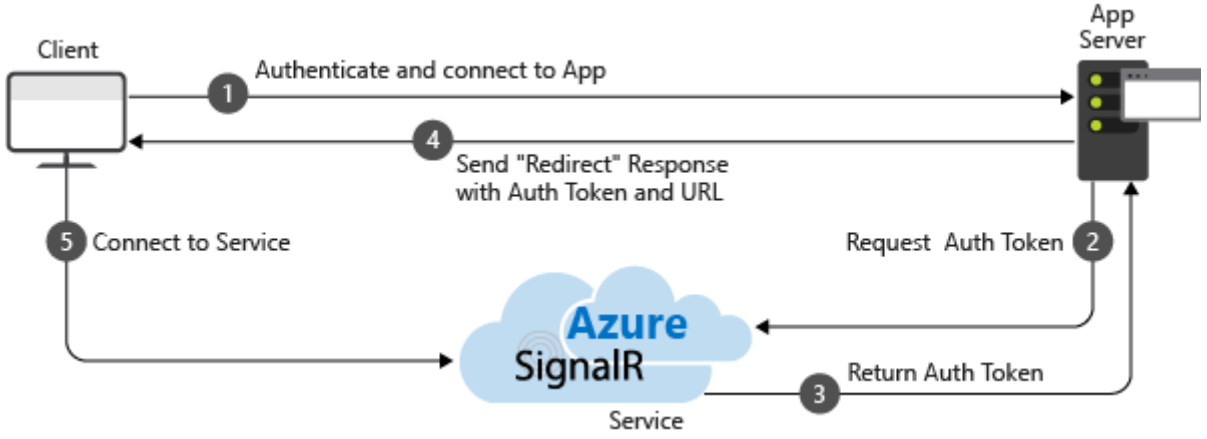
An app that uses SignalR needs to keep track of all its connections, which creates problems for a server farm. Add a server, and it gets new connections that the other servers don't know about. For example, SignalR on each server in the following diagram is unaware of the connections on the other servers. When SignalR on one of the servers wants to send a message to all clients, the message only goes to the clients connected to that server.



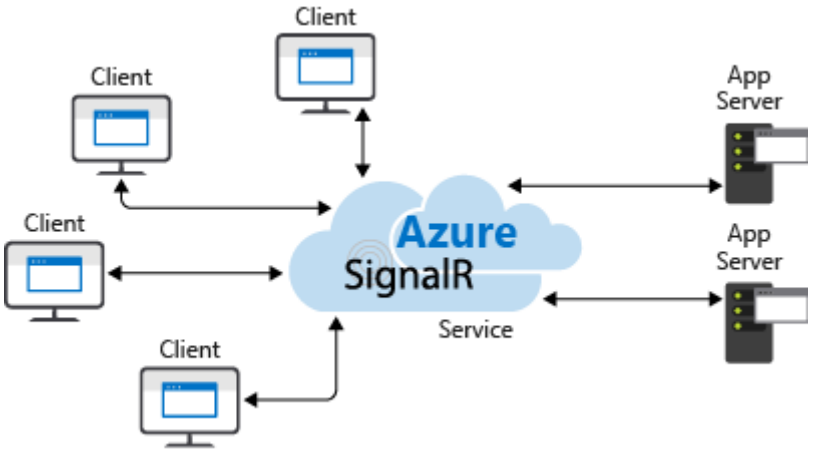
The options for solving this problem are the Azure SignalR Service and Redis backplane.

Azure SignalR Service

The Azure SignalR Service functions as a proxy for real-time traffic and doubles as a backplane when the app is scaled out across multiple servers. Each time a client initiates a connection to the server, the client is redirected to connect to the service. The process is illustrated by the following diagram:



The result is that the service manages all of the client connections, while each server needs only a small constant number of connections to the service, as shown in the following diagram:



This approach to scale-out has several advantages over the Redis backplane alternative:

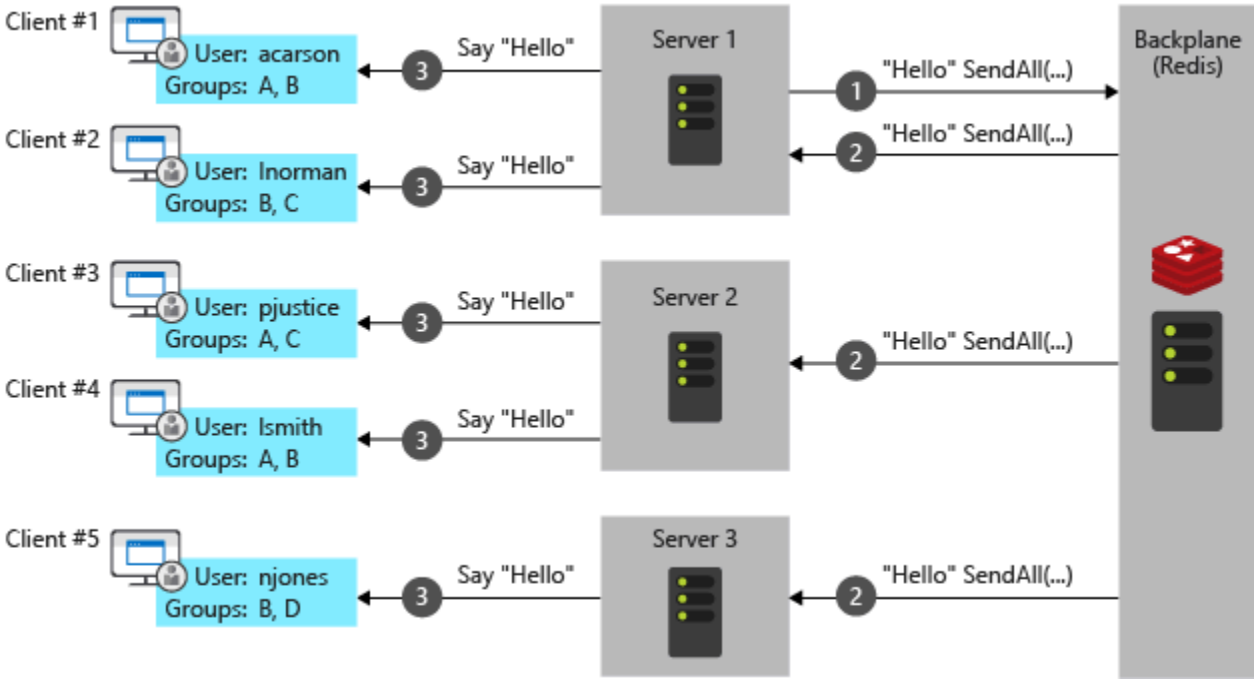
- Sticky sessions, also known as client affinity, is not required, because clients are immediately redirected to the Azure SignalR Service when they connect.
- A SignalR app can scale out based on the number of messages sent, while the Azure SignalR Service scales to handle any number of connections. For example, there could be thousands of clients, but if only a few messages per second are sent, the SignalR app won't need to scale out to multiple servers just to handle the connections themselves.
- A SignalR app won't use significantly more connection resources than a web app without SignalR.

For these reasons, we recommend the Azure SignalR Service for all ASP.NET Core SignalR apps hosted on Azure, including App Service, VMs, and containers.

For more information see the Azure SignalR Service documentation.

Redis backplane

Redis is an in-memory key-value store that supports a messaging system with a publish/subscribe model. The SignalR Redis backplane uses the pub/sub feature to forward messages to other servers. When a client makes a connection, the connection information is passed to the backplane. When a server wants to send a message to all clients, it sends to the backplane. The backplane knows all connected clients and which servers they're on. It sends the message to all clients via their respective servers. This process is illustrated in the following diagram:



The Redis backplane is the recommended scale-out approach for apps hosted on your own infrastructure. If there is significant connection latency between your data center and an Azure data center, Azure SignalR Service may not be a practical option for on-premises apps with low latency or high throughput requirements.

The Azure SignalR Service advantages noted earlier are disadvantages for the Redis backplane:

- Sticky sessions, also known as client affinity, is required, except when **both** of the following are true:
  - All clients are configured to **only** use WebSockets.
  - The SkipNegotiation setting is enabled in the client configuration. Once a connection is initiated on a server, the connection has to stay on that server.
- A SignalR app must scale out based on number of clients even if few messages are being sent.
- A SignalR app uses significantly more connection resources than a web app without SignalR.

IIS limitations on Windows client OS

Windows 10 and Windows 8.x are client operating systems. IIS on client operating systems has a limit of 10 concurrent connections. SignalR's connections are:

- Transient and frequently re-established.
- **Not** disposed immediately when no longer used.

The preceding conditions make it likely to hit the 10 connection limit on a client OS. When a client OS is used for development, we recommend:

- Avoid IIS.
- Use Kestrel or IIS Express as deployment targets.

Linux with Nginx

The following contains the minimum required settings to enable WebSockets, ServerSentEvents, and LongPolling for SignalR:

nginx

```
http {
  map $http_connection $connection_upgrade {
    "~*Upgrade" $http_connection;
    default keep-alive;
  }

  server {
    listen 80;
    server_name example.com *.example.com;

    # Configure the SignalR Endpoint
    location /hubroute {
      # App server url
      proxy_pass http://localhost:5000;

      # Configuration for WebSockets
      proxy_set_header Upgrade $http_upgrade;
      proxy_set_header Connection $connection_upgrade;
      proxy_cache off;
      # WebSockets were implemented after http/1.0
      proxy_http_version 1.1;

      # Configuration for ServerSentEvents
      proxy_buffering off;

      # Configuration for LongPolling or if your KeepAliveInterval is longer
than 60 seconds
      proxy_read_timeout 100s;

      proxy_set_header Host $host;
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header X-Forwarded-Proto $scheme;
    }
  }
}
```

When multiple backend servers are used, sticky sessions must be added to prevent SignalR connections from switching servers when connecting. There are multiple ways to

add sticky sessions in Nginx. Two approaches are shown below depending on what you have available.

The following is added in addition to the previous configuration. In the following examples, `backend` is the name of the group of servers.

With Nginx Open Source, use `ip_hash` to route connections to a server based on the client's IP address:

nginx

```
http {
  upstream backend {
    # App server 1
    server localhost:5000;
    # App server 2
    server localhost:5002;

    ip_hash;
  }
}
```

With Nginx Plus, use `sticky` to add a cookie to requests and pin the user's requests to a server:

nginx

```
http {
  upstream backend {
    # App server 1
    server localhost:5000;
    # App server 2
    server localhost:5002;

    sticky cookie srv_id expires=max domain=.example.com path=/ httponly;
  }
}
```

Finally, change `proxy_pass http://localhost:5000` in the `server` section to `proxy_pass http://backend`.

For more information on WebSockets over Nginx, see NGINX as a WebSocket Proxy.

For more information on load balancing and sticky sessions, see NGINX load balancing.

For more information about ASP.NET Core with Nginx see the following article:

- Host ASP.NET Core on Linux with Nginx

Third-party SignalR backplane providers

- NCache
- Orleans
- Rebus
- SQL Server

Next steps

For more information, see the following resources:

- Azure SignalR Service documentation
- Set up a Redis backplane

Publish an ASP.NET Core SignalR app to Azure App Service

Article • 09/17/2024

By Brady Gaster

Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core.

> **Note**
>
> This article refers to publishing an ASP.NET Core SignalR app from Visual Studio. For more information, see SignalR service for Azure.

Publish the app

This article covers publishing using the tools in Visual Studio. Visual Studio Code users can use Azure CLI commands to publish apps to Azure. For more information, see Publish an ASP.NET Core app to Azure with command line tools.

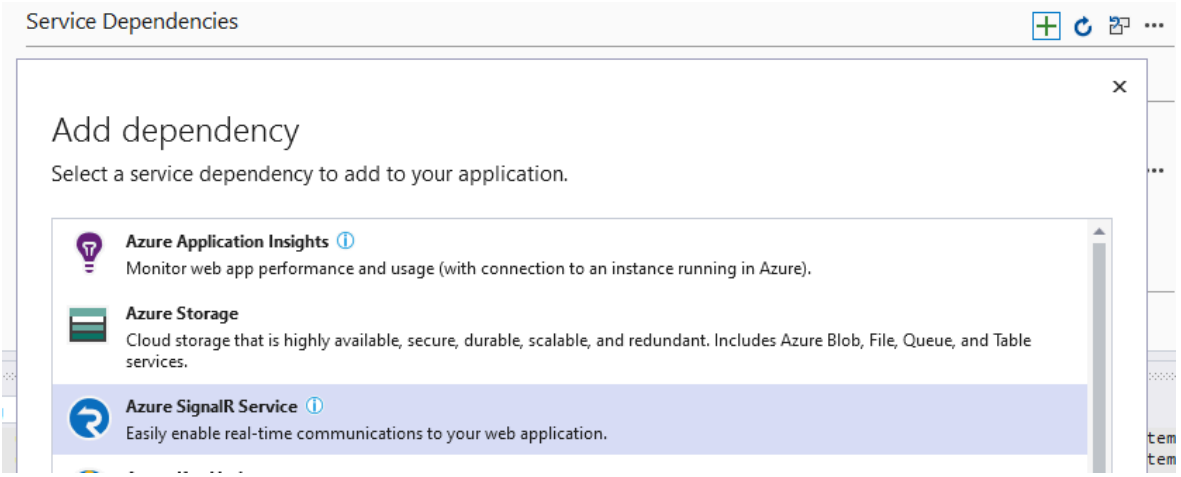
1. Right-click on the project in **Solution Explorer** and select **Publish**.
2. Confirm that **App Service** and **Create new** are selected in the **Pick a publish target** dialog.
3. Select **Create Profile** from the **Publish** button drop down.

   Enter the information described in the following table in the **Create App Service** dialog and select **Create**.

Expand table

Item	Description
Name	Unique name of the app.
Subscription	Azure subscription that the app uses.
Resource Group	Group of related resources to which the app belongs.
Hosting Plan	Pricing plan for the web app.

4. Select **Azure SignalR Service** in the **Service Dependencies** section. Select the + button:



5. In the **Azure SignalR Service** dialog, select **Create a new Azure SignalR Service instance**.

6. Provide a **Name**, **Resource Group**, and **Location**. Return to the **Azure SignalR Service** dialog and select **Add**.

Visual Studio completes the following tasks:

- Creates a Publish Profile containing publish settings.
- Creates an *Azure Web App* with the provided details.
- Publishes the app.
- Launches a browser, which loads the web app.

The format of the app's URL is `{APP SERVICE NAME}.azurewebsites.net`. For example, an app named `SignalRChatApp` has a URL of `https://signalrchatapp.azurewebsites.net`.

If an HTTP *502.2 - Bad Gateway* error occurs when deploying an app that targets a preview .NET Core release, see Deploy ASP.NET Core preview release to Azure App Service to resolve it.

Configure the app in Azure App Service

> **Note**
>
> *This section only applies to apps not using the Azure SignalR Service.*
>
> If the app uses the Azure SignalR Service, the App Service doesn't require the configuration of WebSockets and session affinity, also called Application Request

Routing (ARR) affinity, described in this section. Clients connect their WebSockets to the Azure SignalR Service, not directly to the app.

For apps hosted without the Azure SignalR Service, enable:

- WebSockets to allow the WebSockets transport to function. The default setting is **Off**.
- Session affinity (ARR affinity) to route requests from a user back to the same App Service instance. The default setting is **On**.

1. In the Azure portal, navigate to the web app in **App Services**.
2. Open **Settings** > **Configuration**.
3. Set **Web sockets** to **On**.
4. Verify that **Session affinity** is set to **On**.

App Service Plan limits

WebSockets and other transports are limited based on the App Service Plan selected. For more information, see the *Azure Cloud Services limits* and *App Service limits* sections of the Azure subscription and service limits, quotas, and constraints article.

Additional resources

- What is Azure SignalR Service?
- Overview of ASP.NET Core SignalR
- Host and deploy ASP.NET Core
- Publish an ASP.NET Core app to Azure with Visual Studio
- Publish an ASP.NET Core app to Azure with command line tools
- Host and deploy ASP.NET Core Preview apps on Azure

Set up a Redis backplane for ASP.NET Core SignalR scale-out

Article • 11/04/2024

By Andrew Stanton-Nurse, Brady Gaster, and Tom Dykstra.

This article explains SignalR-specific aspects of setting up a Redis server to use for scaling out an ASP.NET Core SignalR app.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

Set up a Redis backplane

- Deploy a Redis server.

  > ⓘ **Important**
  >
  > For production use, a Redis backplane is recommended only when it runs in the same data center as the SignalR app. Otherwise, network latency degrades performance. If your SignalR app is running in the Azure cloud, we recommend Azure SignalR Service instead of a Redis backplane.

  For more information, see the following resources:
  - ASP.NET Core SignalR production hosting and scaling
  - Redis documentation
  - Azure Redis Cache documentation

- In the SignalR app, install the following NuGet package:
  - `Microsoft.AspNetCore.SignalR.StackExchangeRedis`

- Call AddStackExchangeRedis by adding the following line before the line that calls `builder.Build()`) in the `Program.cs` file.

  ```
  C#

  builder.Services.AddSignalR().AddStackExchangeRedis("
  <your_Redis_connection_string>");
  ```

- Configure options as needed:

  Most options can be set in the connection string or in the ConfigurationOptions object. Options specified in `ConfigurationOptions` override the ones set in the connection string.

  The following example shows how to set options in the `ConfigurationOptions` object. This example adds a channel prefix so that multiple apps can share the same Redis instance, as explained in the following step.

  ```
  C#

  builder.Services.AddSignalR()
    .AddStackExchangeRedis(connectionString, options => {
        options.Configuration.ChannelPrefix =
  RedisChannel.Literal("MyApp");
    });
  ```

  In the preceding code, `options.Configuration` is initialized with whatever was specified in the connection string.

  For information about Redis options, see the StackExchange Redis documentation.

- If you're using one Redis server for multiple SignalR apps, use a different channel prefix for each SignalR app.

  Setting a channel prefix isolates one SignalR app from others that use different channel prefixes. If you don't assign different prefixes, a message sent from one app to all of its own clients will go to all clients of all apps that use the Redis server as a backplane.

- Configure your server farm load balancing software for sticky sessions. Here are some examples of documentation on how to do that:
  - IIS
  - HAProxy
  - Nginx

Redis server errors

When a Redis server goes down, SignalR throws exceptions that indicate messages won't be delivered. Some typical exception messages:

- *Failed writing message*
- *Failed to invoke hub method 'MethodName'*
- *Connection to Redis failed*

SignalR doesn't buffer messages to send them when the server comes back up. Any messages sent while the Redis server is down are lost.

SignalR automatically reconnects when the Redis server is available again.

Custom behavior for connection failures

Here's an example that shows how to handle Redis connection failure events.

C#

```
services.AddSignalR()
        .AddMessagePackProtocol()
        .AddStackExchangeRedis(o =>
        {
            o.ConnectionFactory = async writer =>
            {
                var config = new ConfigurationOptions
                {
                    AbortOnConnectFail = false
                };
                config.EndPoints.Add(IPAddress.Loopback, 0);
                config.SetDefaultPorts();
                var connection = await
ConnectionMultiplexer.ConnectAsync(config, writer);
                connection.ConnectionFailed += (_, e) =>
                {
                    Console.WriteLine("Connection to Redis failed.");
                };

                if (!connection.IsConnected)
                {
                    Console.WriteLine("Did not connect to Redis.");
                }

                return connection;
            };
        });
```

Redis Cluster

Redis Cluster utilizes multiple simultaneously active Redis servers to achieve high availability. When Redis Cluster is used as the backplane for SignalR, messages are delivered to all of the nodes of the cluster without code modifications to the app.

There's a tradeoff between the number of nodes in the cluster and the throughput of the backplane. Increasing the number of nodes increases the availability of the cluster but decreases the throughput because messages must be transmitted to all of the nodes in the cluster.

In the SignalR app, include all of the possible Redis nodes using either of the following approaches:

- List the nodes in the connection string delimited with commas.
- If using custom behavior for connection failures, add the nodes to ConfigurationOptions.Endpoints.

Next steps

For more information, see the following resources:

- ASP.NET Core SignalR production hosting and scaling
- Redis documentation
- StackExchange Redis documentation
- Azure Redis Cache documentation

Host ASP.NET Core SignalR in background services

Article • 06/03/2022

By Dave Pringle and Brady Gaster

This article provides guidance for:

- Hosting SignalR Hubs using a background worker process hosted with ASP.NET Core.
- Sending messages to connected clients from within a .NET Core BackgroundService.

View or download sample code (how to download)

Enable SignalR at app startup

Hosting ASP.NET Core SignalR Hubs in the context of a background worker process is identical to hosting a Hub in an ASP.NET Core web app. In `Program.cs`, calling `builder.Services.AddSignalR` adds the required services to the ASP.NET Core Dependency Injection (DI) layer to support SignalR. The `MapHub` method is called on the `WebApplication` `app` to connect the Hub endpoints in the ASP.NET Core request pipeline.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();
builder.Services.AddHostedService<Worker>();

var app = builder.Build();

app.MapHub<ClockHub>("/hubs/clock");

app.Run();
```

In the preceding example, the `ClockHub` class implements the `Hub<T>` class to create a strongly typed Hub. The `ClockHub` has been configured in `Program.cs` to respond to requests at the endpoint `/hubs/clock`.

For more information on strongly typed Hubs, see Use hubs in SignalR for ASP.NET Core.

> **Note**
>
> This functionality isn't limited to the Hub<T> class. Any class that inherits from Hub, such as DynamicHub, works.

C#

```
public class ClockHub : Hub<IClock>
{
    public async Task SendTimeToClients(DateTime dateTime)
    {
        await Clients.All.ShowTime(dateTime);
    }
}
```

The interface used by the strongly typed `ClockHub` is the `IClock` interface.

C#

```
public interface IClock
{
    Task ShowTime(DateTime currentTime);
}
```

Call a SignalR Hub from a background service

During startup, the `Worker` class, a `BackgroundService`, is enabled using `AddHostedService`.

C#

```
builder.Services.AddHostedService<Worker>();
```

Since SignalR is also enabled up during the startup phase, in which each Hub is attached to an individual endpoint in ASP.NET Core's HTTP request pipeline, each Hub is represented by an `IHubContext<T>` on the server. Using ASP.NET Core's DI features, other classes instantiated by the hosting layer, like `BackgroundService` classes, MVC Controller classes, or Razor page models, can get references to server-side Hubs by accepting instances of `IHubContext<ClockHub, IClock>` during construction.

C#

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;
    private readonly IHubContext<ClockHub, IClock> _clockHub;

    public Worker(ILogger<Worker> logger, IHubContext<ClockHub, IClock>
clockHub)
    {
        _logger = logger;
        _clockHub = clockHub;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {Time}",
DateTime.Now);
            await _clockHub.Clients.All.ShowTime(DateTime.Now);
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

As the `ExecuteAsync` method is called iteratively in the background service, the server's current date and time are sent to the connected clients using the `ClockHub`.

React to SignalR events with background services

Like a Single Page App using the JavaScript client for SignalR, or a .NET desktop app using the ASP.NET Core SignalR .NET Client, a `BackgroundService` or `IHostedService` implementation can also be used to connect to SignalR Hubs and respond to events.

The `ClockHubClient` class implements both the `IClock` interface and the `IHostedService` interface. This way it can be enabled during startup to run continuously and respond to Hub events from the server.

C#

```
public partial class ClockHubClient : IClock, IHostedService
{
}
```

During initialization, the `ClockHubClient` creates an instance of a `HubConnection` and enables the `IClock.ShowTime` method as the handler for the Hub's `ShowTime` event.

C#

```
private readonly ILogger<ClockHubClient> _logger;
private HubConnection _connection;

public ClockHubClient(ILogger<ClockHubClient> logger)
{
    _logger = logger;

    _connection = new HubConnectionBuilder()
        .WithUrl(Strings.HubUrl)
        .Build();

    _connection.On<DateTime>(Strings.Events.TimeSent, ShowTime);
}

public Task ShowTime(DateTime currentTime)
{
    _logger.LogInformation("{CurrentTime}",
currentTime.ToShortTimeString());

    return Task.CompletedTask;
}
```

In the `IHostedService.StartAsync` implementation, the `HubConnection` is started asynchronously.

C#

```
public async Task StartAsync(CancellationToken cancellationToken)
{
    // Loop is here to wait until the server is running
    while (true)
    {
        try
        {
            await _connection.StartAsync(cancellationToken);

            break;
        }
        catch
        {
            await Task.Delay(1000, cancellationToken);
        }
    }
}
```

During the `IHostedService.StopAsync` method, the `HubConnection` is disposed of asynchronously.

```
C#
```

```
public async Task StopAsync(CancellationToken cancellationToken)
{
    await _connection.DisposeAsync();
}
```

Additional resources

- Get started
- Hubs
- Publish to Azure
- Strongly typed Hubs

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core SignalR configuration

Article • 05/02/2024

This article covers ASP.NET Core SignalR configuration.

For Blazor SignalR guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor SignalR guidance.

JSON/MessagePack serialization options

ASP.NET Core SignalR supports two protocols for encoding messages: JSON and MessagePack. Each protocol has serialization configuration options.

JSON serialization can be configured on the server using the AddJsonProtocol extension method. `AddJsonProtocol` can be added after AddSignalR in `Startup.ConfigureServices`. The `AddJsonProtocol` method takes a delegate that receives an `options` object. The PayloadSerializerOptions property on that object is a `System.Text.Json` JsonSerializerOptions object that can be used to configure serialization of arguments and return values. For more information, see the System.Text.Json documentation.

For example, to configure the serializer to not change the casing of property names, rather than the default camel case names, use the following code in `Program.cs`:

C#

```
builder.Services.AddSignalR()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
    });
```

In the .NET client, the same `AddJsonProtocol` extension method exists on HubConnectionBuilder. The `Microsoft.Extensions.DependencyInjection` namespace must be imported to resolve the extension method:

C#

```
// At the top of the file:
using Microsoft.Extensions.DependencyInjection;

// When constructing your connection:
var connection = new HubConnectionBuilder()
    .AddJsonProtocol(options => {
        options.PayloadSerializerOptions.PropertyNamingPolicy = null;
```

```
    })
    .Build();
```

> **Note**
>
> It's not possible to configure JSON serialization in the JavaScript client at this time.

Switch to Newtonsoft.Json

If you need features of `Newtonsoft.Json` that aren't supported in `System.Text.Json`, see Switch to Newtonsoft.Json.

MessagePack serialization options

MessagePack serialization can be configured by providing a delegate to the AddMessagePackProtocol call. See MessagePack in SignalR for more details.

> **Note**
>
> It's not possible to configure MessagePack serialization in the JavaScript client at this time.

Configure server options

The following table describes options for configuring SignalR hubs:

Expand table

Option	Default Value	Description
`ClientTimeoutInterval`	30 seconds	The server considers the client disconnected if it hasn't received a message (including keep-alive) in this interval. It could take longer than this timeout interval for the client to be marked disconnected due to how this is implemented. The recommended value is double the `KeepAliveInterval` value.

Option	Default Value	Description
`HandshakeTimeout`	15 seconds	If the client doesn't send an initial handshake message within this time interval, the connection is closed. This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification.
`KeepAliveInterval`	15 seconds	If the server hasn't sent a message within this interval, a ping message is sent automatically to keep the connection open. When changing `KeepAliveInterval`, change the `ServerTimeout` or `serverTimeoutInMilliseconds` setting on the client. The recommended `ServerTimeout` or `serverTimeoutInMilliseconds` value is double the `KeepAliveInterval` value.
`SupportedProtocols`	All installed protocols	Protocols supported by this hub. By default, all protocols registered on the server are allowed. Protocols can be removed from this list to disable specific protocols for individual hubs.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a Hub method. The default is `false` because these exception messages can contain sensitive information.
`StreamBufferCapacity`	`10`	The maximum number of items that can be buffered for client upload streams. If this limit is reached, the processing of invocations is blocked until the server processes stream items.
`MaximumReceiveMessageSize`	32 KB	Maximum size of a single incoming hub message. Increasing the value might increase the risk of Denial of service (DoS) attacks.
`MaximumParallelInvocationsPerClient`	1	The maximum number of hub methods that each client can call in parallel before queueing.

Option	Default Value	Description
`DisableImplicitFromServicesParameters`	`false`	Hub method arguments are resolved from DI if possible.

Options can be configured for all hubs by providing an options delegate to the `AddSignalR` call in `Program.cs`.

C#

```
builder.Services.AddSignalR(hubOptions =>
{
    hubOptions.EnableDetailedErrors = true;
    hubOptions.KeepAliveInterval = TimeSpan.FromMinutes(1);
});
```

Options for a single hub override the global options provided in `AddSignalR` and can be configured using AddHubOptions:

C#

```
builder.Services.AddSignalR().AddHubOptions<ChatHub>(options =>
{
    options.EnableDetailedErrors = true;
});
```

Advanced HTTP configuration options

Use `HttpConnectionDispatcherOptions` to configure advanced settings related to transports and memory buffer management. These options are configured by passing a delegate to MapHub in `Program.cs`.

C#

```
using Microsoft.AspNetCore.Http.Connections;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
```

```
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chathub", options =>
{
    options.Transports =
        HttpTransportType.WebSockets |
        HttpTransportType.LongPolling;
}
);
app.Run();
```

The following table describes options for configuring ASP.NET Core SignalR's advanced HTTP options:

Expand table

Option	Default Value	Description
`ApplicationMaxBufferSize`	64 KB	The maximum number of bytes received from the client that the server buffers before applying backpressure. Increasing this value allows the server to receive larger messages faster without applying backpressure, but can increase memory consumption.
`TransportMaxBufferSize`	64 KB	The maximum number of bytes sent by the app that the server buffers before observing backpressure. Increasing this value allows the server to buffer larger messages faster without awaiting backpressure, but can increase memory consumption.
`AuthorizationData`	Data automatically gathered from the `Authorize` attributes applied to the Hub class.	A list of IAuthorizeData objects used to determine if a client is authorized to connect to the hub.

Option	Default Value	Description
`Transports`	All Transports are enabled.	A bit flags enum of `HttpTransportType` values that can restrict the transports a client can use to connect.
`LongPolling`	See below.	Additional options specific to the Long Polling transport.
`WebSockets`	See below.	Additional options specific to the WebSockets transport.
`MinimumProtocolVersion`	0	Specify the minimum version of the negotiate protocol. This is used to limit clients to newer versions.
`CloseOnAuthenticationExpiration`	false	Set this option to enable authentication expiration tracking, which will close connections when a token expires.

The Long Polling transport has additional options that can be configured using the `LongPolling` property:

Expand table

Option	Default Value	Description
`PollTimeout`	90 seconds	The maximum amount of time the server waits for a message to send to the client before terminating a single poll request. Decreasing this value causes the client to issue new poll requests more frequently.

The WebSocket transport has additional options that can be configured using the `WebSockets` property:

Expand table

Option	Default Value	Description
`CloseTimeout`	5 seconds	After the server closes, if the client fails to close within this time interval, the connection is terminated.
`SubProtocolSelector`	`null`	A delegate that can be used to set the `Sec-WebSocket-Protocol` header to a custom value. The delegate receives the values requested by the client as input and is expected to return the desired value.

Configure client options

Client options can be configured on the `HubConnectionBuilder` type (available in the .NET and JavaScript clients). It's also available in the Java client, but the `HttpHubConnectionBuilder` subclass is what contains the builder configuration options, as well as on the `HubConnection` itself.

Configure logging

Logging is configured in the .NET Client using the `ConfigureLogging` method. Logging providers and filters can be registered in the same way as they are on the server. See the Logging in ASP.NET Core documentation for more information.

> **Note**
>
> In order to register Logging providers, you must install the necessary packages. See the Built-in logging providers section of the docs for a full list.

For example, to enable Console logging, install the `Microsoft.Extensions.Logging.Console` NuGet package. Call the `AddConsole` extension method:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub")
    .ConfigureLogging(logging => {
        logging.SetMinimumLevel(LogLevel.Information);
        logging.AddConsole();
    })
    .Build();
```

In the JavaScript client, a similar `configureLogging` method exists. Provide a `LogLevel` value indicating the minimum level of log messages to produce. Logs are written to the browser console window.

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging(signalR.LogLevel.Information)
    .build();
```

Instead of a `LogLevel` value, you can also provide a `string` value representing a log level name. This is useful when configuring SignalR logging in environments where you don't have access to the `LogLevel` constants.

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .configureLogging("warn")
    .build();
```

The following table lists the available log levels. The value you provide to `configureLogging` sets the **minimum** log level that will be logged. Messages logged at this level, **or the levels listed after it in the table**, will be logged.

Expand table

String	LogLevel
`trace`	`LogLevel.Trace`
`debug`	`LogLevel.Debug`
`info` or `information`	`LogLevel.Information`
`warn` or `warning`	`LogLevel.Warning`
`error`	`LogLevel.Error`
`critical`	`LogLevel.Critical`
`none`	`LogLevel.None`

> **Note**
>
> To disable logging entirely, specify `signalR.LogLevel.None` in the `configureLogging` method.

For more information on logging, see the SignalR Diagnostics documentation.

The SignalR Java client uses the SLF4J library for logging. It's a high-level logging API that allows users of the library to choose their own specific logging implementation by bringing in a specific logging dependency. The following code snippet shows how to use `java.util.logging` with the SignalR Java client.

Gradle

```
implementation 'org.slf4j:slf4j-jdk14:1.7.25'
```

If you don't configure logging in your dependencies, SLF4J loads a default no-operation logger with the following warning message:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

This can safely be ignored.

Configure allowed transports

The transports used by SignalR can be configured in the `WithUrl` call (`withUrl` in JavaScript). A bitwise-OR of the values of `HttpTransportType` can be used to restrict the client to only use the specified transports. All transports are enabled by default.

For example, to disable the Server-Sent Events transport, but allow WebSockets and Long Polling connections:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", HttpTransportType.WebSockets |
HttpTransportType.LongPolling)
    .Build();
```

In the JavaScript client, transports are configured by setting the `transport` field on the options object provided to `withUrl`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", { transport: signalR.HttpTransportType.WebSockets |
signalR.HttpTransportType.LongPolling })
    .build();
```

In this version of the Java client WebSockets is the only available transport.

In the Java client, the transport is selected with the `withTransport` method on the `HttpHubConnectionBuilder`. The Java client defaults to using the WebSockets transport.

Java

```
HubConnection hubConnection =
HubConnectionBuilder.create("https://example.com/chathub")
    .withTransport(TransportEnum.WEBSOCKETS)
    .build();
```

> **Note**
>
> The SignalR Java client doesn't support transport fallback yet.

Configure bearer authentication

To provide authentication data along with SignalR requests, use the `AccessTokenProvider` option (`accessTokenFactory` in JavaScript) to specify a function that returns the desired access token. In the .NET Client, this access token is passed in as an HTTP "Bearer Authentication" token (Using the `Authorization` header with a type of `Bearer`). In the JavaScript client, the access token is used as a Bearer token, **except** in a few cases where browser APIs restrict the ability to apply headers (specifically, in Server-Sent Events and WebSockets requests). In these cases, the access token is provided as a query string value `access_token`.

In the .NET client, the `AccessTokenProvider` option can be specified using the options delegate in `WithUrl`:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.AccessTokenProvider = async () => {
            // Get and return the access token.
        };
    })
    .Build();
```

In the JavaScript client, the access token is configured by setting the `accessTokenFactory` field on the options object in `withUrl`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        accessTokenFactory: () => {
            // Get and return the access token.
            // This function can return a JavaScript Promise if asynchronous
            // logic is required to retrieve the access token.
        }
    })
    .build();
```

In the SignalR Java client, you can configure a bearer token to use for authentication by providing an access token factory to the HttpHubConnectionBuilder. Use withAccessTokenFactory to provide an RxJava Single<String>. With a call to Single.defer, you can write logic to produce access tokens for your client.

Java

```
HubConnection hubConnection =
HubConnectionBuilder.create("https://example.com/chathub")
    .withAccessTokenProvider(Single.defer(() -> {
        // Your logic here.
        return Single.just("An Access Token");
    })).build();
```

Configure timeout and keep-alive options

Additional options for configuring timeout and keep-alive behavior:

.NET

Expand table

Option	Default value	Description
`WithServerTimeout`	30 seconds (30,000 milliseconds)	Timeout for server activity and is set directly on HubConnectionBuilder. If the server hasn't sent a message in this interval, the client considers the server disconnected and triggers the `Closed` event (`onclose` in JavaScript). This value must be large enough for a ping message to be sent from the server **and** received by the client within the timeout interval. The recommended value is a number at least double the server's keep-alive

Option	Default value	Description
		interval (`WithKeepAliveInterval`) value to allow time for pings to arrive.
`HandshakeTimeout`	15 seconds	Timeout for initial server handshake and is available on the `HubConnection` object itself. If the server doesn't send a handshake response in this interval, the client cancels the handshake and triggers the `Closed` event (`onclose` in JavaScript). This is an advanced setting that should only be modified if handshake timeout errors are occurring due to severe network latency. For more detail on the handshake process, see the SignalR Hub Protocol Specification .
`WithKeepAliveInterval`	15 seconds	Determines the interval at which the client sends ping messages and is set directly on HubConnectionBuilder. This setting allows the server to detect hard disconnects, such as when a client unplugs their computer from the network. Sending any message from the client resets the timer to the start of the interval. If the client hasn't sent a message in the `ClientTimeoutInterval` set on the server, the server considers the client disconnected.

In the .NET Client, timeout values are specified as `TimeSpan` values.

The following example shows values that are double the default values:

C#

```
var builder = new HubConnectionBuilder()
    .WithUrl(Navigation.ToAbsoluteUri("/chathub"))
    .WithServerTimeout(TimeSpan.FromSeconds(60))
    .WithKeepAliveInterval(TimeSpan.FromSeconds(30))
    .Build();

builder.On<string, string>("ReceiveMessage", (user, message) => ...

await builder.StartAsync();
```

Configure stateful reconnect

SignalR stateful reconnect reduces the perceived downtime of clients that have a temporary disconnect in their network connection, such as when switching network connections or a short temporary loss in access.

Stateful reconnect achieves this by:

- Temporarily buffering data on the server and client.
- Acknowledging messages received (ACK-ing) by both the server and client.
- Recognizing when a connection is up and replaying messages that might have been sent while the connection was down.

Stateful reconnect is available in ASP.NET Core 8.0 and later.

Opt in to stateful reconnect at both the server hub endpoint and the client:

- Update the server hub endpoint configuration to enable the `AllowStatefulReconnects` option:

  C#

  ```
  app.MapHub<MyHub>("/hubName", options =>
  {
      options.AllowStatefulReconnects = true;
  });
  ```

  Optionally, the maximum buffer size in bytes allowed by the server can be set globally or for a specific hub with the `StatefulReconnectBufferSize` option:

  The `StatefulReconnectBufferSize` option set globally:

  C#

  ```
  builder.AddSignalR(o => o.StatefulReconnectBufferSize = 1000);
  ```

  The `StatefulReconnectBufferSize` option set for a specific hub:

  C#

  ```
  builder.AddSignalR().AddHubOptions<MyHub>(o =>
  o.StatefulReconnectBufferSize = 1000);
  ```

  The `StatefulReconnectBufferSize` option is optional with a default of 100,000 bytes.

- Update JavaScript or TypeScript client code to enable the `withStatefulReconnect` option:

  JavaScript

  ```
  const builder = new signalR.HubConnectionBuilder()
    .withUrl("/hubname")
    .withStatefulReconnect({ bufferSize: 1000 });  // Optional, defaults
  to 100,000
  const connection = builder.build();
  ```

  The `bufferSize` option is optional with a default of 100,000 bytes.

- Update .NET client code to enable the `WithStatefulReconnect` option:

  C#

  ```
    var builder = new HubConnectionBuilder()
        .WithUrl("<hub url>")
        .WithStatefulReconnect();
    builder.Services.Configure<HubConnectionOptions>(o =>
  o.StatefulReconnectBufferSize = 1000);
    var hubConnection = builder.Build();
  ```

  The `StatefulReconnectBufferSize` option is optional with a default of 100,000 bytes.

Configure additional options

Additional options can be configured in the `WithUrl` (`withUrl` in JavaScript) method on `HubConnectionBuilder` or on the various configuration APIs on the `HttpHubConnectionBuilder` in the Java client:

.NET

Expand table

.NET Option	Default value	Description
`AccessTokenProvider`	`null`	A function returning a string that is provided as a Bearer authentication token in HTTP requests.

.NET Option	Default value	Description
`SkipNegotiation`	`false`	Set this to `true` to skip the negotiation step. **Only supported when the WebSockets transport is the only enabled transport**. This setting can't be enabled when using the Azure SignalR Service.
`ClientCertificates`	Empty	A collection of TLS certificates to send to authenticate requests.
`Cookies`	Empty	A collection of HTTP cookies to send with every HTTP request.
`Credentials`	Empty	Credentials to send with every HTTP request.
`CloseTimeout`	5 seconds	WebSockets only. The maximum amount of time the client waits after closing for the server to acknowledge the close request. If the server doesn't acknowledge the close within this time, the client disconnects.
`Headers`	Empty	A Map of additional HTTP headers to send with every HTTP request.
`HttpMessageHandlerFactory`	`null`	A delegate that can be used to configure or replace the `HttpMessageHandler` used to send HTTP requests. Not used for WebSocket connections. This delegate must return a non-null value, and it receives the default value as a parameter. Either modify settings on that default value and return it, or return a new `HttpMessageHandler` instance. **When replacing the handler make sure to copy the settings you want to keep from the provided handler, otherwise, the configured options (such as Cookies and Headers) won't apply to the new handler.**
`Proxy`	`null`	An HTTP proxy to use when sending HTTP requests.
`UseDefaultCredentials`	`false`	Set this boolean to send the default credentials for HTTP and WebSockets requests. This enables the use of Windows authentication.
`WebSocketConfiguration`	`null`	A delegate that can be used to configure additional WebSocket options. Receives an instance of ClientWebSocketOptions that can be used to configure the options.
`ApplicationMaxBufferSize`	1 MB	The maximum number of bytes received from the server that the client buffers before applying

.NET Option	Default value	Description
		backpressure. Increasing this value allows the client to receive larger messages faster without applying backpressure, but can increase memory consumption.
`TransportMaxBufferSize`	1 MB	The maximum number of bytes sent by the user application that the client buffers before observing backpressure. Increasing this value allows the client to buffer larger messages faster without awaiting backpressure, but can increase memory consumption.

In the .NET Client, these options can be modified by the options delegate provided to `WithUrl`:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options => {
        options.Headers["Foo"] = "Bar";
        options.SkipNegotiation = true;
        options.Transports = HttpTransportType.WebSockets;
        options.Cookies.Add(new Cookie(/* ... */);
        options.ClientCertificates.Add(/* ... */);
    })
    .Build();
```

In the JavaScript Client, these options can be provided in a JavaScript object provided to `withUrl`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub", {
        // "Foo: Bar" will not be sent with WebSockets or Server-Sent Events
requests
        headers: { "Foo": "Bar" },
        transport: signalR.HttpTransportType.LongPolling
    })
    .build();
```

In the Java client, these options can be configured with the methods on the `HttpHubConnectionBuilder` returned from the `HubConnectionBuilder.create("HUB URL")`

Java

```
HubConnection hubConnection =
HubConnectionBuilder.create("https://example.com/chathub")
        .withHeader("Foo", "Bar")
        .shouldSkipNegotiate(true)
        .withHandshakeResponseTimeout(30*1000)
        .build();
```

Additional resources

- Get started with ASP.NET Core SignalR
- Use hubs in ASP.NET Core SignalR
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client
- Use MessagePack Hub Protocol in SignalR for ASP.NET Core
- ASP.NET Core SignalR supported platforms

Authentication and authorization in ASP.NET Core SignalR

Article • 10/07/2023

Authenticate users connecting to a SignalR hub

SignalR can be used with ASP.NET Core authentication to associate a user with each connection. In a hub, authentication data can be accessed from the HubConnectionContext.User property. Authentication allows the hub to call methods on all connections associated with a user. For more information, see Manage users and groups in SignalR. Multiple connections may be associated with a single user.

The following code is an example that uses SignalR and ASP.NET Core authentication:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using SignalRAuthenticationSample.Data;
using SignalRAuthenticationSample.Hubs;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
```

```
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chat");

app.Run();
```

> **Note**
>
> If a token expires during the lifetime of a connection, by default the connection continues to work. `LongPolling` and `ServerSentEvent` connections fail on subsequent requests if they don't send new access tokens. For connections to close when the authentication token expires, set CloseOnAuthenticationExpiration.

Cookie authentication

In a browser-based app, cookie authentication allows existing user credentials to automatically flow to SignalR connections. When using the browser client, no extra configuration is needed. If the user is logged in to an app, the SignalR connection automatically inherits this authentication.

Cookies are a browser-specific way to send access tokens, but non-browser clients can send them. When using the .NET Client, the `Cookies` property can be configured in the `.WithUrl` call to provide a cookie. However, using cookie authentication from the .NET client requires the app to provide an API to exchange authentication data for a cookie.

Bearer token authentication

The client can provide an access token instead of using a cookie. The server validates the token and uses it to identify the user. This validation is done only when the connection is established. During the life of the connection, the server doesn't automatically revalidate to check for token revocation.

In the JavaScript client, the token can be provided using the accessTokenFactory option.

TypeScript

```
// Connect, using the token we got.
this.connection = new signalR.HubConnectionBuilder()
    .withUrl("/hubs/chat", { accessTokenFactory: () => this.loginToken })
    .build();
```

In the .NET client, there's a similar AccessTokenProvider property that can be used to configure the token:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options =>
    {
        options.AccessTokenProvider = () => Task.FromResult(_myAccessToken);
    })
    .Build();
```

> **Note**
>
> The access token function provided is called before **every** HTTP request made by SignalR. If the token needs to be renewed in order to keep the connection active, do so from within this function and return the updated token. The token may need to be renewed so it doesn't expire during the connection.

In standard web APIs, bearer tokens are sent in an HTTP header. However, SignalR is unable to set these headers in browsers when using some transports. When using WebSockets and Server-Sent Events, the token is transmitted as a query string parameter.

Built-in JWT authentication

On the server, bearer token authentication is configured using the JWT Bearer middleware:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.SignalR;
using Microsoft.EntityFrameworkCore;
using SignalRAuthenticationSample.Data;
using SignalRAuthenticationSample.Hubs;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using SignalRAuthenticationSample;
```

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddAuthentication(options =>
{
    // Identity made Cookie authentication the default.
    // However, we want JWT Bearer Auth to be the default.
    options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
```

```
}).AddJwtBearer(options =>
  {
      // Configure the Authority to the expected value for
      // the authentication provider. This ensures the token
      // is appropriately validated.
      options.Authority = "Authority URL"; // TODO: Update URL

      // We have to hook the OnMessageReceived event in order to
      // allow the JWT authentication handler to read the access
      // token from the query string when a WebSocket or
      // Server-Sent Events request comes in.

      // Sending the access token in the query string is required when using
WebSockets or ServerSentEvents
      // due to a limitation in Browser APIs. We restrict it to only calls
to the
      // SignalR hub in this code.
      // See https://docs.microsoft.com/aspnet/core/signalr/security#access-
token-logging
      // for more information about security considerations when using
      // the query string to transmit the access token.
      options.Events = new JwtBearerEvents
      {
          OnMessageReceived = context =>
          {
              var accessToken = context.Request.Query["access_token"];

              // If the request is for our hub...
              var path = context.HttpContext.Request.Path;
              if (!string.IsNullOrEmpty(accessToken) &&
                  (path.StartsWithSegments("/hubs/chat")))
              {
                  // Read the token out of the query string
                  context.Token = accessToken;
              }
              return Task.CompletedTask;
          }
      };
  });
```

```
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

// Change to use Name as the user identifier for SignalR
// WARNING: This requires that the source of your JWT token
// ensures that the Name claim is unique!
// If the Name claim isn't unique, users could receive messages
// intended for a different user!
builder.Services.AddSingleton<IUserIdProvider, NameUserIdProvider>();

// Change to use email as the user identifier for SignalR
// builder.Services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>
();

// WARNING: use *either* the NameUserIdProvider *or* the
```

```
// EmailBasedUserIdProvider, but do not use both.

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapHub<ChatHub>("/chatHub");

app.Run();
```

> ⓘ Note
>
> The query string is used on browsers when connecting with WebSockets and Server-Sent Events due to browser API limitations. When using HTTPS, query string values are secured by the TLS connection. However, many servers log query string values. For more information, see Security considerations in ASP.NET Core SignalR. SignalR uses headers to transmit tokens in environments which support them (such as the .NET and Java clients).

Identity Server JWT authentication

When using Duende IdentityServer add a PostConfigureOptions<TOptions> service to the project:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Extensions.Options;
public class ConfigureJwtBearerOptions :
IPostConfigureOptions<JwtBearerOptions>
{
```

```
    public void PostConfigure(string name, JwtBearerOptions options)
    {
        var originalOnMessageReceived = options.Events.OnMessageReceived;
        options.Events.OnMessageReceived = async context =>
        {
            await originalOnMessageReceived(context);

            if (string.IsNullOrEmpty(context.Token))
            {
                var accessToken = context.Request.Query["access_token"];
                var path = context.HttpContext.Request.Path;

                if (!string.IsNullOrEmpty(accessToken) &&
                    path.StartsWithSegments("/hubs"))
                {
                    context.Token = accessToken;
                }
            }
        };
    }
}
```

Register the service after adding services for authentication (AddAuthentication) and the authentication handler for Identity Server (AddIdentityServerJwt):

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Extensions.DependencyInjection.Extensions;
using SignalRAuthenticationSample.Hubs;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication()
    .AddIdentityServerJwt();
builder.Services.TryAddEnumerable(
    ServiceDescriptor.Singleton<IPostConfigureOptions<JwtBearerOptions>,
        ConfigureJwtBearerOptions>());

builder.Services.AddRazorPages();

var app = builder.Build();

// Code removed for brevity.
```

Cookies vs. bearer tokens

Cookies are specific to browsers. Sending them from other kinds of clients adds complexity compared to sending bearer tokens. Cookie authentication isn't recommended unless the app only needs to authenticate users from the browser client.

Bearer token authentication is the recommended approach when using clients other than the browser client.

Windows authentication

If Windows authentication is configured in the app, SignalR can use that identity to secure hubs. However, to send messages to individual users, add a custom User ID provider. The Windows authentication system doesn't provide the "Name Identifier" claim. SignalR uses the claim to determine the user name.

Add a new class that implements `IUserIdProvider` and retrieve one of the claims from the user to use as the identifier. For example, to use the "Name" claim (which is the Windows username in the form `[Domain]/[Username]`), create the following class:

C#

```
public class NameUserIdProvider : IUserIdProvider
{
    public string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.Identity?.Name;
    }
}
```

Rather than `ClaimTypes.Name`, use any value from the `User`, such as the Windows SID identifier, etc.

> **Note**
>
> The value chosen must be unique among all the users in the system. Otherwise, a message intended for one user could end up going to a different user.

Register this component in `Program.cs`:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;
using Microsoft.AspNetCore.SignalR;
using SignalRAuthenticationSample;

var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;

services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
   .AddNegotiate();
```

```
services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
});
services.AddRazorPages();

services.AddSignalR();
services.AddSingleton<IUserIdProvider, NameUserIdProvider>();

var app = builder.Build();

// Code removed for brevity.
```

In the .NET Client, Windows Authentication must be enabled by setting the UseDefaultCredentials property:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/chathub", options =>
    {
        options.UseDefaultCredentials = true;
    })
    .Build();
```

Windows authentication is supported in Microsoft Edge, but not in all browsers. For example, in Chrome and Safari, attempting to use Windows authentication and WebSockets fails. When Windows authentication fails, the client attempts to fall back to other transports which might work.

Use claims to customize identity handling

An app that authenticates users can derive SignalR user IDs from user claims. To specify how SignalR creates user IDs, implement `IUserIdProvider` and register the implementation.

The sample code demonstrates how to use claims to select the user's email address as the identifying property.

> **Note**
>
> The value chosen must be unique among all the users in the system. Otherwise, a message intended for one user could end up going to a different user.

C#

```
public class EmailBasedUserIdProvider : IUserIdProvider
{
    public virtual string GetUserId(HubConnectionContext connection)
    {
        return connection.User?.FindFirst(ClaimTypes.Email)?.Value!;
    }
}
```

The account registration adds a claim with type `ClaimsTypes.Email` to the ASP.NET identity database.

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");
    ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync())

.ToList();
    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.Email,
CancellationToken.None);
        await _emailStore.SetEmailAsync(user, Input.Email,
CancellationToken.None);
        var result = await _userManager.CreateAsync(user, Input.Password);

        // Add the email claim and value for this user.
        await _userManager.AddClaimAsync(user, new Claim(ClaimTypes.Email,
Input.Email));

        // Remaining code removed for brevity.
```

Register this component in `Program.cs`:

C#

```
builder.Services.AddSingleton<IUserIdProvider, EmailBasedUserIdProvider>();
```

Authorize users to access hubs and hub methods

By default, all methods in a hub can be called by an unauthenticated user. To require authentication, apply the AuthorizeAttribute attribute to the hub:

C#

```
[Authorize]
public class ChatHub: Hub
{
}
```

The constructor arguments and properties of the `[Authorize]` attribute can be used to restrict access to only users matching specific authorization policies. For example, with the custom authorization policy called `MyAuthorizationPolicy`, only users matching that policy can access the hub using the following code:

C#

```
[Authorize("MyAuthorizationPolicy")]
public class ChatPolicyHub : Hub
{
    public override async Task OnConnectedAsync()
    {
        await Clients.All.SendAsync("ReceiveSystemMessage",
                                    $"{Context.UserIdentifier} joined.");
        await base.OnConnectedAsync();
    }
    // Code removed for brevity.
```

The `[Authorize]` attribute can be applied to individual hub methods. If the current user doesn't match the policy applied to the method, an error is returned to the caller:

C#

```
[Authorize]
public class ChatHub : Hub
{
    public async Task Send(string message)
    {
        // ... send a message to all users ...
    }

    [Authorize("Administrators")]
    public void BanUser(string userName)
    {
        // ... ban a user from the chat room (something only Administrators
can do) ...
    }
}
```

Use authorization handlers to customize hub method authorization

SignalR provides a custom resource to authorization handlers when a hub method requires authorization. The resource is an instance of HubInvocationContext. The `HubInvocationContext` includes the HubCallerContext, the name of the hub method being invoked, and the arguments to the hub method.

Consider the example of a chat room allowing multiple organization sign-in via Microsoft Entra ID. Anyone with a Microsoft account can sign in to chat, but only members of the owning organization should be able to ban users or view users' chat histories. Furthermore, we might want to restrict some functionality from specific users. Note how the `DomainRestrictedRequirement` serves as a custom IAuthorizationRequirement. Now that the `HubInvocationContext` resource parameter is being passed in, the internal logic can inspect the context in which the Hub is being called and make decisions on allowing the user to execute individual Hub methods:

C#

```
[Authorize]
public class ChatHub : Hub
{
    public void SendMessage(string message)
    {
    }

    [Authorize("DomainRestricted")]
    public void BanUser(string username)
    {
    }

    [Authorize("DomainRestricted")]
    public void ViewUserHistory(string username)
    {
    }
}
```

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.SignalR;

namespace SignalRAuthenticationSample;

public class DomainRestrictedRequirement :
    AuthorizationHandler<DomainRestrictedRequirement, HubInvocationContext>,
```

```
    IAuthorizationRequirement
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
        DomainRestrictedRequirement requirement,
        HubInvocationContext resource)
    {
        if (context.User.Identity != null &&
          !string.IsNullOrEmpty(context.User.Identity.Name) &&
          IsUserAllowedToDoThis(resource.HubMethodName,
                                context.User.Identity.Name) &&
          context.User.Identity.Name.EndsWith("@microsoft.com"))
        {
                context.Succeed(requirement);

        }
        return Task.CompletedTask;
    }

    private bool IsUserAllowedToDoThis(string hubMethodName,
        string currentUsername)
    {
        return !(currentUsername.Equals("asdf42@microsoft.com") &&
            hubMethodName.Equals("banUser",
StringComparison.OrdinalIgnoreCase));
    }
}
```

In `Program.cs`, add the new policy, providing the custom `DomainRestrictedRequirement` requirement as a parameter to create the `DomainRestricted` policy:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using SignalRAuthenticationSample;
using SignalRAuthenticationSample.Data;
using SignalRAuthenticationSample.Hubs;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
var services = builder.Services;

services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
services.AddDatabaseDeveloperPageExceptionFilter();

services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

```
services.AddAuthorization(options =>
    {
        options.AddPolicy("DomainRestricted", policy =>
        {
            policy.Requirements.Add(new DomainRestrictedRequirement());
        });
    });

services.AddRazorPages();

var app = builder.Build();

// Code removed for brevity.
```

In the preceding example, the `DomainRestrictedRequirement` class is both an `IAuthorizationRequirement` and its own `AuthorizationHandler` for that requirement. It's acceptable to split these two components into separate classes to separate concerns. A benefit of the example's approach is there's no need to inject the `AuthorizationHandler` during startup, as the requirement and the handler are the same thing.

Additional resources

- Bearer Token Authentication in ASP.NET Core
- Resource-based Authorization
- View or download sample code (how to download)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Security considerations in ASP.NET Core SignalR

Article • 02/20/2024

By Andrew Stanton-Nurse

This article provides information on securing SignalR.

Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) can be used to allow cross-origin SignalR connections in the browser. If JavaScript code is hosted on a different domain from the SignalR app, CORS middleware must be enabled to allow the JavaScript to connect to the SignalR app. Allow cross-origin requests only from domains you trust or control. For example:

- Your site is hosted on `http://www.example.com`
- Your SignalR app is hosted on `http://signalr.example.com`

CORS should be configured in the SignalR app to only allow the origin `www.example.com`.

For more information on configuring CORS, see Enable Cross-Origin Requests (CORS). SignalR **requires** the following CORS policies:

- Allow the specific expected origins. Allowing any origin is possible but is **not** secure or recommended.
- HTTP methods `GET` and `POST` must be allowed.
- Credentials must be allowed in order for cookie-based sticky sessions to work correctly. They must be enabled even when authentication isn't used.

However, in 5.0 we have provided an option in the TypeScript client to not use credentials. The option to not use credentials should only be used when you know 100% that credentials like Cookies are not needed in your app (cookies are used by azure app service when using multiple servers for sticky sessions).

For example, the following highlighted CORS policy allows a SignalR browser client hosted on `https://example.com` to access the SignalR app hosted on `https://signalr.example.com`:

C#

```
using SignalRChat.Hubs;

var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      policy =>
                      {
                          policy.WithOrigins("http://example.com");
                          policy.WithMethods("GET", "POST");
                          policy.AllowCredentials();
                      });
});

// Add services to the container.
builder.Services.AddRazorPages();
builder.Services.AddSignalR();

var app = builder.Build();

app.MapHub<ChatHub>("/chatHub");
```

In the previous example, the CORS policy is customized to allow specific origins, methods, and credentials. For more information on customizing CORS policies and middleware in ASP.NET Core, see CORS middleware: CORS with named policy and middleware.

WebSocket Origin Restriction

The protections provided by CORS don't apply to WebSockets. For origin restriction on WebSockets, read WebSockets origin restriction.

ConnectionId

Exposing `ConnectionId` can lead to malicious impersonation if the SignalR server or client version is ASP.NET Core 2.2 or earlier. If the SignalR server and client version are ASP.NET Core 3.0 or later, the `ConnectionToken` rather than the `ConnectionId` must be kept secret. The `ConnectionToken` is purposely not exposed in any API. It can be difficult to ensure that older SignalR clients aren't connecting to the server, so even if your SignalR server version is ASP.NET Core 3.0 or later, the `ConnectionId` shouldn't be exposed.

Access token logging

When using WebSockets or Server-Sent Events, the browser client sends the access token in the query string. Receiving the access token via query string is generally as secure as using the standard `Authorization` header. Always use HTTPS to ensure a secure end-to-end connection between the client and the server. Many web servers log the URL for each request, including the query string. Logging the URLs may log the access token. ASP.NET Core logs the URL for each request by default, which includes the query string. For example:

Output

```
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:5000/chathub?
access_token=1234
```

If you have concerns about logging this data with your server logs, you can disable this logging entirely by configuring the `Microsoft.AspNetCore.Hosting` logger to the `Warning` level or above (these messages are written at `Info` level). For more information, see Apply log filter rules in code for more information. If you still want to log certain request information, you can write middleware to log the data that you require and filter out the `access_token` query string value (if present).

Exceptions

Exception messages are generally considered sensitive data that shouldn't be revealed to a client. By default, SignalR doesn't send the details of an exception thrown by a hub method to the client. Instead, the client receives a generic message indicating an error occurred. Exception message delivery to the client can be overridden (for example in development or test) with EnableDetailedErrors. Exception messages should not be exposed to the client in production apps.

Buffer management

SignalR uses per-connection buffers to manage incoming and outgoing messages. By default, SignalR limits these buffers to 32 KB. The largest message a client or server can send is 32 KB. The maximum memory consumed by a connection for messages is 32 KB. If your messages are always smaller than 32 KB, you can reduce the limit, which:

- Prevents a client from being able to send a larger message.
- The server will never need to allocate large buffers to accept messages.

If your messages are larger than 32 KB, you can increase the limit. Increasing this limit means:

- The client can cause the server to allocate large memory buffers.
- Server allocation of large buffers may reduce the number of concurrent connections.

There are limits for incoming and outgoing messages, both can be configured on the HttpConnectionDispatcherOptions object configured in `MapHub`:

- `ApplicationMaxBufferSize` represents the maximum number of bytes from the client that the server buffers. If the client attempts to send a message larger than this limit, the connection may be closed.
- `TransportMaxBufferSize` represents the maximum number of bytes the server can send. If the server attempts to send a message (including return values from hub methods) larger than this limit, an exception will be thrown.

Setting the limit to `0` disables the limit. Removing the limit allows a client to send a message of any size. Malicious clients sending large messages can cause excess memory to be allocated. Excess memory usage can significantly reduce the number of concurrent connections.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use MessagePack Hub Protocol in SignalR for ASP.NET Core

Article • 01/08/2025

This article assumes the reader is familiar with the topics covered in Get started with ASP.NET Core SignalR.

What is MessagePack?

MessagePack is a fast and compact binary serialization format. It's useful when performance and bandwidth are a concern because it creates smaller messages than JSON. The binary messages are unreadable when looking at network traces and logs unless the bytes are passed through a MessagePack parser. SignalR has built-in support for the MessagePack format and provides APIs for the client and server to use.

Configure MessagePack on the server

To enable the MessagePack Hub Protocol on the server, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package in your app. In the `Startup.ConfigureServices` method, add `AddMessagePackProtocol` to the `AddSignalR` call to enable MessagePack support on the server.

C#

```
services.AddSignalR()
    .AddMessagePackProtocol();
```

> **Note**
>
> JSON is enabled by default. Adding MessagePack enables support for both JSON and MessagePack clients.

To customize how MessagePack formats data, `AddMessagePackProtocol` takes a delegate for configuring options. In that delegate, the `SerializerOptions` property is used to configure MessagePack serialization options. For more information on how the resolvers work, visit the MessagePack library at MessagePack-CSharp. Attributes can be used on the objects you want to serialize to define how they should be handled.

C#

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        options.SerializerOptions = MessagePackSerializerOptions.Standard
            .WithResolver(new CustomResolver())
            .WithSecurity(MessagePackSecurity.UntrustedData);
    });
```

> ⚠ **Warning**
>
> We strongly recommend reviewing CVE-2020-5234 and applying the recommended patches. For example, calling `.WithSecurity(MessagePackSecurity.UntrustedData)` when replacing the `SerializerOptions`.

Configure MessagePack on the client

> ⓘ **Note**
>
> JSON is enabled by default for the supported clients. Clients can only support a single protocol. Adding MessagePack support replaces any previously configured protocols.

.NET client

To enable MessagePack in the .NET Client, install the `Microsoft.AspNetCore.SignalR.Protocols.MessagePack` package and call `AddMessagePackProtocol` on `HubConnectionBuilder`.

C#

```
using Microsoft.AspNetCore.SignalR.Client;
using Microsoft.Extensions.DependencyInjection;

var hubConnection = new HubConnectionBuilder()
                        .WithUrl("/chathub")
                        .AddMessagePackProtocol()
                        .Build();
```

> **Note**
>
> This `AddMessagePackProtocol` call takes a delegate for configuring options just like the server.

JavaScript client

MessagePack support for the JavaScript client is provided by the @microsoft/signalr-protocol-msgpack npm package. Install the package by executing the following command in a command shell:

Bash

```
npm install @microsoft/signalr-protocol-msgpack
```

After installing the npm package, the module can be used directly via a JavaScript module loader or imported into the browser by referencing the following file:

node_modules\@microsoft\signalr-protocol-msgpack\dist\browser\signalr-protocol-msgpack.js

The following required javaScript files must be referenced in the order shown below:

HTML

```
<script src="~/lib/signalr/signalr.js"></script>
<script src="~/lib/signalr/signalr-protocol-msgpack.js"></script>
```

Adding `.withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())` to the `HubConnectionBuilder` configures the client to use the MessagePack protocol when connecting to a server.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withHubProtocol(new signalR.protocols.msgpack.MessagePackHubProtocol())
    .build();
```

At this time, there are no configuration options for the MessagePack protocol on the JavaScript client.

Java client

To enable MessagePack with Java, install the `com.microsoft.signalr.messagepack` package. When using Gradle, add the following line to the `dependencies` section of the *build.gradle* file:

Gradle

```
implementation 'com.microsoft.signalr.messagepack:signalr-messagepack:5.0.0'
```

When using Maven, add the following lines inside the `<dependencies>` element of the `pom.xml` file:

XML

```
<dependency>
    <groupId>com.microsoft.signalr.messagepack</groupId>
    <artifactId>signalr</artifactId>
    <version>5.0.0</version>
</dependency>
```

Call `withHubProtocol(new MessagePackHubProtocol())` on `HubConnectionBuilder`.

Java

```
HubConnection messagePackConnection = HubConnectionBuilder.create("YOUR HUB
URL HERE")
    .withHubProtocol(new MessagePackHubProtocol())
    .build();
```

MessagePack considerations

There are a few issues to be aware of when using the MessagePack Hub Protocol.

MessagePack is case-sensitive

The MessagePack protocol is case-sensitive. For example, consider the following C# class:

C#

```
public class ChatMessage
{
    public string Sender { get; }
```

```
    public string Message { get; }
}
```

When sending from the JavaScript client, you must use `PascalCased` property names, since the casing must match the C# class exactly. For example:

JavaScript

```
connection.invoke("SomeMethod", { Sender: "Sally", Message: "Hello!" });
```

Using `camelCased` names won't properly bind to the C# class. You can work around this by using the `Key` attribute to specify a different name for the MessagePack property. For more information, see the MessagePack-CSharp documentation.

DateTime.Kind is not preserved when serializing/deserializing

The MessagePack protocol doesn't provide a way to encode the `Kind` value of a `DateTime`. As a result, when deserializing a date, the MessagePack Hub Protocol will convert to the UTC format if the `DateTime.Kind` is `DateTimeKind.Local` otherwise it will not touch the time and pass it as is. If you're working with `DateTime` values, we recommend converting to UTC before sending them. Convert them from UTC to local time when you receive them.

MessagePack support in "ahead-of-time" compilation environment

The MessagePack-CSharp library used by the .NET client and server uses code generation to optimize serialization. As a result, it isn't supported by default on environments that use "ahead-of-time" compilation, such as NET Multi-platform App UI (.NET MAUI) or Unity. It's possible to use MessagePack in these environments by "pre-generating" the serializer/deserializer code. For more information, see the MessagePack-CSharp documentation. Once you have pre-generated the serializers, you can register them using the configuration delegate passed to `AddMessagePackProtocol`:

C#

```
services.AddSignalR()
    .AddMessagePackProtocol(options =>
    {
        StaticCompositeResolver.Instance.Register(
            MessagePack.Resolvers.GeneratedResolver.Instance,
```

```
            MessagePack.Resolvers.StandardResolver.Instance
        );
        options.SerializerOptions = MessagePackSerializerOptions.Standard
            .WithResolver(StaticCompositeResolver.Instance)
            .WithSecurity(MessagePackSecurity.UntrustedData);
    });
```

Type checks are more strict in MessagePack

The JSON Hub Protocol will perform type conversions during deserialization. For example, if the incoming object has a property value that is a number (`{ foo: 42 }`) but the property on the .NET class is of type `string`, the value will be converted. However, MessagePack doesn't perform this conversion and will throw an exception that can be seen in server-side logs (and in the console):

```
InvalidDataException: Error binding arguments. Make sure that the types of
the provided values match the types of the hub method being invoked.
```

For more information on this limitation, see GitHub issue aspnet/SignalR#2937.

Chars and Strings in Java

In the java client, `char` objects will be serialized as one-character `String` objects. This is in contrast with the C# and JavaScript client, which serialize them as `short` objects. The MessagePack spec itself does not define behavior for `char` objects, so it is up to the library author to determine how to serialize them. The difference in behavior between our clients is a result of the libraries we used for our implementations.

Additional resources

- ASP.NET Core SignalR .NET Client
- ASP.NET Core SignalR JavaScript client

Use streaming in ASP.NET Core SignalR

Article • 07/05/2022

By Brennan Conroy

ASP.NET Core SignalR supports streaming from client to server and from server to client. This is useful for scenarios where fragments of data arrive over time. When streaming, each fragment is sent to the client or server as soon as it becomes available, rather than waiting for all of the data to become available.

View or download sample code (how to download)

Set up a hub for streaming

A hub method automatically becomes a streaming hub method when it returns IAsyncEnumerable<T>, ChannelReader<T>, `Task<IAsyncEnumerable<T>>`, or `Task<ChannelReader<T>>`.

Server-to-client streaming

Streaming hub methods can return `IAsyncEnumerable<T>` in addition to `ChannelReader<T>`. The simplest way to return `IAsyncEnumerable<T>` is by making the hub method an async iterator method as the following sample demonstrates. Hub async iterator methods can accept a `CancellationToken` parameter that's triggered when the client unsubscribes from the stream. Async iterator methods avoid problems common with Channels, such as not returning the `ChannelReader` early enough or exiting the method without completing the ChannelWriter<T>.

> **Note**
>
> The following sample requires C# 8.0 or later.

C#

```
public class AsyncEnumerableHub : Hub
{
    public async IAsyncEnumerable<int> Counter(
        int count,
        int delay,
        [EnumeratorCancellation]
        CancellationToken cancellationToken)
```

```
    {
        for (var i = 0; i < count; i++)
        {
            // Check the cancellation token regularly so that the server
will stop
            // producing items if the client disconnects.
            cancellationToken.ThrowIfCancellationRequested();

            yield return i;

            // Use the cancellationToken in other APIs that accept
cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
}
```

The following sample shows the basics of streaming data to the client using Channels. Whenever an object is written to the ChannelWriter<T>, the object is immediately sent to the client. At the end, the `ChannelWriter` is completed to tell the client the stream is closed.

> **Note**
>
> Write to the `ChannelWriter<T>` on a background thread and return the `ChannelReader` as soon as possible. Other hub invocations are blocked until a `ChannelReader` is returned.
>
> Wrap logic in a try ... catch statement. Complete the `Channel` in a finally block. If you want to flow an error, capture it inside the `catch` block and write it in the `finally` block.

C#

```
public ChannelReader<int> Counter(
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    var channel = Channel.CreateUnbounded<int>();

    // We don't want to await WriteItemsAsync, otherwise we'd end up waiting
    // for all the items to be written before returning the channel back to
    // the client.
    _ = WriteItemsAsync(channel.Writer, count, delay, cancellationToken);

    return channel.Reader;
```

```
}

private async Task WriteItemsAsync(
    ChannelWriter<int> writer,
    int count,
    int delay,
    CancellationToken cancellationToken)
{
    Exception localException = null;
    try
    {
        for (var i = 0; i < count; i++)
        {
            await writer.WriteAsync(i, cancellationToken);

            // Use the cancellationToken in other APIs that accept
cancellation
            // tokens so the cancellation can flow down to them.
            await Task.Delay(delay, cancellationToken);
        }
    }
    catch (Exception ex)
    {
        localException = ex;
    }
    finally
    {
        writer.Complete(localException);
    }
}
```

Server-to-client streaming hub methods can accept a `CancellationToken` parameter that's triggered when the client unsubscribes from the stream. Use this token to stop the server operation and release any resources if the client disconnects before the end of the stream.

Client-to-server streaming

A hub method automatically becomes a client-to-server streaming hub method when it accepts one or more objects of type ChannelReader<T> or IAsyncEnumerable<T>. The following sample shows the basics of reading streaming data sent from the client. Whenever the client writes to the ChannelWriter<T>, the data is written into the `ChannelReader` on the server from which the hub method is reading.

C#

```
public async Task UploadStream(ChannelReader<string> stream)
{
    while (await stream.WaitToReadAsync())
```

```
    {
        while (stream.TryRead(out var item))
        {
            // do something with the stream item
            Console.WriteLine(item);
        }
    }
}
```

An IAsyncEnumerable<T> version of the method follows.

> ⓘ Note
>
> The following sample requires C# 8.0 or later.

C#

```
public async Task UploadStream(IAsyncEnumerable<string> stream)
{
    await foreach (var item in stream)
    {
        Console.WriteLine(item);
    }
}
```

.NET client

Server-to-client streaming

The `StreamAsync` and `StreamAsChannelAsync` methods on `HubConnection` are used to invoke server-to-client streaming methods. Pass the hub method name and arguments defined in the hub method to `StreamAsync` or `StreamAsChannelAsync`. The generic parameter on `StreamAsync<T>` and `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method. An object of type `IAsyncEnumerable<T>` or `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

A `StreamAsync` example that returns `IAsyncEnumerable<int>`:

C#

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation
message to
// the server, which will trigger the corresponding token in the hub method.
```

```
var cancellationTokenSource = new CancellationTokenSource();
var stream = hubConnection.StreamAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

await foreach (var count in stream)
{
    Console.WriteLine($"{count}");
}

Console.WriteLine("Streaming completed");
```

A corresponding `StreamAsChannelAsync` example that returns `ChannelReader<int>`:

C#

```
// Call "Cancel" on this CancellationTokenSource to send a cancellation
message to
// the server, which will trigger the corresponding token in the hub method.
var cancellationTokenSource = new CancellationTokenSource();
var channel = await hubConnection.StreamAsChannelAsync<int>(
    "Counter", 10, 500, cancellationTokenSource.Token);

// Wait asynchronously for data to become available
while (await channel.WaitToReadAsync())
{
    // Read all currently available data synchronously, before waiting for
more data
    while (channel.TryRead(out var count))
    {
        Console.WriteLine($"{count}");
    }
}

Console.WriteLine("Streaming completed");
```

In the previous code:

- The `StreamAsChannelAsync` method on `HubConnection` is used to invoke a server-to-client streaming method. Pass the hub method name and arguments defined in the hub method to `StreamAsChannelAsync`.
- The generic parameter on `StreamAsChannelAsync<T>` specifies the type of objects returned by the streaming method.
- A `ChannelReader<T>` is returned from the stream invocation and represents the stream on the client.

Client-to-server streaming

There are two ways to invoke a client-to-server streaming hub method from the .NET client. You can either pass in an `IAsyncEnumerable<T>` or a `ChannelReader` as an argument to `SendAsync`, `InvokeAsync`, or `StreamAsChannelAsync`, depending on the hub method invoked.

Whenever data is written to the `IAsyncEnumerable` or `ChannelWriter` object, the hub method on the server receives a new item with the data from the client.

If using an `IAsyncEnumerable` object, the stream ends after the method returning stream items exits.

> ⓘ **Note**
>
> The following sample requires C# 8.0 or later.

C#

```
async IAsyncEnumerable<string> clientStreamData()
{
    for (var i = 0; i < 5; i++)
    {
        var data = await FetchSomeData();
        yield return data;
    }
    //After the for loop has completed and the local function exits the
stream completion will be sent.
}

await connection.SendAsync("UploadStream", clientStreamData());
```

Or if you're using a `ChannelWriter`, you complete the channel with `channel.Writer.Complete()`:

C#

```
var channel = Channel.CreateBounded<string>(10);
await connection.SendAsync("UploadStream", channel.Reader);
await channel.Writer.WriteAsync("some data");
await channel.Writer.WriteAsync("some more data");
channel.Writer.Complete();
```

JavaScript client

Server-to-client streaming

JavaScript clients call server-to-client streaming methods on hubs with `connection.stream`. The `stream` method accepts two arguments:

- The name of the hub method. In the following example, the hub method name is `Counter`.
- Arguments defined in the hub method. In the following example, the arguments are a count for the number of stream items to receive and the delay between stream items.

`connection.stream` returns an `IStreamResult`, which contains a `subscribe` method. Pass an `IStreamSubscriber` to `subscribe` and set the `next`, `error`, and `complete` callbacks to receive notifications from the `stream` invocation.

JavaScript

```
connection.stream("Counter", 10, 500)
    .subscribe({
        next: (item) => {
            var li = document.createElement("li");
            li.textContent = item;
            document.getElementById("messagesList").appendChild(li);
        },
        complete: () => {
            var li = document.createElement("li");
            li.textContent = "Stream completed";
            document.getElementById("messagesList").appendChild(li);
        },
        error: (err) => {
            var li = document.createElement("li");
            li.textContent = err;
            document.getElementById("messagesList").appendChild(li);
        },
});
```

To end the stream from the client, call the `dispose` method on the `ISubscription` that's returned from the `subscribe` method. Calling this method causes cancellation of the `CancellationToken` parameter of the Hub method, if you provided one.

Client-to-server streaming

JavaScript clients call client-to-server streaming methods on hubs by passing in a `Subject` as an argument to `send`, `invoke`, or `stream`, depending on the hub method

invoked. The `Subject` is a class that looks like a `Subject`. For example in RxJS, you can use the Subject class from that library.

JavaScript

```
const subject = new signalR.Subject();
yield connection.send("UploadStream", subject);
var iteration = 0;
const intervalHandle = setInterval(() => {
    iteration++;
    subject.next(iteration.toString());
    if (iteration === 10) {
        clearInterval(intervalHandle);
        subject.complete();
    }
}, 500);
```

Calling `subject.next(item)` with an item writes the item to the stream, and the hub method receives the item on the server.

To end the stream, call `subject.complete()`.

Java client

Server-to-client streaming

The SignalR Java client uses the `stream` method to invoke streaming methods. `stream` accepts three or more arguments:

- The expected type of the stream items.
- The name of the hub method.
- Arguments defined in the hub method.

Java

```
hubConnection.stream(String.class, "ExampleStreamingHubMethod", "Arg1")
    .subscribe(
        (item) -> {/* Define your onNext handler here. */ },
        (error) -> {/* Define your onError handler here. */},
        () -> {/* Define your onCompleted handler here. */});
```

The `stream` method on `HubConnection` returns an Observable of the stream item type. The Observable type's `subscribe` method is where `onNext`, `onError` and `onCompleted` handlers are defined.

Client-to-server streaming

The SignalR Java client can call client-to-server streaming methods on hubs by passing in an Observable as an argument to `send`, `invoke`, or `stream`, depending on the hub method invoked.

Java

```
ReplaySubject<String> stream = ReplaySubject.create();
hubConnection.send("UploadStream", stream);
stream.onNext("FirstItem");
stream.onNext("SecondItem");
stream.onComplete();
```

Calling `stream.onNext(item)` with an item writes the item to the stream, and the hub method receives the item on the server.

To end the stream, call `stream.onComplete()`.

Additional resources

- Hubs
- .NET client
- JavaScript client
- Publish to Azure

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Differences between ASP.NET SignalR and ASP.NET Core SignalR

Article • 06/03/2022

ASP.NET Core SignalR isn't compatible with clients or servers for ASP.NET SignalR. This article details features which have been removed or changed in ASP.NET Core SignalR.

How to identify the SignalR version

Expand table

	ASP.NET SignalR	ASP.NET Core SignalR
Server NuGet package	Microsoft.AspNet.SignalR	None. Included in the Microsoft.AspNetCore.App shared framework.
Client NuGet packages	Microsoft.AspNet.SignalR.Client Microsoft.AspNet.SignalR.JS	Microsoft.AspNetCore.SignalR.Client
JavaScript client npm package	signalr	@microsoft/signalr
Java client	GitHub Repository (deprecated)	Maven package com.microsoft.signalr
Server app type	ASP.NET (System.Web) or OWIN Self-Host	ASP.NET Core
Supported server platforms	.NET Framework 4.5 or later	.NET Core 3.0 or later

Feature differences

Automatic reconnects

In ASP.NET SignalR:

- By default, SignalR attempts to reconnect to the server if the connection is dropped.

In ASP.NET Core SignalR:

- Automatic reconnects are opt-in with both the .NET client and the JavaScript client:

C#

```
HubConnection connection = new HubConnectionBuilder()
    .WithUrl(new Uri("http://127.0.0.1:5000/chathub"))
    .WithAutomaticReconnect()
    .Build();
```

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/chathub")
    .withAutomaticReconnect()
    .build();
```

Protocol support

ASP.NET Core SignalR supports JSON, as well as a new binary protocol based on MessagePack. Additionally, custom protocols can be created.

Transports

The Forever Frame transport isn't supported in ASP.NET Core SignalR.

Differences on the server

The ASP.NET Core SignalR server-side libraries are included in Microsoft.AspNetCore.App, which is used in the **ASP.NET Core Web Application** template for both Razor and MVC projects.

ASP.NET Core SignalR is an ASP.NET Core middleware. It must be configured by calling AddSignalR in `Startup.ConfigureServices`.

C#

```
services.AddSignalR()
```

To configure routing, map routes to hubs inside the UseEndpoints method call in the `Startup.Configure` method.

C#

```
app.UseRouting();

app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>("/hub");
});
```

Sticky sessions

The scaleout model for ASP.NET SignalR allows clients to reconnect and send messages to any server in the farm. In ASP.NET Core SignalR, the client must interact with the same server for the duration of the connection. For scaleout using Redis, that means sticky sessions are required. For scaleout using Azure SignalR Service, sticky sessions aren't required because the service handles connections to clients.

Single hub per connection

In ASP.NET Core SignalR, the connection model has been simplified. Connections are made directly to a single hub, rather than a single connection being used to share access to multiple hubs.

Streaming

ASP.NET Core SignalR now supports streaming data from the hub to the client.

State

The ability to pass arbitrary state between clients and the hub (often called `HubState`) has been removed, as well as support for progress messages. There is no counterpart of hub proxies at the moment.

PersistentConnection removal

In ASP.NET Core SignalR, the PersistentConnection class has been removed.

GlobalHost

ASP.NET Core has dependency injection (DI) built into the framework. Services can use DI to access the HubContext. The `GlobalHost` object that is used in ASP.NET SignalR to get a `HubContext` doesn't exist in ASP.NET Core SignalR.

HubPipeline

ASP.NET Core SignalR doesn't have support for `HubPipeline` modules.

Differences on the client

TypeScript

The ASP.NET Core SignalR client is written in TypeScript. You can write in JavaScript or TypeScript when using the JavaScript client.

The JavaScript client is hosted at npm

In ASP.NET versions, the JavaScript client was obtained through a NuGet package in Visual Studio. In the ASP.NET Core versions, the @microsoft/signalr npm package contains the JavaScript libraries. This package isn't included in the **ASP.NET Core Web Application** template. Use npm to obtain and install the `@microsoft/signalr` npm package.

Console

```
npm init -y
npm install @microsoft/signalr
```

jQuery

The dependency on jQuery has been removed, however projects can still use jQuery.

Internet Explorer support

ASP.NET Core SignalR doesn't support Microsoft Internet Explorer, whereas ASP.NET SignalR supports Microsoft Internet Explorer 8 or later. For more information, see ASP.NET Core SignalR supported platforms.

JavaScript client method syntax

The JavaScript syntax has changed from the ASP.NET version of SignalR. Rather than using the `$connection` object, create a connection using the HubConnectionBuilder API.

JavaScript

```
const connection = new signalR.HubConnectionBuilder()
    .withUrl("/hub")
    .build();
```

Use the on method to specify client methods that the hub can call.

JavaScript

```
connection.on("ReceiveMessage", (user, message) => {
    const msg = message.replace(/&/g, "&amp;").replace(/</g,
"&lt;").replace(/>/g, "&gt;");
    const encodedMsg = `${user} says ${msg}`;
    console.log(encodedMsg);
});
```

After creating the client method, start the hub connection. Chain a catch method to log or handle errors.

JavaScript

```
connection.start().catch(err => console.error(err));
```

Hub proxies

Hub proxies are no longer automatically generated. Instead, the method name is passed into the invoke API as a string.

.NET and other clients

The Microsoft.AspNetCore.SignalR.Client NuGet package contains the .NET client libraries for ASP.NET Core SignalR.

Use the HubConnectionBuilder to create and build an instance of a connection to a hub.

C#

```
connection = new HubConnectionBuilder()
    .WithUrl("url")
    .Build();
```

Scaleout differences

ASP.NET SignalR supports SQL Server and Redis. ASP.NET Core SignalR supports Azure SignalR Service and Redis.

ASP.NET

- SignalR scaleout with Azure Service Bus
- SignalR scaleout with Redis
- SignalR scaleout with SQL Server

ASP.NET Core

- Azure SignalR Service
- Redis Backplane

Additional resources

- Hubs
- JavaScript client
- .NET client
- Supported platforms

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

WebSockets support in ASP.NET Core

Article • 04/25/2024

This article explains how to get started with WebSockets in ASP.NET Core. WebSocket (RFC 6455) is a protocol that enables two-way persistent communication channels over TCP connections. It's used in apps that benefit from fast, real-time communication, such as chat, dashboard, and game apps.

View or download sample code (how to download, how to run).

Http/2 WebSockets support

Using WebSockets over HTTP/2 takes advantage of new features such as:

- Header compression.
- Multiplexing, which reduces the time and resources needed when making multiple requests to the server.

These supported features are available in Kestrel on all HTTP/2 enabled platforms. The version negotiation is automatic in browsers and Kestrel, so no new APIs are needed.

.NET 7 introduced WebSockets over HTTP/2 support for Kestrel, the SignalR JavaScript client, and SignalR with Blazor WebAssembly.

> **Note**
>
> HTTP/2 WebSockets use CONNECT requests rather than GET, so your own routes and controllers may need updating. For more information, see Add HTTP/2 WebSockets support for existing controllers in this article.
>
> Chrome and Edge have HTTP/2 WebSockets enabled by default, and you can enable it in FireFox on the `about:config` page with the `network.http.spdy.websockets` flag.

WebSockets were originally designed for HTTP/1.1 but have since been adapted to work over HTTP/2. (RFC 8441)

SignalR

ASP.NET Core SignalR is a library that simplifies adding real-time web functionality to apps. It uses WebSockets whenever possible.

For most applications, we recommend SignalR rather than raw WebSockets. SignalR:

- Provides transport fallback for environments where WebSockets isn't available.
- Provides a basic remote procedure call app model.
- Has no significant performance disadvantage compared to using raw WebSockets in most scenarios.

WebSockets over HTTP/2 is supported for:

- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR with Blazor WebAssembly

For some apps, gRPC on .NET provides an alternative to WebSockets.

Prerequisites

- Any OS that supports ASP.NET Core:
  - Windows 7 / Windows Server 2008 or later
  - Linux
  - macOS
- If the app runs on Windows with IIS:
  - Windows 8 / Windows Server 2012 or later
  - IIS 8 / IIS 8 Express
  - WebSockets must be enabled. See the IIS/IIS Express support section.
- If the app runs on HTTP.sys:
  - Windows 8 / Windows Server 2012 or later
- For supported browsers, see Can I use.

Configure the middleware

Add the WebSockets middleware in `Program.cs`:

```
C#

app.UseWebSockets();
```

The following settings can be configured:

- KeepAliveInterval - How frequently to send "ping" frames to the client to ensure proxies keep the connection open. The default is two minutes.
- AllowedOrigins - A list of allowed Origin header values for WebSocket requests. By default, all origins are allowed. For more information, see WebSocket origin restriction in this article.

C#

```
var webSocketOptions = new WebSocketOptions
{
    KeepAliveInterval = TimeSpan.FromMinutes(2)
};

app.UseWebSockets(webSocketOptions);
```

Accept WebSocket requests

Somewhere later in the request life cycle (later in `Program.cs` or in an action method, for example) check if it's a WebSocket request and accept the WebSocket request.

The following example is from later in `Program.cs`:

C#

```
app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            using var webSocket = await
context.WebSockets.AcceptWebSocketAsync();
            await Echo(webSocket);
        }
        else
        {
            context.Response.StatusCode = StatusCodes.Status400BadRequest;
        }
    }
    else
    {
        await next(context);
    }

});
```

A WebSocket request could come in on any URL, but this sample code only accepts requests for `/ws`.

A similar approach can be taken in a controller method:

C#

```
public class WebSocketController : ControllerBase
{
    [Route("/ws")]
    public async Task Get()
    {
        if (HttpContext.WebSockets.IsWebSocketRequest)
        {
            using var webSocket = await
HttpContext.WebSockets.AcceptWebSocketAsync();
            await Echo(webSocket);
        }
        else
        {
            HttpContext.Response.StatusCode =
StatusCodes.Status400BadRequest;
        }
    }
```

When using a WebSocket, you **must** keep the middleware pipeline running for the duration of the connection. If you attempt to send or receive a WebSocket message after the middleware pipeline ends, you may get an exception like the following:

```
System.Net.WebSockets.WebSocketException (0x80004005): The remote party
closed the WebSocket connection without completing the close handshake. --->
System.ObjectDisposedException: Cannot write to the response body, the
response has completed.
Object name: 'HttpResponseStream'.
```

If you're using a background service to write data to a WebSocket, make sure you keep the middleware pipeline running. Do this by using a TaskCompletionSource<TResult>. Pass the `TaskCompletionSource` to your background service and have it call TrySetResult when you finish with the WebSocket. Then `await` the Task property during the request, as shown in the following example:

C#

```
app.Run(async (context) =>
{
    using var webSocket = await context.WebSockets.AcceptWebSocketAsync();
```

```
    var socketFinishedTcs = new TaskCompletionSource<object>();

    BackgroundSocketProcessor.AddSocket(webSocket, socketFinishedTcs);

    await socketFinishedTcs.Task;
});
```

The WebSocket closed exception can also happen when returning too soon from an action method. When accepting a socket in an action method, wait for the code that uses the socket to complete before returning from the action method.

Never use `Task.Wait`, `Task.Result`, or similar blocking calls to wait for the socket to complete, as that can cause serious threading issues. Always use `await`.

Add HTTP/2 WebSockets support for existing controllers

.NET 7 introduced WebSockets over HTTP/2 support for Kestrel, the SignalR JavaScript client, and SignalR with Blazor WebAssembly. HTTP/2 WebSockets use CONNECT requests rather than GET. If you previously used `[HttpGet("/path")]` on your controller action method for Websocket requests, update it to use `[Route("/path")]` instead.

C#

```
public class WebSocketController : ControllerBase
{
    [Route("/ws")]
    public async Task Get()
    {
        if (HttpContext.WebSockets.IsWebSocketRequest)
        {
            using var webSocket = await
HttpContext.WebSockets.AcceptWebSocketAsync();
            await Echo(webSocket);
        }
        else
        {
            HttpContext.Response.StatusCode =
StatusCodes.Status400BadRequest;
        }
    }
```

Compression

> ⚠ **Warning**
>
> Enabling compression over encrypted connections can make an app subject to CRIME/BREACH attacks. If sending sensitive information, avoid enabling compression or use `WebSocketMessageFlags.DisableCompression` when calling `WebSocket.SendAsync`. This applies to both sides of the WebSocket. Note that the WebSockets API in the browser doesn't have configuration for disabling compression per send.

If compression of messages over WebSockets is desired, then the accept code must specify that it allows compression as follows:

```
C#

using (var webSocket = await context.WebSockets.AcceptWebSocketAsync(
    new WebSocketAcceptContext { DangerousEnableCompression = true }))
{

}
```

`WebSocketAcceptContext.ServerMaxWindowBits` and `WebSocketAcceptContext.DisableServerContextTakeover` are advanced options that control how the compression works.

Compression is negotiated between the client and server when first establishing a connection. You can read more about the negotiation in the Compression Extensions for WebSocket RFC.

> ⓘ **Note**
>
> If the compression negotiation isn't accepted by either the server or client, the connection is still established. However, the connection doesn't use compression when sending and receiving messages.

Send and receive messages

The `AcceptWebSocketAsync` method upgrades the TCP connection to a WebSocket connection and provides a WebSocket object. Use the `WebSocket` object to send and receive messages.

The code shown earlier that accepts the WebSocket request passes the `WebSocket` object to an `Echo` method. The code receives a message and immediately sends back the same message. Messages are sent and received in a loop until the client closes the connection:

C#

```
private static async Task Echo(WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    var receiveResult = await webSocket.ReceiveAsync(
        new ArraySegment<byte>(buffer), CancellationToken.None);

    while (!receiveResult.CloseStatus.HasValue)
    {
        await webSocket.SendAsync(
            new ArraySegment<byte>(buffer, 0, receiveResult.Count),
            receiveResult.MessageType,
            receiveResult.EndOfMessage,
            CancellationToken.None);

        receiveResult = await webSocket.ReceiveAsync(
            new ArraySegment<byte>(buffer), CancellationToken.None);
    }

    await webSocket.CloseAsync(
        receiveResult.CloseStatus.Value,
        receiveResult.CloseStatusDescription,
        CancellationToken.None);
}
```

When accepting the WebSocket connection before beginning the loop, the middleware pipeline ends. Upon closing the socket, the pipeline unwinds. That is, the request stops moving forward in the pipeline when the WebSocket is accepted. When the loop is finished and the socket is closed, the request proceeds back up the pipeline.

Handle client disconnects

The server isn't automatically informed when the client disconnects due to loss of connectivity. The server receives a disconnect message only if the client sends it, which can't be done if the internet connection is lost. If you want to take some action when that happens, set a timeout after nothing is received from the client within a certain time window.

If the client isn't always sending messages and you don't want to time out just because the connection goes idle, have the client use a timer to send a ping message every X

seconds. On the server, if a message hasn't arrived within 2*X seconds after the previous one, terminate the connection and report that the client disconnected. Wait for twice the expected time interval to leave extra time for network delays that might hold up the ping message.

WebSocket origin restriction

The protections provided by CORS don't apply to WebSockets. Browsers do **not**:

- Perform CORS pre-flight requests.
- Respect the restrictions specified in `Access-Control` headers when making WebSocket requests.

However, browsers do send the `Origin` header when issuing WebSocket requests. Applications should be configured to validate these headers to ensure that only WebSockets coming from the expected origins are allowed.

If you're hosting your server on "https://server.com" and hosting your client on "https://client.com", add "https://client.com" to the AllowedOrigins list for WebSockets to verify.

C#

```
var webSocketOptions = new WebSocketOptions
{
    KeepAliveInterval = TimeSpan.FromMinutes(2)
};

webSocketOptions.AllowedOrigins.Add("https://client.com");
webSocketOptions.AllowedOrigins.Add("https://www.client.com");

app.UseWebSockets(webSocketOptions);
```

> ⓘ Note
>
> The `Origin` header is controlled by the client and, like the `Referer` header, can be faked. Do **not** use these headers as an authentication mechanism.

IIS/IIS Express support

Windows Server 2012 or later and Windows 8 or later with IIS/IIS Express 8 or later has support for the WebSocket protocol, but not for WebSockets over HTTP/2.

> **Note**
>
> WebSockets are always enabled when using IIS Express.

Enabling WebSockets on IIS

To enable support for the WebSocket protocol on Windows Server 2012 or later:

> **Note**
>
> These steps are not required when using IIS Express

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**.
2. Select **Role-based or Feature-based Installation**. Select **Next**.
3. Select the appropriate server (the local server is selected by default). Select **Next**.
4. Expand **Web Server (IIS)** in the **Roles** tree, expand **Web Server**, and then expand **Application Development**.
5. Select **WebSocket Protocol**. Select **Next**.
6. If additional features aren't needed, select **Next**.
7. Select **Install**.
8. When the installation completes, select **Close** to exit the wizard.

To enable support for the WebSocket protocol on Windows 8 or later:

> **Note**
>
> These steps are not required when using IIS Express

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the following nodes: **Internet Information Services** > **World Wide Web Services** > **Application Development Features**.
3. Select the **WebSocket Protocol** feature. Select **OK**.

Disable WebSocket when using socket.io on Node.js

If using the WebSocket support in socket.io on Node.js, disable the default IIS WebSocket module using the `webSocket` element in *web.config* or *applicationHost.config*.

If this step isn't performed, the IIS WebSocket module attempts to handle the WebSocket communication rather than Node.js and the app.

XML

```
<system.webServer>
  <webSocket enabled="false" />
</system.webServer>
```

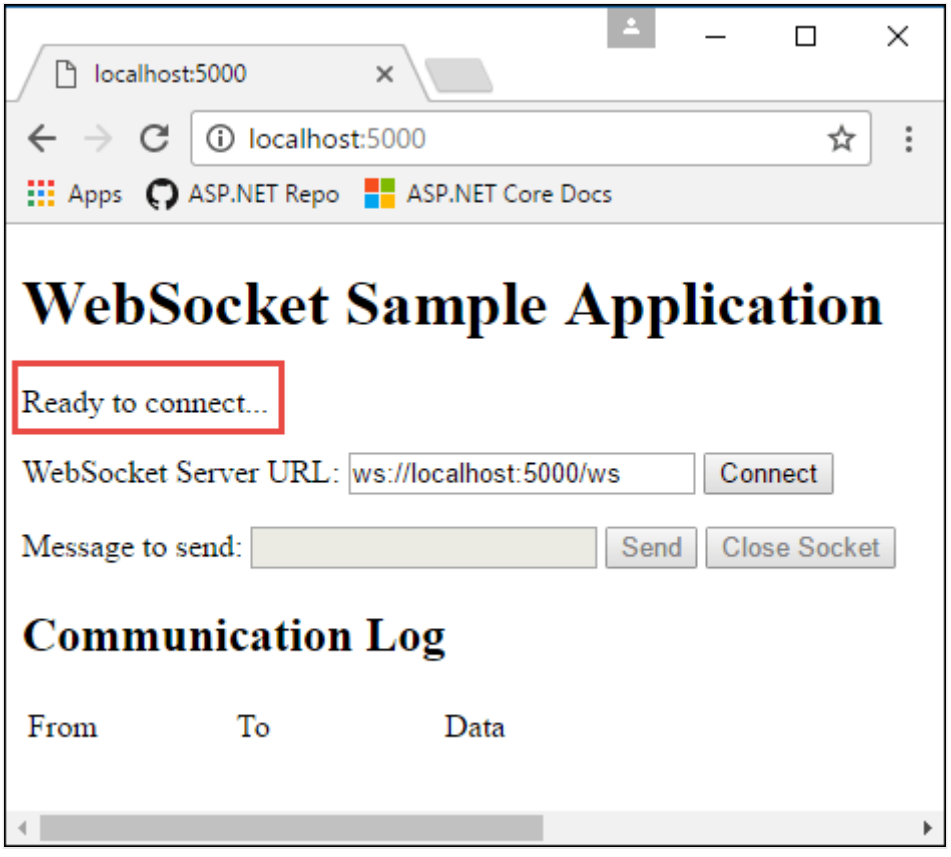
Sample app

The sample app that accompanies this article is an echo app. It has a webpage that makes WebSocket connections, and the server resends any messages it receives back to the client. The sample app supports WebSockets over HTTP/2 when using a targeted framework of .NET 7 or later.

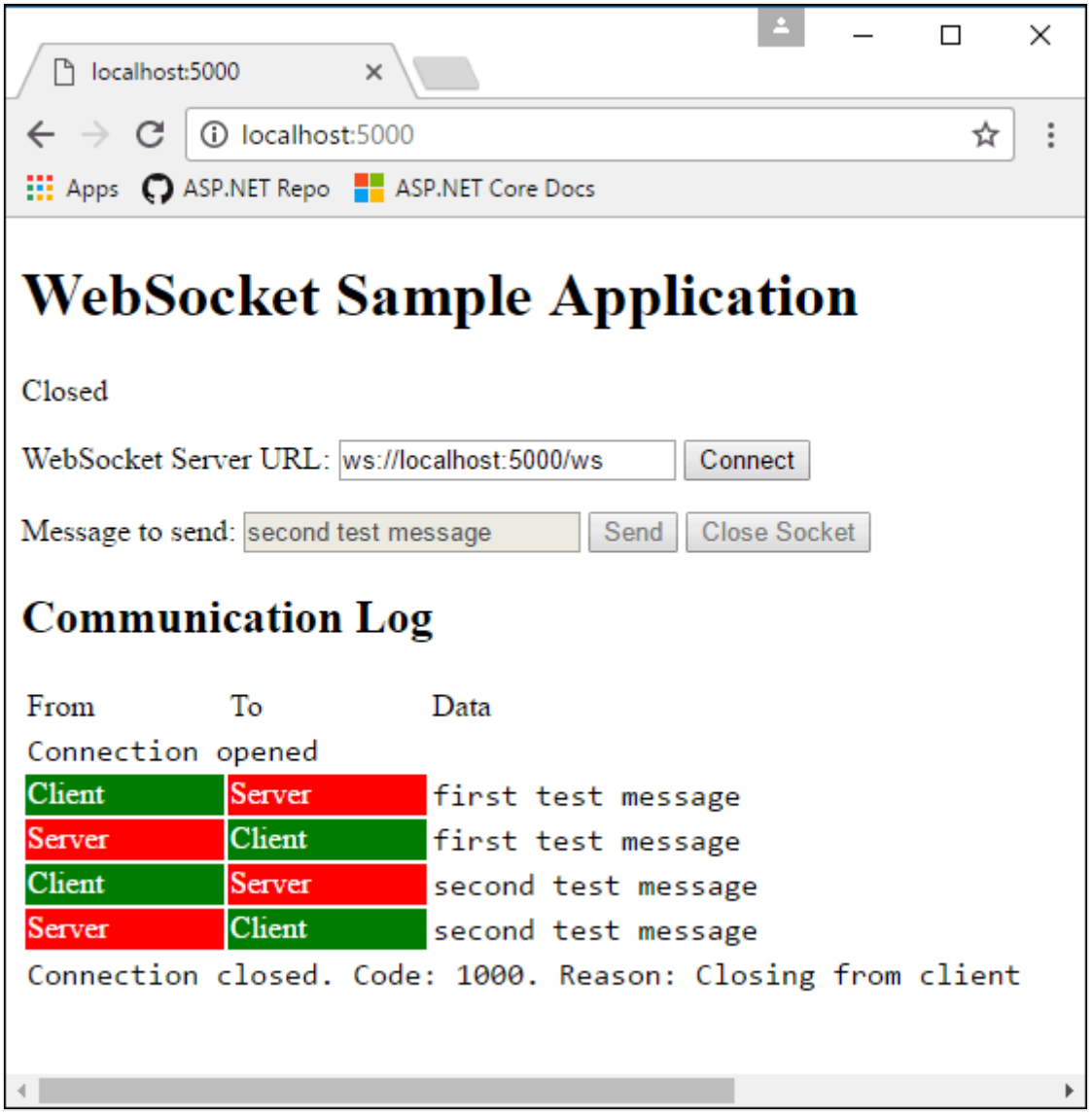
Run the app:

- To run app in Visual Studio: Open the sample project in Visual Studio, and press Ctrl+F5 to run without the debugger.
- To run the app in a command shell: Run the command dotnet run and navigate in a browser to `http://localhost:<port>`.

The webpage shows the connection status:



Select **Connect** to send a WebSocket request to the URL shown. Enter a test message and select **Send**. When done, select **Close Socket**. The **Communication Log** section reports each open, send, and close action as it happens.



Logging and diagnostics in ASP.NET Core SignalR

Article • 10/30/2023

By Andrew Stanton-Nurse

This article provides guidance for gathering diagnostics from your ASP.NET Core SignalR app to help troubleshoot issues.

Server-side logging

> ⚠ **Warning**
>
> Server-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

Since SignalR is part of ASP.NET Core, it uses the ASP.NET Core logging system. In the default configuration, SignalR logs very little information, but this can configured. See the documentation on ASP.NET Core logging for details on configuring ASP.NET Core logging.

SignalR uses two logger categories:

- `Microsoft.AspNetCore.SignalR`: For logs related to Hub Protocols, activating Hubs, invoking methods, and other Hub-related activities.
- `Microsoft.AspNetCore.Http.Connections`: For logs related to transports, such as WebSockets, Long Polling, Server-Sent Events, and low-level SignalR infrastructure.

To enable detailed logs from SignalR, configure both of the preceding prefixes to the `Debug` level in your `appsettings.json` file by adding the following items to the `LogLevel` sub-section in `Logging`:

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Debug",
            "System": "Information",
            "Microsoft": "Information",
            "Microsoft.AspNetCore.SignalR": "Debug",
            "Microsoft.AspNetCore.Http.Connections": "Debug"
```

```
        }
    }
}
```

You can also configure this in code in your `CreateWebHostBuilder` method:

C#

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.AddFilter("Microsoft.AspNetCore.SignalR",
LogLevel.Debug);
            logging.AddFilter("Microsoft.AspNetCore.Http.Connections",
LogLevel.Debug);
        })
        .UseStartup<Startup>();
```

If you aren't using JSON-based configuration, set the following configuration values in your configuration system:

- `Logging:LogLevel:Microsoft.AspNetCore.SignalR` = `Debug`
- `Logging:LogLevel:Microsoft.AspNetCore.Http.Connections` = `Debug`

Check the documentation for your configuration system to determine how to specify nested configuration values. For example, when using environment variables, two `_` characters are used instead of the `:` (for example, `Logging__LogLevel__Microsoft.AspNetCore.SignalR`).

We recommend using the `Debug` level when gathering more detailed diagnostics for your app. The `Trace` level produces very low-level diagnostics and is rarely needed to diagnose issues in your app.

Access server-side logs

How you access server-side logs depends on the environment in which you're running.

As a console app outside IIS

If you're running in a console app, the Console logger should be enabled by default. SignalR logs will appear in the console.

Within IIS Express from Visual Studio

Visual Studio displays the log output in the **Output** window. Select the **ASP.NET Core Web Server** drop down option.

Azure App Service

Enable the **Application Logging (Filesystem)** option in the **Diagnostics logs** section of the Azure App Service portal and configure the **Level** to `Verbose`. Logs should be available from the **Log streaming** service and in logs on the file system of the App Service. For more information, see Azure log streaming.

Other environments

If the app is deployed to another environment (for example, Docker, Kubernetes, or Windows Service), see Logging in .NET Core and ASP.NET Core for more information on how to configure logging providers suitable for the environment.

JavaScript client logging

> ⚠ **Warning**
>
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

When using the JavaScript client, you can configure logging options using the `configureLogging` method on `HubConnectionBuilder`:

JavaScript

```
let connection = new signalR.HubConnectionBuilder()
    .withUrl("/my/hub/url")
    .configureLogging(signalR.LogLevel.Debug)
    .build();
```

To disable framework logging, specify `signalR.LogLevel.None` in the `configureLogging` method. Note that some logging is emitted directly by the browser and can't be disabled via setting the log level.

The following table shows log levels available to the JavaScript client. Setting the log level to one of these values enables logging at that level and all levels above it in the

table.

Expand table

Level	Description
`None`	No messages are logged.
`Critical`	Messages that indicate a failure in the entire app.
`Error`	Messages that indicate a failure in the current operation.
`Warning`	Messages that indicate a non-fatal problem.
`Information`	Informational messages.
`Debug`	Diagnostic messages useful for debugging.
`Trace`	Very detailed diagnostic messages designed for diagnosing specific issues.

Once you've configured the verbosity, the logs will be written to the Browser Console (or Standard Output in a NodeJS app).

If you want to send logs to a custom logging system, you can provide a JavaScript object implementing the `ILogger` interface. The only method that needs to be implemented is `log`, which takes the level of the event and the message associated with the event. For example:

TypeScript

```
import { ILogger, LogLevel, HubConnectionBuilder } from
"@microsoft/signalr";

export class MyLogger implements ILogger {
    log(logLevel: LogLevel, message: string) {
        // Use `message` and `logLevel` to record the log message to your
own system
    }
}

// later on, when configuring your connection...

let connection = new HubConnectionBuilder()
    .withUrl("/my/hub/url")
    .configureLogging(new MyLogger())
    .build();
```

.NET client logging

> ⚠ **Warning**
>
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

To get logs from the .NET client, you can use the `ConfigureLogging` method on `HubConnectionBuilder`. This works the same way as the `ConfigureLogging` method on `WebHostBuilder` and `HostBuilder`. You can configure the same logging providers you use in ASP.NET Core. However, you have to manually install and enable the NuGet packages for the individual logging providers.

To add .NET client logging to a Blazor WebAssembly app, see ASP.NET Core Blazor logging.

Console logging

In order to enable Console logging, add the Microsoft.Extensions.Logging.Console package. Then, use the `AddConsole` method to configure the console logger:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Log to the Console
        logging.AddConsole();

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug);
    })
    .Build();
```

Debug output window logging

You can also configure logs to go to the **Output** window in Visual Studio. Install the Microsoft.Extensions.Logging.Debug package and use the `AddDebug` method:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
```

```
        // Log to the Output Window
        logging.AddDebug();

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug)
    })
    .Build();
```

Other logging providers

SignalR supports other logging providers such as Serilog, Seq, NLog, or any other logging system that integrates with `Microsoft.Extensions.Logging`. If your logging system provides an `ILoggerProvider`, you can register it with `AddProvider`:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Log to your custom provider
        logging.AddProvider(new MyCustomLoggingProvider());

        // This will set ALL logging to Debug level
        logging.SetMinimumLevel(LogLevel.Debug)
    })
    .Build();
```

Control verbosity

If you are logging from other places in your app, changing the default level to `Debug` may be too verbose. You can use a Filter to configure the logging level for SignalR logs. This can be done in code, in much the same way as on the server:

C#

```
var connection = new HubConnectionBuilder()
    .WithUrl("https://example.com/my/hub/url")
    .ConfigureLogging(logging =>
    {
        // Register your providers

        // Set the default log level to Information, but to Debug for
SignalR-related loggers.
        logging.SetMinimumLevel(LogLevel.Information);
```

```
        logging.AddFilter("Microsoft.AspNetCore.SignalR", LogLevel.Debug);
        logging.AddFilter("Microsoft.AspNetCore.Http.Connections",
LogLevel.Debug);
    })
    .Build();
```

Network traces

> ⚠ **Warning**
>
> A network trace contains the full contents of every message sent by your app. **Never** post raw network traces from production apps to public forums like GitHub.

If you encounter an issue, a network trace can sometimes provide a lot of helpful information. This is particularly useful if you're going to file an issue on our issue tracker.

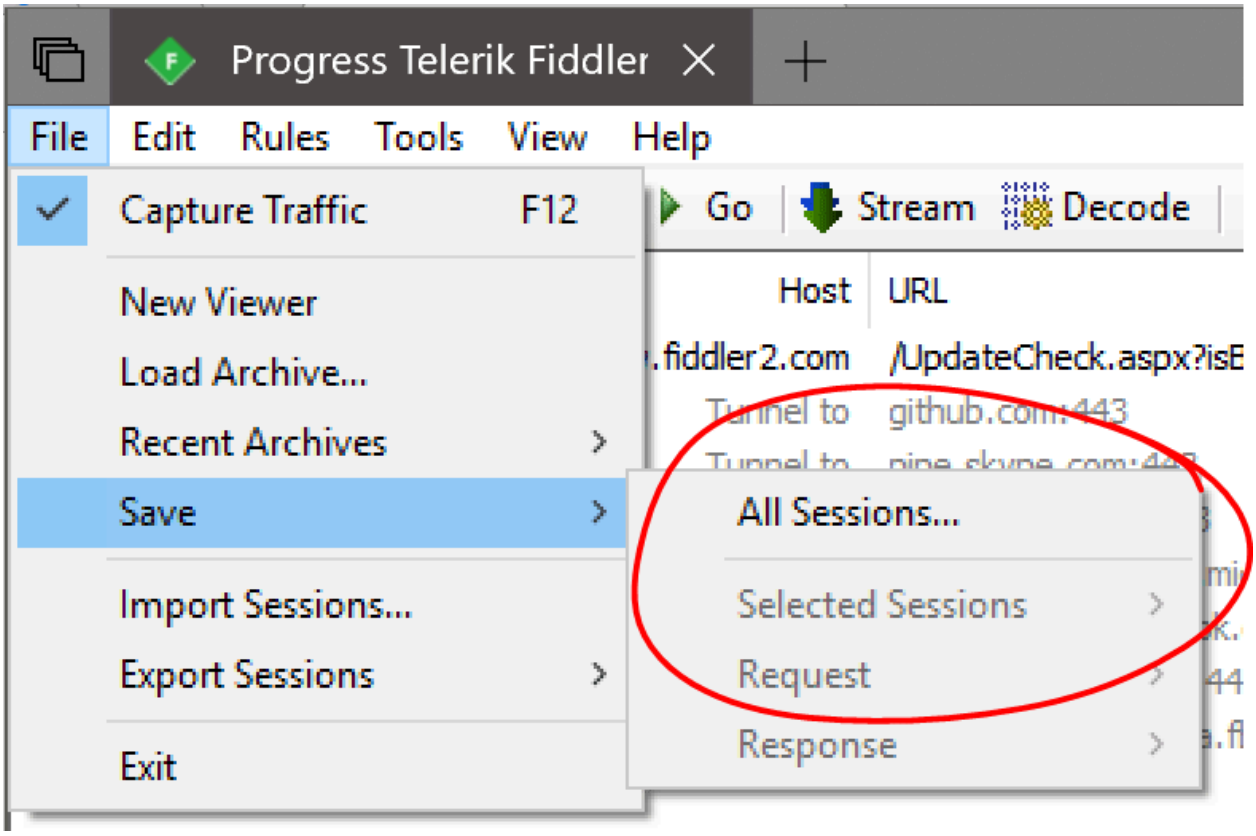
Collect a network trace with Fiddler (preferred option)

This method works for all apps.

Fiddler is a very powerful tool for collecting HTTP traces. Install it from telerik.com/fiddler, launch it, and then run your app and reproduce the issue. Fiddler is available for Windows, and there are beta versions for macOS and Linux.

If you connect using HTTPS, there are some extra steps to ensure Fiddler can decrypt the HTTPS traffic. For more details, see the Fiddler documentation.

Once you've collected the trace, you can export the trace by choosing **File** > **Save** > **All Sessions** from the menu bar.



Collect a network trace with tcpdump (macOS and Linux only)

This method works for all apps.

You can collect raw TCP traces using tcpdump by running the following command from a command shell. You may need to be `root` or prefix the command with `sudo` if you get a permissions error:

Console

```
tcpdump -i [interface] -w trace.pcap
```

Replace `[interface]` with the network interface you wish to capture on. Usually, this is something like `/dev/eth0` (for your standard Ethernet interface) or `/dev/lo0` (for localhost traffic). For more information, see the `tcpdump` man page on your host system.

Collect a network trace in the browser

This method only works for browser-based apps.

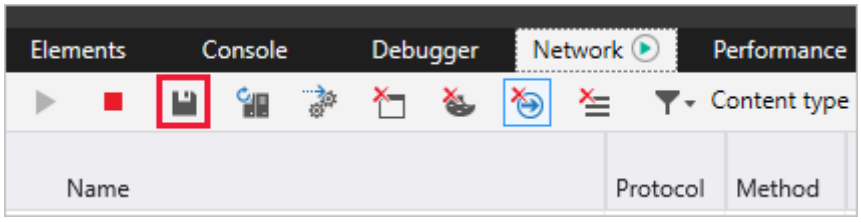
Most browser developer tools consoles have a "Network" tab that allows you to capture network activity between the browser and the server. However, these traces don't

include WebSocket and Server-Sent Event messages. If you are using those transports, using a tool like Fiddler or TcpDump (described below) is a better approach.

Microsoft Edge and Internet Explorer

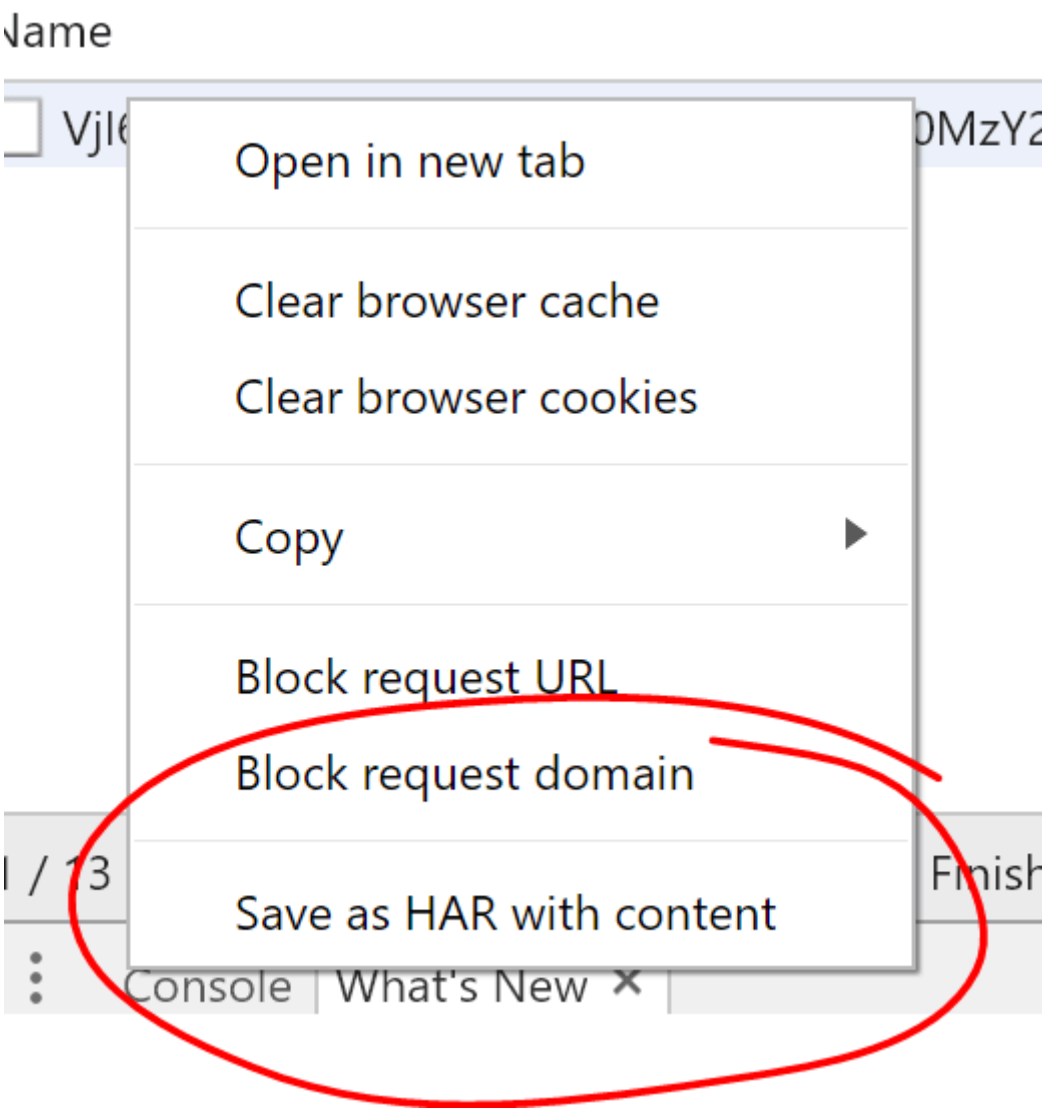
(The instructions are the same for both Edge and Internet Explorer)

1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Click the Save icon in the toolbar to export the trace as a "HAR" file:



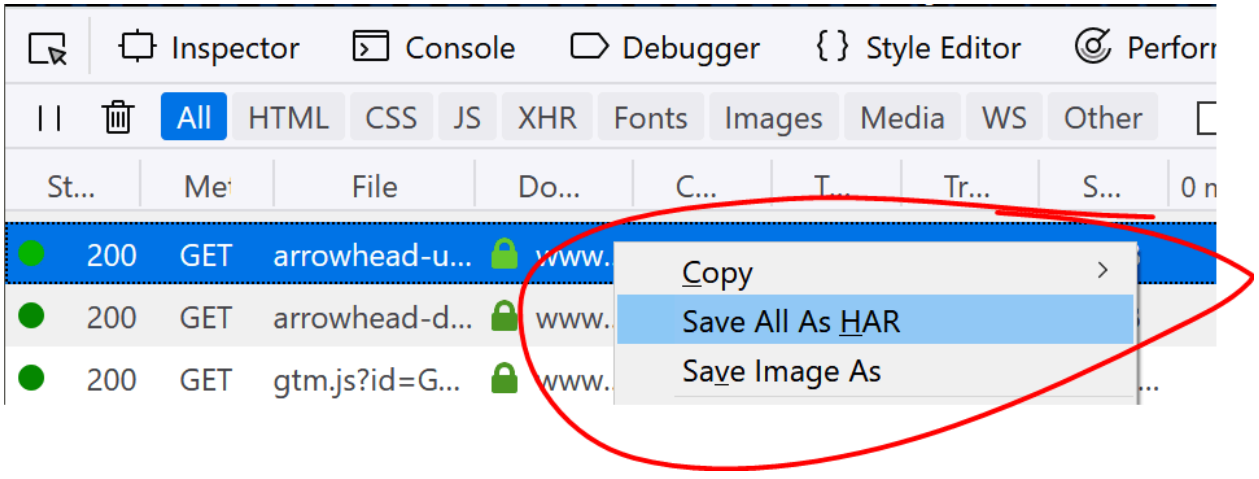
Google Chrome

1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Right click anywhere in the list of requests and choose "Save as HAR with content":



Mozilla Firefox

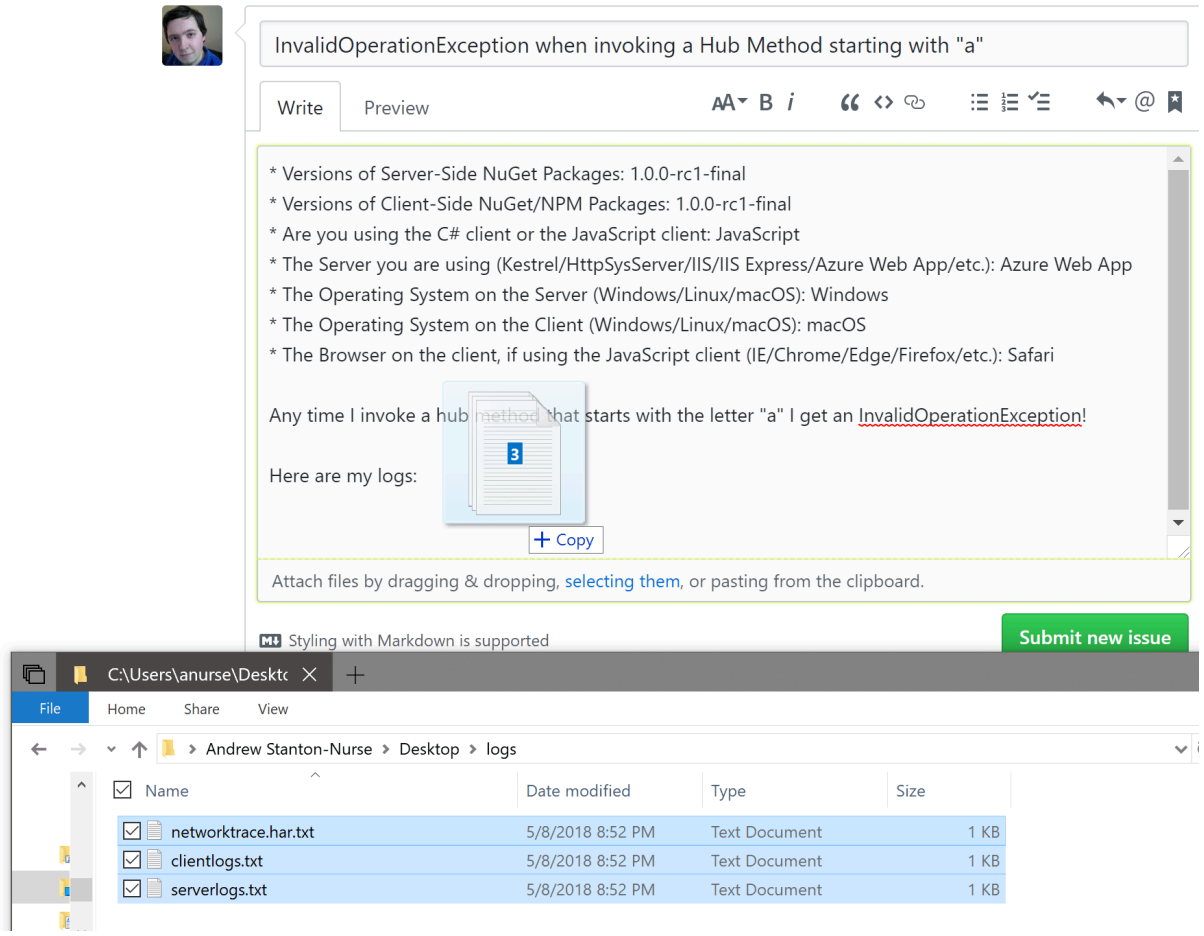
1. Press F12 to open the Dev Tools
2. Click the Network Tab
3. Refresh the page (if needed) and reproduce the problem
4. Right click anywhere in the list of requests and choose "Save All As HAR"



Attach diagnostics files to GitHub issues

You can attach Diagnostics files to GitHub issues by renaming them so they have a `.txt` extension and then dragging and dropping them on to the issue.

> **Note**
>
> Please don't paste the content of log files or network traces into a GitHub issue. These logs and traces can be quite large, and GitHub usually truncates them.



Metrics

Metrics is a representation of data measures over intervals of time. For example, requests per second. Metrics data allows observation of the state of an app at a high level. .NET gRPC metrics are emitted using EventCounter.

SignalR server metrics

SignalR server metrics are reported on the Microsoft.AspNetCore.Http.Connections event source.

Expand table

Name	Description
`connections-started`	Total connections started
`connections-stopped`	Total connections stopped
`connections-timed-out`	Total connections timed out
`current-connections`	Current connections
`connections-duration`	Average connection duration

Observe metrics

dotnet-counters is a performance monitoring tool for ad-hoc health monitoring and first-level performance investigation. Monitor a .NET app with `Microsoft.AspNetCore.Http.Connections` as the provider name. For example:

Console

```
> dotnet-counters monitor --process-id 37016
Microsoft.AspNetCore.Http.Connections

Press p to pause, r to resume, q to quit.
    Status: Running
[Microsoft.AspNetCore.Http.Connections]
    Average Connection Duration (ms)       16,040.56
    Current Connections                            1
    Total Connections Started                      8
    Total Connections Stopped                      7
    Total Connections Timed Out                    0
```

Additional resources

- ASP.NET Core SignalR configuration
- ASP.NET Core SignalR JavaScript client
- ASP.NET Core SignalR .NET Client

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

Troubleshoot connection errors

Article • 01/05/2023

This section provides help with errors that can occur when trying to establish a connection to an ASP.NET Core SignalR hub.

Response code 404

When using WebSockets and `skipNegotiation = true`

log

```
WebSocket connection to 'wss://xxx/HubName' failed: Error during WebSocket
handshake: Unexpected response code: 404
```

- When using multiple servers without sticky sessions, the connection can start on one server and then switch to another server. The other server is not aware of the previous connection.
- Verify the client is connecting to the correct endpoint. For example, the server is hosted at `http://127.0.0.1:5000/hub/myHub` and client is trying to connect to `http://127.0.0.1:5000/myHub`.
- If the connection uses the ID and takes too long to send a request to the server after the negotiate, the server:
  - Deletes the ID.
  - Returns a 404.

Response code 400 or 503

For the following error:

log

```
WebSocket connection to 'wss://xxx/HubName' failed: Error during WebSocket
handshake: Unexpected response code: 400

Error: Failed to start the connection: Error: There was an error with the
transport.
```

This error is usually caused by a client using only the WebSockets transport but the WebSocket protocol isn't enabled on the server.

Response code 307

When using WebSockets and `skipNegotiation = true`

log

```
WebSocket connection to 'ws://xxx/HubName' failed: Error during WebSocket
handshake: Unexpected response code: 307
```

This error can also happen during the negotiate request.

Common cause:

- App is configured to enforce HTTPS by calling `UseHttpsRedirection` in `Startup`, or enforces HTTPS via URL rewrite rule.

Possible solution:

- Change the URL on the client side from "http" to "https".
  `.withUrl("https://xxx/HubName")`

Response code 405

Http status code 405 - Method Not Allowed

- The app doesn't have CORS enabled

Response code 0

Http status code 0 - Usually a CORS issue, no status code is given

log

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
remote resource at http://localhost:5000/default/negotiate?
negotiateVersion=1. (Reason: CORS header 'Access-Control-Allow-Origin'
missing).
```

- Add the expected origins to `.WithOrigins(...)`

log

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
remote resource at http://localhost:5000/default/negotiate?
```

```
negotiateVersion=1. (Reason: expected 'true' in CORS header 'Access-Control-
Allow-Credentials').
```

- Add `.AllowCredentials()` to your CORS policy. Cannot use `.AllowAnyOrigin()` or `.WithOrigins("*")` with this option

Response code 413

Http status code 413 - Payload Too Large

This is often caused by having an access token that is over 4k.

- If using the Azure SignalR Service, reduce the token size by customizing the claims being sent through the Service with:

C#

```
.AddAzureSignalR(options =>
{
    options.ClaimsProvider = context => context.User.Claims;
});
```

Transient network failures

Transient network failures may close the SignalR connection. The server may interpret the closed connection as a graceful client disconnect. To get more info on why a client disconnected in those cases gather logs from the client and server.

Additional resources

- SignalR Hub Protocol

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Overview for gRPC on .NET

Article • 07/31/2024

By James Newton-King

gRPC is a language agnostic, high-performance Remote Procedure Call (RPC) framework.

The main benefits of gRPC are:

- Modern, high-performance, lightweight RPC framework.
- Contract-first API development, using Protocol Buffers by default, allowing for language agnostic implementations.
- Tooling available for many languages to generate strongly-typed servers and clients.
- Supports client, server, and bi-directional streaming calls.
- Reduced network usage with Protobuf binary serialization.

These benefits make gRPC ideal for:

- Lightweight microservices where efficiency is critical.
- Polyglot systems where multiple languages are required for development.
- Point-to-point real-time services that need to handle streaming requests or responses.

C# Tooling support for `.proto` files

gRPC uses a contract-first approach to API development. Services and messages are defined in `.proto` files:

ProtoBuf

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
```

```
  string message = 1;
}
```

.NET types for services, clients, and messages are automatically generated by including `.proto` files in a project:

- Add a package reference to Grpc.Tools package.
- Add `.proto` files to the `<Protobuf>` item group.

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" />
</ItemGroup>
```

For more information on gRPC tooling support, see gRPC services with C#.

gRPC services on ASP.NET Core

gRPC services can be hosted on ASP.NET Core. Services have full integration with ASP.NET Core features such as logging, dependency injection (DI), authentication, and authorization.

Add gRPC services to an ASP.NET Core app

gRPC requires the Grpc.AspNetCore package. For information on configuring gRPC in a .NET app, see Configure gRPC.

The gRPC service project template

The **ASP.NET Core gRPC Service** project template provides a starter service:

C#

```
public class GreeterService : Greeter.GreeterBase
{
    private readonly ILogger<GreeterService> _logger;

    public GreeterService(ILogger<GreeterService> logger)
    {
        _logger = logger;
    }

    public override Task<HelloReply> SayHello(HelloRequest request,
        ServerCallContext context)
```

```
    {
        _logger.LogInformation("Saying hello to {Name}", request.Name);
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

`GreeterService` inherits from the `GreeterBase` type, which is generated from the `Greeter` service in the `.proto` file. The service is made accessible to clients in `Program.cs`:

C#

```
app.MapGrpcService<GreeterService>();
```

To learn more about gRPC services on ASP.NET Core, see gRPC services with ASP.NET Core.

Call gRPC services with a .NET client

gRPC clients are concrete client types that are generated from .proto files. The concrete gRPC client has methods that translate to the gRPC service in the `.proto` file.

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);

var response = await client.SayHelloAsync(
    new HelloRequest { Name = "World" });

Console.WriteLine(response.Message);
```

A gRPC client is created using a channel, which represents a long-lived connection to a gRPC service. A channel can be created using `GrpcChannel.ForAddress`.

For more information on creating clients, and calling different service methods, see Call gRPC services with the .NET client.

Additional resources

- gRPC services with C#

- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client
- gRPC client factory integration in .NET
- Create a .NET Core gRPC client and server in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Create a gRPC client and server in ASP.NET Core

Article • 11/16/2023

This tutorial shows how to create a .NET Core gRPC client and an ASP.NET Core gRPC Server. At the end, you'll have a gRPC client that communicates with the gRPC Greeter service.

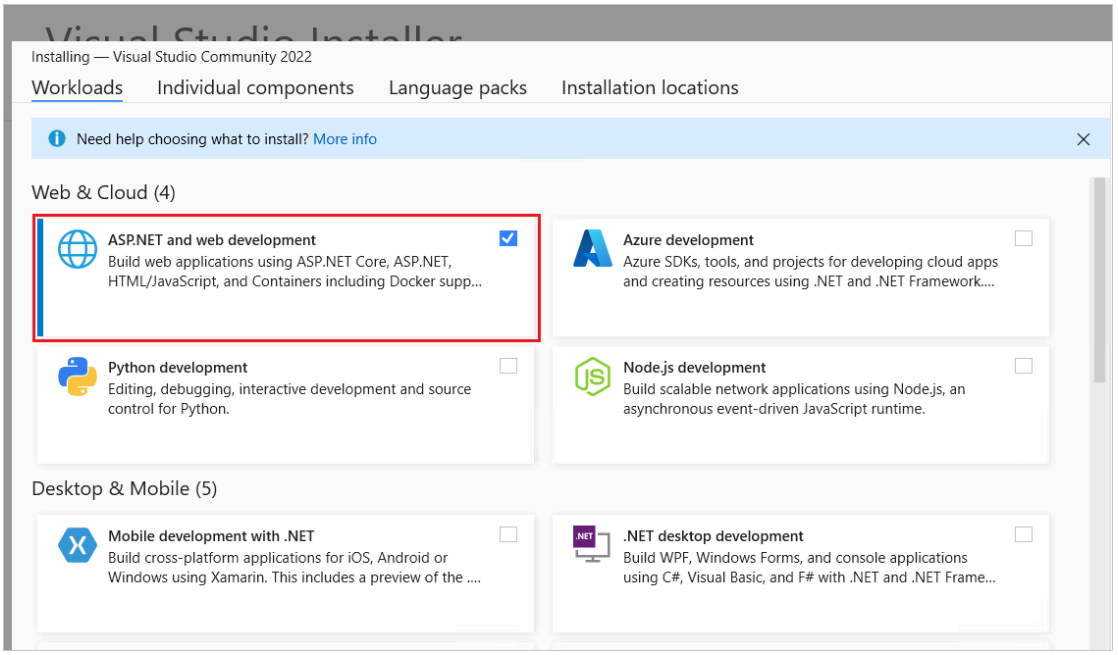
In this tutorial, you:

- ✓ Create a gRPC Server.
- ✓ Create a gRPC client.
- ✓ Test the gRPC client with the gRPC Greeter service.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Create a gRPC service

Visual Studio

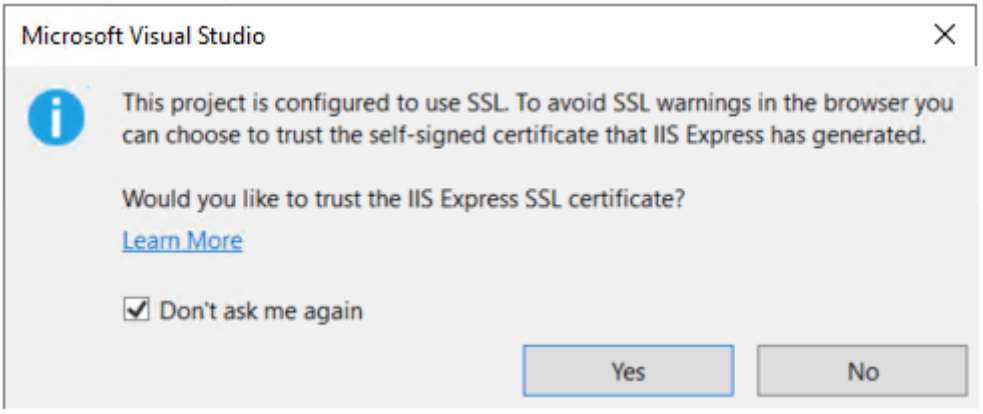
- Start Visual Studio 2022 and select **New Project**.
- In the **Create a new project** dialog, search for `gRPC`. Select **ASP.NET Core gRPC Service** and select **Next**.
- In the **Configure your new project** dialog, enter `GrpcGreeter` for **Project name**. It's important to name the project *GrpcGreeter* so the namespaces match when you copy and paste code.
- Select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Run the service

Visual Studio

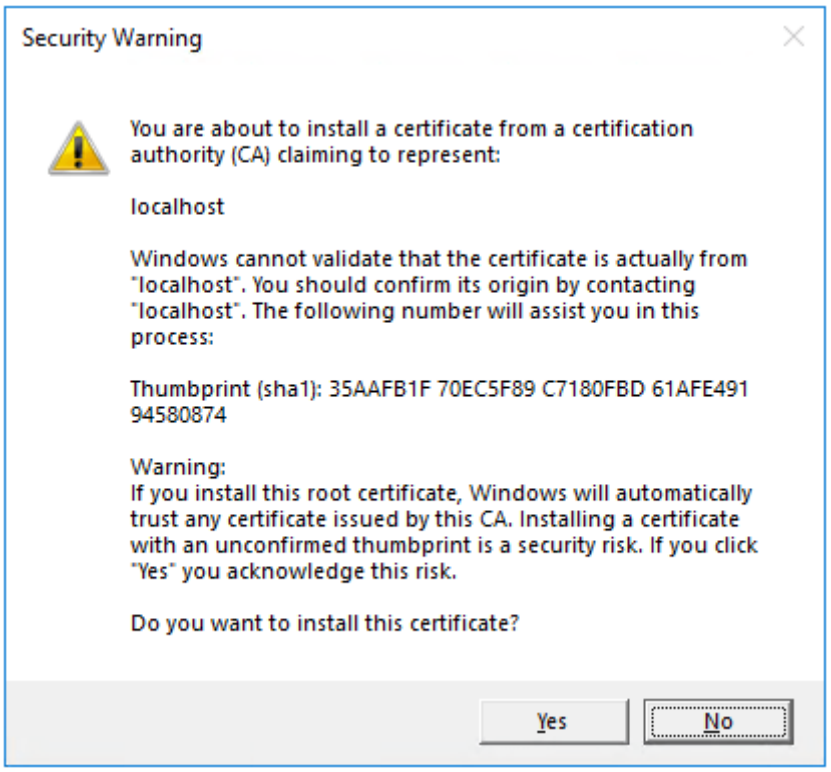
- Press Ctrl+F5 to run without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:

  Select **Yes** if you trust the IIS Express SSL certificate.

  The following dialog is displayed:



Select **Yes** if you agree to trust the development certificate.

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio:

- Starts Kestrel server.
- Launches a browser.
- Navigates to `http://localhost:port`, such as `http://localhost:7042`.
  - *port*: A randomly assigned port number for the app.
  - `localhost`: The standard hostname for the local computer. Localhost only serves web requests from the local computer.

The logs show the service listening on `https://localhost:<port>`, where `<port>` is the localhost port number randomly assigned when the project is created and set in `Properties/launchSettings.json`.

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

> **Note**
>
> The gRPC template is configured to use Transport Layer Security (TLS). gRPC clients need to use HTTPS to call the server. The gRPC service localhost port number is randomly assigned when the project is created and set in the *Properties\launchSettings.json* file of the gRPC service project.

Examine the project files

GrpcGreeter project files:

- `Protos/greet.proto`: defines the `Greeter` gRPC and is used to generate the gRPC server assets. For more information, see Introduction to gRPC.
- `Services` folder: Contains the implementation of the `Greeter` service.
- `appSettings.json`: Contains configuration data such as the protocol used by Kestrel. For more information, see Configuration in ASP.NET Core.
- `Program.cs`, which contains:
  - The entry point for the gRPC service. For more information, see .NET Generic Host in ASP.NET Core.
  - Code that configures app behavior. For more information, see App startup.

Create the gRPC client in a .NET console app

Visual Studio

- Open a second instance of Visual Studio and select **New Project**.
- In the **Create a new project** dialog, select **Console App**, and select **Next**.
- In the **Project name** text box, enter **GrpcGreeterClient** and select **Next**.
- In the **Additional information** dialog, select **.NET 8.0 (Long Term Support)** and then select **Create**.

Add required NuGet packages

The gRPC client project requires the following NuGet packages:

- Grpc.Net.Client, which contains the .NET Core client.
- Google.Protobuf, which contains protobuf message APIs for C#.

- Grpc.Tools, which contain C# tooling support for protobuf files. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`.

Visual Studio

Install the packages using either the Package Manager Console (PMC) or Manage NuGet Packages.

PMC option to install packages

- From Visual Studio, select **Tools** > **NuGet Package Manager** > **Package Manager Console**
- From the **Package Manager Console** window, run `cd GrpcGreeterClient` to change directories to the folder containing the `GrpcGreeterClient.csproj` files.
- Run the following commands:

PowerShell

```
Install-Package Grpc.Net.Client
Install-Package Google.Protobuf
Install-Package Grpc.Tools
```

Manage NuGet Packages option to install packages

- Right-click the project in **Solution Explorer** > **Manage NuGet Packages**.
- Select the **Browse** tab.
- Enter **Grpc.Net.Client** in the search box.
- Select the **Grpc.Net.Client** package from the **Browse** tab and select **Install**.
- Repeat for `Google.Protobuf` and `Grpc.Tools`.

Add greet.proto

- Create a *Protos* folder in the gRPC client project.
- Copy the *Protos\greet.proto* file from the gRPC Greeter service to the *Protos* folder in the gRPC client project.
- Update the namespace inside the `greet.proto` file to the project's namespace:

JSON

```
option csharp_namespace = "GrpcGreeterClient";
```

- Edit the `GrpcGreeterClient.csproj` project file:

Visual Studio

Right-click the project and select **Edit Project File**.

- Add an item group with a `<Protobuf>` element that refers to the *greet.proto* file:

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
</ItemGroup>
```

Create the Greeter client

- Build the client project to create the types in the `GrpcGreeterClient` namespace.

> **Note**
>
> The `GrpcGreeterClient` types are generated automatically by the build process. The tooling package Grpc.Tools generates the following files based on the *greet.proto* file:
>
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\Greet.cs`: The protocol buffer code which populates, serializes and retrieves the request and response message types.
> - `GrpcGreeterClient\obj\Debug\[TARGET_FRAMEWORK]\Protos\GreetGrpc.cs`: Contains the generated client classes.
>
> For more information on the C# assets automatically generated by Grpc.Tools, see gRPC services with C#: Generated C# assets.

- Update the gRPC client `Program.cs` file with the following code.

C#

```
using System.Threading.Tasks;
using Grpc.Net.Client;
using GrpcGreeterClient;

// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

- In the preceding highlighted code, replace the localhost port number `7042` with the `HTTPS` port number specified in `Properties/launchSettings.json` within the `GrpcGreeter` service project.

`Program.cs` contains the entry point and logic for the gRPC client.

The Greeter client is created by:

- Instantiating a `GrpcChannel` containing the information for creating the connection to the gRPC service.
- Using the `GrpcChannel` to construct the Greeter client:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

The Greeter client calls the asynchronous `SayHello` method. The result of the `SayHello` call is displayed:

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
```

```
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

Test the gRPC client with the gRPC Greeter service

Update the `appsettings.Development.json` file by adding the following highlighted lines:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
      ,"Microsoft.AspNetCore.Hosting": "Information",
      "Microsoft.AspNetCore.Routing.EndpointMiddleware": "Information"
    }
  }
}
```

Visual Studio

- In the Greeter service, press `Ctrl+F5` to start the server without the debugger.
- In the `GrpcGreeterClient` project, press `Ctrl+F5` to start the client without the debugger.

The client sends a greeting to the service with a message containing its name, *GreeterClient*. The service sends the message "Hello GreeterClient" as a response. The "Hello GreeterClient" response is displayed in the command prompt:

Console

```
Greeting: Hello GreeterClient
Press any key to exit...
```

The gRPC service records the details of the successful call in the logs written to the command prompt:

Console

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:<port>
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path:
C:\GH\aspnet\docs\4\Docs\aspnetcore\tutorials\grpc\grpc-
start\sample\GrpcGreeter
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST https://localhost:
<port>/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 78.32260000000001ms 200 application/grpc
```

> **Note**
>
> The code in this article requires the ASP.NET Core HTTPS development certificate to secure the gRPC service. If the .NET gRPC client fails with the message `The remote certificate is invalid according to the validation procedure.` or `The SSL connection could not be established.`, the development certificate isn't trusted. To fix this issue, see Call a gRPC service with an untrusted/invalid certificate.

Next steps

- View or download the completed sample code for this tutorial (how to download).
- Overview for gRPC on .NET
- gRPC services with C#
- Migrate gRPC from C-core to gRPC for .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

gRPC services with C#

Article • 07/31/2024

This document outlines the concepts needed to write gRPC apps in C#. The topics covered here apply to both C-core -based and ASP.NET Core-based gRPC apps.

proto file

gRPC uses a contract-first approach to API development. Protocol buffers (protobuf) are used as the Interface Definition Language (IDL) by default. The `.proto` file contains:

- The definition of the gRPC service.
- The messages sent between clients and servers.

For more information on the syntax of protobuf files, see Create Protobuf messages for .NET apps.

For example, consider the *greet.proto* file used in Get started with gRPC service:

- Defines a `Greeter` service.
- The `Greeter` service defines a `SayHello` call.
- `SayHello` sends a `HelloRequest` message and receives a `HelloReply` message:

ProtoBuf

```
syntax = "proto3";

option csharp_namespace = "GrpcGreeter";

package greet;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply);
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

Add a `.proto` file to a C# app

The `.proto` file is included in a project by adding it to the `<Protobuf>` item group:

XML

```xml
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

By default, a `<Protobuf>` reference generates a concrete client and a service base class. The reference element's `GrpcServices` attribute can be used to limit C# asset generation. Valid `GrpcServices` options are:

- `Both` (default when not present)
- `Server`
- `Client`
- `None`

C# Tooling support for `.proto` files

The tooling package Grpc.Tools is required to generate the C# assets from `.proto` files. The generated assets (files):

- Are generated on an as-needed basis each time the project is built.
- Aren't added to the project or checked into source control.
- Are a build artifact contained in the *obj* directory.

This package is required by both the server and client projects. The `Grpc.AspNetCore` metapackage includes a reference to `Grpc.Tools`. Server projects can add `Grpc.AspNetCore` using the Package Manager in Visual Studio or by adding a `<PackageReference>` to the project file:

XML

```xml
<PackageReference Include="Grpc.AspNetCore" Version="2.32.0" />
```

Client projects should directly reference `Grpc.Tools` alongside the other packages required to use the gRPC client. The tooling package isn't required at runtime, so the dependency is marked with `PrivateAssets="All"`:

XML

```
<PackageReference Include="Google.Protobuf" Version="3.18.0" />
<PackageReference Include="Grpc.Net.Client" Version="2.52.0" />
<PackageReference Include="Grpc.Tools" Version="2.40.0">
  <IncludeAssets>runtime; build; native; contentfiles; analyzers;
buildtransitive</IncludeAssets>
  <PrivateAssets>all</PrivateAssets>
</PackageReference>
```

Generated C# assets

The tooling package generates the C# types representing the messages defined in the included `.proto` files.

For server-side assets, an abstract service base type is generated. The base type contains the definitions of all the gRPC calls contained in the `.proto` file. Create a concrete service implementation that derives from this base type and implements the logic for the gRPC calls. For the `greet.proto`, the example described previously, an abstract `GreeterBase` type that contains a virtual `SayHello` method is generated. A concrete implementation `GreeterService` overrides the method and implements the logic handling the gRPC call.

C#

```
public class GreeterService : Greeter.GreeterBase
{
    private readonly ILogger<GreeterService> _logger;
    public GreeterService(ILogger<GreeterService> logger)
    {
        _logger = logger;
    }

    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

For client-side assets, a concrete client type is generated. The gRPC calls in the `.proto` file are translated into methods on the concrete type, which can be called. For the `greet.proto`, the example described previously, a concrete `GreeterClient` type is generated. Call `GreeterClient.SayHelloAsync` to initiate a gRPC call to the server.

C#

```
// The port number must match the port of the gRPC server.
using var channel = GrpcChannel.ForAddress("https://localhost:7042");
var client = new Greeter.GreeterClient(channel);
var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
Console.WriteLine("Greeting: " + reply.Message);
Console.WriteLine("Press any key to exit...");
Console.ReadKey();
```

By default, server and client assets are generated for each `.proto` file included in the `<Protobuf>` item group. To ensure only the server assets are generated in a server project, the `GrpcServices` attribute is set to `Server`.

XML

```
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

Similarly, the attribute is set to `Client` in client projects.

Additional resources

- Overview for gRPC on .NET
- Create a .NET Core gRPC client and server in ASP.NET Core
- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Create gRPC services and methods

Article • 07/31/2024

By James Newton-King

This document explains how to create gRPC services and methods in C#. Topics include:

- How to define services and methods in `.proto` files.
- Generated code using gRPC C# tooling.
- Implementing gRPC services and methods.

Create new gRPC services

gRPC services with C# introduced gRPC's contract-first approach to API development. Services and messages are defined in `.proto` files. C# tooling then generates code from `.proto` files. For server-side assets, an abstract base type is generated for each service, along with classes for any messages.

The following `.proto` file:

- Defines a `Greeter` service.
- The `Greeter` service defines a `SayHello` call.
- `SayHello` sends a `HelloRequest` message and receives a `HelloReply` message

ProtoBuf

```
syntax = "proto3";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

C# tooling generates the C# `GreeterBase` base type:

C#

```
public abstract partial class GreeterBase
{
    public virtual Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        throw new RpcException(new Status(StatusCode.Unimplemented, ""));
    }
}

public class HelloRequest
{
    public string Name { get; set; }
}

public class HelloReply
{
    public string Message { get; set; }
}
```

By default the generated `GreeterBase` doesn't do anything. Its virtual `SayHello` method will return an `UNIMPLEMENTED` error to any clients that call it. For the service to be useful an app must create a concrete implementation of `GreeterBase`:

C#

```
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        return Task.FromResult(new HelloReply { Message = $"Hello
{request.Name}" });
    }
}
```

The `ServerCallContext` gives the context for a server-side call.

The service implementation is registered with the app. If the service is hosted by ASP.NET Core gRPC, it should be added to the routing pipeline with the `MapGrpcService` method.

C#

```
app.MapGrpcService<GreeterService>();
```

See gRPC services with ASP.NET Core for more information.

Implement gRPC methods

A gRPC service can have different types of methods. How messages are sent and received by a service depends on the type of method defined. The gRPC method types are:

- Unary
- Server streaming
- Client streaming
- Bi-directional streaming

Streaming calls are specified with the `stream` keyword in the `.proto` file. `stream` can be placed on a call's request message, response message, or both.

ProtoBuf

```
syntax = "proto3";

service ExampleService {
  // Unary
  rpc UnaryCall (ExampleRequest) returns (ExampleResponse);

  // Server streaming
  rpc StreamingFromServer (ExampleRequest) returns (stream ExampleResponse);

  // Client streaming
  rpc StreamingFromClient (stream ExampleRequest) returns (ExampleResponse);

  // Bi-directional streaming
  rpc StreamingBothWays (stream ExampleRequest) returns (stream
ExampleResponse);
}
```

Each call type has a different method signature. Overriding generated methods from the abstract base service type in a concrete implementation ensures the correct arguments and return type are used.

Unary method

A unary method has the request message as a parameter, and returns the response. A unary call is complete when the response is returned.

C#

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request,
    ServerCallContext context)
{
```

```
    var response = new ExampleResponse();
    return Task.FromResult(response);
}
```

Unary calls are the most similar to actions on web API controllers. One important difference gRPC methods have from actions is gRPC methods are not able to bind parts of a request to different method arguments. gRPC methods always have one message argument for the incoming request data. Multiple values can still be sent to a gRPC service by adding fields to the request message:

ProtoBuf

```
message ExampleRequest {
    int32 pageIndex = 1;
    int32 pageSize = 2;
    bool isDescending = 3;
}
```

Server streaming method

A server streaming method has the request message as a parameter. Because multiple messages can be streamed back to the caller, `responseStream.WriteAsync` is used to send response messages. A server streaming call is complete when the method returns.

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    for (var i = 0; i < 5; i++)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

The client has no way to send additional messages or data once the server streaming method has started. Some streaming methods are designed to run forever. For continuous streaming methods, a client can cancel the call when it's no longer needed. When cancellation happens the client sends a signal to the server and the ServerCallContext.CancellationToken is raised. The `CancellationToken` token should be used on the server with async methods so that:

- Any asynchronous work is canceled together with the streaming call.

- The method exits quickly.

```
C#
```

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    while (!context.CancellationToken.IsCancellationRequested)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1),
context.CancellationToken);
    }
}
```

Client streaming method

A client streaming method starts *without* the method receiving a message. The `requestStream` parameter is used to read messages from the client. A client streaming call is complete when a response message is returned:

```
C#
```

```
public override async Task<ExampleResponse> StreamingFromClient(
    IAsyncStreamReader<ExampleRequest> requestStream, ServerCallContext
context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        // ...
    }
    return new ExampleResponse();
}
```

Bi-directional streaming method

A bi-directional streaming method starts *without* the method receiving a message. The `requestStream` parameter is used to read messages from the client. The method can choose to send messages with `responseStream.WriteAsync`. A bi-directional streaming call is complete when the method returns:

```
C#
```

```
public override async Task
StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
```

```
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    await foreach (var message in requestStream.ReadAllAsync())
    {
        await responseStream.WriteAsync(new ExampleResponse());
    }
}
```

The preceding code:

- Sends a response for each request.
- Is a basic usage of bi-directional streaming.

It is possible to support more complex scenarios, such as reading requests and sending responses simultaneously:

C#

```
public override async Task
StreamingBothWays(IAsyncStreamReader<ExampleRequest> requestStream,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    // Read requests in a background task.
    var readTask = Task.Run(async () =>
    {
        await foreach (var message in requestStream.ReadAllAsync())
        {
            // Process request.
        }
    });

    // Send responses until the client signals that it is complete.
    while (!readTask.IsCompleted)
    {
        await responseStream.WriteAsync(new ExampleResponse());
        await Task.Delay(TimeSpan.FromSeconds(1),
context.CancellationToken);
    }
}
```

In a bi-directional streaming method, the client and service can send messages to each other at any time. The best implementation of a bi-directional method varies depending upon requirements.

Access gRPC request headers

A request message is not the only way for a client to send data to a gRPC service. Header values are available in a service using `ServerCallContext.RequestHeaders`.

C#

```
public override Task<ExampleResponse> UnaryCall(ExampleRequest request,
    ServerCallContext context)
{
    var userAgent = context.RequestHeaders.GetValue("user-agent");
    // ...

    return Task.FromResult(new ExampleResponse());
}
```

Multi-threading with gRPC streaming methods

There are important considerations to implementing gRPC streaming methods that use multiple threads.

Reader and writer thread safety

`IAsyncStreamReader<TMessage>` and `IServerStreamWriter<TMessage>` can each be used by only one thread at a time. For a streaming gRPC method, multiple threads can't read new messages with `requestStream.MoveNext()` simultaneously. And multiple threads can't write new messages with `responseStream.WriteAsync(message)` simultaneously.

A safe way to enable multiple threads to interact with a gRPC method is to use the producer-consumer pattern with System.Threading.Channels.

C#

```
public override async Task DownloadResults(DataRequest request,
        IServerStreamWriter<DataResult> responseStream, ServerCallContext
context)
{
    var channel = Channel.CreateBounded<DataResult>(new
BoundedChannelOptions(capacity: 5));

    var consumerTask = Task.Run(async () =>
    {
        // Consume messages from channel and write to response stream.
        await foreach (var message in channel.Reader.ReadAllAsync())
        {
            await responseStream.WriteAsync(message);
        }
    });
```

```
    var dataChunks = request.Value.Chunk(size: 10);

    // Write messages to channel from multiple threads.
    await Task.WhenAll(dataChunks.Select(
        async c =>
        {
            var message = new DataResult { BytesProcessed = c.Length };
            await channel.Writer.WriteAsync(message);
        }));

    // Complete writing and wait for consumer to complete.
    channel.Writer.Complete();
    await consumerTask;
}
```

The preceding gRPC server streaming method:

- Creates a bounded channel for producing and consuming `DataResult` messages.
- Starts a task to read messages from the channel and write them to the response stream.
- Writes messages to the channel from multiple threads.

> **Note**
>
> Bidirectional streaming methods take `IAsyncStreamReader<TMessage>` and `IServerStreamWriter<TMessage>` as arguments. It's safe to use these types on separate threads from each other.

Interacting with a gRPC method after a call ends

A gRPC call ends on the server once the gRPC method exits. The following arguments passed to gRPC methods aren't safe to use after the call has ended:

- `ServerCallContext`
- `IAsyncStreamReader<TMessage>`
- `IServerStreamWriter<TMessage>`

If a gRPC method starts background tasks that use these types, it must complete the tasks before the gRPC method exits. Continuing to use the context, stream reader, or stream writer after the gRPC method exists causes errors and unpredictable behavior.

In the following example, the server streaming method could write to the response stream after the call has finished:

```
```

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    _ = Task.Run(async () =>
    {
        for (var i = 0; i < 5; i++)
        {
            await responseStream.WriteAsync(new ExampleResponse());
            await Task.Delay(TimeSpan.FromSeconds(1));
        }
    });

    await PerformLongRunningWorkAsync();
}
```

For the previous example, the solution is to await the write task before exiting the method:

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    var writeTask = Task.Run(async () =>
    {
        for (var i = 0; i < 5; i++)
        {
            await responseStream.WriteAsync(new ExampleResponse());
            await Task.Delay(TimeSpan.FromSeconds(1));
        }
    });

    await PerformLongRunningWorkAsync();

    await writeTask;
}
```

Additional resources

- gRPC services with C#
- Call gRPC services with the .NET client

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Create Protobuf messages for .NET apps

Article • 09/27/2024

By James Newton-King and Mark Rendle

gRPC uses Protobuf as its Interface Definition Language (IDL). Protobuf IDL is a language neutral format for specifying the messages sent and received by gRPC services. Protobuf messages are defined in `.proto` files. This document explains how Protobuf concepts map to .NET.

Protobuf messages

Messages are the main data transfer object in Protobuf. They are conceptually similar to .NET classes.

ProtoBuf

```
syntax = "proto3";

option csharp_namespace = "Contoso.Messages";

message Person {
    int32 id = 1;
    string first_name = 2;
    string last_name = 3;
}
```

The preceding message definition specifies three fields as name-value pairs. Like properties on .NET types, each field has a name and a type. The field type can be a Protobuf scalar value type, for example `int32`, or another message.

The Protobuf style guide recommends using `underscore_separated_names` for field names. New Protobuf messages created for .NET apps should follow the Protobuf style guidelines. .NET tooling automatically generates .NET types that use .NET naming standards. For example, a `first_name` Protobuf field generates a `FirstName` .NET property.

In addition to a name, each field in the message definition has a unique number. Field numbers are used to identify fields when the message is serialized to Protobuf. Serializing a small number is faster than serializing the entire field name. Because field numbers identify a field it is important to take care when changing them. For more information about changing Protobuf messages see Versioning gRPC services.

When an app is built the Protobuf tooling generates .NET types from `.proto` files. The `Person` message generates a .NET class:

C#

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

For more information about Protobuf messages see the Protobuf language guide.

Scalar Value Types

Protobuf supports a range of native scalar value types. The following table lists them all with their equivalent C# type:

Expand table

Protobuf type	C# type
`double`	`double`
`float`	`float`
`int32`	`int`
`int64`	`long`
`uint32`	`uint`
`uint64`	`ulong`
`sint32`	`int`
`sint64`	`long`
`fixed32`	`uint`
`fixed64`	`ulong`
`sfixed32`	`int`
`sfixed64`	`long`
`bool`	`bool`

Protobuf type	C# type
`string`	`string`
`bytes`	`ByteString`

Scalar values always have a default value and can't be set to `null`. This constraint includes `string` and `ByteString` which are C# classes. `string` defaults to an empty string value and `ByteString` defaults to an empty bytes value. Attempting to set them to `null` throws an error.

Nullable wrapper types can be used to support null values.

Dates and times

The native scalar types don't provide for date and time values, equivalent to .NET's DateTimeOffset, DateTime, and TimeSpan. These types can be specified by using some of Protobuf's *Well-Known Types* extensions. These extensions provide code generation and runtime support for complex field types across the supported platforms.

The following table shows the date and time types:

Expand table

.NET type	Protobuf Well-Known Type
`DateTimeOffset`	`google.protobuf.Timestamp`
`DateTime`	`google.protobuf.Timestamp`
`TimeSpan`	`google.protobuf.Duration`

ProtoBuf

```
syntax = "proto3";

import "google/protobuf/duration.proto";
import "google/protobuf/timestamp.proto";

message Meeting {
    string subject = 1;
    google.protobuf.Timestamp start = 2;
    google.protobuf.Duration duration = 3;
}
```

The generated properties in the C# class aren't the .NET date and time types. The properties use the `Timestamp` and `Duration` classes in the `Google.Protobuf.WellKnownTypes` namespace. These classes provide methods for converting to and from `DateTimeOffset`, `DateTime`, and `TimeSpan`.

C#

```
// Create Timestamp and Duration from .NET DateTimeOffset and TimeSpan.
var meeting = new Meeting
{
    Time = Timestamp.FromDateTimeOffset(meetingTime), // also FromDateTime()
    Duration = Duration.FromTimeSpan(meetingLength)
};

// Convert Timestamp and Duration to .NET DateTimeOffset and TimeSpan.
var time = meeting.Time.ToDateTimeOffset();
var duration = meeting.Duration?.ToTimeSpan();
```

> ⓘ **Note**
>
> The `Timestamp` type works with UTC times. `DateTimeOffset` values always have an offset of zero, and the `DateTime.Kind` property is always `DateTimeKind.Utc`.

Nullable types

The Protobuf code generation for C# uses the native types, such as `int` for `int32`. So the values are always included and can't be `null`.

For values that require explicit `null`, such as using `int?` in C# code, Protobuf's Well-Known Types include wrappers that are compiled to nullable C# types. To use them, import `wrappers.proto` into your `.proto` file, like the following code:

ProtoBuf

```
syntax = "proto3";

import "google/protobuf/wrappers.proto";

message Person {
    // ...
    google.protobuf.Int32Value age = 5;
}
```

`wrappers.proto` types aren't exposed in generated properties. Protobuf automatically maps them to appropriate .NET nullable types in C# messages. For example, a `google.protobuf.Int32Value` field generates an `int?` property. Reference type properties like `string` and `ByteString` are unchanged except `null` can be assigned to them without error.

The following table shows the complete list of wrapper types with their equivalent C# type:

Expand table

C# type	Well-Known Type wrapper
`bool?`	`google.protobuf.BoolValue`
`double?`	`google.protobuf.DoubleValue`
`float?`	`google.protobuf.FloatValue`
`int?`	`google.protobuf.Int32Value`
`long?`	`google.protobuf.Int64Value`
`uint?`	`google.protobuf.UInt32Value`
`ulong?`	`google.protobuf.UInt64Value`
`string`	`google.protobuf.StringValue`
`ByteString`	`google.protobuf.BytesValue`

Bytes

Binary payloads are supported in Protobuf with the `bytes` scalar value type. A generated property in C# uses `ByteString` as the property type.

Use `ByteString.CopyFrom(byte[] data)` to create a new instance from a byte array:

C#

```
var data = await File.ReadAllBytesAsync(path);

var payload = new PayloadResponse();
payload.Data = ByteString.CopyFrom(data);
```

`ByteString` data is accessed directly using `ByteString.Span` or `ByteString.Memory`. Or call `ByteString.ToByteArray()` to convert an instance back into a byte array:

C#

```
var payload = await client.GetPayload(new PayloadRequest());

await File.WriteAllBytesAsync(path, payload.Data.ToByteArray());
```

Decimals

Protobuf doesn't natively support the .NET `decimal` type, just `double` and `float`. There's an ongoing discussion in the Protobuf project about the possibility of adding a standard decimal type to the Well-Known Types, with platform support for languages and frameworks that support it. Nothing has been implemented yet.

It's possible to create a message definition to represent the `decimal` type that works for safe serialization between .NET clients and servers. But developers on other platforms would have to understand the format being used and implement their own handling for it.

Creating a custom decimal type for Protobuf

ProtoBuf

```
package CustomTypes;

// Example: 12345.6789 -> { units = 12345, nanos = 678900000 }
message DecimalValue {

    // Whole units part of the amount
    int64 units = 1;

    // Nano units of the amount (10^-9)
    // Must be same sign as units
    sfixed32 nanos = 2;
}
```

The `nanos` field represents values from `0.999_999_999` to `-0.999_999_999`. For example, the `decimal` value `1.5m` would be represented as `{ units = 1, nanos = 500_000_000 }`. This is why the `nanos` field in this example uses the `sfixed32` type, which encodes more efficiently than `int32` for larger values. If the `units` field is negative, the `nanos` field should also be negative.

> **Note**
>
> Additional algorithms are available for encoding `decimal` values as byte strings. The algorithm used by `DecimalValue`:
>
> - Is easy to understand.
> - Isn't affected by big-endian or little-endian on different platforms.
> - Supports decimal numbers ranging from positive `9,223,372,036,854,775,807.999999999` to negative `9,223,372,036,854,775,808.999999999` with a maximum precision of nine decimal places, which isn't the full range of a `decimal`.

Conversion between this type and the BCL `decimal` type might be implemented in C# like this:

C#

```
namespace CustomTypes
{
    public partial class DecimalValue
    {
        private const decimal NanoFactor = 1_000_000_000;
        public DecimalValue(long units, int nanos)
        {
            Units = units;
            Nanos = nanos;
        }

        public static implicit operator decimal(CustomTypes.DecimalValue
grpcDecimal)
        {
            return grpcDecimal.Units + grpcDecimal.Nanos / NanoFactor;
        }

        public static implicit operator CustomTypes.DecimalValue(decimal
value)
        {
            var units = decimal.ToInt64(value);
            var nanos = decimal.ToInt32((value - units) * NanoFactor);
            return new CustomTypes.DecimalValue(units, nanos);
        }
    }
}
```

The preceding code:

- Adds a partial class for `DecimalValue`. The partial class is combined with `DecimalValue` generated from the `.proto` file. The generated class declares the `Units` and `Nanos` properties.
- Has implicit operators for converting between `DecimalValue` and the BCL `decimal` type.

Collections

Lists

Lists in Protobuf are specified by using the `repeated` prefix keyword on a field. The following example shows how to create a list:

ProtoBuf

```
message Person {
    // ...
    repeated string roles = 8;
}
```

In the generated code, `repeated` fields are represented by the `Google.Protobuf.Collections.RepeatedField<T>` generic type.

C#

```
public class Person
{
    // ...
    public RepeatedField<string> Roles { get; }
}
```

`RepeatedField<T>` implements IList<T>. So you can use LINQ queries or convert it to an array or a list. `RepeatedField<T>` properties don't have a public setter. Items should be added to the existing collection.

C#

```
var person = new Person();

// Add one item.
person.Roles.Add("user");

// Add all items from another collection.
```

```
var roles = new [] { "admin", "manager" };
person.Roles.Add(roles);
```

Dictionaries

The .NET IDictionary<TKey,TValue> type is represented in Protobuf using `map<key_type, value_type>`.

ProtoBuf

```
message Person {
    // ...
    map<string, string> attributes = 9;
}
```

In generated .NET code, `map` fields are represented by the `Google.Protobuf.Collections.MapField<TKey, TValue>` generic type. `MapField<TKey, TValue>` implements IDictionary<TKey,TValue>. Like `repeated` properties, `map` properties don't have a public setter. Items should be added to the existing collection.

C#

```
var person = new Person();

// Add one item.
person.Attributes["created_by"] = "James";

// Add all items from another collection.
var attributes = new Dictionary<string, string>
{
    ["last_modified"] = DateTime.UtcNow.ToString()
};
person.Attributes.Add(attributes);
```

Unstructured and conditional messages

Protobuf is a contract-first messaging format. An app's messages, including its fields and types, must be specified in `.proto` files when the app is built. Protobuf's contract-first design is great at enforcing message content but can limit scenarios where a strict contract isn't required:

- Messages with unknown payloads. For example, a message with a field that could contain any message.

- Conditional messages. For example, a message returned from a gRPC service might be a success result or an error result.
- Dynamic values. For example, a message with a field that contains an unstructured collection of values, similar to JSON.

Protobuf offers language features and types to support these scenarios.

Any

The `Any` type lets you use messages as embedded types without having their `.proto` definition. To use the `Any` type, import `any.proto`.

ProtoBuf

```
import "google/protobuf/any.proto";

message Status {
    string message = 1;
    google.protobuf.Any detail = 2;
}
```

C#

```
// Create a status with a Person message set to detail.
var status = new ErrorStatus();
status.Detail = Any.Pack(new Person { FirstName = "James" });

// Read Person message from detail.
if (status.Detail.Is(Person.Descriptor))
{
    var person = status.Detail.Unpack<Person>();
    // ...
}
```

Oneof

`oneof` fields are a language feature. The compiler handles the `oneof` keyword when it generates the message class. Using `oneof` to specify a response message that could either return a `Person` or `Error` might look like this:

ProtoBuf

```
message Person {
    // ...
}
```

```
message Error {
    // ...
}

message ResponseMessage {
  oneof result {
    Error error = 1;
    Person person = 2;
  }
}
```

Fields within the `oneof` set must have unique field numbers in the overall message declaration.

When using `oneof`, the generated C# code includes an enum that specifies which of the fields has been set. You can test the enum to find which field is set. Fields that aren't set return `null` or the default value, rather than throwing an exception.

C#

```
var response = await client.GetPersonAsync(new RequestMessage());

switch (response.ResultCase)
{
    case ResponseMessage.ResultOneofCase.Person:
        HandlePerson(response.Person);
        break;
    case ResponseMessage.ResultOneofCase.Error:
        HandleError(response.Error);
        break;
    default:
        throw new ArgumentException("Unexpected result.");
}
```

Value

The `Value` type represents a dynamically typed value. It can be either `null`, a number, a string, a boolean, a dictionary of values (`Struct`), or a list of values (`ValueList`). `Value` is a Protobuf Well-Known Type that uses the previously discussed `oneof` feature. To use the `Value` type, import `struct.proto`.

ProtoBuf

```
import "google/protobuf/struct.proto";

message Status {
```

```
    // ...
    google.protobuf.Value data = 3;
}
```

C#

```
// Create dynamic values.
var status = new Status();
status.Data = Value.ForStruct(new Struct
{
    Fields =
    {
        ["enabled"] = Value.ForBool(true),
        ["metadata"] = Value.ForList(
            Value.ForString("value1"),
            Value.ForString("value2"))
    }
});

// Read dynamic values.
switch (status.Data.KindCase)
{
    case Value.KindOneofCase.StructValue:
        foreach (var field in status.Data.StructValue.Fields)
        {
            // Read struct fields...
        }
        break;
    // ...
}
```

Using `Value` directly can be verbose. An alternative way to use `Value` is with Protobuf's built-in support for mapping messages to JSON. Protobuf's `JsonFormatter` and `JsonWriter` types can be used with any Protobuf message. `Value` is particularly well suited to being converted to and from JSON.

This is the JSON equivalent of the previous code:

C#

```
// Create dynamic values from JSON.
var status = new Status();
status.Data = Value.Parser.ParseJson(@"{
    ""enabled"": true,
    ""metadata"": [ ""value1"", ""value2"" ]
}");

// Convert dynamic values to JSON.
// JSON can be read with a library like System.Text.Json or Newtonsoft.Json
```

```
var json = JsonFormatter.Default.Format(status.Data);
var document = JsonDocument.Parse(json);
```

Additional resources

- Protobuf language guide
- Versioning gRPC services

Versioning gRPC services

Article • 07/31/2024

By James Newton-King

New features added to an app can require gRPC services provided to clients to change, sometimes in unexpected and breaking ways. When gRPC services change:

- Consideration should be given on how changes impact clients.
- A versioning strategy to support changes should be implemented.

Backwards compatibility

The gRPC protocol is designed to support services that change over time. Generally, additions to gRPC services and methods are non-breaking. Non-breaking changes allow existing clients to continue working without changes. Changing or deleting gRPC services are breaking changes. When gRPC services have breaking changes, clients using that service have to be updated and redeployed.

Making non-breaking changes to a service has a number of benefits:

- Existing clients continue to run.
- Avoids work involved with notifying clients of breaking changes, and updating them.
- Only one version of the service needs to be documented and maintained.

Non-breaking changes

These changes are non-breaking at a gRPC protocol level and .NET binary level.

- **Adding a new service**
- **Adding a new method to a service**
- **Adding a field to a request message** - Fields added to a request message are deserialized with the default value on the server when not set. To be a non-breaking change, the service must succeed when the new field isn't set by older clients.
- **Adding a field to a response message** - If an older client hasn't been updated with the new field, the value is deserialized into the response message's unknown fields collection.
- **Adding a value to an enum** - Enums are serialized as a numeric value. New enum values are deserialized on the client to the enum value without an enum name. To

be a non-breaking change, older clients must run correctly when receiving the new enum value.

Binary breaking changes

The following changes are non-breaking at a gRPC protocol level, but the client needs to be updated if it upgrades to the latest `.proto` contract or client .NET assembly. Binary compatibility is important if you plan to publish a gRPC library to NuGet.

- **Removing a field** - Values from a removed field are deserialized to a message's unknown fields. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract. It's important that a removed field number isn't accidentally reused in the future. To ensure this doesn't happen, specify deleted field numbers and names on the message using Protobuf's reserved keyword.
- **Renaming a message** - Message names aren't typically sent on the network, so this isn't a gRPC protocol breaking change. The client will need to be updated if it upgrades to the latest contract. One situation where message names **are** sent on the network is with Any fields, when the message name is used to identify the message type.
- **Nesting or unnesting a message** - Message types can be nested. Nesting or unnesting a message changes its message name. Changing how a message type is nested has the same impact on compatibility as renaming.
- **Changing csharp_namespace** - Changing `csharp_namespace` will change the namespace of generated .NET types. This isn't a gRPC protocol breaking change, but the client needs to be updated if it upgrades to the latest contract.

Protocol breaking changes

The following items are protocol and binary breaking changes:

- **Renaming a field** - With Protobuf content, the field names are only used in generated code. The field number is used to identify fields on the network. Renaming a field isn't a protocol breaking change for Protobuf. However, if a server is using JSON content then renaming a field is a breaking change.
- **Changing a field data type** - Changing a field's data type to an incompatible type will cause errors when deserializing the message. Even if the new data type is compatible, it's likely the client needs to be updated to support the new type if it upgrades to the latest contract.
- **Changing a field number** - With Protobuf payloads, the field number is used to identify fields on the network.

- **Renaming a package, service or method** - gRPC uses the package name, service name, and method name to build the URL. The client gets an *UNIMPLEMENTED* status from the server.
- **Removing a service or method** - The client gets an *UNIMPLEMENTED* status from the server when calling the removed method.

Behavior breaking changes

When making non-breaking changes, you must also consider whether older clients can continue working with the new service behavior. For example, adding a new field to a request message:

- Isn't a protocol breaking change.
- Returning an error status on the server if the new field isn't set makes it a breaking change for old clients.

Behavior compatibility is determined by your app-specific code.

Version number services

Services should strive to remain backwards compatible with old clients. Eventually changes to your app may require breaking changes. Breaking old clients and forcing them to be updated along with your service isn't a good user experience. A way to maintain backwards compatibility while making breaking changes is to publish multiple versions of a service.

gRPC supports an optional package specifier, which functions much like a .NET namespace. In fact, the `package` will be used as the .NET namespace for generated .NET types if `option csharp_namespace` is not set in the `.proto` file. The package can be used to specify a version number for your service and its messages:

ProtoBuf

```
syntax = "proto3";

package greet.v1;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}
```

```
message HelloReply {
  string message = 1;
}
```

The package name is combined with the service name to identify a service address. A service address allows multiple versions of a service to be hosted side-by-side:

- `greet.v1.Greeter`
- `greet.v2.Greeter`

Implementations of the versioned service are registered in `Startup.cs`:

C#

```
app.UseEndpoints(endpoints =>
{
    // Implements greet.v1.Greeter
    endpoints.MapGrpcService<GreeterServiceV1>();

    // Implements greet.v2.Greeter
    endpoints.MapGrpcService<GreeterServiceV2>();
});
```

Including a version number in the package name gives you the opportunity to publish a *v2* version of your service with breaking changes, while continuing to support older clients who call the *v1* version. Once clients have updated to use the *v2* service, you can choose to remove the old version. When planning to publish multiple versions of a service:

- Avoid breaking changes if reasonable.
- Don't update the version number unless making breaking changes.
- Do update the version number when you make breaking changes.

Publishing multiple versions of a service duplicates it. To reduce duplication, consider moving business logic from the service implementations to a centralized location that can be reused by the old and new implementations:

C#

```
using Greet.V1;
using Grpc.Core;
using System.Threading.Tasks;

namespace Services
{
    public class GreeterServiceV1 : Greeter.GreeterBase
    {
```

```
        private readonly IGreeter _greeter;
        public GreeterServiceV1(IGreeter greeter)
        {
            _greeter = greeter;
        }

        public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
        {
            return Task.FromResult(new HelloReply
            {
                Message = _greeter.GetHelloMessage(request.Name)
            });
        }
    }
}
```

Services and messages generated with different package names are **different .NET types**. Moving business logic to a centralized location requires mapping messages to common types.

Additional resources

- Create Protobuf messages for .NET apps

Test gRPC services in ASP.NET Core

Article • 07/31/2024

By: James Newton-King

Testing is an important aspect of building stable and maintainable software. This article discusses how to test ASP.NET Core gRPC services.

There are three common approaches for testing gRPC services:

- **Unit testing**: Test gRPC services directly from a unit testing library.
- **Integration testing**: The gRPC app is hosted in TestServer, an in-memory test server from the Microsoft.AspNetCore.TestHost package. gRPC services are tested by calling them using a gRPC client from a unit testing library.
- **Manual testing**: Test gRPC servers with ad hoc calls. For information about how to use command-line and UI tooling with gRPC services, see Test gRPC services with gRPCurl and gRPCui in ASP.NET Core.

In unit testing, only the gRPC service is involved. Dependencies injected into the service must be mocked. In integration testing, the gRPC service and its auxiliary infrastructure are part of the test. This includes app startup, dependency injection, routing and authentication, and authorization.

Example testable service

To demonstrate service tests, review the following service in the sample app.

View or download sample code (how to download)

The `TesterService` returns greetings using gRPC's four method types.

C#

```
public class TesterService : Tester.TesterBase
{
    private readonly IGreeter _greeter;

    public TesterService(IGreeter greeter)
    {
        _greeter = greeter;
    }

    public override Task<HelloReply> SayHelloUnary(HelloRequest request,
        ServerCallContext context)
    {
```

```
        var message = _greeter.Greet(request.Name);
        return Task.FromResult(new HelloReply { Message = message });
    }

    public override async Task SayHelloServerStreaming(HelloRequest request,
        IServerStreamWriter<HelloReply> responseStream, ServerCallContext
context)
    {
        var i = 0;
        while (!context.CancellationToken.IsCancellationRequested)
        {
            var message = _greeter.Greet($"{request.Name} {++i}");
            await responseStream.WriteAsync(new HelloReply { Message =
message });

            await Task.Delay(1000);
        }
    }

    public override async Task<HelloReply> SayHelloClientStreaming(
        IAsyncStreamReader<HelloRequest> requestStream, ServerCallContext
context)
    {
        var names = new List<string>();

        await foreach (var request in requestStream.ReadAllAsync())
        {
            names.Add(request.Name);
        }

        var message = _greeter.Greet(string.Join(", ", names));
        return new HelloReply { Message = message };
    }

    public override async Task SayHelloBidirectionalStreaming(
        IAsyncStreamReader<HelloRequest> requestStream,
        IServerStreamWriter<HelloReply> responseStream,
        ServerCallContext context)
    {
        await foreach (var request in requestStream.ReadAllAsync())
        {
            await responseStream.WriteAsync(
                new HelloReply { Message = _greeter.Greet(request.Name) });
        }
    }
}
```

The preceding gRPC service:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IGreeter`.

- Can be tested with a mocked `IGreeter` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Integration tests in ASP.NET Core.

Unit test gRPC services

A unit test library can directly test gRPC services by calling its methods. Unit tests test a gRPC service in isolation.

C#

```
[Fact]
public async Task SayHelloUnaryTest()
{
    // Arrange
    var mockGreeter = new Mock<IGreeter>();
    mockGreeter.Setup(
        m => m.Greet(It.IsAny<string>())).Returns((string s) => $"Hello
{s}");
    var service = new TesterService(mockGreeter.Object);

    // Act
    var response = await service.SayHelloUnary(
        new HelloRequest { Name = "Joe" }, TestServerCallContext.Create());

    // Assert
    mockGreeter.Verify(v => v.Greet("Joe"));
    Assert.Equal("Hello Joe", response.Message);
}
```

The preceding unit test:

- Mocks `IGreeter` using Moq.
- Executes the `SayHelloUnary` method with a request message and a `ServerCallContext`. All service methods have a `ServerCallContext` argument. In this test, the type is provided using the `TestServerCallContext.Create()` helper method. This helper method is included in the sample code.
- Makes assertions:
  - Verifies the request name is passed to `IGreeter`.
  - The service returns the expected reply message.

Unit test `HttpContext` in gRPC methods

gRPC methods can access a request's HttpContext using the `ServerCallContext.GetHttpContext` extension method. To unit test a method that uses `HttpContext`, the context must be configured in test setup. If HttpContext isn't configured then `GetHttpContext` returns `null`.

To configure a `HttpContext` during test setup, create a new instance and add it to `ServerCallContext.UserState` collection using the `__HttpContext` key.

C#

```
var httpContext = new DefaultHttpContext();

var serverCallContext = TestServerCallContext.Create();
serverCallContext.UserState["__HttpContext"] = httpContext;
```

Execute service methods with this call context to use the configured `HttpContext` instance.

Integration test gRPC services

Integration tests evaluate an app's components on a broader level than unit tests. The gRPC app is hosted in TestServer, an in-memory test server from the Microsoft.AspNetCore.TestHost package.

A unit test library starts the gRPC app and then gRPC services are tested using the gRPC client.

The sample code contains infrastructure to make integration testing possible:

- The `GrpcTestFixture<TStartup>` class configures the ASP.NET Core host and starts the gRPC app in an in-memory test server.
- The `IntegrationTestBase` class is the base type that integration tests inherit from. It contains the fixture state and APIs for creating a gRPC client to call the gRPC app.

C#

```
[Fact]
public async Task SayHelloUnaryTest()
{
    // Arrange
    var client = new Tester.TesterClient(Channel);

    // Act
    var response = await client.SayHelloUnaryAsync(new HelloRequest { Name =
```

```
"Joe" });

    // Assert
    Assert.Equal("Hello Joe", response.Message);
}
```

The preceding integration test:

- Creates a gRPC client using the channel provided by `IntegrationTestBase`. This type is included in the sample code.
- Calls the `SayHelloUnary` method using the gRPC client.
- Asserts the service returns the expected reply message.

Inject mock dependencies

Use `ConfigureWebHost` on the fixture to override dependencies. Overriding dependencies is useful when an external dependency is unavailable in the test environment. For example, an app that uses an external payment gateway shouldn't call the external dependency when executing tests. Instead, use a mock gateway for the test.

C#

```
public MockedGreeterServiceTests(GrpcTestFixture<Startup> fixture,
    ITestOutputHelper outputHelper) : base(fixture, outputHelper)
{
    var mockGreeter = new Mock<IGreeter>();
    mockGreeter.Setup(
        m => m.Greet(It.IsAny<string>())).Returns((string s) =>
        {
            if (string.IsNullOrEmpty(s))
            {
                throw new ArgumentException("Name not provided.");
            }
            return $"Test {s}";
        });

    Fixture.ConfigureWebHost(builder =>
    {
        builder.ConfigureServices(
            services => services.AddSingleton(mockGreeter.Object));
    });
}

[Fact]
public async Task SayHelloUnaryTest_MockGreeter_Success()
{
    // Arrange
    var client = new Tester.TesterClient(Channel);
```

```
    // Act
    var response = await client.SayHelloUnaryAsync(
        new HelloRequest { Name = "Joe" });

    // Assert
    Assert.Equal("Test Joe", response.Message);
}
```

The preceding integration test:

- In the test class's (`MockedGreeterServiceTests`) constructor:
  - Mocks `IGreeter` using Moq.
  - Overrides the `IGreeter` registered with dependency injection using `ConfigureWebHost`.
- Calls the `SayHelloUnary` method using the gRPC client.
- Asserts the expected reply message based on the mock `IGreeter` instance.

Additional resources

- Test gRPC services with gRPCurl and gRPCui in ASP.NET Core
- Mock gRPC client in tests

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Call gRPC services with the .NET client

Article • 01/08/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

A .NET gRPC client library is available in the Grpc.Net.Client NuGet package. This document explains how to:

- Configure a gRPC client to call gRPC services.
- Make gRPC calls to unary, server streaming, client streaming, and bi-directional streaming methods.

Configure gRPC client

gRPC clients are concrete client types that are generated from .proto files. The concrete gRPC client has methods that translate to the gRPC service in the `.proto` file. For example, a service called `Greeter` generates a `GreeterClient` type with methods to call the service.

A gRPC client is created from a channel. Start by using `GrpcChannel.ForAddress` to create a channel, and then use the channel to create a gRPC client:

```
C#

var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greet.GreeterClient(channel);
```

A channel represents a long-lived connection to a gRPC service. When a channel is created, it's configured with options related to calling a service. For example, the `HttpClient` used to make calls, the maximum send and receive message size, and logging can be specified on `GrpcChannelOptions` and used with `GrpcChannel.ForAddress`. For a complete list of options, see client configuration options.

```
C#

var channel = GrpcChannel.ForAddress("https://localhost:5001");

var greeterClient = new Greet.GreeterClient(channel);
```

```
var counterClient = new Count.CounterClient(channel);

// Use clients to call gRPC services
```

Configure TLS

A gRPC client must use the same connection-level security as the called service. gRPC client Transport Layer Security (TLS) is configured when the gRPC channel is created. A gRPC client throws an error when it calls a service and the connection-level security of the channel and service don't match.

To configure a gRPC channel to use TLS, ensure the server address starts with `https`. For example, `GrpcChannel.ForAddress("https://localhost:5001")` uses HTTPS protocol. The gRPC channel automatically negotiates a connection secured by TLS and uses a secure connection to make gRPC calls.

> **Tip**
>
> gRPC supports client certificate authentication over TLS. For information on configuring client certificates with a gRPC channel, see **Authentication and authorization in gRPC for ASP.NET Core**.

To call unsecured gRPC services, ensure the server address starts with `http`. For example, `GrpcChannel.ForAddress("http://localhost:5000")` uses HTTP protocol. In .NET Core 3.1, additional configuration is required to call insecure gRPC services with the .NET client.

Client performance

Channel and client performance and usage:

- Creating a channel can be an expensive operation. Reusing a channel for gRPC calls provides performance benefits.
- A channel manages connections to the server. If the connection is closed or lost, the channel automatically reconnects the next time a gRPC call is made.
- gRPC clients are created with channels. gRPC clients are lightweight objects and don't need to be cached or reused.
- Multiple gRPC clients can be created from a channel, including different types of clients.
- A channel and clients created from the channel can safely be used by multiple threads.

- Clients created from the channel can make multiple simultaneous calls.

`GrpcChannel.ForAddress` isn't the only option for creating a gRPC client. If calling gRPC services from an ASP.NET Core app, consider gRPC client factory integration. gRPC integration with `HttpClientFactory` offers a centralized alternative to creating gRPC clients.

Make gRPC calls

A gRPC call is initiated by calling a method on the client. The gRPC client will handle message serialization and addressing the gRPC call to the correct service.

gRPC has different types of methods. How the client is used to make a gRPC call depends on the type of method called. The gRPC method types are:

- Unary
- Server streaming
- Client streaming
- Bi-directional streaming

Unary call

A unary call starts with the client sending a request message. A response message is returned when the service finishes.

C#

```
var client = new Greet.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = "World"
});

Console.WriteLine("Greeting: " + response.Message);
// Greeting: Hello World
```

Each unary service method in the `.proto` file will result in two .NET methods on the concrete gRPC client type for calling the method: an asynchronous method and a blocking method. For example, on `GreeterClient` there are two ways of calling `SayHello`:

- `GreeterClient.SayHelloAsync` - calls `Greeter.SayHello` service asynchronously. Can be awaited.
- `GreeterClient.SayHello` - calls `Greeter.SayHello` service and blocks until complete. Don't use in asynchronous code.

Server streaming call

A server streaming call starts with the client sending a request message. `ResponseStream.MoveNext()` reads messages streamed from the service. The server streaming call is complete when `ResponseStream.MoveNext()` returns `false`.

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

while (await call.ResponseStream.MoveNext())
{
    Console.WriteLine("Greeting: " + call.ResponseStream.Current.Message);
    // "Greeting: Hello World" is written multiple times
}
```

When using C# 8 or later, the `await foreach` syntax can be used to read messages. The `IAsyncStreamReader<T>.ReadAllAsync()` extension method reads all messages from the response stream:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });

await foreach (var response in call.ResponseStream.ReadAllAsync())
{
    Console.WriteLine("Greeting: " + response.Message);
    // "Greeting: Hello World" is written multiple times
}
```

The type returned from starting a server streaming call implements `IDisposable`. Always dispose a streaming call to ensure it's stopped and all resources are cleaned up.

Client streaming call

A client streaming call starts *without* the client sending a message. The client can choose to send messages with `RequestStream.WriteAsync`. When the client has finished sending messages, `RequestStream.CompleteAsync()` should be called to notify the service. The call is finished when the service returns a response message.

C#

```
var client = new Counter.CounterClient(channel);
using var call = client.AccumulateCount();

for (var i = 0; i < 3; i++)
{
    await call.RequestStream.WriteAsync(new CounterRequest { Count = 1 });
}
await call.RequestStream.CompleteAsync();

var response = await call;
Console.WriteLine($"Count: {response.Count}");
// Count: 3
```

The type returned from starting a client streaming call implements `IDisposable`. Always dispose a streaming call to ensure it's stopped and all resources are cleaned up.

Bi-directional streaming call

A bi-directional streaming call starts *without* the client sending a message. The client can choose to send messages with `RequestStream.WriteAsync`. Messages streamed from the service are accessible with `ResponseStream.MoveNext()` or `ResponseStream.ReadAllAsync()`. The bi-directional streaming call is complete when the `ResponseStream` has no more messages.

C#

```
var client = new Echo.EchoClient(channel);
using var call = client.Echo();

Console.WriteLine("Starting background task to receive messages");
var readTask = Task.Run(async () =>
{
    await foreach (var response in call.ResponseStream.ReadAllAsync())
    {
        Console.WriteLine(response.Message);
        // Echo messages sent to the service
    }
});

Console.WriteLine("Starting to send messages");
Console.WriteLine("Type a message to echo then press enter.");
while (true)
{
    var result = Console.ReadLine();
    if (string.IsNullOrEmpty(result))
    {
        break;
    }
```

```
    await call.RequestStream.WriteAsync(new EchoMessage { Message = result
});
}

Console.WriteLine("Disconnecting");
await call.RequestStream.CompleteAsync();
await readTask;
```

For best performance, and to avoid unnecessary errors in the client and service, try to complete bi-directional streaming calls gracefully. A bi-directional call completes gracefully when the server has finished reading the request stream and the client has finished reading the response stream. The preceding sample call is one example of a bi-directional call that ends gracefully. In the call, the client:

1. Starts a new bi-directional streaming call by calling `EchoClient.Echo`.
2. Creates a background task to read messages from the service using `ResponseStream.ReadAllAsync()`.
3. Sends messages to the server with `RequestStream.WriteAsync`.
4. Notifies the server it has finished sending messages with `RequestStream.CompleteAsync()`.
5. Waits until the background task has read all incoming messages.

During a bi-directional streaming call, the client and service can send messages to each other at any time. The best client logic for interacting with a bi-directional call varies depending upon the service logic.

The type returned from starting a bi-directional streaming call implements `IDisposable`. Always dispose a streaming call to ensure it's stopped and all resources are cleaned up.

Access gRPC headers

gRPC calls return response headers. HTTP response headers pass name/value metadata about a call that isn't related the returned message.

Headers are accessible using `ResponseHeadersAsync`, which returns a collection of metadata. Headers are typically returned with the response message; therefore, you must await them.

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHelloAsync(new HelloRequest { Name = "World" });

var headers = await call.ResponseHeadersAsync;
```

```
var myValue = headers.GetValue("my-trailer-name");

var response = await call.ResponseAsync;
```

`ResponseHeadersAsync` usage:

- Must await the result of `ResponseHeadersAsync` to get the headers collection.
- Doesn't have to be accessed before `ResponseAsync` (or the response stream when streaming). If a response has been returned, then `ResponseHeadersAsync` returns headers instantly.
- Will throw an exception if there was a connection or server error and headers weren't returned for the gRPC call.

Access gRPC trailers

gRPC calls may return response trailers. Trailers are used to provide name/value metadata about a call. Trailers provide similar functionality to HTTP headers, but are received at the end of the call.

Trailers are accessible using `GetTrailers()`, which returns a collection of metadata. Trailers are returned after the response is complete. Therefore, you must await all response messages before accessing the trailers.

Unary and client streaming calls must await `ResponseAsync` before calling `GetTrailers()`:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHelloAsync(new HelloRequest { Name = "World" });
var response = await call.ResponseAsync;

Console.WriteLine("Greeting: " + response.Message);
// Greeting: Hello World

var trailers = call.GetTrailers();
var myValue = trailers.GetValue("my-trailer-name");
```

Server and bidirectional streaming calls must finish awaiting the response stream before calling `GetTrailers()`:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHellos(new HelloRequest { Name = "World" });
```

```
await foreach (var response in call.ResponseStream.ReadAllAsync())
{
    Console.WriteLine("Greeting: " + response.Message);
    // "Greeting: Hello World" is written multiple times
}

var trailers = call.GetTrailers();
var myValue = trailers.GetValue("my-trailer-name");
```

Trailers are also accessible from `RpcException`. A service may return trailers together with a non-OK gRPC status. In this situation, the trailers are retrieved from the exception thrown by the gRPC client:

C#

```
var client = new Greet.GreeterClient(channel);
string myValue = null;

try
{
    using var call = client.SayHelloAsync(new HelloRequest { Name = "World"
});
    var response = await call.ResponseAsync;

    Console.WriteLine("Greeting: " + response.Message);
    // Greeting: Hello World

    var trailers = call.GetTrailers();
    myValue = trailers.GetValue("my-trailer-name");
}
catch (RpcException ex)
{
    var trailers = ex.Trailers;
    myValue = trailers.GetValue("my-trailer-name");
}
```

Configure deadline

Configuring a gRPC call deadline is recommended because it provides an upper limit on how long a call can run for. It stops misbehaving services from running forever and exhausting server resources. Deadlines are a useful tool for building reliable apps.

Configure `CallOptions.Deadline` to set a deadline for a gRPC call:

C#

```
var client = new Greet.GreeterClient(channel);
```

```
try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = "World" },
        deadline: DateTime.UtcNow.AddSeconds(5));

    // Greeting: Hello World
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded)
{
    Console.WriteLine("Greeting timeout.");
}
```

For more information, see Reliable gRPC services with deadlines and cancellation.

Additional resources

- gRPC client factory integration in .NET
- Reliable gRPC services with deadlines and cancellation
- gRPC services with C#

gRPC client factory integration in .NET

Article • 10/29/2022

By James Newton-King

gRPC integration with `HttpClientFactory` offers a centralized way to create gRPC clients. It can be used as an alternative to configuring stand-alone gRPC client instances. Factory integration is available in the Grpc.Net.ClientFactory NuGet package.

The factory offers the following benefits:

- Provides a central location for configuring logical gRPC client instances.
- Manages the lifetime of the underlying `HttpClientMessageHandler`.
- Automatic propagation of deadline and cancellation in an ASP.NET Core gRPC service.

Register gRPC clients

To register a gRPC client, the generic `AddGrpcClient` extension method can be used within an instance of WebApplicationBuilder at the app's entry point in `Program.cs`, specifying the gRPC typed client class and service address:

C#

```
builder.Services.AddGrpcClient<Greeter.GreeterClient>(o =>
{
    o.Address = new Uri("https://localhost:5001");
});
```

The gRPC client type is registered as transient with dependency injection (DI). The client can now be injected and consumed directly in types created by DI. ASP.NET Core MVC controllers, SignalR hubs and gRPC services are places where gRPC clients can automatically be injected:

C#

```
public class AggregatorService : Aggregator.AggregatorBase
{
    private readonly Greeter.GreeterClient _client;

    public AggregatorService(Greeter.GreeterClient client)
    {
        _client = client;
    }
```

```
    public override async Task SayHellos(HelloRequest request,
        IServerStreamWriter<HelloReply> responseStream, ServerCallContext
context)
    {
        // Forward the call on to the greeter service
        using (var call = _client.SayHellos(request))
        {
            await foreach (var response in
call.ResponseStream.ReadAllAsync())
            {
                await responseStream.WriteAsync(response);
            }
        }
    }
}
```

Configure HttpHandler

`HttpClientFactory` creates the `HttpMessageHandler` used by the gRPC client. Standard `HttpClientFactory` methods can be used to add outgoing request middleware or to configure the underlying `HttpClientHandler` of the `HttpClient`:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        var handler = new HttpClientHandler();
        handler.ClientCertificates.Add(LoadCertificate());
        return handler;
    });
```

For more information, see Make HTTP requests using IHttpClientFactory.

Configure Interceptors

gRPC interceptors can be added to clients using the `AddInterceptor` method.

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
```

```
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddInterceptor<LoggingInterceptor>();
```

The preceding code:

- Registers the `GreeterClient` type.
- Configures a `LoggingInterceptor` for this client. `LoggingInterceptor` is created once and shared between `GreeterClient` instances.

By default, an interceptor is created once and shared between clients. This behavior can be overridden by specifying a scope when registering an interceptor. The client factory can be configured to create a new interceptor for each client by specifying `InterceptorScope.Client`.

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddInterceptor<LoggingInterceptor>(InterceptorScope.Client);
```

Creating client scoped interceptors is useful when an interceptor requires scoped or transient scoped services from DI.

A gRPC interceptor or channel credentials can be used to send `Authorization` metadata with each request. For more information about configuring authentication, see Send a bearer token with gRPC client factory.

Configure Channel

Additional configuration can be applied to a channel using the `ConfigureChannel` method:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigureChannel(o =>
```

```
    {
        o.Credentials = new CustomCredentials();
    });
```

`ConfigureChannel` is passed a `GrpcChannelOptions` instance. For more information, see configure client options.

> **Note**
>
> Some properties are set on `GrpcChannelOptions` before the `ConfigureChannel` callback is run:
>
> - `HttpHandler` is set to the result from ConfigurePrimaryHttpMessageHandler.
> - `LoggerFactory` is set to the ILoggerFactory resolved from DI.
>
> These values can be overridden by `ConfigureChannel`.

Call credentials

An authentication header can be added to gRPC calls using the `AddCallCredentials` method:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials((context, metadata) =>
    {
        if (!string.IsNullOrEmpty(_token))
        {
            metadata.Add("Authorization", $"Bearer {_token}");
        }
        return Task.CompletedTask;
    });
```

For more information about configuring call credentials, see Bearer token with gRPC client factory.

Deadline and cancellation propagation

gRPC clients created by the factory in a gRPC service can be configured with `EnableCallContextPropagation()` to automatically propagate the deadline and cancellation token to child calls. The `EnableCallContextPropagation()` extension method is available in the Grpc.AspNetCore.Server.ClientFactory NuGet package.

Call context propagation works by reading the deadline and cancellation token from the current gRPC request context and automatically propagating them to outgoing calls made by the gRPC client. Call context propagation is an excellent way of ensuring that complex, nested gRPC scenarios always propagate the deadline and cancellation.

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation();
```

By default, `EnableCallContextPropagation` raises an error if the client is used outside the context of a gRPC call. The error is designed to alert you that there isn't a call context to propagate. If you want to use the client outside of a call context, suppress the error when the client is configured with `SuppressContextNotFoundErrors`:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation(o => o.SuppressContextNotFoundErrors =
true);
```

For more information about deadlines and RPC cancellation, see Reliable gRPC services with deadlines and cancellation.

Named clients

Typically, a gRPC client type is registered once and then injected directly into a type's constructor by DI. However, there are scenarios where it's useful to have multiple configurations for one client. For example, a client that makes gRPC calls with and without authentication.

Multiple clients with the same type can be registered by giving each client a name. Each named client can have its own configuration. The generic `AddGrpcClient` extension method has an overload that includes a name parameter:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>("Greeter", o =>
    {
        o.Address = new Uri("https://localhost:5001");
    });

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>("GreeterAuthenticated", o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigureChannel(o =>
    {
        o.Credentials = new CustomCredentials();
    });
```

The preceding code:

- Registers the `GreeterClient` type twice, specifying a unique name for each.
- Configures different settings for each named client. The `GreeterAuthenticated` registration adds credentials to the channel so that gRPC calls made with it are authenticated.

A named gRPC client is created in app code using `GrpcClientFactory`. The type and name of the desired client is specified using the generic `GrpcClientFactory.CreateClient` method:

C#

```
public class AggregatorService : Aggregator.AggregatorBase
{
    private readonly Greeter.GreeterClient _client;

    public AggregatorService(GrpcClientFactory grpcClientFactory)
    {
        _client = grpcClientFactory.CreateClient<Greeter.GreeterClient>
("GreeterAuthenticated");
    }
}
```

Additional resources

- Call gRPC services with the .NET client
- Reliable gRPC services with deadlines and cancellation
- Make HTTP requests using IHttpClientFactory in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Reliable gRPC services with deadlines and cancellation

Article • 07/31/2024

By James Newton-King

Deadlines and cancellation are features used by gRPC clients to abort in-progress calls. This article discusses why deadlines and cancellation are important, and how to use them in .NET gRPC apps.

Deadlines

A deadline allows a gRPC client to specify how long it will wait for a call to complete. When a deadline is exceeded, the call is canceled. Setting a deadline is important because it provides an upper limit on how long a call can run for. It stops misbehaving services from running forever and exhausting server resources. Deadlines are a useful tool for building reliable apps and should be configured.

Deadline configuration:

- A deadline is configured using `CallOptions.Deadline` when a call is made.
- There is no default deadline value. gRPC calls aren't time limited unless a deadline is specified.
- A deadline is the UTC time of when the deadline is exceeded. For example, `DateTime.UtcNow.AddSeconds(5)` is a deadline of 5 seconds from now.
- If a past or current time is used then the call immediately exceeds the deadline.
- The deadline is sent with the gRPC call to the service and is independently tracked by both the client and the service. It is possible that a gRPC call completes on one machine, but by the time the response has returned to the client the deadline has been exceeded.

If a deadline is exceeded, the client and service have different behavior:

- The client immediately aborts the underlying HTTP request and throws a `DeadlineExceeded` error. The client app can choose to catch the error and display a timeout message to the user.
- On the server, the executing HTTP request is aborted and ServerCallContext.CancellationToken is raised. Although the HTTP request is aborted, the gRPC call continues to run on the server until the method completes. It's important that the cancellation token is passed to async methods so they are

cancelled along with the call. For example, passing a cancellation token to async database queries and HTTP requests. Passing a cancellation token allows the canceled call to complete quickly on the server and free up resources for other calls.

Configure `CallOptions.Deadline` to set a deadline for a gRPC call:

C#

```
var client = new Greet.GreeterClient(channel);

try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = "World" },
        deadline: DateTime.UtcNow.AddSeconds(5));

    // Greeting: Hello World
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded)
{
    Console.WriteLine("Greeting timeout.");
}
```

Using `ServerCallContext.CancellationToken` in a gRPC service:

C#

```
public override async Task<HelloReply> SayHello(HelloRequest request,
    ServerCallContext context)
{
    var user = await _databaseContext.GetUserAsync(request.Name,
        context.CancellationToken);

    return new HelloReply { Message = "Hello " + user.DisplayName };
}
```

Deadlines and retries

When a gRPC call is configured with retry fault handling and a deadline, the deadline tracks time across all retries for a gRPC call. If the deadline is exceeded, a gRPC call immediately aborts the underlying HTTP request, skips any remaining retries, and throws a `DeadlineExceeded` error.

Propagating deadlines

When a gRPC call is made from an executing gRPC service, the deadline should be propagated. For example:

1. Client app calls `FrontendService.GetUser` with a deadline.
2. `FrontendService` calls `UserService.GetUser`. The deadline specified by the client should be specified with the new gRPC call.
3. `UserService.GetUser` receives the deadline. It correctly times-out if the client app's deadline is exceeded.

The call context provides the deadline with `ServerCallContext.Deadline`:

C#

```
public override async Task<UserResponse> GetUser(UserRequest request,
    ServerCallContext context)
{
    var client = new User.UserServiceClient(_channel);
    var response = await client.GetUserAsync(
        new UserRequest { Id = request.Id },
        deadline: context.Deadline);

    return response;
}
```

Manually propagating deadlines can be cumbersome. The deadline needs to be passed to every call, and it's easy to accidentally miss. An automatic solution is available with gRPC client factory. Specifying `EnableCallContextPropagation`:

- Automatically propagates the deadline and cancellation token to child calls.
- Doesn't propagate the deadline if the child call specifies a smaller deadline. For example, a propagated deadline of 10 seconds isn't used if a child call specifies a new deadline of 5 seconds using `CallOptions.Deadline`. When multiple deadlines are available, the smallest deadline is used.
- Is an excellent way of ensuring that complex, nested gRPC scenarios always propagate the deadline and cancellation.

C#

```
services
    .AddGrpcClient<User.UserServiceClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .EnableCallContextPropagation();
```

For more information, see gRPC client factory integration in .NET.

Cancellation

Cancellation allows a gRPC client to cancel long running calls that are no longer needed. For example, a gRPC call that streams realtime updates is started when the user visits a page on a website. The stream should be canceled when the user navigates away from the page.

A gRPC call can be canceled in the client by passing a cancellation token with CallOptions.CancellationToken or calling `Dispose` on the call.

C#

```
private AsyncServerStreamingCall<HelloReply> _call;

public void StartStream()
{
    _call = client.SayHellos(new HelloRequest { Name = "World" });

    // Read response in background task.
    _ = Task.Run(async () =>
    {
        await foreach (var response in _call.ResponseStream.ReadAllAsync())
        {
            Console.WriteLine("Greeting: " + response.Message);
        }
    });
}

public void StopStream()
{
    _call.Dispose();
}
```

gRPC services that can be cancelled should:

- Pass `ServerCallContext.CancellationToken` to async methods. Canceling async methods allows the call on the server to complete quickly.
- Propagate the cancellation token to child calls. Propagating the cancellation token ensures that child calls are canceled with their parent. gRPC client factory and `EnableCallContextPropagation()` automatically propagates the cancellation token.

Additional resources

- Call gRPC services with the .NET client
- gRPC client factory integration in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Transient fault handling with gRPC retries

Article • 07/31/2024

By James Newton-King

gRPC retries is a feature that allows gRPC clients to automatically retry failed calls. This article discusses how to configure a retry policy to make resilient, fault tolerant gRPC apps in .NET.

gRPC retries requires Grpc.Net.Client version 2.36.0 or later.

Transient fault handling

gRPC calls can be interrupted by transient faults. Transient faults include:

- Momentary loss of network connectivity.
- Temporary unavailability of a service.
- Timeouts due to server load.

When a gRPC call is interrupted, the client throws an `RpcException` with details about the error. The client app must catch the exception and choose how to handle the error.

C#

```
var client = new Greeter.GreeterClient(channel);
try
{
    var response = await client.SayHelloAsync(
        new HelloRequest { Name = ".NET" });

    Console.WriteLine("From server: " + response.Message);
}
catch (RpcException ex)
{
    // Write logic to inspect the error and retry
    // if the error is from a transient fault.
}
```

Duplicating retry logic throughout an app is verbose and error prone. Fortunately the .NET gRPC client has a built-in support for automatic retries.

Configure a gRPC retry policy

A retry policy is configured once when a gRPC channel is created:

```C#
var defaultMethodConfig = new MethodConfig
{
    Names = { MethodName.Default },
    RetryPolicy = new RetryPolicy
    {
        MaxAttempts = 5,
        InitialBackoff = TimeSpan.FromSeconds(1),
        MaxBackoff = TimeSpan.FromSeconds(5),
        BackoffMultiplier = 1.5,
        RetryableStatusCodes = { StatusCode.Unavailable }
    }
};

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
{
    ServiceConfig = new ServiceConfig { MethodConfigs = {
defaultMethodConfig } }
});
```

The preceding code:

- Creates a `MethodConfig`. Retry policies can be configured per-method and methods are matched using the `Names` property. This method is configured with `MethodName.Default`, so it's applied to all gRPC methods called by this channel.
- Configures a retry policy. This policy instructs clients to automatically retry gRPC calls that fail with the status code `Unavailable`.
- Configures the created channel to use the retry policy by setting `GrpcChannelOptions.ServiceConfig`.

gRPC clients created with the channel will automatically retry failed calls:

```C#
var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(
    new HelloRequest { Name = ".NET" });

Console.WriteLine("From server: " + response.Message);
```

When retries are valid

Calls are retried when:

- The failing status code matches a value in `RetryableStatusCodes`.
- The previous number of attempts is less than `MaxAttempts`.
- The call hasn't been commited.
- The deadline hasn't been exceeded.

A gRPC call becomes committed in two scenarios:

- The client receives response headers. Response headers are sent by the server when `ServerCallContext.WriteResponseHeadersAsync` is called, or when the first message is written to the server response stream.
- The client's outgoing message (or messages if streaming) has exceeded the client's maximum buffer size. `MaxRetryBufferSize` and `MaxRetryBufferPerCallSize` are configured on the channel.

Committed calls won't retry, regardless of the status code or the previous number of attempts.

Streaming calls

Streaming calls can be used with gRPC retries, but there are important considerations when they are used together:

- **Server streaming**, **bidirectional streaming**: Streaming RPCs that return multiple messages from the server won't retry after the first message has been received. Apps must add additional logic to manually re-establish server and bidirectional streaming calls.
- **Client streaming**, **bidirectional streaming**: Streaming RPCs that send multiple messages to the server won't retry if the outgoing messages have exceeded the client's maximum buffer size. The maximum buffer size can be increased with configuration.

For more information, see When retries are valid.

Retry backoff delay

The backoff delay between retry attempts is configured with `InitialBackoff`, `MaxBackoff`, and `BackoffMultiplier`. More information about each option is available in the gRPC retry options section.

The actual delay between retry attempts is randomized. A randomized delay between 0 and the current backoff determines when the next retry attempt is made. Consider that even with exponential backoff configured, increasing the current backoff between

attempts, the actual delay between attempts isn't always larger. The delay is randomized to prevent retries from multiple calls from clustering together and potentially overloading the server.

Detect retries with metadata

gRPC retries can be detected by the presence of `grpc-previous-rpc-attempts` metadata. The `grpc-previous-rpc-attempts` metadata:

- Is automatically added to retried calls and sent to the server.
- Value represents the number of preceding retry attempts.
- Value is always an integer.

Consider the following retry scenario:

1. Client makes a gRPC call to the server.
2. Server fails and returns a retriable status code response.
3. Client retries the gRPC call. Because there was one previous attempt, `grpc-previous-rpc-attempts` metadata has a value of `1`. Metadata is sent to the server with the retry.
4. Server succeeds and returns OK.
5. Client reports success. `grpc-previous-rpc-attempts` is in the response metadata and has a value of `1`.

The `grpc-previous-rpc-attempts` metadata is not present on the initial gRPC call, is `1` for the first retry, `2` for the second retry, and so on.

gRPC retry options

The following table describes options for configuring gRPC retry policies:

Expand table

Option	Description
`MaxAttempts`	The maximum number of call attempts, including the original attempt. This value is limited by `GrpcChannelOptions.MaxRetryAttempts` which defaults to 5. A value is required and must be greater than 1.
`InitialBackoff`	The initial backoff delay between retry attempts. A randomized delay between 0 and the current backoff determines when the next retry attempt is made. After each attempt, the current backoff is multiplied by `BackoffMultiplier`. A value is required and must be greater than zero.

Option	Description
`MaxBackoff`	The maximum backoff places an upper limit on exponential backoff growth. A value is required and must be greater than zero.
`BackoffMultiplier`	The backoff will be multiplied by this value after each retry attempt and will increase exponentially when the multiplier is greater than 1. A value is required and must be greater than zero.
`RetryableStatusCodes`	A collection of status codes. A gRPC call that fails with a matching status will be automatically retried. For more information about status codes, see Status codes and their use in gRPC . At least one retryable status code is required.

Hedging

Hedging is an alternative retry strategy. Hedging enables aggressively sending multiple copies of a single gRPC call without waiting for a response. Hedged gRPC calls may be executed multiple times on the server and the first successful result is used. It's important that hedging is only enabled for methods that are safe to execute multiple times without adverse effect.

Hedging has pros and cons when compared to retries:

- An advantage to hedging is it might return a successful result faster. It allows for multiple simultaneously gRPC calls and will complete when the first successful result is available.
- A disadvantage to hedging is it can be wasteful. Multiple calls could be made and all succeed. Only the first result is used and the rest are discarded.

Configure a gRPC hedging policy

A hedging policy is configured like a retry policy. Note that a hedging policy can't be combined with a retry policy.

C#

```
var defaultMethodConfig = new MethodConfig
{
    Names = { MethodName.Default },
    HedgingPolicy = new HedgingPolicy
    {
        MaxAttempts = 5,
        NonFatalStatusCodes = { StatusCode.Unavailable }
    }
```

```
};

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
{
    ServiceConfig = new ServiceConfig { MethodConfigs = {
defaultMethodConfig } }
});
```

gRPC hedging options

The following table describes options for configuring gRPC hedging policies:

Expand table

Option	Description
`MaxAttempts`	The hedging policy will send up to this number of calls. `MaxAttempts` represents the total number of all attempts, including the original attempt. This value is limited by `GrpcChannelOptions.MaxRetryAttempts` which defaults to 5. A value is required and must be 2 or greater.
`HedgingDelay`	The first call is sent immediately, subsequent hedging calls are delayed by this value. When the delay is set to zero or `null`, all hedged calls are sent immediately. `HedgingDelay` is optional and defaults to zero. A value must be zero or greater.
`NonFatalStatusCodes`	A collection of status codes which indicate other hedge calls may still succeed. If a non-fatal status code is returned by the server, hedged calls will continue. Otherwise, outstanding requests will be canceled and the error returned to the app. For more information about status codes, see Status codes and their use in gRPC.

Additional resources

- Call gRPC services with the .NET client
- Retry general guidance - Best practices for cloud applications

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

gRPC client-side load balancing

Article • 07/31/2024

By James Newton-King

Client-side load balancing is a feature that allows gRPC clients to distribute load optimally across available servers. This article discusses how to configure client-side load balancing to create scalable, high-performance gRPC apps in .NET.

Client-side load balancing requires:

- .NET 5 or later.
- Grpc.Net.Client version 2.45.0 or later.

Configure gRPC client-side load balancing

Client-side load balancing is configured when a channel is created. The two components to consider when using load balancing:

- The resolver, which resolves the addresses for the channel. Resolvers support getting addresses from an external source. This is also known as service discovery.
- The load balancer, which creates connections and picks the address that a gRPC call will use.

Built-in implementations of resolvers and load balancers are included in Grpc.Net.Client. Load balancing can also be extended by writing custom resolvers and load balancers.

Addresses, connections and other load balancing state is stored in a `GrpcChannel` instance. A channel must be reused when making gRPC calls for load balancing to work correctly.

> **Note**
>
> Some load balancing configuration uses dependency injection (DI). Apps that don't use DI can create a ServiceCollection instance.
>
> If an app already has DI setup, like an ASP.NET Core website, then types should be registered with the existing DI instance. `GrpcChannelOptions.ServiceProvider` is configured by getting an IServiceProvider from DI.

Configure resolver

The resolver is configured using the address a channel is created with. The URI scheme of the address specifies the resolver.

Expand table

Scheme	Type	Description
`dns`	`DnsResolverFactory`	Resolves addresses by querying the hostname for DNS address records.
`static`	`StaticResolverFactory`	Resolves addresses that the app has specified. Recommended if an app already knows the addresses it calls.

A channel doesn't directly call a URI that matches a resolver. Instead, a matching resolver is created and used to resolve the addresses.

For example, using `GrpcChannel.ForAddress("dns:///my-example-host", new GrpcChannelOptions { Credentials = ChannelCredentials.Insecure })`:

- The `dns` scheme maps to `DnsResolverFactory`. A new instance of a DNS resolver is created for the channel.
- The resolver makes a DNS query for `my-example-host` and gets two results: `127.0.0.100` and `127.0.0.101`.
- The load balancer uses `127.0.0.100:80` and `127.0.0.101:80` to create connections and make gRPC calls.

DnsResolverFactory

The `DnsResolverFactory` creates a resolver designed to get addresses from an external source. DNS resolution is commonly used to load balance over pod instances that have a Kubernetes headless services.

C#

```
var channel = GrpcChannel.ForAddress(
    "dns:///my-example-host",
    new GrpcChannelOptions { Credentials = ChannelCredentials.Insecure });
var client = new Greet.GreeterClient(channel);

var response = await client.SayHelloAsync(new HelloRequest { Name = "world"
});
```

The preceding code:

- Configures the created channel with the address `dns:///my-example-host`.
  - The `dns` scheme maps to `DnsResolverFactory`.
  - `my-example-host` is the hostname to resolve.
  - No port is specified in the address, so gRPC calls are sent to port 80. This is the default port for unsecured channels. A port can optionally be specified after the hostname. For example, `dns:///my-example-host:8080` configures gRPC calls to be sent to port 8080.
- Doesn't specify a load balancer. The channel defaults to a pick first load balancer.
- Starts the gRPC call `SayHello`:
  - DNS resolver gets addresses for the hostname `my-example-host`.
  - Pick first load balancer attempts to connect to one of the resolved addresses.
  - The call is sent to the first address the channel successfully connects to.

DNS address caching

Performance is important when load balancing. The latency of resolving addresses is eliminated from gRPC calls by caching the addresses. A resolver will be invoked when making the first gRPC call, and subsequent calls use the cache.

Addresses are automatically refreshed if a connection is interrupted. Refreshing is important in scenarios where addresses change at runtime. For example, in Kubernetes a restarted pod triggers the DNS resolver to refresh and get the pod's new address.

By default, a DNS resolver is refreshed if a connection is interrupted. The DNS resolver can also optionally refresh itself on a periodic interval. This can be useful for quickly detecting new pod instances.

C#

```
services.AddSingleton<ResolverFactory>(
    sp => new DnsResolverFactory(refreshInterval:
TimeSpan.FromSeconds(30)));
```

The preceding code creates a `DnsResolverFactory` with a refresh interval and registers it with dependency injection. For more information on using a custom-configured resolver, see Configure custom resolvers and load balancers.

StaticResolverFactory

A static resolver is provided by `StaticResolverFactory`. This resolver:

- Doesn't call an external source. Instead, the client app configures the addresses.
- Is designed for situations where an app already knows the addresses it calls.

C#

```
var factory = new StaticResolverFactory(addr => new[]
{
    new BalancerAddress("localhost", 80),
    new BalancerAddress("localhost", 81)
});

var services = new ServiceCollection();
services.AddSingleton<ResolverFactory>(factory);

var channel = GrpcChannel.ForAddress(
    "static:///my-example-host",
    new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Insecure,
        ServiceProvider = services.BuildServiceProvider()
    });
var client = new Greet.GreeterClient(channel);
```

The preceding code:

- Creates a `StaticResolverFactory`. This factory knows about two addresses: `localhost:80` and `localhost:81`.
- Registers the factory with dependency injection (DI).
- Configures the created channel with:
  - The address `static:///my-example-host`. The `static` scheme maps to a static resolver.
  - Sets `GrpcChannelOptions.ServiceProvider` with the DI service provider.

This example creates a new ServiceCollection for DI. Suppose an app already has DI setup, like an ASP.NET Core website. In that case, types should be registered with the existing DI instance. `GrpcChannelOptions.ServiceProvider` is configured by getting an IServiceProvider from DI.

Configure load balancer

A load balancer is specified in a service config using the `ServiceConfig.LoadBalancingConfigs` collection. Two load balancers are built-in and map to load balancer config names:

Expand table

Name	Type	Description
`pick_first`	`PickFirstLoadBalancerFactory`	Attempts to connect to addresses until a connection is successfully made. gRPC calls are all made to the first successful connection.
`round_robin`	`RoundRobinLoadBalancerFactory`	Attempts to connect to all addresses. gRPC calls are distributed across all successful connections using round-robin logic.

`service config` is an abbreviation of service configuration and is represented by the `ServiceConfig` type. There are a couple of ways a channel can get a `service config` with a load balancer configured:

- An app can specify a `service config` when a channel is created using `GrpcChannelOptions.ServiceConfig`.
- Alternatively, a resolver can resolve a `service config` for a channel. This feature allows an external source to specify how its callers should perform load balancing. Whether a resolver supports resolving a `service config` is dependent on the resolver implementation. Disable this feature with `GrpcChannelOptions.DisableResolverServiceConfig`.
- If no `service config` is provided, or the `service config` doesn't have a load balancer configured, the channel defaults to `PickFirstLoadBalancerFactory`.

C#

```
var channel = GrpcChannel.ForAddress(
    "dns:///my-example-host",
    new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Insecure,
        ServiceConfig = new ServiceConfig { LoadBalancingConfigs = { new
RoundRobinConfig() } }
    });
var client = new Greet.GreeterClient(channel);

var response = await client.SayHelloAsync(new HelloRequest { Name = "world"
});
```

The preceding code:

- Specifies a `RoundRobinLoadBalancerFactory` in the `service config`.
- Starts the gRPC call `SayHello`:
  - `DnsResolverFactory` creates a resolver that gets addresses for the hostname `my-example-host`.

- Round-robin load balancer attempts to connect to all resolved addresses.
- gRPC calls are distributed evenly using round-robin logic.

Configure channel credentials

A channel must know whether gRPC calls are sent using transport security. `http` and `https` are no longer part of the address, the scheme now specifies a resolver, so `Credentials` must be configured on channel options when using load balancing.

- `ChannelCredentials.SecureSsl` - gRPC calls are secured with Transport Layer Security (TLS). Equivalent to an `https` address.
- `ChannelCredentials.Insecure` - gRPC calls don't use transport security. Equivalent to an `http` address.

C#

```
var channel = GrpcChannel.ForAddress(
    "dns:///my-example-host",
    new GrpcChannelOptions { Credentials = ChannelCredentials.Insecure });
var client = new Greet.GreeterClient(channel);

var response = await client.SayHelloAsync(new HelloRequest { Name = "world"
});
```

Use load balancing with gRPC client factory

gRPC client factory can be configured to use load balancing:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("dns:///my-example-host");
    })
    .ConfigureChannel(o => o.Credentials = ChannelCredentials.Insecure);

builder.Services.AddSingleton<ResolverFactory>(
    sp => new DnsResolverFactory(refreshInterval:
TimeSpan.FromSeconds(30)));

var app = builder.Build();
```

The preceding code:

- Configures the client with a load-balancing address.
- Specifies channel credentials.
- Registers DI types with the app's IServiceCollection.

Write custom resolvers and load balancers

Client-side load balancing is extensible:

- Implement `Resolver` to create a custom resolver and resolve addresses from a new data source.
- Implement `LoadBalancer` to create a custom load balancer with new load balancing behavior.

> **ⓘ Important**
>
> The APIs used to extend client-side load balancing are experimental. They can change without notice.

Create a custom resolver

A resolver:

- Implements `Resolver` and is created by a `ResolverFactory`. Create a custom resolver by implementing these types.
- Is responsible for resolving the addresses a load balancer uses.
- Can optionally provide a service configuration.

C#

```
public class FileResolver : PollingResolver
{
    private readonly Uri _address;
    private readonly int _port;

    public FileResolver(Uri address, int defaultPort, ILoggerFactory
loggerFactory)
        : base(loggerFactory)
    {
        _address = address;
        _port = defaultPort;
    }
```

```
    public override async Task ResolveAsync(CancellationToken
cancellationToken)
    {
        // Load JSON from a file on disk and deserialize into endpoints.
        var jsonString = await File.ReadAllTextAsync(_address.LocalPath);
        var results = JsonSerializer.Deserialize<string[]>(jsonString);
        var addresses = results.Select(r => new BalancerAddress(r,
_port)).ToArray();

        // Pass the results back to the channel.
        Listener(ResolverResult.ForResult(addresses));
    }
}

public class FileResolverFactory : ResolverFactory
{
    // Create a FileResolver when the URI has a 'file' scheme.
    public override string Name => "file";

    public override Resolver Create(ResolverOptions options)
    {
        return new FileResolver(options.Address, options.DefaultPort,
options.LoggerFactory);
    }
}
```

In the preceding code:

- `FileResolverFactory` implements `ResolverFactory`. It maps to the `file` scheme and creates `FileResolver` instances.
- `FileResolver` implements `PollingResolver`. `PollingResolver` is an abstract base type that makes it easy to implement a resolver with asynchronous logic by overriding `ResolveAsync`.
- In `ResolveAsync`:
  - The file URI is converted to a local path. For example, `file:///c:/addresses.json` becomes `c:\addresses.json`.
  - JSON is loaded from disk and converted into a collection of addresses.
  - Listener is called with results to let the channel know that addresses are available.

Create a custom load balancer

A load balancer:

- Implements `LoadBalancer` and is created by a `LoadBalancerFactory`. Create a custom load balancer and factory by implementing these types.
- Is given addresses from a resolver and creates `Subchannel` instances.

- Tracks state about the connection and creates a `SubchannelPicker`. The channel internally uses the picker to pick addresses when making gRPC calls.

The `SubchannelsLoadBalancer` is:

- An abstract base class that implements `LoadBalancer`.
- Manages creating `Subchannel` instances from addresses.
- Makes it easy to implement a custom picking policy over a collection of subchannels.

C#

```
public class RandomBalancer : SubchannelsLoadBalancer
{
    public RandomBalancer(IChannelControlHelper controller, ILoggerFactory
loggerFactory)
        : base(controller, loggerFactory)
    {
    }

    protected override SubchannelPicker CreatePicker(List<Subchannel>
readySubchannels)
    {
        return new RandomPicker(readySubchannels);
    }

    private class RandomPicker : SubchannelPicker
    {
        private readonly List<Subchannel> _subchannels;

        public RandomPicker(List<Subchannel> subchannels)
        {
            _subchannels = subchannels;
        }

        public override PickResult Pick(PickContext context)
        {
            // Pick a random subchannel.
            return
PickResult.ForSubchannel(_subchannels[Random.Shared.Next(0,
_subchannels.Count)]);
        }
    }
}

public class RandomBalancerFactory : LoadBalancerFactory
{
    // Create a RandomBalancer when the name is 'random'.
    public override string Name => "random";

    public override LoadBalancer Create(LoadBalancerOptions options)
    {
```

```
        return new RandomBalancer(options.Controller,
options.LoggerFactory);
    }
}
```

In the preceding code:

- `RandomBalancerFactory` implements `LoadBalancerFactory`. It maps to the `random` policy name and creates `RandomBalancer` instances.
- `RandomBalancer` implements `SubchannelsLoadBalancer`. It creates a `RandomPicker` that randomly picks a subchannel.

Configure custom resolvers and load balancers

Custom resolvers and load balancers need to be registered with dependency injection (DI) when they are used. There are a couple of options:

- If an app is already using DI, such as an ASP.NET Core web app, they can be registered with the existing DI configuration. An IServiceProvider can be resolved from DI and passed to the channel using `GrpcChannelOptions.ServiceProvider`.
- If an app isn't using DI then create:
  - A ServiceCollection with types registered with it.
  - A service provider using BuildServiceProvider.

C#

```
var services = new ServiceCollection();
services.AddSingleton<ResolverFactory, FileResolverFactory>();
services.AddSingleton<LoadBalancerFactory, RandomLoadBalancerFactory>();

var channel = GrpcChannel.ForAddress(
    "file:///c:/data/addresses.json",
    new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Insecure,
        ServiceConfig = new ServiceConfig { LoadBalancingConfigs = { new
LoadBalancingConfig("random") } },
        ServiceProvider = services.BuildServiceProvider()
    });
var client = new Greet.GreeterClient(channel);
```

The preceding code:

- Creates a `ServiceCollection` and registers new resolver and load balancer implementations.

- Creates a channel configured to use the new implementations:
  - `ServiceCollection` is built into an `IServiceProvider` and set to `GrpcChannelOptions.ServiceProvider`.
  - Channel address is `file:///c:/data/addresses.json`. The `file` scheme maps to `FileResolverFactory`.
  - `service config` load balancer name is `random`. Maps to `RandomLoadBalancerFactory`.

Why load balancing is important

HTTP/2 multiplexes multiple calls on a single TCP connection. If gRPC and HTTP/2 are used with a network load balancer (NLB), the connection is forwarded to a server, and all gRPC calls are sent to that one server. The other server instances on the NLB are idle.

Network load balancers are a common solution for load balancing because they are fast and lightweight. For example, Kubernetes by default uses a network load balancer to balance connections between pod instances. However, network load balancers are not effective at distributing load when used with gRPC and HTTP/2.

Proxy or client-side load balancing?

gRPC and HTTP/2 can be effectively load balanced using either an application load balancer proxy or client-side load balancing. Both of these options allow individual gRPC calls to be distributed across available servers. Deciding between proxy and client-side load balancing is an architectural choice. There are pros and cons for each.

- **Proxy**: gRPC calls are sent to the proxy, the proxy makes a load balancing decision, and the gRPC call is sent on to the final endpoint. The proxy is responsible for knowing about endpoints. Using a proxy adds:
  - An additional network hop to gRPC calls.
  - Latency and consumes additional resources.
  - Proxy server must be setup and configured correctly.

- **Client-side load balancing**: The gRPC client makes a load balancing decision when a gRPC call is started. The gRPC call is sent directly to the final endpoint. When using client-side load balancing:
  - The client is responsible for knowing about available endpoints and making load balancing decisions.
  - Additional client configuration is required.
  - High-performance, load balanced gRPC calls eliminate the need for a proxy.

Additional resources

- Call gRPC services with the .NET client

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use gRPC client with .NET Standard 2.0

Article • 01/16/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By James Newton-King

This article discusses how to use the .NET gRPC client with .NET implementations that support .NET Standard 2.0.

.NET implementations

The following .NET implementations (or later) support Grpc.Net.Client but don't have full support for HTTP/2:

- .NET Core 2.1
- .NET Framework 4.6.1
- Mono 5.4
- Universal Windows Platform 10.0.16299
- Unity 2018.1

The .NET gRPC client can call services from these .NET implementations with some additional configuration.

HttpHandler configuration

An HTTP provider must be configured using `GrpcChannelOptions.HttpHandler`. If a handler isn't configured, an error is thrown:

> `System.PlatformNotSupportedException`: gRPC requires extra configuration to successfully make RPC calls on .NET implementations that don't have support for gRPC over HTTP/2. An HTTP provider must be specified using `GrpcChannelOptions.HttpHandler`. The configured HTTP provider must either support HTTP/2 or be configured to use gRPC-Web.

.NET implementations that don't support HTTP/2, such as UWP and Unity, can use gRPC-Web as an alternative.

```C#
var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
    {
        HttpHandler = new GrpcWebHandler(new HttpClientHandler())
    });

var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = ".NET"
});
```

Clients can also be created using the gRPC client factory. An HTTP provider is configured using the ConfigurePrimaryHttpMessageHandler extension method.

```C#
builder.Services
    .AddGrpcClient<Greet.GreeterClient>(options =>
    {
        options.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(
        () => new GrpcWebHandler(new HttpClientHandler()));
```

For more information, see Configure gRPC-Web with the .NET gRPC client.

> ⓘ **Important**
>
> gRPC-Web requires the client ***and*** server to support it. gRPC-Web can be **quickly configured by an ASP.NET Core gRPC server**. Other gRPC server implementations require a proxy to support gRPC-Web.

.NET Framework

.NET Framework has limited support for gRPC over HTTP/2. To enable gRPC over HTTP/2 on .NET Framework, configure the channel to use WinHttpHandler.

Requirements and restrictions to using `WinHttpHandler`:

- Windows 11 or later, Windows Server 2019 or later.
  - gRPC client is fully supported on Windows 11 or later.

    - gRPC client is partially supported on Windows Server 2019 and Windows Server 2022. Unary and server streaming methods are supported. Client and bidirectional streaming methods are ***not*** supported.
- A reference to System.Net.Http.WinHttpHandler version 6.0.1 or later.
- Configure `WinHttpHandler` on the channel using `GrpcChannelOptions.HttpHandler`.
- .NET Framework 4.6.1 or later.
- Only gRPC calls over TLS are supported.

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
    {
        HttpHandler = new WinHttpHandler()
    });

var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(new HelloRequest { Name = ".NET"
});
```

gRPC C# core-library

An alternative option for .NET Framework has been to use gRPC C# core-library to make gRPC calls. gRPC C# core-library is:

- A third party library that supports making gRPC calls over HTTP/2 on .NET Framework.
- Not supported by Microsoft.
- In maintenance mode and will be deprecated in favour of gRPC for .NET.
- Not recommended for new apps.

Additional resources

- Call gRPC services with the .NET client
- Use gRPC in browser apps
- gRPC C# core-library

Mock gRPC client in tests

Article • 07/31/2024

By: James Newton-King

Testing is an important aspect of building stable and maintainable software. Part of writing high-quality tests is removing external dependencies. This article discusses using mock gRPC clients in tests to remove gRPC calls to external servers.

Example testable client app

To demonstrate client app tests, review the following type in the sample app.

View or download sample code (how to download)

The `Worker` is a BackgroundService that makes calls to a gRPC server.

C#

```
public class Worker : BackgroundService
{
    private readonly Tester.TesterClient _client;
    private readonly IGreetRepository _greetRepository;

    public Worker(Tester.TesterClient client, IGreetRepository
greetRepository)
    {
        _client = client;
        _greetRepository = greetRepository;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        var count = 0;
        while (!stoppingToken.IsCancellationRequested)
        {
            count++;

            var reply = await _client.SayHelloUnaryAsync(
                new HelloRequest { Name = $"Worker {count}" });

            _greetRepository.SaveGreeting(reply.Message);

            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

The preceding type:

- Follows the Explicit Dependencies Principle.
- `TesterClient` is generated automatically by the tooling package Grpc.Tools based on the *test.proto* file, during the build process.
- Expects dependency injection (DI) to provide instances of `TesterClient` and `IGreetRepository`. The app is configured to use the gRPC client factory to create `TesterClient`.
- Can be tested with a mocked `IGreetRepository` service and `TesterClient` client using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Integration tests in ASP.NET Core.

For more information on the C# assets automatically generated by Grpc.Tools, see gRPC services with C#: Generated C# assets.

Mock a gRPC client

gRPC clients are concrete client types that are generated from .proto files. The concrete gRPC client has methods that translate to the gRPC service in the `.proto` file. For example, a service called `Greeter` generates a `GreeterClient` type with methods to call the service.

A mocking framework can mock a gRPC client type. When a mocked client is passed to the type, the test uses the mocked method instead of sending a gRPC call to a server.

C#

```
[Fact]
public async Task Greeting_Success_RepositoryCalled()
{
    // Arrange
    var mockRepository = new Mock<IGreetRepository>();

    var mockCall = CallHelpers.CreateAsyncUnaryCall(new HelloReply { Message
= "Test" });
    var mockClient = new Mock<Tester.TesterClient>();
    mockClient
        .Setup(m => m.SayHelloUnaryAsync(
            It.IsAny<HelloRequest>(), null, null, CancellationToken.None))
        .Returns(mockCall);

    var worker = new Worker(mockClient.Object, mockRepository.Object);

    // Act
    await worker.StartAsync(CancellationToken.None);
```

```
    // Assert
    mockRepository.Verify(v => v.SaveGreeting("Test"));
}
```

The preceding unit test:

- Mocks `IGreetRepository` and `TesterClient` using Moq.
- Starts the worker.
- Verifies `SaveGreeting` is called with the greeting message returned by the mocked `TesterClient`.

Additional resources

- Test gRPC services with gRPCurl and gRPCui in ASP.NET Core
- Test gRPC services in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

gRPC services with ASP.NET Core

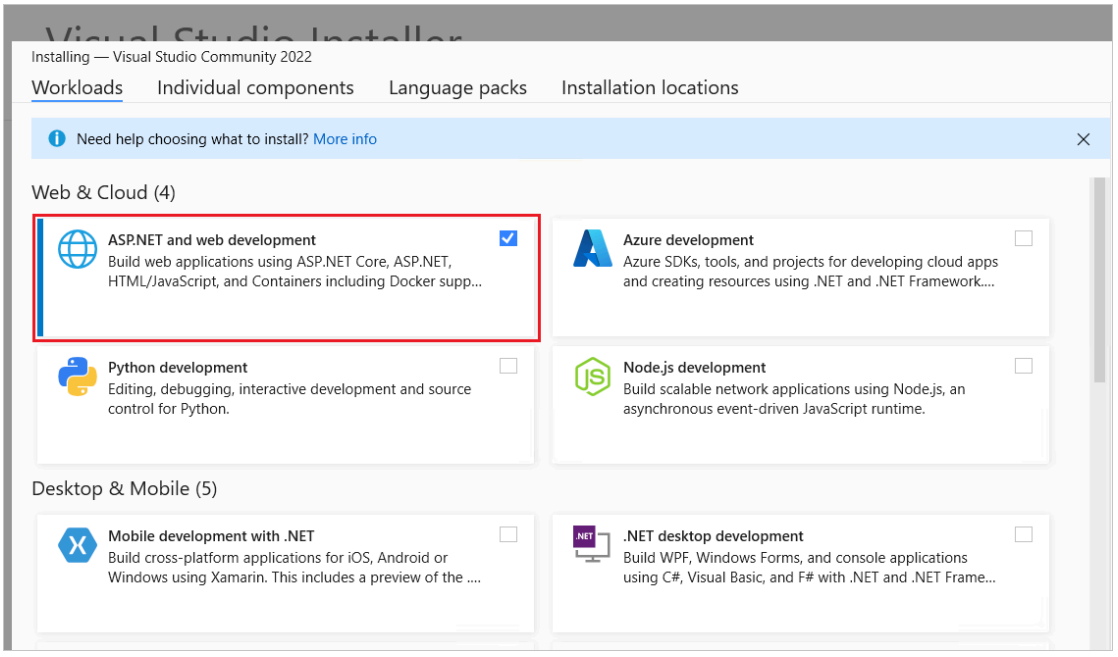
Article • 09/25/2023

This document shows how to get started with gRPC services using ASP.NET Core.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Get started with gRPC service in ASP.NET Core

View or download sample code (how to download).

Visual Studio

See Get started with gRPC services for detailed instructions on how to create a gRPC project.

Add gRPC services to an ASP.NET Core app

gRPC requires the Grpc.AspNetCore package.

Configure gRPC

In `Program.cs`:

- gRPC is enabled with the `AddGrpc` method.
- Each gRPC service is added to the routing pipeline through the `MapGrpcService` method.

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

// Additional configuration is required to successfully run gRPC on macOS.
// For instructions on how to configure Kestrel and gRPC clients on macOS,
visit https://go.microsoft.com/fwlink/?linkid=2099682

// Add services to the container.
builder.Services.AddGrpc();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.MapGrpcService<GreeterService>();
app.MapGet("/", () => "Communication with gRPC endpoints must be made
through a gRPC client. To learn how to create a client, visit:
https://go.microsoft.com/fwlink/?linkid=2086909");

app.Run();
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

ASP.NET Core middleware and features share the routing pipeline, therefore an app can be configured to serve additional request handlers. The additional request handlers, such as MVC controllers, work in parallel with the configured gRPC services.

Server options

gRPC services can be hosted by all built-in ASP.NET Core servers.

- ✓ Kestrel
- ✓ TestServer

✔ IIS†
✔ HTTP.sys†

†Requires .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later.

For more information about choosing the right server for an ASP.NET Core app, see Web server implementations in ASP.NET Core.

Kestrel

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel focuses on high performance and memory utilization, but it doesn't have some of the advanced features in HTTP.sys such as port sharing.

Kestrel gRPC endpoints:

- Require HTTP/2.
- Should be secured with Transport Layer Security (TLS).

HTTP/2

gRPC requires HTTP/2. gRPC for ASP.NET Core validates HttpRequest.Protocol is `HTTP/2`.

Kestrel supports HTTP/2 on most modern operating systems. Kestrel endpoints are configured to support HTTP/1.1 and HTTP/2 connections by default.

TLS

Kestrel endpoints used for gRPC should be secured with TLS. In development, an endpoint secured with TLS is automatically created at `https://localhost:5001` when the ASP.NET Core development certificate is present. No configuration is required. An `https` prefix verifies the Kestrel endpoint is using TLS.

In production, TLS must be explicitly configured. In the following `appsettings.json` example, an HTTP/2 endpoint secured with TLS is provided:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
```

```
        "Protocols": "Http2",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      }
    }
  }
}
```

Alternatively, Kestrel endpoints can be configured in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>
{
    options.Listen(IPAddress.Any, 5001, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
        listenOptions.UseHttps("<path to .pfx file>",
            "<certificate password>");
    });
});
```

For more information on enabling TLS with Kestrel, see Kestrel HTTPS endpoint configuration.

Protocol negotiation

TLS is used for more than securing communication. The TLS Application-Layer Protocol Negotiation (ALPN) handshake is used to negotiate the connection protocol between the client and the server when an endpoint supports multiple protocols. This negotiation determines whether the connection uses HTTP/1.1 or HTTP/2.

If an HTTP/2 endpoint is configured without TLS, the endpoint's ListenOptions.Protocols must be set to `HttpProtocols.Http2`. An endpoint with multiple protocols, such as `HttpProtocols.Http1AndHttp2` for example, can't be used without TLS because there's no negotiation. All connections to the unsecured endpoint default to HTTP/1.1, and gRPC calls fail.

For more information on enabling HTTP/2 and TLS with Kestrel, see Kestrel endpoint configuration.

> **Note**

macOS doesn't support ASP.NET Core gRPC with TLS before .NET 8. Additional configuration is required to successfully run gRPC services on macOS when using .NET 7 or earlier. For more information, see Unable to start ASP.NET Core gRPC app on macOS.

IIS

Internet Information Services (IIS) is a flexible, secure and manageable Web Server for hosting web apps, including ASP.NET Core. .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later are required to host gRPC services with IIS.

IIS must be configured to use TLS and HTTP/2. For more information, see Use ASP.NET Core with HTTP/2 on IIS.

HTTP.sys

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later are required to host gRPC services with HTTP.sys.

HTTP.sys must be configured to use TLS and HTTP/2. For more information, see HTTP.sys web server HTTP/2 support.

Host gRPC in non-ASP.NET Core projects

An ASP.NET Core gRPC server is typically created from the gRPC template. The project file created by the template uses `Microsoft.NET.SDK.Web` as the SDK:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.47.0" />
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  </ItemGroup>

</Project>
```

The `Microsoft.NET.SDK.Web` SDK value automatically adds a reference to the ASP.NET Core framework. The reference allows the app to use ASP.NET Core types required to host a server.

You can add a gRPC server to non-ASP.NET Core projects with the following project file settings:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.47.0" />
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />

    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

The preceding project file:

- Doesn't use `Microsoft.NET.SDK.Web` as the SDK.
- Adds a framework reference to `Microsoft.AspNetCore.App`.
  - The framework reference allows non-ASP.NET Core apps, such as Windows Services, WPF apps, or WinForms apps to use ASP.NET Core APIs.
  - The app can now use ASP.NET Core APIs to start an ASP.NET Core server.
- Adds gRPC requirements:
  - NuGet package reference to Grpc.AspNetCore.
  - `.proto` file.

For more information about using the `Microsoft.AspNetCore.App` framework reference, see Use the ASP.NET Core shared framework.

Integration with ASP.NET Core APIs

gRPC services have full access to the ASP.NET Core features such as Dependency Injection (DI) and Logging. For example, the service implementation can resolve a logger service from the DI container via the constructor:

C#

```csharp
public class GreeterService : Greeter.GreeterBase
{
    public GreeterService(ILogger<GreeterService> logger)
    {
    }
}
```

By default, the gRPC service implementation can resolve other DI services with any lifetime (Singleton, Scoped, or Transient).

Resolve HttpContext in gRPC methods

The gRPC API provides access to some HTTP/2 message data, such as the method, host, header, and trailers. Access is through the `ServerCallContext` argument passed to each gRPC method:

C#

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(
        HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name
        });
    }
}
```

`ServerCallContext` doesn't provide full access to `HttpContext` in all ASP.NET APIs. The `GetHttpContext` extension method provides full access to the `HttpContext` representing the underlying HTTP/2 message in ASP.NET APIs:

C#

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(
        HelloRequest request, ServerCallContext context)
    {
        var httpContext = context.GetHttpContext();
        var clientCertificate = httpContext.Connection.ClientCertificate;

        return Task.FromResult(new HelloReply
        {
            Message = "Hello " + request.Name + " from " +
clientCertificate.Issuer
        });
    }
}
```

Additional resources

- Create a .NET Core gRPC client and server in ASP.NET Core
- Overview for gRPC on .NET
- gRPC services with C#
- Kestrel web server in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

gRPC on .NET supported platforms

Article • 01/16/2025

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By James Newton-King

This article discusses the requirements and supported platforms for using gRPC with .NET. There are different requirements for the two major gRPC workloads:

- Hosting gRPC services in ASP.NET Core
- Calling gRPC from .NET client apps

Wire-formats

gRPC takes advantage of advanced features available in HTTP/2. HTTP/2 isn't supported everywhere, but a second wire-format using HTTP/1.1 is available for gRPC:

- application/grpc - gRPC over HTTP/2 is how gRPC is typically used.
- application/grpc-web - gRPC-Web modifies the gRPC protocol to be compatible with HTTP/1.1. gRPC-Web can be used in more places. gRPC-Web can be used by browser apps and in networks without complete support for HTTP/2. Two advanced gRPC features are no longer supported: client streaming and bidirectional streaming.

gRPC on .NET supports both wire-formats. `application/grpc` is used by default. gRPC-Web must be configured on the client and the server for successful gRPC-Web calls. For information on setting up gRPC-Web, see gRPC-Web in ASP.NET Core gRPC apps.

ASP.NET Core gRPC server requirements

Hosting gRPC services with ASP.NET Core requires .NET Core 3.x or later.

- ✓ .NET 5 or later
- ✓ .NET Core 3

ASP.NET Core gRPC services can be hosted on all operating system that .NET Core supports.

- ✔ Windows
- ✔ Linux
- ✔ macOS

Supported ASP.NET Core servers

All built-in ASP.NET Core servers are supported.

- ✔ Kestrel
- ✔ TestServer
- ✔ IIS†
- ✔ HTTP.sys†

†Requires .NET 5 and Windows 11 Build 22000 or Windows Server 2022 Build 20348 or later.

For information about configuring ASP.NET Core servers to run gRPC, see gRPC services with ASP.NET Core.

Azure services

- ✔ Azure Kubernetes Service (AKS)
- ✔ Azure Container Apps
- ✔ Azure App Service †

†gRPC requires a Linux-based environment on Azure App Service. See How-to deploy a .NET 6 gRPC app on App Service for Azure App Service deployment information.

.NET gRPC client requirements

The Grpc.Net.Client package supports gRPC calls over HTTP/2 on .NET Core 3 and .NET 5 or later.

Limited support is available for gRPC over HTTP/2 on .NET Framework. Other .NET versions such as UWP and Unity don't have required HTTP/2 support, and must use gRPC-Web instead.

The following table lists .NET implementations and their gRPC client support:

Expand table

.NET implementation	gRPC over HTTP/2	gRPC-Web
.NET 5 or later	✔	✔
.NET Core 3	✔	✔
.NET Core 2.1	✘	✔
.NET Framework 4.6.1	⚠†	✔
Blazor WebAssembly	✘	✔
Mono 5.4	✘	✔
Universal Windows Platform 10.0.16299	✘	✔
Unity 2018.1	✘	✔

†.NET Framework requires configuration of WinHttpHandler and Windows 11 or later, Windows Server 2019 or later. For more information, see Make gRPC calls on .NET Framework.

Using `Grpc.Net.Client` with gRPC-Web requires additional configuration. For more information:

- Configure gRPC-Web with the .NET gRPC client
- Use gRPC client with .NET Standard 2.0

> **Important**
>
> gRPC-Web requires the client ***and*** server to support it. gRPC-Web can be **quickly configured by an ASP.NET Core gRPC server**. Other gRPC server implementations require a proxy to support gRPC-Web.

Additional resources

- Use gRPC client with .NET Standard 2.0
- gRPC C# core-library

Use gRPC in browser apps

Article • 07/31/2024

By James Newton-King

It's not possible to directly call a gRPC service from a browser. gRPC uses HTTP/2 features, and no browser provides the level of control required over web requests to support a gRPC client.

gRPC on ASP.NET Core offers two browser-compatible solutions, gRPC-Web and gRPC JSON transcoding.

gRPC-Web

gRPC-Web allows browser apps to call gRPC services with the gRPC-Web client and Protobuf.

- Similar to normal gRPC, but it has a slightly different wire-protocol, which makes it compatible with HTTP/1.1 and browsers.
- Requires the browser app to generate a gRPC client from a `.proto` file.
- Allows browser apps to benefit from the high-performance and low network usage of binary messages.

.NET has built-in support for gRPC-Web. For more information, see gRPC-Web in ASP.NET Core gRPC apps.

gRPC JSON transcoding

gRPC JSON transcoding allows browser apps to call gRPC services as if they were RESTful APIs with JSON.

- The browser app doesn't need to generate a gRPC client or know anything about gRPC.
- RESTful APIs are automatically created from gRPC services by annotating the `.proto` file with HTTP metadata.
- Allows an app to support both gRPC and JSON web APIs without duplicating the effort of building separate services for both.

.NET has built-in support for creating JSON web APIs from gRPC services. For more information, see gRPC JSON transcoding in ASP.NET Core gRPC apps.

> **Note**
>
> gRPC JSON transcoding requires .NET 7 or later.

Additional resources

- gRPC-Web in ASP.NET Core gRPC apps
- gRPC JSON transcoding in ASP.NET Core gRPC apps

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

gRPC-Web in ASP.NET Core gRPC apps

Article • 12/04/2023

By James Newton-King

Learn how to configure an existing ASP.NET Core gRPC service to be callable from browser apps, using the gRPC-Web protocol. gRPC-Web allows browser JavaScript and Blazor apps to call gRPC services. It's not possible to call an HTTP/2 gRPC service from a browser-based app. gRPC services hosted in ASP.NET Core can be configured to support gRPC-Web alongside HTTP/2 gRPC.

For instructions on adding a gRPC service to an existing ASP.NET Core app, see Add gRPC services to an ASP.NET Core app.

For instructions on creating a gRPC project, see Create a .NET Core gRPC client and server in ASP.NET Core.

ASP.NET Core gRPC-Web versus Envoy

There are two choices for how to add gRPC-Web to an ASP.NET Core app:

- Support gRPC-Web alongside gRPC HTTP/2 in ASP.NET Core. This option uses middleware provided by the Grpc.AspNetCore.Web package.
- Use the Envoy proxy's gRPC-Web support to translate gRPC-Web to gRPC HTTP/2. The translated call is then forwarded onto the ASP.NET Core app.

There are pros and cons to each approach. If an app's environment is already using Envoy as a proxy, it might make sense to also use Envoy to provide gRPC-Web support. For a basic solution for gRPC-Web that only requires ASP.NET Core, `Grpc.AspNetCore.Web` is a good choice.

Configure gRPC-Web in ASP.NET Core

gRPC services hosted in ASP.NET Core can be configured to support gRPC-Web alongside HTTP/2 gRPC. gRPC-Web doesn't require any changes to services. The only modification is in setting the middleware in `Program.cs`.

To enable gRPC-Web with an ASP.NET Core gRPC service:

- Add a reference to the Grpc.AspNetCore.Web package.

- Configure the app to use gRPC-Web by adding `UseGrpcWeb` and `EnableGrpcWeb` to `Program.cs`:

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();

var app = builder.Build();

app.UseGrpcWeb();

app.MapGrpcService<GreeterService>().EnableGrpcWeb();
app.MapGet("/", () => "This gRPC service is gRPC-Web enabled and is callable
from browser apps using the gRPC-Web protocol");

app.Run();
```

The preceding code:

- Adds the gRPC-Web middleware, `UseGrpcWeb`, after routing and before endpoints.
- Specifies that the `endpoints.MapGrpcService<GreeterService>()` method supports gRPC-Web with `EnableGrpcWeb`.

Alternatively, the gRPC-Web middleware can be configured so that all services support gRPC-Web by default and `EnableGrpcWeb` isn't required. Specify `new GrpcWebOptions { DefaultEnabled = true }` when the middleware is added.

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();

var app = builder.Build();

app.UseGrpcWeb(new GrpcWebOptions { DefaultEnabled = true });

app.MapGrpcService<GreeterService>().EnableGrpcWeb();
app.MapGet("/", () => "All gRPC service are supported by default in this
example, and are callable from browser apps using the gRPC-Web protocol");

app.Run();
```

> **Note**
>
> There is a known issue that causes gRPC-Web to fail when hosted by HTTP.sys in .NET Core 3.x.
>
> A workaround to get gRPC-Web working on HTTP.sys is available in Grpc-web experimental and UseHttpSys()? (grpc/grpc-dotnet #853).

gRPC-Web and CORS

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction applies to making gRPC-Web calls with browser apps. For example, a browser app served by `https://www.contoso.com` is blocked from calling gRPC-Web services hosted on `https://services.contoso.com`. Cross-Origin Resource Sharing (CORS) can be used to relax this restriction.

To allow a browser app to make cross-origin gRPC-Web calls, set up CORS in ASP.NET Core. Use the built-in CORS support, and expose gRPC-specific headers with WithExposedHeaders.

C#

```
using GrpcGreeter.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();

builder.Services.AddCors(o => o.AddPolicy("AllowAll", builder =>
{
    builder.AllowAnyOrigin()
           .AllowAnyMethod()
           .AllowAnyHeader()
           .WithExposedHeaders("Grpc-Status", "Grpc-Message", "Grpc-
Encoding", "Grpc-Accept-Encoding");
}));

var app = builder.Build();

app.UseGrpcWeb();
app.UseCors();

app.MapGrpcService<GreeterService>().EnableGrpcWeb()
                                    .RequireCors("AllowAll");

app.MapGet("/", () => "This gRPC service is gRPC-Web enabled, CORS enabled,
and is callable from browser apps using the gRPC-Web protocol");
```

```
app.Run();
```

The preceding code:

- Calls `AddCors` to add CORS services and configure a CORS policy that exposes gRPC-specific headers.
- Calls `UseCors` to add the CORS middleware after routing configuration and before endpoints configuration.
- Specifies that the `endpoints.MapGrpcService<GreeterService>()` method supports CORS with `RequireCors`.

gRPC-Web and streaming

Traditional gRPC over HTTP/2 supports client, server and bidirectional streaming. gRPC-Web offers limited support for streaming:

- gRPC-Web browser clients don't support calling client streaming and bidirectional streaming methods.
- gRPC-Web .NET clients don't support calling client streaming and bidirectional streaming methods over HTTP/1.1.
- ASP.NET Core gRPC services hosted on Azure App Service and IIS don't support bidirectional streaming.

When using gRPC-Web, we only recommend the use of unary methods and server streaming methods.

HTTP protocol

The ASP.NET Core gRPC service template, included in the .NET SDK, creates an app that's only configured for HTTP/2. This is a good default when an app only supports traditional gRPC over HTTP/2. gRPC-Web, however, works with both HTTP/1.1 and HTTP/2. Some platforms, such as UWP or Unity, can't use HTTP/2. To support all client apps, configure the server to enable HTTP/1.1 and HTTP/2.

Update the default protocol in `appsettings.json`:

JSON

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
```

```
    }
  }
}
```

Alternatively, configure Kestrel endpoints in startup code.

Enabling HTTP/1.1 and HTTP/2 on the same port requires TLS for protocol negotiation. For more information, see ASP.NET Core gRPC protocol negotiation.

Call gRPC-Web from the browser

Browser apps can use gRPC-Web to call gRPC services. There are some requirements and limitations when calling gRPC services with gRPC-Web from the browser:

- The server must contain configuration to support gRPC-Web.
- Client streaming and bidirectional streaming calls aren't supported. Server streaming is supported.
- Calling gRPC services on a different domain requires CORS configuration on the server.

JavaScript gRPC-Web client

A JavaScript gRPC-Web client exists. For instructions on how to use gRPC-Web from JavaScript, see write JavaScript client code with gRPC-Web.

Configure gRPC-Web with the .NET gRPC client

The .NET gRPC client can be configured to make gRPC-Web calls. This is useful for Blazor WebAssembly apps, which are hosted in the browser and have the same HTTP limitations of JavaScript code. Calling gRPC-Web with a .NET client is the same as HTTP/2 gRPC. The only modification is how the channel is created.

To use gRPC-Web:

- Add a reference to the Grpc.Net.Client.Web package.
- Ensure the reference to Grpc.Net.Client package is version 2.29.0 or later.
- Configure the channel to use the `GrpcWebHandler`:

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:53305", new
GrpcChannelOptions
{
```

```
    HttpHandler = new GrpcWebHandler(new HttpClientHandler())
});

var client = new Greeter.GreeterClient(channel);
var response = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
```

The preceding code:

- Configures a channel to use gRPC-Web.
- Creates a client and makes a call using the channel.

`GrpcWebHandler` has the following configuration options:

- `InnerHandler`: The underlying HttpMessageHandler that makes the gRPC HTTP request, for example, `HttpClientHandler`.
- `GrpcWebMode`: An enumeration type that specifies whether the gRPC HTTP request `Content-Type` is `application/grpc-web` or `application/grpc-web-text`.
  - `GrpcWebMode.GrpcWeb` configures sending content without encoding. Default value.
  - `GrpcWebMode.GrpcWebText` configures base64-encoded content. Required for server streaming calls in browsers.
- `HttpVersion`: HTTP protocol `Version` used to set HttpRequestMessage.Version on the underlying gRPC HTTP request. gRPC-Web doesn't require a specific version and doesn't override the default unless specified.

> **ⓘ Important**
>
> Generated gRPC clients have synchronous and asynchronous methods for calling unary methods. For example, `SayHello` is synchronous, and `SayHelloAsync` is asynchronous. Asynchronous methods are always required in Blazor WebAssembly. Calling a synchronous method in a Blazor WebAssembly app causes the app to become unresponsive.

Use gRPC client factory with gRPC-Web

Create a .NET client compatible with gRPC-Web using the gRPC client factory:

- Add package references to the project file for the following packages:
  - Grpc.Net.Client.Web
  - Grpc.Net.ClientFactory

- Register a gRPC client with dependency injection (DI) using the generic `AddGrpcClient` extension method. In a Blazor WebAssembly app, services are registered with DI in `Program.cs`.
- Configure `GrpcWebHandler` using the ConfigurePrimaryHttpMessageHandler extension method.

C#

```
builder.Services
    .AddGrpcClient<Greet.GreeterClient>(options =>
    {
        options.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(
        () => new GrpcWebHandler(new HttpClientHandler()));
```

For more information, see gRPC client factory integration in .NET.

Additional resources

- gRPC for Web Clients GitHub project
- Enable Cross-Origin Requests (CORS) in ASP.NET Core
- gRPC JSON transcoding in ASP.NET Core gRPC apps

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

gRPC JSON transcoding in ASP.NET Core

Article • 07/31/2024

By James Newton-King

gRPC is a high-performance Remote Procedure Call (RPC) framework. gRPC uses HTTP/2, streaming, Protobuf, and message contracts to create high-performance, real-time services.

One limitation with gRPC is that not every platform can use it. Browsers don't fully support HTTP/2, making REST APIs and JSON the primary way to get data into browser apps. Despite the benefits that gRPC brings, REST APIs and JSON have an important place in modern apps. Building gRPC ***and*** JSON Web APIs adds unwanted overhead to app development.

This document discusses how to create JSON Web APIs using gRPC services.

Overview

gRPC JSON transcoding is an extension for ASP.NET Core that creates RESTful JSON APIs for gRPC services. Once configured, transcoding allows apps to call gRPC services with familiar HTTP concepts:

- HTTP verbs
- URL parameter binding
- JSON requests/responses

gRPC can still be used to call services.

> **Note**
>
> gRPC JSON transcoding replaces **gRPC HTTP API**, an alternative experimental extension.

Usage

1. Add a package reference to Microsoft.AspNetCore.Grpc.JsonTranscoding.
2. Register transcoding in server startup code by adding `AddJsonTranscoding`: In the `Program.cs` file, change `builder.Services.AddGrpc();` to

`builder.Services.AddGrpc().AddJsonTranscoding();`.

3. Add `<IncludeHttpRuleProtos>true</IncludeHttpRuleProtos>` to the property group in the project file (`.csproj`):

```JSON
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>true</InvariantGlobalization>
    <IncludeHttpRuleProtos>true</IncludeHttpRuleProtos>
  </PropertyGroup>
```

4. Annotate gRPC methods in your `.proto` files with HTTP bindings and routes:

```ProtoBuf
syntax = "proto3";

option csharp_namespace = "GrpcServiceTranscoding";
import "google/api/annotations.proto";

package greet;

// The greeting service definition.
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```

The `SayHello` gRPC method can now be invoked as gRPC and as a JSON Web API:

- Request: `GET /v1/greeter/world`

- Response: `{ "message": "Hello world" }`

If the server is configured to write logs for each request, server logs show that a gRPC service executes the HTTP call. Transcoding maps the incoming HTTP request to a gRPC message and converts the response message to JSON.

text

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET https://localhost:5001/v1/greeter/world
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /v1/greeter/{name}'
info: Server.GreeterService[0]
      Sending hello to world
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /v1/greeter/{name}'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 1.996ms 200 application/json
```

Annotate gRPC methods

gRPC methods must be annotated with an HTTP rule before they support transcoding. The HTTP rule includes information about how to call the gRPC method, such as the HTTP method and route.

ProtoBuf

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}
```

The proceeding example:

- Defines a `Greeter` service with a `SayHello` method. The method has an HTTP rule specified using the name `google.api.http`.
- The method is accessible with `GET` requests and the `/v1/greeter/{name}` route.
- The `name` field on the request message is bound to a route parameter.

Many options are available for customizing how a gRPC method binds to a RESTful API. For more information about annotating gRPC methods and customizing JSON, see Configure HTTP and JSON for gRPC JSON transcoding.

Streaming methods

Traditional gRPC over HTTP/2 supports streaming in all directions. Transcoding is limited to server streaming only. Client streaming and bidirectional streaming methods aren't supported.

Server streaming methods use line-delimited JSON. Each message written using `WriteAsync` is serialized to JSON and followed by a new line.

The following server streaming method writes three messages:

C#

```
public override async Task StreamingFromServer(ExampleRequest request,
    IServerStreamWriter<ExampleResponse> responseStream, ServerCallContext
context)
{
    for (var i = 1; i <= 3; i++)
    {
        await responseStream.WriteAsync(new ExampleResponse { Text =
$"Message {i}" });
        await Task.Delay(TimeSpan.FromSeconds(1));
    }
}
```

The client receives three line-delimited JSON objects:

JSON

```
{"Text":"Message 1"}
{"Text":"Message 2"}
{"Text":"Message 3"}
```

Note that the `WriteIndented` JSON setting doesn't apply to server streaming methods. Pretty printing adds new lines and whitespace to JSON, which can't be used with line-delimited JSON.

View or download an ASP.NET Core gPRC transcoding and streaming app sample.

HTTP protocol

The ASP.NET Core gRPC service template, included in the .NET SDK, creates an app that's only configured for HTTP/2. HTTP/2 is a good default when an app only supports traditional gRPC over HTTP/2. Transcoding, however, works with both HTTP/1.1 and

HTTP/2. Some platforms, such as UWP or Unity, can't use HTTP/2. To support all client apps, configure the server to enable HTTP/1.1 and HTTP/2.

Update the default protocol in `appsettings.json`:

JSON

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
  }
}
```

Alternatively, configure Kestrel endpoints in startup code.

Enabling HTTP/1.1 and HTTP/2 on the same port requires TLS for protocol negotiation. For more information about configuring HTTP protocols in a gRPC app, see ASP.NET Core gRPC protocol negotiation.

gRPC JSON transcoding vs gRPC-Web

Both transcoding and gRPC-Web allow gRPC services to be called from a browser. However, the way each does this is different:

- gRPC-Web lets browser apps call gRPC services from the browser with the gRPC-Web client and Protobuf. gRPC-Web requires the browser app to generate a gRPC client and has the advantage of sending small, fast Protobuf messages.
- Transcoding allows browser apps to call gRPC services as if they were RESTful APIs with JSON. The browser app doesn't need to generate a gRPC client or know anything about gRPC.

The previous `Greeter` service can be called using browser JavaScript APIs:

JavaScript

```
var name = nameInput.value;

fetch('/v1/greeter/' + name)
  .then((response) => response.json())
  .then((result) => {
    console.log(result.message);
    // Hello world
  });
```

grpc-gateway

grpc-gateway is another technology for creating RESTful JSON APIs from gRPC services. It uses the same `.proto` annotations to map HTTP concepts to gRPC services.

grpc-gateway uses code generation to create a reverse-proxy server. The reverse proxy translates RESTful calls into gRPC+Protobuf and sends the calls over HTTP/2 to the gRPC service. The benefit of this approach is the gRPC service doesn't know about the RESTful JSON APIs. Any gRPC server can use grpc-gateway.

Meanwhile, gRPC JSON transcoding runs inside an ASP.NET Core app. It deserializes JSON into Protobuf messages, then invokes the gRPC service directly. Transcoding in ASP.NET Core offers advantages to .NET app developers:

- Less complex: Both gRPC services and mapped RESTful JSON API run out of one ASP.NET Core app.
- Better performance: Transcoding deserializes JSON to Protobuf messages and invokes the gRPC service directly. There are significant performance benefits in doing this in-process versus making a new gRPC call to a different server.
- Lower cost: Fewer servers result in a smaller monthly hosting bill.

For installation and usage of grpc-gateway, see the grpc-gateway README.

Additional resources

- Configure HTTP and JSON for gRPC JSON transcoding ASP.NET Core apps
- Use OpenAPI with gRPC JSON transcoding ASP.NET Core apps
- Use gRPC in browser apps
- gRPC-Web in ASP.NET Core gRPC apps

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Configure HTTP and JSON for gRPC JSON transcoding

Article • 07/31/2024

By James Newton-King

gRPC JSON transcoding creates RESTful JSON web APIs from gRPC methods. It uses annotations and options for customizing how a RESTful API maps to the gRPC methods.

HTTP rules

gRPC methods must be annotated with an HTTP rule before they support transcoding. The HTTP rule includes information about calling the gRPC method as a RESTful API, such as the HTTP method and route.

ProtoBuf

```
import "google/api/annotations.proto";

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}
```

An HTTP rule is:

- An annotation on gRPC methods.
- Identified by the name `google.api.http`.
- Imported from the `google/api/annotations.proto` file. The google/api/http.proto and google/api/annotations.proto files need to be in the project.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

HTTP method

The HTTP method is specified by setting the route to the matching HTTP method field name:

- `get`
- `put`
- `post`
- `delete`
- `patch`

The `custom` field allows for other HTTP methods.

In the following example, the `CreateAddress` method is mapped to `POST` with the specified route:

ProtoBuf

```
service Address {
  rpc CreateAddress (CreateAddressRequest) returns (CreateAddressReply) {
    option (google.api.http) = {
      post: "/v1/address",
      body: "*"
    };
  }
}
```

Route

gRPC JSON transcoding routes support route parameters. For example, `{name}` in a route binds to the `name` field on the request message.

To bind a field on a nested message, specify the path to the field. In the following example, `{params.org}` binds to the `org` field on the `IssueParams` message:

ProtoBuf

```
service Repository {
  rpc GetIssue (GetIssueRequest) returns (GetIssueReply) {
    option (google.api.http) = {
      get: "/{apiVersion}/{params.org}/{params.repo}/issue/{params.issueId}"
    };
  }
}

message GetIssueRequest {
```

```
  int32 api_version = 1;
  IssueParams params = 2;
}
message IssueParams {
  string org = 1;
  string repo = 2;
  int32 issueId = 3;
}
```

Transcoding routes and ASP.NET Core routes have a similar syntax and feature set. However, some ASP.NET Core routing features aren't supported by transcoding. These include:

- Route constraints
- Default values
- Optional parameters
- Complex segments

Request body

Transcoding deserializes the request body JSON to the request message. The `body` field specifies how the HTTP request body maps to the request message. The value is either the name of the request field whose value is mapped to the HTTP request body or `*` for mapping all request fields.

In the following example, the HTTP request body is deserialized to the `address` field:

ProtoBuf

```
service Address {
  rpc AddAddress (AddAddressRequest) returns (AddAddressReply) {
    option (google.api.http) = {
      post: "/{apiVersion}/address",
      body: "address"
    };
  }
}

message AddAddressRequest {
  int32 api_version = 1;
  Address address = 2;
}
message Address {
  string street = 1;
  string city = 2;
  string country = 3;
}
```

Query parameters

Any fields in the request message that aren't bound by route parameters or the request body can be set using HTTP query parameters.

ProtoBuf

```
service Repository {
  rpc GetIssues (GetIssuesRequest) returns (GetIssuesReply) {
    option (google.api.http) = {
      get: "/v1/{org}/{repo}/issue"
    };
  }
}

message GetIssuesRequest {
  string org = 1;
  string repo = 2;
  string text = 3;
  PageParams page = 4;
}
message PageParams {
  int32 index = 1;
  int32 size = 2;
}
```

In the preceding example:

- `org` and `repo` fields are bound from route parameters.
- Other fields, such as `text` and the nested fields from `page`, can be bound from the query string: `?text=value&page.index=0&page.size=10`

Response body

By default, transcoding serializes the entire response message as JSON. The `response_body` field allows serialization of a subset of the response message.

ProtoBuf

```
service Address {
  rpc GetAddress (GetAddressRequest) returns (GetAddressReply) {
    option (google.api.http) = {
      get: "/v1/address/{id}",
      response_body: "address"
    };
  }
}
```

```
message GetAddressReply {
  int32 version = 1;
  Address address = 2;
}
message Address {
  string street = 1;
  string city = 2;
  string country = 3;
}
```

In the preceding example, the `address` field is serialized to the response body as JSON.

Specification

For more information about customizing gRPC transcoding, see the HttpRule specification.

Customize JSON

Messages are converted to and from JSON using the JSON mapping in the Protobuf specification. Protobuf's JSON mapping is a standardized way to convert between JSON and Protobuf, and all serialization follows these rules.

However, gRPC JSON transcoding offers some limited options for customizing JSON with GrpcJsonSettings, as shown in the following table.

Expand table

Option	Default Value	Description
IgnoreDefaultValues	`false`	If set to `true`, fields with default values are ignored during serialization.
WriteEnumsAsIntegers	`false`	If set to `true`, enum values are written as integers instead of strings.
WriteInt64sAsStrings	`false`	If set to `true`, `Int64` and `UInt64` values are written as strings instead of numbers.
WriteIndented	`false`	If set to `true`, JSON is written using pretty printing. This option doesn't affect streaming methods, which write line-delimited JSON messages and can't use pretty printing.

C#

```
builder.Services.AddGrpc().AddJsonTranscoding(o =>
{
    o.JsonSettings.WriteIndented = true;
});
```

In the `.proto` file, the `json_name` field option customizes a field's name when it's serialized as JSON, as in the following example:

ProtoBuf

```
message TestMessage {
  string my_field = 1 [json_name="customFieldName"];
}
```

Transcoding doesn't support advanced JSON customization. Apps requiring precise JSON structure control should consider using ASP.NET Core Web API.

Additional resources

- gRPC JSON transcoding in ASP.NET Core gRPC apps
- HttpRule specification

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

gRPC JSON transcoding documentation with Swagger / OpenAPI

Article • 07/31/2024

By James Newton-King

OpenAPI (Swagger) is a language-agnostic specification for describing REST APIs. gRPC JSON transcoding supports generating OpenAPI from transcoded RESTful APIs. The Microsoft.AspNetCore.Grpc.Swagger package:

- Integrates gRPC JSON transcoding with Swashbuckle.
- Is experimental in .NET 7 to allow us to explore the best way to provide OpenAPI support.

Get started

To enable OpenAPI with gRPC JSON transcoding:

1. Add a package reference to Microsoft.AspNetCore.Grpc.Swagger. The version must be 0.3.0-xxx or later.
2. Configure Swashbuckle in startup. The `AddGrpcSwagger` method configures Swashbuckle to include gRPC endpoints.

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddGrpc().AddJsonTranscoding();
builder.Services.AddGrpcSwagger();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1",
        new OpenApiInfo { Title = "gRPC transcoding", Version = "v1" });
});

var app = builder.Build();
app.UseSwagger();
if (app.Environment.IsDevelopment())
{
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });
}
app.MapGrpcService<GreeterService>();
```

```
app.Run();
```

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Add OpenAPI descriptions from `.proto` comments

Generate OpenAPI descriptions from comments in the `.proto` contract, as in the following example:

ProtoBuf

```
// My amazing greeter service.
service Greeter {
  // Sends a greeting.
  rpc SayHello (HelloRequest) returns (HelloReply) {
    option (google.api.http) = {
      get: "/v1/greeter/{name}"
    };
  }
}

message HelloRequest {
  // Name to say hello to.
  string name = 1;
}
message HelloReply {
  // Hello reply message.
  string message = 1;
}
```

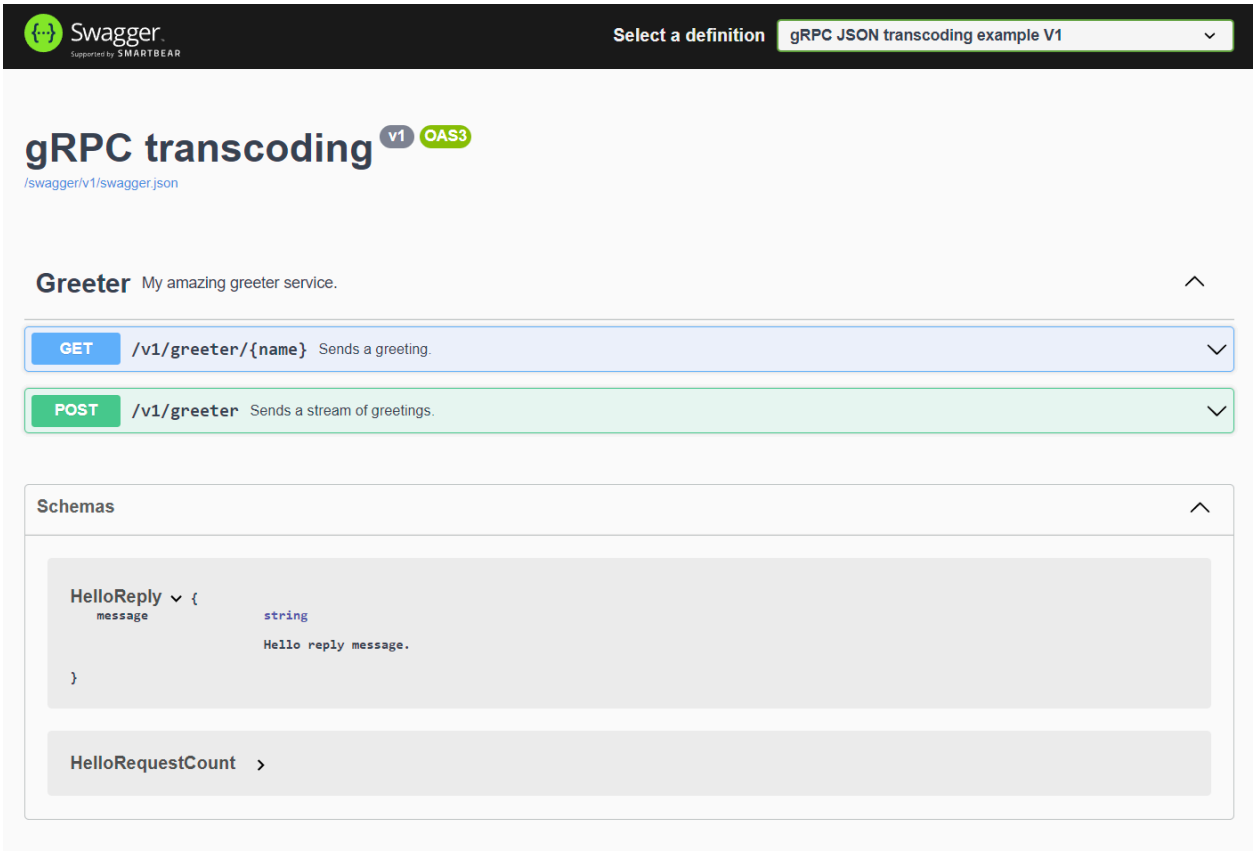
To enable gRPC OpenAPI comments:

1. Enable the XML documentation file in the server project with `<GenerateDocumentationFile>true</GenerateDocumentationFile>`.
2. Configure `AddSwaggerGen` to read the generated XML file. Pass the XML file path to `IncludeXmlComments` and `IncludeGrpcXmlComments`, as in the following example:

C#

```
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1",
        new OpenApiInfo { Title = "gRPC transcoding", Version = "v1" });

    var filePath = Path.Combine(System.AppContext.BaseDirectory,
"Server.xml");
    c.IncludeXmlComments(filePath);
    c.IncludeGrpcXmlComments(filePath, includeControllerXmlComments: true);
});
```

To confirm that Swashbuckle is generating OpenAPI with descriptions for the RESTful gRPC services, start the app and navigate to the Swagger UI page:



Additional resources

- gRPC JSON transcoding in ASP.NET Core gRPC apps
- OpenAPI homepage
- Swashbuckle.AspNetCore GitHub repository

gRPC for .NET configuration

Article • 07/31/2024

Configure services options

gRPC services are configured with `AddGrpc` in `Startup.cs`. Configuration options are in the Grpc.AspNetCore.Server package.

The following table describes options for configuring gRPC services:

Expand table

Option	Default Value	Description
`MaxSendMessageSize`	`null`	The maximum message size in bytes that can be sent from the server. Attempting to send a message that exceeds the configured maximum message size results in an exception. When set to `null`, the message size is unlimited.
`MaxReceiveMessageSize`	4 MB	The maximum message size in bytes that can be received by the server. If the server receives a message that exceeds this limit, it throws an exception. Increasing this value allows the server to receive larger messages, but can negatively impact memory consumption. When set to `null`, the message size is unlimited.
`EnableDetailedErrors`	`false`	If `true`, detailed exception messages are returned to clients when an exception is thrown in a service method. The default is `false`. Setting `EnableDetailedErrors` to `true` can leak sensitive information.
`CompressionProviders`	gzip	A collection of compression providers used to compress and decompress messages. Custom compression providers can be created and added to the collection. The default configured providers support **gzip** compression.
`ResponseCompressionAlgorithm`	`null`	The compression algorithm used to compress messages sent from the server. The algorithm must match a compression provider in `CompressionProviders`. For the algorithm to compress a response, the client must indicate it supports the

Option	Default Value	Description
		algorithm by sending it in the **grpc-accept-encoding** header.
`ResponseCompressionLevel`	`null`	The compress level used to compress messages sent from the server.
`Interceptors`	None	A collection of interceptors that are run with each gRPC call. Interceptors are run in the order they are registered. Globally configured interceptors are run before interceptors configured for a single service. Interceptors have a per-request lifetime by default. The interceptor constructor is called and parameters are resolved from dependency injection (DI). An interceptor type can also be registered with DI to override how it is created and its lifetime. Interceptors offer similar functionalities compared to ASP.NET Core middleware. For more information, see gRPC Interceptors vs. Middleware.
`IgnoreUnknownServices`	`false`	If `true`, calls to unknown services and methods don't return an **UNIMPLEMENTED** status, and the request passes to the next registered middleware in ASP.NET Core.

Options can be configured for all services by providing an options delegate to the `AddGrpc` call in `Startup.ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.EnableDetailedErrors = true;
        options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
        options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
    });
}
```

Options for a single service override the global options provided in `AddGrpc` and can be configured using `AddServiceOptions<TService>`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc().AddServiceOptions<MyService>(options =>
    {
        options.MaxReceiveMessageSize = 2 * 1024 * 1024; // 2 MB
        options.MaxSendMessageSize = 5 * 1024 * 1024; // 5 MB
    });
}
```

Service interceptors have a per-request lifetime by default. Registering the interceptor type with DI overrides how an interceptor is created and its lifetime.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.Interceptors.Add<LoggingInterceptor>();
    });
    services.AddSingleton<LoggingInterceptor>();
}
```

ASP.NET Core server options

`Grpc.AspNetCore.Server` is hosted by an ASP.NET Core web server. There are a number of options for ASP.NET Core servers, including Kestrel, IIS and HTTP.sys. Each server offers additional options for how HTTP requests are served.

The server used by an ASP.NET Core app is configured in app startup code. The default server is Kestrel.

For more information about the different servers and their configuration options, see:

- Kestrel web server in ASP.NET Core
- HTTP.sys web server implementation in ASP.NET Core
- Host ASP.NET Core on Windows with IIS

Configure client options

gRPC client configuration is set on `GrpcChannelOptions`. Configuration options are in the Grpc.Net.Client package.

The following table describes options for configuring gRPC channels:

Expand table

Option	Default Value	Description
`HttpHandler`	New instance	The `HttpMessageHandler` used to make gRPC calls. A client can be set to configure a custom `HttpClientHandler` or add additional handlers to the HTTP pipeline for gRPC calls. If no `HttpMessageHandler` is specified, a new `HttpClientHandler` instance is created for the channel with automatic disposal.
`HttpClient`	`null`	The `HttpClient` used to make gRPC calls. This setting is an alternative to `HttpHandler`.
`DisposeHttpClient`	`false`	If set to `true` and an `HttpMessageHandler` or `HttpClient` is specified, then either the `HttpHandler` or `HttpClient`, respectively, is disposed when the `GrpcChannel` is disposed.
`LoggerFactory`	`null`	The `LoggerFactory` used by the client to log information about gRPC calls. A `LoggerFactory` instance can be resolved from dependency injection or created using `LoggerFactory.Create`. For examples of configuring logging, see Logging and diagnostics in gRPC on .NET.
`MaxSendMessageSize`	`null`	The maximum message size in bytes that can be sent from the client. Attempting to send a message that exceeds the configured maximum message size results in an exception. When set to `null`, the message size is unlimited.
`MaxReceiveMessageSize`	4 MB	The maximum message size in bytes that can be received by the client. If the client receives a message that exceeds this limit, it throws an exception. Increasing this value allows the client to receive larger messages, but can negatively impact memory consumption. When set to `null`, the message size is unlimited.
`Credentials`	`null`	A `ChannelCredentials` instance. Credentials are used to add

Option	Default Value	Description
		authentication metadata to gRPC calls.
`CompressionProviders`	gzip	A collection of compression providers used to compress and decompress messages. Custom compression providers can be created and added to the collection. The default configured providers support **gzip** compression.
`ThrowOperationCanceledOnCancellation`	`false`	If set to `true`, clients throw OperationCanceledException when a call is canceled or its deadline is exceeded.
`UnsafeUseInsecureChannelCallCredentials`	`false`	If set to `true`, `CallCredentials` are applied to gRPC calls made by an insecure channel. Sending authentication headers over an insecure connection has security implications and shouldn't be done in production environments.
`MaxRetryAttempts`	5	The maximum retry attempts. This value limits any retry and hedging attempt values specified in the service config. Setting this value alone doesn't enable retries. Retries are enabled in the service config, which can be done using `ServiceConfig`. A `null` value removes the maximum retry attempts limit. For more information about retries, see Transient fault handling with gRPC retries.
`MaxRetryBufferSize`	16 MB	The maximum buffer size in bytes that can be used to store sent messages when retrying or hedging calls. If the buffer limit is exceeded, then no more retry attempts are made and all hedging calls but one will be canceled. This limit is applied across all calls made using the channel. A `null` value removes the maximum retry buffer size limit.
`MaxRetryBufferPerCallSize`	1 MB	The maximum buffer size in bytes that can be used to store sent messages when retrying or hedging calls. If the buffer limit is exceeded, then no more retry attempts are made and all hedging calls but one will be canceled. This limit is applied to

Option	Default Value	Description
		one call. A `null` value removes the maximum retry buffer size limit per call.
`ServiceConfig`	`null`	The service config for a gRPC channel. A service config can be used to configure gRPC retries.

The following code:

- Sets the maximum send and receive message size on the channel.
- Creates a client.

C#

```
static async Task Main(string[] args)
{
    var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
    {
        MaxReceiveMessageSize = 5 * 1024 * 1024, // 5 MB
        MaxSendMessageSize = 2 * 1024 * 1024 // 2 MB
    });
    var client = new Greeter.GreeterClient(channel);

    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
}
```

Note that client interceptors aren't configured with `GrpcChannelOptions`. Instead, client interceptors are configured using the `Intercept` extension method with a channel. This extension method is in the `Grpc.Core.Interceptors` namespace.

C#

```
static async Task Main(string[] args)
{
    var channel = GrpcChannel.ForAddress("https://localhost:5001");
    var callInvoker = channel.Intercept(new LoggingInterceptor());
    var client = new Greeter.GreeterClient(callInvoker);

    var reply = await client.SayHelloAsync(
                      new HelloRequest { Name = "GreeterClient" });
    Console.WriteLine("Greeting: " + reply.Message);
}
```

System.Net handler options

`Grpc.Net.Client` uses a HTTP transport derived from `HttpMessageHandler` to make HTTP requests. Each handler offers additional options for how HTTP requests are made.

The handler is configured on a channel and can be overridden by setting `GrpcChannelOptions.HttpHandler`. .NET Core 3 and .NET 5 or later uses SocketsHttpHandler by default. gRPC client apps on .NET Framework should configure WinHttpHandler.

For more information about the different handlers and their configuration options, see:

- System.Net.Http.SocketsHttpHandler
- System.Net.Http.WinHttpHandler

Additional resources

- gRPC services with ASP.NET Core
- Call gRPC services with the .NET client
- Logging and diagnostics in gRPC on .NET
- Create a .NET Core gRPC client and server in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Authentication and authorization in gRPC for ASP.NET Core

Article • 07/16/2024

By James Newton-King

View or download sample code (how to download)

Authenticate users calling a gRPC service

gRPC can be used with ASP.NET Core authentication to associate a user with each call.

The following is an example of `Program.cs` which uses gRPC and ASP.NET Core authentication:

C#

```
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapGrpcService<GreeterService>();
```

> **Note**
>
> The order in which you register the ASP.NET Core authentication middleware matters. Always call `UseAuthentication` and `UseAuthorization` after `UseRouting` and before `UseEndpoints`.

The authentication mechanism your app uses during a call needs to be configured. Authentication configuration is added in `Program.cs` and will be different depending upon the authentication mechanism your app uses.

Once authentication has been setup, the user can be accessed in a gRPC service methods via the `ServerCallContext`.

C#

```
public override Task<BuyTicketsResponse> BuyTickets(
    BuyTicketsRequest request, ServerCallContext context)
{
```

```
    var user = context.GetHttpContext().User;

    // ... access data from ClaimsPrincipal ...
}
```

Bearer token authentication

The client can provide an access token for authentication. The server validates the token and uses it to identify the user.

On the server, bearer token authentication is configured using the JWT Bearer middleware.

In the .NET gRPC client, the token can be sent with calls by using the `Metadata` collection. Entries in the `Metadata` collection are sent with a gRPC call as HTTP headers:

C#

```
public bool DoAuthenticatedCall(
    Ticketer.TicketerClient client, string token)
{
    var headers = new Metadata();
    headers.Add("Authorization", $"Bearer {token}");

    var request = new BuyTicketsRequest { Count = 1 };
    var response = await client.BuyTicketsAsync(request, headers);

    return response.Success;
}
```

Set the bearer token with `CallCredentials`

Configuring `ChannelCredentials` on a channel is an alternative way to send the token to the service with gRPC calls. A `ChannelCredentials` can include `CallCredentials`, which provide a way to automatically set `Metadata`.

Benefits of using `CallCredentials`:

- Authentication is centrally configured on the channel. The token doesn't need to be manually provided to the gRPC call.
- The `CallCredentials.FromInterceptor` callback is asynchronous. Call credentials can fetch a credential token from an external system if required. Asynchronous

methods inside the callback should use the `CancellationToken` on `AuthInterceptorContext`.

> **Note**
>
> `CallCredentials` are only applied if the channel is secured with TLS. Sending authentication headers over an insecure connection has security implications and shouldn't be done in production environments. An app can configure a channel to ignore this behavior and always use `CallCredentials` by setting `UnsafeUseInsecureChannelCallCredentials` on a channel.

The credential in the following example configures the channel to send the token with every gRPC call:

C#

```
private static GrpcChannel CreateAuthenticatedChannel(ITokenProvder
tokenProvider)
{
    var credentials = CallCredentials.FromInterceptor(async (context,
metadata) =>
    {
        var token = await
tokenProvider.GetTokenAsync(context.CancellationToken);
        metadata.Add("Authorization", $"Bearer {token}");
    });

    var channel = GrpcChannel.ForAddress(address, new GrpcChannelOptions
    {
        Credentials = ChannelCredentials.Create(new SslCredentials(),
credentials)
    });
    return channel;
}
```

Bearer token with gRPC client factory

gRPC client factory can create clients that send a bearer token using `AddCallCredentials`. This method is available in Grpc.Net.ClientFactory version 2.46.0 or later.

The delegate passed to `AddCallCredentials` is executed for each gRPC call:

C#

```
builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials((context, metadata) =>
    {
        if (!string.IsNullOrEmpty(_token))
        {
            metadata.Add("Authorization", $"Bearer {_token}");
        }
        return Task.CompletedTask;
    });
```

Dependency injection (DI) can be combined with `AddCallCredentials`. An overload passes `IServiceProvider` to the delegate, which can be used to get a service constructed from DI using scoped and transient services.

Consider an app that has:

- A user-defined `ITokenProvider` for getting a bearer token. `ITokenProvider` is registered in DI with a scoped lifetime.
- gRPC client factory is configured to create clients that are injected into gRPC services and Web API controllers.
- gRPC calls should use `ITokenProvider` to get a bearer token.

C#

```
public interface ITokenProvider
{
    Task<string> GetTokenAsync(CancellationToken cancellationToken);
}

public class AppTokenProvider : ITokenProvider
{
    private string _token;

    public async Task<string> GetTokenAsync(CancellationToken
cancellationToken)
    {
        if (_token == null)
        {
            // App code to resolve the token here.
        }

        return _token;
    }
}
```

C#

```
builder.Services.AddScoped<ITokenProvider, AppTokenProvider>();

builder.Services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .AddCallCredentials(async (context, metadata, serviceProvider) =>
    {
        var provider = serviceProvider.GetRequiredService<ITokenProvider>();
        var token = await provider.GetTokenAsync(context.CancellationToken);
        metadata.Add("Authorization", $"Bearer {token}");
    }));
```

The preceding code:

- Defines `ITokenProvider` and `AppTokenProvider`. These types handle resolving the authentication token for gRPC calls.
- Registers the `AppTokenProvider` type with DI in a scoped lifetime. `AppTokenProvider` caches the token so that only the first call in the scope is required to calculate it.
- Registers the `GreeterClient` type with client factory.
- Configures `AddCallCredentials` for this client. The delegate is executed each time a call is made and adds the token returned by `ITokenProvider` to the metadata.

Client certificate authentication

A client could alternatively provide a client certificate for authentication. Certificate authentication happens at the TLS level, long before it ever gets to ASP.NET Core. When the request enters ASP.NET Core, the client certificate authentication package allows you to resolve the certificate to a `ClaimsPrincipal`.

> **Note**
>
> Configure the server to accept client certificates. For information on accepting client certificates in Kestrel, IIS, and Azure, see Configure certificate authentication in ASP.NET Core.

In the .NET gRPC client, the client certificate is added to `HttpClientHandler` that is then used to create the gRPC client:

C#

```
public Ticketer.TicketerClient CreateClientWithCert(
    string baseAddress,
    X509Certificate2 certificate)
{
    // Add client cert to the handler
    var handler = new HttpClientHandler();
    handler.ClientCertificates.Add(certificate);

    // Create the gRPC channel
    var channel = GrpcChannel.ForAddress(baseAddress, new GrpcChannelOptions
    {
        HttpHandler = handler
    });

    return new Ticketer.TicketerClient(channel);
}
```

Other authentication mechanisms

Many ASP.NET Core supported authentication mechanisms work with gRPC:

- Microsoft Entra ID
- Client Certificate
- IdentityServer
- JWT Token
- OAuth 2.0
- OpenID Connect
- WS-Federation

For more information on configuring authentication on the server, see ASP.NET Core authentication.

Configuring the gRPC client to use authentication will depend on the authentication mechanism you are using. The previous bearer token and client certificate examples show a couple of ways the gRPC client can be configured to send authentication metadata with gRPC calls:

- Strongly typed gRPC clients use `HttpClient` internally. Authentication can be configured on HttpClientHandler, or by adding custom HttpMessageHandler instances to the `HttpClient`.
- Each gRPC call has an optional `CallOptions` argument. Custom headers can be sent using the option's headers collection.

> ⓘ Note

Windows Authentication (NTLM/Kerberos/Negotiate) can't be used with gRPC. gRPC requires HTTP/2, and HTTP/2 doesn't support Windows Authentication.

Authorize users to access services and service methods

By default, all methods in a service can be called by unauthenticated users. To require authentication, apply the [Authorize] attribute to the service:

C#

```
[Authorize]
public class TicketerService : Ticketer.TicketerBase
{
}
```

You can use the constructor arguments and properties of the `[Authorize]` attribute to restrict access to only users matching specific authorization policies. For example, if you have a custom authorization policy called `MyAuthorizationPolicy`, ensure that only users matching that policy can access the service using the following code:

C#

```
[Authorize("MyAuthorizationPolicy")]
public class TicketerService : Ticketer.TicketerBase
{
}
```

Individual service methods can have the `[Authorize]` attribute applied as well. If the current user doesn't match the policies applied to **both** the method and the class, an error is returned to the caller:

C#

```
[Authorize]
public class TicketerService : Ticketer.TicketerBase
{
    public override Task<AvailableTicketsResponse> GetAvailableTickets(
        Empty request, ServerCallContext context)
    {
        // ... buy tickets for the current user ...
    }

    [Authorize("Administrators")]
```

```
    public override Task<BuyTicketsResponse> RefundTickets(
        BuyTicketsRequest request, ServerCallContext context)
    {
        // ... refund tickets (something only Administrators can do) ..
    }
}
```

Additional resources

- Bearer Token authentication in ASP.NET Core
- Configure Client Certificate authentication in ASP.NET Core
- Configure interceptors in a gRPC client factory in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Error handling with gRPC

Article • 07/31/2024

By James Newton-King

This article discusses error handling and gRPC:

- Built-in error handling capabilities using gRPC status codes and error messages.
- Sending complex, structured error information using rich error handling.

Built-in error handling

gRPC calls communicate success or failure with a status code. When a gRPC call completes successfully the server returns an `OK` status to the client. If an error occurs, gRPC returns:

- An error status code, such as `CANCELLED` or `UNAVAILABLE`.
- An optional string error message.

The types commonly used with error handling are:

- `StatusCode`: An enumeration of gRPC status codes. `OK` signals success; other values are failure.
- `Status`: A `struct` that combines a `StatusCode` and an optional string error message. The error message provides further details about what happened.
- `RpcException`: An exception type that has `Status` value. This exception is thrown in gRPC server methods and caught by gRPC clients.

Built-in error handling only supports a status code and string description. To send complex error information from the server to the client, use rich error handling.

Throw server errors

A gRPC server call always returns a status. The server automatically returns `OK` when a method completes successfully.

cs

```
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
```

```
    {
        return Task.FromResult(new HelloReply { Message = $"Hello
{request.Name}" });
    }

    public override async Task SayHelloStreaming(HelloRequest request,
        IServerStreamWriter<HelloReply> responseStream, ServerCallContext
context)
    {
        for (var i = 0; i < 5; i++)
        {
            await responseStream.WriteAsync(new HelloReply { Message =
$"Hello {request.Name} {i}" });
            await Task.Delay(TimeSpan.FromSeconds(1));
        }
    }
}
```

The preceding code:

- Implements the unary `SayHello` method that completes successfully when it returns a response message.
- Implements the server streaming `SayHelloStreaming` method that completes successfully when the method finished.

Server error status

gRPC methods return an error status code by throwing an exception. When an `RpcException` is thrown on the server, its status code and description is returned to the client:

cs

```
public class GreeterService : GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        if (string.IsNullOrEmpty(request.Name))
        {
            throw new RpcException(new Status(StatusCode.InvalidArgument,
"Name is required."));
        }
        return Task.FromResult(new HelloReply { Message = $"Hello
{request.Name}" });
    }
}
```

Thrown exception types that aren't `RpcException` also cause the call to fail, but with an `UNKNOWN` status code and a generic message `Exception was thrown by handler`.

`Exception was thrown by handler` is sent to the client rather than the exception message to prevent exposing potentially sensitive information. To see a more descriptive error message in a development environment, configure EnableDetailedErrors.

Handle client errors

When a gRPC client makes a call, the status code is automatically validated when accessing the response. For example, awaiting a unary gRPC call returns the message sent by the server if the call is successful, and throws an `RpcException` if there's a failure. Catch `RpcException` to handle errors in a client:

cs

```
var client = new Greet.GreeterClient(channel);

try
{
    var response = await client.SayHelloAsync(new HelloRequest { Name =
"World" });
    Console.WriteLine("Greeting: " + response.Message);
}
catch (RpcException ex)
{
    Console.WriteLine("Status code: " + ex.Status.StatusCode);
    Console.WriteLine("Message: " + ex.Status.Detail);
}
```

The preceding code:

- Makes a unary gRPC call to the `SayHello` method.
- Writes the response message to the console if it's successful.
- Catches `RpcException` and writes out the error details on failure.

Error scenarios

Errors are represented by `RpcException` with an error status code and optional detail message. `RpcException` is thrown in many scenarios:

- The call failed on the server and the server sent an error status code. For example, the gRPC client started a call that was missing required data from the request message and the server returns an `INVALID_ARGUMENT` status code.

- An error occurred inside the client when making the gRPC call. For example, a client makes a gRPC call, can't connect to the server, and throws an error with a status of `UNAVAILABLE`.
- The CancellationToken passed to the gRPC call is canceled. The gRPC call is stopped and the client throws an error with a status of `CANCELLED`.
- A gRPC call exceeds its configured deadline. The gRPC call is stopped and the client throws an error with a status of `DEADLINE_EXCEEDED`.

Rich error handling

Rich error handling allows complex, structured information to be sent with error messages. For example, validation of incoming message fields that returns a list of invalid field names and descriptions. The google.rpc.Status error model is often used to send complex error information between gRPC apps.

gRPC on .NET supports a rich error model using the Grpc.StatusProto package. This package has methods for creating rich error models on the server and reading them by a client. The rich error model builds on top of gRPC's built-in handling capabilities and they can be used side-by-side.

> **Important**
>
> Errors are included in headers, and total headers in responses are often limited to 8 KB (8,192 bytes). Ensure that the headers containing errors do not exceed 8 KB.

Creating rich errors on the server

Rich errors are created from `Google.Rpc.Status`. This type is ***different*** from `Grpc.Core.Status`.

`Google.Rpc.Status` has status, message, and details fields. The most important field is details, which is a repeating field of Any values. Details are where complex payloads are added.

Although any message type can be used as a payload, it's recommended to use one of the standard error payloads:

- `BadRequest`
- `PreconditionFailure`
- `ErrorInfo`

- `ResourceInfo`
- `QuotaFailure`

Grpc.StatusProto includes the `ToRpcException` a helper method to convert `Google.Rpc.Status` to an error. Throw the error from the gRPC server method:

cs

```
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        ArgumentNotNullOrEmpty(request.Name);

        return Task.FromResult(new HelloReply { Message = "Hello " +
request.Name });
    }

    public static void ArgumentNotNullOrEmpty(string value,
[CallerArgumentExpression(nameof(value))] string? paramName = null)
    {
        if (string.IsNullOrEmpty(value))
        {
            var status = new Google.Rpc.Status
            {
                Code = (int)Code.InvalidArgument,
                Message = "Bad request",
                Details =
                {
                    Any.Pack(new BadRequest
                    {
                        FieldViolations =
                        {
                            new BadRequest.Types.FieldViolation { Field =
paramName, Description = "Value is null or empty" }
                        }
                    })
                }
            };
            throw status.ToRpcException();
        }
    }
}
```

Reading rich errors by a client

Rich errors are read from the `RpcException` caught in the client. Catch the exception and use helper methods provided by `Grpc.StatusCode` to get its `Google.Rpc.Status` instance:

cs

```cs
var client = new Greet.GreeterClient(channel);

try
{
    var reply = await client.SayHelloAsync(new HelloRequest { Name = name
});
    Console.WriteLine("Greeting: " + reply.Message);
}
catch (RpcException ex)
{
    Console.WriteLine($"Server error: {ex.Status.Detail}");
    var badRequest = ex.GetRpcStatus()?.GetDetail<BadRequest>();
    if (badRequest != null)
    {
        foreach (var fieldViolation in badRequest.FieldViolations)
        {
            Console.WriteLine($"Field: {fieldViolation.Field}");
            Console.WriteLine($"Description: {fieldViolation.Description}");
        }
    }
}
```

The preceding code:

- Makes a gRPC call inside a try/catch that catches `RpcException`.
- Calls `GetRpcStatus()` to attempt to get the rich error model from the exception.
- Calls `GetDetail<BadRequest>()` to attempt to get a `BadRequest` payload from the rich error.

Additional resources

- Create gRPC services and methods
- Call gRPC services with the .NET client
- gRPC status codes

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

gRPC interceptors on .NET

Article • 07/31/2024

By Ernest Nguyen

Interceptors are a gRPC concept that allows apps to interact with incoming or outgoing gRPC calls. They offer a way to enrich the request processing pipeline.

Interceptors are configured for a channel or service and executed automatically for each gRPC call. Since interceptors are transparent to the user's application logic, they're an excellent solution for common cases, such as logging, monitoring, authentication, and validation.

`Interceptor` type

Interceptors can be implemented for both gRPC servers and clients by creating a class that inherits from the `Interceptor` type:

```
C#

public class ExampleInterceptor : Interceptor
{
}
```

By default, the `Interceptor` base class doesn't do anything. Add behavior to an interceptor by overriding the appropriate base class methods in an interceptor implementation.

Client interceptors

gRPC client interceptors intercept outgoing RPC invocations. They provide access to the sent request, the incoming response, and the context for a client-side call.

`Interceptor` methods to override for client:

- `BlockingUnaryCall`: Intercepts a blocking invocation of an unary RPC.
- `AsyncUnaryCall`: Intercepts an asynchronous invocation of an unary RPC.
- `AsyncClientStreamingCall`: Intercepts an asynchronous invocation of a client-streaming RPC.
- `AsyncServerStreamingCall`: Intercepts an asynchronous invocation of a server-streaming RPC.

- `AsyncDuplexStreamingCall`: Intercepts an asynchronous invocation of a bidirectional-streaming RPC.

> ⚠ **Warning**
>
> Although both `BlockingUnaryCall` and `AsyncUnaryCall` refer to unary RPCs, they aren't interchangeable. A blocking invocation isn't intercepted by `AsyncUnaryCall`, and an asynchronous invocation isn't intercepted by a `BlockingUnaryCall`.

Create a client gRPC interceptor

The following code presents a basic example of intercepting an asynchronous invocation of a unary call:

C#

```
public class ClientLoggingInterceptor : Interceptor
{
    private readonly ILogger _logger;

    public ClientLoggingInterceptor(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<ClientLoggingInterceptor>();
    }

    public override AsyncUnaryCall<TResponse> AsyncUnaryCall<TRequest,
TResponse>(
        TRequest request,
        ClientInterceptorContext<TRequest, TResponse> context,
        AsyncUnaryCallContinuation<TRequest, TResponse> continuation)
    {
        _logger.LogInformation("Starting call. Type/Method: {Type} /
{Method}",
            context.Method.Type, context.Method.Name);
        return continuation(request, context);
    }
}
```

Overriding `AsyncUnaryCall`:

- Intercepts an asynchronous unary call.
- Logs details about the call.
- Calls the `continuation` parameter passed into the method. This invokes the next interceptor in the chain or the underlying call invoker if this is the last interceptor.

Methods on `Interceptor` for each kind of service method have different signatures. However, the concept behind `continuation` and `context` parameters remains the same:

- `continuation` is a delegate which invokes the next interceptor in the chain or the underlying call invoker (if there is no interceptor left in the chain). It isn't an error to call it zero or multiple times. Interceptors aren't required to return a call representation (`AsyncUnaryCall` in case of unary RPC) returned from the `continuation` delegate. Omitting the delegate call and returning your own instance of call representation breaks the interceptors' chain and returns the associated response immediately.
- `context` carries scoped values associated with the client-side call. Use `context` to pass metadata, such as security principals, credentials, or tracing data. Moreover, `context` carries information about deadlines and cancellation. For more information, see Reliable gRPC services with deadlines and cancellation.

Awaiting response in client interceptor

An interceptor can await the response in unary and client streaming calls by updating the `AsyncUnaryCall<TResponse>.ResponseAsync` or `AsyncClientStreamingCall<TRequest, TResponse>.ResponseAsync` value.

C#

```
public class ErrorHandlerInterceptor : Interceptor
{
    public override AsyncUnaryCall<TResponse> AsyncUnaryCall<TRequest,
TResponse>(
        TRequest request,
        ClientInterceptorContext<TRequest, TResponse> context,
        AsyncUnaryCallContinuation<TRequest, TResponse> continuation)
    {
        var call = continuation(request, context);

        return new AsyncUnaryCall<TResponse>(
            HandleResponse(call.ResponseAsync),
            call.ResponseHeadersAsync,
            call.GetStatus,
            call.GetTrailers,
            call.Dispose);
    }

    private async Task<TResponse> HandleResponse<TResponse>(Task<TResponse>
inner)
    {
        try
        {
            return await inner;
```

```
        }
        catch (Exception ex)
        {
            throw new InvalidOperationException("Custom error", ex);
        }
    }
}
```

The preceding code:

- Creates a new interceptor that overrides `AsyncUnaryCall`.
- Overriding `AsyncUnaryCall`:
  - Calls the `continuation` parameter to invoke the next item in the interceptor chain.
  - Creates a new `AsyncUnaryCall<TResponse>` instance based on the result of the continuation.
  - Wraps the `ResponseAsync` task using the `HandleResponse` method.
  - Awaits the response with `HandleResponse`. Awaiting the response allows logic to be added after the client received the response. By awaiting the response in a try-catch block, errors from calls can be logged.

For more information on how to create a client interceptor, see the ClientLoggerInterceptor.cs example in the grpc/grpc-dotnet GitHub repository.

Configure client interceptors

gRPC client interceptors are configured on a channel.

The following code:

- Creates a channel by using `GrpcChannel.ForAddress`.
- Uses the `Intercept` extension method to configure the channel to use the interceptor. Note that this method returns a `CallInvoker`. Strongly-typed gRPC clients can be created from an invoker just like a channel.
- Creates a client from the invoker. gRPC calls made by the client automatically execute the interceptor.

C#

```
using var channel = GrpcChannel.ForAddress("https://localhost:5001");
var invoker = channel.Intercept(new ClientLoggerInterceptor());

var client = new Greeter.GreeterClient(invoker);
```

The `Intercept` extension method can be chained to configure multiple interceptors for a channel. Alternatively, there is an `Intercept` overload that accepts multiple interceptors. Any number of interceptors can be executed for a single gRPC call, as the following example demonstrates:

C#

```
var invoker = channel
    .Intercept(new ClientTokenInterceptor())
    .Intercept(new ClientMonitoringInterceptor())
    .Intercept(new ClientLoggerInterceptor());
```

Interceptors are invoked in reverse order of the chained `Intercept` extension methods. In the preceding code, interceptors are invoked in the following order:

1. `ClientLoggerInterceptor`
2. `ClientMonitoringInterceptor`
3. `ClientTokenInterceptor`

For information on how to configure interceptors with gRPC client factory, see gRPC client factory integration in .NET.

Server interceptors

gRPC server interceptors intercept incoming RPC requests. They provide access to the incoming request, the outgoing response, and the context for a server-side call.

`Interceptor` methods to override for server:

- `UnaryServerHandler`: Intercepts a unary RPC.
- `ClientStreamingServerHandler`: Intercepts a client-streaming RPC.
- `ServerStreamingServerHandler`: Intercepts a server-streaming RPC.
- `DuplexStreamingServerHandler`: Intercepts a bidirectional-streaming RPC.

Create a server gRPC interceptor

The following code presents an example of an intercepting an incoming unary RPC:

C#

```
public class ServerLoggerInterceptor : Interceptor
{
    private readonly ILogger _logger;
```

```
    public ServerLoggerInterceptor(ILogger<ServerLoggerInterceptor> logger)
    {
        _logger = logger;
    }

    public override async Task<TResponse> UnaryServerHandler<TRequest,
TResponse>(
        TRequest request,
        ServerCallContext context,
        UnaryServerMethod<TRequest, TResponse> continuation)
    {
        _logger.LogInformation("Starting receiving call. Type/Method: {Type}
/ {Method}",
            MethodType.Unary, context.Method);
        try
        {
            return await continuation(request, context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, $"Error thrown by {context.Method}.");
            throw;
        }
    }
}
```

Overriding `UnaryServerHandler`:

- Intercepts an incoming unary call.
- Logs details about the call.
- Calls the `continuation` parameter passed into the method. This invokes the next interceptor in the chain or the service handler if this is the last interceptor.
- Logs any exceptions. Awaiting the continuation allows logic to be added after the service method has executed. By awaiting the continuation in a try-catch block, errors from methods can be logged.

The signature of both client and server interceptors methods are similar:

- `continuation` stands for a delegate for an incoming RPC calling the next interceptor in the chain or the service handler (if there is no interceptor left in the chain). Similar to client interceptors, you can call it any time and there's no need to return a response directly from the continuation delegate. Outbound logic can be added after a service handler has executed by awaiting the continuation.
- `context` carries metadata associated with the server-side call, such as request metadata, deadlines and cancellation, or RPC result.

For more information on how to create a server interceptor, see the ServerLoggerInterceptor.cs example in the grpc/grpc-dotnet GitHub repository.

Configure server interceptors

gRPC server interceptors are configured at startup. The following code:

- Adds gRPC to the app with `AddGrpc`.
- Configures `ServerLoggerInterceptor` for all services by adding it to the service option's `Interceptors` collection.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.Interceptors.Add<ServerLoggerInterceptor>();
    });
}
```

An interceptor can also be configured for a specific service by using `AddServiceOptions` and specifying the service type.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services
        .AddGrpc()
        .AddServiceOptions<GreeterService>(options =>
        {
            options.Interceptors.Add<ServerLoggerInterceptor>();
        });
}
```

Interceptors are run in the order that they're added to the `InterceptorCollection`. If both global and single service interceptors are configured, then globally-configured interceptors are run before those configured for a single service.

By default, gRPC server interceptors have a per-request lifetime. Overriding this behavior is possible through registering the interceptor type with dependency injection. The following example registers the `ServerLoggerInterceptor` with a singleton lifetime:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.Interceptors.Add<ServerLoggerInterceptor>();
    });

    services.AddSingleton<ServerLoggerInterceptor>();
}
```

gRPC Interceptors versus Middleware

ASP.NET Core middleware offers similar functionalities compared to interceptors in C-core-based gRPC apps. ASP.NET Core middleware and interceptors are conceptually similar. Both:

- Are used to construct a pipeline that handles a gRPC request.
- Allow work to be performed before or after the next component in the pipeline.
- Provide access to `HttpContext`:
  - In middleware, the `HttpContext` is a parameter.
  - In interceptors, the `HttpContext` can be accessed using the `ServerCallContext` parameter with the `ServerCallContext.GetHttpContext` extension method. This feature is specific to interceptors running in ASP.NET Core.

gRPC Interceptor differences from ASP.NET Core Middleware:

- Interceptors:
  - Operate on the gRPC layer of abstraction using the ServerCallContext.
  - Provide access to:
    - The deserialized message sent to a call.
    - The message returned from the call before it's serialized.
  - Can catch and handle exceptions thrown from gRPC services.
- Middleware:
  - Runs for all HTTP requests.
  - Runs before gRPC interceptors.
  - Operates on the underlying HTTP/2 messages.
  - Can only access bytes from the request and response streams.

Additional resources

- Overview for gRPC on .NET
- Create gRPC services and methods

- Call gRPC services with the .NET client
- Example of how to use gRPC on the client and server (grpc/grpc-dotnet GitHub repository)
- Configure interceptors in a gRPC client factory in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Logging and diagnostics in gRPC on .NET

Article • 06/03/2022

By James Newton-King

This article provides guidance for gathering diagnostics from a gRPC app to help troubleshoot issues. Topics covered include:

- **Logging** - Structured logs written to .NET Core logging. ILogger is used by app frameworks to write logs, and by users for their own logging in an app.
- **Tracing** - Events related to an operation written using `DiaganosticSource` and `Activity`. Traces from diagnostic source are commonly used to collect app telemetry by libraries such as Application Insights and OpenTelemetry.
- **Metrics** - Representation of data measures over intervals of time, for example, requests per second. Metrics are emitted using `EventCounter` and can be observed using dotnet-counters command-line tool or with Application Insights.

Logging

gRPC services and the gRPC client write logs using .NET Core logging. Logs are a good place to start when debugging unexpected behavior in service and client apps.

gRPC services logging

> ⚠ **Warning**
>
> Server-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

Since gRPC services are hosted on ASP.NET Core, it uses the ASP.NET Core logging system. In the default configuration, gRPC logs minimal information, but logging can be configured. See the documentation on ASP.NET Core logging for details on configuring ASP.NET Core logging.

gRPC adds logs under the `Grpc` category. To enable detailed logs from gRPC, configure the `Grpc` prefixes to the `Debug` level in the `appsettings.json` file by adding the following items to the `LogLevel` subsection in `Logging`:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Grpc": "Debug"
    }
  }
}
```

Logging can also be configured in `Program.cs` with `ConfigureLogging`:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.AddFilter("Grpc", LogLevel.Debug);
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

When not using JSON-based configuration, set the following configuration value in the configuration system:

- `Logging:LogLevel:Grpc` = `Debug`

Check the documentation for your configuration system to determine how to specify nested configuration values. For example, when using environment variables, two `_` characters are used instead of the `:` (for example, `Logging__LogLevel__Grpc`).

We recommend using the `Debug` level when gathering detailed diagnostics for an app. The `Trace` level produces low-level diagnostics and is rarely needed to diagnose issues.

Sample logging output

Here is an example of console output at the `Debug` level of a gRPC service:

Console

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/2 POST
https://localhost:5001/Greet.Greeter/SayHello application/grpc
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint 'gRPC - /Greet.Greeter/SayHello'
dbug: Grpc.AspNetCore.Server.ServerCallHandler[1]
      Reading message.
info: GrpcService.GreeterService[0]
      Hello World
dbug: Grpc.AspNetCore.Server.ServerCallHandler[6]
      Sending message.
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint 'gRPC - /Greet.Greeter/SayHello'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 1.4113ms 200 application/grpc
```

Access server-side logs

How server-side logs are accessed depends on the app's environment.

As a console app

If you're running in a console app, the Console logger should be enabled by default. gRPC logs will appear in the console.

Other environments

If the app is deployed to another environment (for example, Docker, Kubernetes, or Windows Service), see Logging in .NET Core and ASP.NET Core for more information on how to configure logging providers suitable for the environment.

gRPC client logging

> ⚠ **Warning**
>
> Client-side logs may contain sensitive information from your app. **Never** post raw logs from production apps to public forums like GitHub.

To get logs from the .NET client, set the `GrpcChannelOptions.LoggerFactory` property when the client's channel is created. When calling a gRPC service from an ASP.NET Core app, the logger factory can be resolved from dependency injection (DI):

C#

```
[ApiController]
[Route("[controller]")]
public class GreetingController : ControllerBase
{
    private ILoggerFactory _loggerFactory;

    public GreetingController(ILoggerFactory loggerFactory)
    {
        _loggerFactory = loggerFactory;
    }

    [HttpGet]
    public async Task<ActionResult<string>> Get(string name)
    {
        var channel = GrpcChannel.ForAddress("https://localhost:5001",
            new GrpcChannelOptions { LoggerFactory = _loggerFactory });
        var client = new Greeter.GreeterClient(channel);

        var reply = await client.SayHelloAsync(new HelloRequest { Name =
name });
        return Ok(reply.Message);
    }
}
```

An alternative way to enable client logging is to use the gRPC client factory to create the client. A gRPC client registered with the client factory and resolved from DI will automatically use the app's configured logging.

If the app isn't using DI, then create a new `ILoggerFactory` instance with LoggerFactory.Create. To access this method, add the Microsoft.Extensions.Logging package to your app.

C#

```
var loggerFactory = LoggerFactory.Create(logging =>
{
    logging.AddConsole();
    logging.SetMinimumLevel(LogLevel.Debug);
});

var channel = GrpcChannel.ForAddress("https://localhost:5001",
    new GrpcChannelOptions { LoggerFactory = loggerFactory });

var client = Greeter.GreeterClient(channel);
```

gRPC client log scopes

The gRPC client adds a logging scope to logs made during a gRPC call. The scope has metadata related to the gRPC call:

- **GrpcMethodType** - The gRPC method type. Possible values are names from `Grpc.Core.MethodType` enum. For example, `Unary`.
- **GrpcUri** - The relative URI of the gRPC method. For example, */greet.Greeter/SayHellos*.

Sample logging output

Here is an example of console output at the `Debug` level of a gRPC client:

Console

```
dbug: Grpc.Net.Client.Internal.GrpcCall[1]
      Starting gRPC call. Method type: 'Unary', URI:
'https://localhost:5001/Greet.Greeter/SayHello'.
dbug: Grpc.Net.Client.Internal.GrpcCall[6]
      Sending message.
dbug: Grpc.Net.Client.Internal.GrpcCall[1]
      Reading message.
dbug: Grpc.Net.Client.Internal.GrpcCall[4]
      Finished gRPC call.
```

Tracing

gRPC services and the gRPC client provide information about gRPC calls using DiagnosticSource and Activity.

- .NET gRPC uses an activity to represent a gRPC call.
- Tracing events are written to the diagnostic source at the start and stop of the gRPC call activity.
- Tracing doesn't capture information about when messages are sent over the lifetime of gRPC streaming calls.

gRPC service tracing

gRPC services are hosted on ASP.NET Core, which reports events about incoming HTTP requests. gRPC specific metadata is added to the existing HTTP request diagnostics that ASP.NET Core provides.

- Diagnostic source name is `Microsoft.AspNetCore`.
- Activity name is `Microsoft.AspNetCore.Hosting.HttpRequestIn`.

    - Name of the gRPC method invoked by the gRPC call is added as a tag with the name `grpc.method`.
    - Status code of the gRPC call when it is complete is added as a tag with the name `grpc.status_code`.

gRPC client tracing

The .NET gRPC client uses `HttpClient` to make gRPC calls. Although `HttpClient` writes diagnostic events, the .NET gRPC client provides a custom diagnostic source, activity, and events so that complete information about a gRPC call can be collected.

- Diagnostic source name is `Grpc.Net.Client`.
- Activity name is `Grpc.Net.Client.GrpcOut`.
    - Name of the gRPC method invoked by the gRPC call is added as a tag with the name `grpc.method`.
    - Status code of the gRPC call when it is complete is added as a tag with the name `grpc.status_code`.

Collecting tracing

The easiest way to use `DiagnosticSource` is to configure a telemetry library such as Application Insights or OpenTelemetry in your app. The library will process information about gRPC calls along-side other app telemetry.

Tracing can be viewed in a managed service like Application Insights, or run as your own distributed tracing system. OpenTelemetry supports exporting tracing data to Jaeger and Zipkin.

`DiagnosticSource` can consume tracing events in code using `DiagnosticListener`. For information about listening to a diagnostic source with code, see the DiagnosticSource user's guide.

> **Note**
>
> Telemetry libraries do not capture gRPC specific `Grpc.Net.Client.GrpcOut` telemetry currently. Work to improve telemetry libraries capturing this tracing is ongoing.

Metrics

Metrics is a representation of data measures over intervals of time, for example, requests per second. Metrics data allows observation of the state of an app at a high level. .NET gRPC metrics are emitted using `EventCounter`.

gRPC service metrics

gRPC server metrics are reported on `Grpc.AspNetCore.Server` event source.

Expand table

Name	Description
`total-calls`	Total Calls
`current-calls`	Current Calls
`calls-failed`	Total Calls Failed
`calls-deadline-exceeded`	Total Calls Deadline Exceeded
`messages-sent`	Total Messages Sent
`messages-received`	Total Messages Received
`calls-unimplemented`	Total Calls Unimplemented

ASP.NET Core also provides its own metrics on `Microsoft.AspNetCore.Hosting` event source.

gRPC client metrics

gRPC client metrics are reported on `Grpc.Net.Client` event source.

Expand table

Name	Description
`total-calls`	Total Calls
`current-calls`	Current Calls
`calls-failed`	Total Calls Failed
`calls-deadline-exceeded`	Total Calls Deadline Exceeded
`messages-sent`	Total Messages Sent

Name	Description
`messages-received`	Total Messages Received

Observe metrics

dotnet-counters is a performance monitoring tool for ad-hoc health monitoring and first-level performance investigation. Monitor a .NET app with either `Grpc.AspNetCore.Server` or `Grpc.Net.Client` as the provider name.

Console

```
> dotnet-counters monitor --process-id 1902 Grpc.AspNetCore.Server

Press p to pause, r to resume, q to quit.
    Status: Running
[Grpc.AspNetCore.Server]
    Total Calls                                300
    Current Calls                              5
    Total Calls Failed                         0
    Total Calls Deadline Exceeded              0
    Total Messages Sent                        295
    Total Messages Received                    300
    Total Calls Unimplemented                  0
```

Another way to observe gRPC metrics is to capture counter data using Application Insights's Microsoft.ApplicationInsights.EventCounterCollector package. Once setup, Application Insights collects common .NET counters at runtime. gRPC's counters are not collected by default, but App Insights can be customized to include additional counters.

Specify the gRPC counters for Application Insight to collect in `Startup.cs`:

C#

```
    using Microsoft.ApplicationInsights.Extensibility.EventCounterCollector;

    public void ConfigureServices(IServiceCollection services)
    {
        //... other code...

        services.ConfigureTelemetryModule<EventCounterCollectionModule>(
            (module, o) =>
            {
                // Configure App Insights to collect gRPC counters gRPC
services hosted in an ASP.NET Core app
                module.Counters.Add(new
EventCounterCollectionRequest("Grpc.AspNetCore.Server", "current-calls"));
                module.Counters.Add(new
```

```
EventCounterCollectionRequest("Grpc.AspNetCore.Server", "total-calls"));
                module.Counters.Add(new
EventCounterCollectionRequest("Grpc.AspNetCore.Server", "calls-failed"));
            }
        );
    }
```

Additional resources

- Logging in .NET Core and ASP.NET Core
- gRPC for .NET configuration
- gRPC client factory integration in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Security considerations in gRPC for ASP.NET Core

Article • 07/31/2024

By James Newton-King

This article provides information on securing gRPC with .NET Core.

Transport security

gRPC messages are sent and received using HTTP/2. We recommend:

- Transport Layer Security (TLS) be used to secure messages in production gRPC apps.
- gRPC services should only listen and respond over secured ports.

TLS is configured in Kestrel. For more information on configuring Kestrel endpoints, see Kestrel endpoint configuration.

A TLS termination proxy can be combined with TLS. The benefits of using TLS termination should be considered against the security risks of sending unsecured HTTP requests between apps in the private network.

Exceptions

Exception messages are generally considered sensitive data that shouldn't be revealed to a client. By default, gRPC doesn't send the details of an exception thrown by a gRPC service to the client. Instead, the client receives a generic message indicating an error occurred. Exception message delivery to the client can be overridden (for example, in development or test) with EnableDetailedErrors. Exception messages shouldn't be exposed to the client in production apps.

Message size limits

Incoming messages to gRPC clients and services are loaded into memory. Message size limits are a mechanism to help prevent gRPC from consuming excessive resources.

gRPC uses per-message size limits to manage incoming and outgoing messages. By default, gRPC limits incoming messages to 4 MB. There is no limit on outgoing messages.

On the server, gRPC message limits can be configured for all services in an app with `AddGrpc`:

```C#
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.MaxReceiveMessageSize = 1 * 1024 * 1024; // 1 MB
        options.MaxSendMessageSize = 1 * 1024 * 1024; // 1 MB
    });
}
```

Limits can also be configured for an individual service using `AddServiceOptions<TService>`. For more information on configuring message size limits, see gRPC configuration.

Client certificate validation

Client certificates are initially validated when the connection is established. By default, Kestrel doesn't perform additional validation of a connection's client certificate.

We recommend that gRPC services secured by client certificates use the Microsoft.AspNetCore.Authentication.Certificate package. ASP.NET Core certification authentication will perform additional validation on a client certificate, including:

- Certificate has a valid extended key use (EKU)
- Is within its validity period
- Check certificate revocation

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Performance best practices with gRPC

Article • 06/07/2024

By James Newton-King

gRPC is designed for high-performance services. This document explains how to get the best performance possible from gRPC.

Reuse gRPC channels

A gRPC channel should be reused when making gRPC calls. Reusing a channel allows calls to be multiplexed through an existing HTTP/2 connection.

If a new channel is created for each gRPC call then the amount of time it takes to complete can increase significantly. Each call will require multiple network round-trips between the client and the server to create a new HTTP/2 connection:

1. Opening a socket
2. Establishing TCP connection
3. Negotiating TLS
4. Starting HTTP/2 connection
5. Making the gRPC call

Channels are safe to share and reuse between gRPC calls:

- gRPC clients are created with channels. gRPC clients are lightweight objects and don't need to be cached or reused.
- Multiple gRPC clients can be created from a channel, including different types of clients.
- A channel and clients created from the channel can safely be used by multiple threads.
- Clients created from the channel can make multiple simultaneous calls.

gRPC client factory offers a centralized way to configure channels. It automatically reuses underlying channels. For more information, see gRPC client factory integration in .NET.

Connection concurrency

HTTP/2 connections typically have a limit on the number of maximum concurrent streams (active HTTP requests) on a connection at one time. By default, most servers

set this limit to 100 concurrent streams.

A gRPC channel uses a single HTTP/2 connection, and concurrent calls are multiplexed on that connection. When the number of active calls reaches the connection stream limit, additional calls are queued in the client. Queued calls wait for active calls to complete before they are sent. Applications with high load, or long running streaming gRPC calls, could see performance issues caused by calls queuing because of this limit.

.NET 5 introduces the SocketsHttpHandler.EnableMultipleHttp2Connections property. When set to `true`, additional HTTP/2 connections are created by a channel when the concurrent stream limit is reached. When a `GrpcChannel` is created its internal `SocketsHttpHandler` is automatically configured to create additional HTTP/2 connections. If an app configures its own handler, consider setting `EnableMultipleHttp2Connections` to `true`:

C#

```
var channel = GrpcChannel.ForAddress("https://localhost", new
GrpcChannelOptions
{
    HttpHandler = new SocketsHttpHandler
    {
        EnableMultipleHttp2Connections = true,

        // ...configure other handler settings
    }
});
```

.NET Framework apps that make gRPC calls must be configured to use WinHttpHandler. .NET Framework apps can set the WinHttpHandler.EnableMultipleHttp2Connections property to `true` to create additional connections.

There are a couple of workarounds for .NET Core 3.1 apps:

- Create separate gRPC channels for areas of the app with high load. For example, the `Logger` gRPC service might have a high load. Use a separate channel to create the `LoggerClient` in the app.
- Use a pool of gRPC channels, for example, create a list of gRPC channels. `Random` is used to pick a channel from the list each time a gRPC channel is needed. Using `Random` randomly distributes calls over multiple connections.

ⓘ Important

Increasing the maximum concurrent stream limit on the server is another way to solve this problem. In Kestrel this is configured with MaxStreamsPerConnection.

Increasing the maximum concurrent stream limit is not recommended. Too many streams on a single HTTP/2 connection introduces new performance issues:

- Thread contention between streams trying to write to the connection.
- Connection packet loss causes all calls to be blocked at the TCP layer.

`ServerGarbageCollection` in client apps

The .NET garbage collector has two modes: workstation garbage collection (GC) and server garbage collection. Each is each tuned for different workloads. ASP.NET Core apps use server GC by default.

Highly concurrent apps generally perform better with server GC. If a gRPC client app is sending and receiving a high number of gRPC calls at the same time, then there may be a performance benefit in updating the app to use server GC.

To enable server GC, set `<ServerGarbageCollection>` in the app's project file:

XML

```xml
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

For more information about garbage collection, see Workstation and server garbage collection.

> ⓘ Note
>
> ASP.NET Core apps use server GC by default. Enabling `<ServerGarbageCollection>` is only useful in non-server gRPC client apps, for example in a gRPC client console app.

Load balancing

Some load balancers don't work effectively with gRPC. L4 (transport) load balancers operate at a connection level, by distributing TCP connections across endpoints. This approach works well for loading balancing API calls made with HTTP/1.1. Concurrent

calls made with HTTP/1.1 are sent on different connections, allowing calls to be load balanced across endpoints.

Because L4 load balancers operate at a connection level, they don't work well with gRPC. gRPC uses HTTP/2, which multiplexes multiple calls on a single TCP connection. All gRPC calls over that connection go to one endpoint.

There are two options to effectively load balance gRPC:

- Client-side load balancing
- L7 (application) proxy load balancing

> **Note**
>
> Only gRPC calls can be load balanced between endpoints. Once a streaming gRPC call is established, all messages sent over the stream go to one endpoint.

Client-side load balancing

With client-side load balancing, the client knows about endpoints. For each gRPC call, it selects a different endpoint to send the call to. Client-side load balancing is a good choice when latency is important. There's no proxy between the client and the service, so the call is sent to the service directly. The downside to client-side load balancing is that each client must keep track of the available endpoints that it should use.

Lookaside client load balancing is a technique where load balancing state is stored in a central location. Clients periodically query the central location for information to use when making load balancing decisions.

For more information, see gRPC client-side load balancing.

Proxy load balancing

An L7 (application) proxy works at a higher level than an L4 (transport) proxy. L7 proxies understand HTTP/2. The proxy receives gRPC calls multiplexed on one HTTP/2 connection and distributes them across multiple backend endpoints. Using a proxy is simpler than client-side load balancing, but adds extra latency to gRPC calls.

There are many L7 proxies available. Some options are:

- Envoy - A popular open source proxy.
- Linkerd - Service mesh for Kubernetes.

- YARP: Yet Another Reverse Proxy - An open source proxy written in .NET.

Inter-process communication

gRPC calls between a client and service are usually sent over TCP sockets. TCP is great for communicating across a network, but inter-process communication (IPC) is more efficient when the client and service are on the same machine.

Consider using a transport like Unix domain sockets or named pipes for gRPC calls between processes on the same machine. For more information, see Inter-process communication with gRPC.

Keep alive pings

Keep alive pings can be used to keep HTTP/2 connections alive during periods of inactivity. Having an existing HTTP/2 connection ready when an app resumes activity allows for the initial gRPC calls to be made quickly, without a delay caused by the connection being reestablished.

Keep alive pings are configured on SocketsHttpHandler:

C#

```
var handler = new SocketsHttpHandler
{
    PooledConnectionIdleTimeout = Timeout.InfiniteTimeSpan,
    KeepAlivePingDelay = TimeSpan.FromSeconds(60),
    KeepAlivePingTimeout = TimeSpan.FromSeconds(30),
    EnableMultipleHttp2Connections = true
};

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions
{
    HttpHandler = handler
});
```

The preceding code configures a channel that sends a keep alive ping to the server every 60 seconds during periods of inactivity. The ping ensures the server and any proxies in use won't close the connection because of inactivity.

> **Note**

Keep alive pings only help keep the connection alive. Long-running gRPC calls on the connection may still be terminated by the server or intermediary proxies for inactivity.

Flow control

HTTP/2 flow control is a feature that prevents apps from being overwhelmed with data. When using flow control:

- Each HTTP/2 connection and request has an available buffer window. The buffer window is how much data the app can receive at once.
- Flow control activates if the buffer window is filled up. When activated, the sending app pauses sending more data.
- Once the receiving app has processed data, then space in the buffer window is available. The sending app resumes sending data.

Flow control can have a negative impact on performance when receiving large messages. If the buffer window is smaller than incoming message payloads or there's latency between the client and server, then data can be sent in start/stop bursts.

Flow control performance issues can be fixed by increasing buffer window size. In Kestrel, this is configured with InitialConnectionWindowSize and InitialStreamWindowSize at app startup:

C#

```
builder.WebHost.ConfigureKestrel(options =>
{
    var http2 = options.Limits.Http2;
    http2.InitialConnectionWindowSize = 1024 * 1024 * 2; // 2 MB
    http2.InitialStreamWindowSize = 1024 * 1024; // 1 MB
});
```

Recommendations:

- If a gRPC service often receives messages larger than 768 KB, Kestrel's default stream window size, then consider increasing the connection and stream window size.
- The connection window size should always be equal to or greater than the stream window size. A stream is part of the connection, and the sender is limited by both.

For more information about how flow control works, see HTTP/2 Flow Control (blog post).

> **Important**
>
> Increasing Kestrel's window size allows Kestrel to buffer more data on behalf of the app, which possibly increases memory usage. Avoid configuring an unnecessarily large window size.

Gracefully complete streaming calls

Try to complete streaming calls gracefully. Gracefully completing calls avoids unnecessary errors and allow servers to reuse internal data structures between requests.

A call is completed gracefully when the client and server have finished sending messages and the peer has read all the messages.

Client request stream:

1. The client has finished writing messages to the request stream and completes the stream with `call.RequestStream.CompleteAsync()`.
2. The server has read all messages from the request stream. Depending on how you're reading messages, either `requestStream.MoveNext()` returns `false` or `requestStream.ReadAllAsync()` has finished.

Server response stream:

1. The server has finished writing messages to the response stream and the server method has exited.
2. The client has read all messages from the response stream. Depending on how you're reading messages, either `call.ResponseStream.MoveNext()` returns `false` or `call.ResponseStream.ReadAllAsync()` has finished.

For an example of gracefully completing a bi-direction streaming call, see making a bi-directional streaming call.

Server streaming calls don't have a request stream. This means that the only way a client can communicate to the server that the stream should stop is by canceling it. If the overhead from canceled calls is impacting the app then consider changing the server streaming call to a bi-directional streaming call. In a bi-directional streaming call the client completing the request stream can be a signal to the server to end the call.

Dispose streaming calls

Always dispose streaming calls once they're no longer needed. The type returned when starting streaming calls implements `IDisposable`. Disposing a call once it is no longer needed ensures it is stopped and all resources are cleaned up.

In the following example, the using declaration on the `AccumulateCount()` call ensures it's always disposed if an unexpected error occurs.

C#

```
var client = new Counter.CounterClient(channel);
using var call = client.AccumulateCount();

for (var i = 0; i < 3; i++)
{
    await call.RequestStream.WriteAsync(new CounterRequest { Count = 1 });
}
await call.RequestStream.CompleteAsync();

var response = await call;
Console.WriteLine($"Count: {response.Count}");
// Count: 3
```

Ideally streaming calls should be completed gracefully. Disposing the call ensures the HTTP request between the client and the server is canceled if an unexpected error occurs. Streaming calls that are accidentally left running don't just leak memory and resources on the client, but are left running on the server as well. Many leaked streaming calls could impact the stability of the app.

Disposing of a streaming call that has already completed gracefully doesn't have any negative impact.

Replace unary calls with streaming

gRPC bidirectional streaming can be used to replace unary gRPC calls in high-performance scenarios. Once a bidirectional stream has started, streaming messages back and forth is faster than sending messages with multiple unary gRPC calls. Streamed messages are sent as data on an existing HTTP/2 request and eliminates the overhead of creating a new HTTP/2 request for each unary call.

Example service:

C#

```
public override async Task SayHello(IAsyncStreamReader<HelloRequest>
requestStream,
    IServerStreamWriter<HelloReply> responseStream, ServerCallContext
```

```
context)
{
    await foreach (var request in requestStream.ReadAllAsync())
    {
        var helloReply = new HelloReply { Message = "Hello " + request.Name
};

        await responseStream.WriteAsync(helloReply);
    }
}
```

Example client:

C#

```
var client = new Greet.GreeterClient(channel);
using var call = client.SayHello();

Console.WriteLine("Type a name then press enter.");
while (true)
{
    var text = Console.ReadLine();

    // Send and receive messages over the stream
    await call.RequestStream.WriteAsync(new HelloRequest { Name = text });
    await call.ResponseStream.MoveNext();

    Console.WriteLine($"Greeting: {call.ResponseStream.Current.Message}");
}
```

Replacing unary calls with bidirectional streaming for performance reasons is an advanced technique and is not appropriate in many situations.

Using streaming calls is a good choice when:

1. High throughput or low latency is required.
2. gRPC and HTTP/2 are identified as a performance bottleneck.
3. A worker in the client is sending or receiving regular messages with a gRPC service.

Be aware of the additional complexity and limitations of using streaming calls instead of unary:

1. A stream can be interrupted by a service or connection error. Logic is required to restart stream if there is an error.
2. `RequestStream.WriteAsync` is not safe for multi-threading. Only one message can be written to a stream at a time. Sending messages from multiple threads over a single stream requires a producer/consumer queue like Channel<T> to marshall messages.

3. A gRPC streaming method is limited to receiving one type of message and sending one type of message. For example, `rpc StreamingCall(stream RequestMessage) returns (stream ResponseMessage)` receives `RequestMessage` and sends `ResponseMessage`. Protobuf's support for unknown or conditional messages using `Any` and `oneof` can work around this limitation.

Binary payloads

Binary payloads are supported in Protobuf with the `bytes` scalar value type. A generated property in C# uses `ByteString` as the property type.

ProtoBuf

```
syntax = "proto3";

message PayloadResponse {
    bytes data = 1;
}
```

Protobuf is a binary format that efficiently serializes large binary payloads with minimal overhead. Text based formats like JSON require encoding bytes to base64 and add 33% to the message size.

When working with large `ByteString` payloads there are some best practices to avoid unnecessary copies and allocations that are discussed below.

Send binary payloads

`ByteString` instances are normally created using `ByteString.CopyFrom(byte[] data)`. This method allocates a new `ByteString` and a new `byte[]`. Data is copied into the new byte array.

Additional allocations and copies can be avoided by using `UnsafeByteOperations.UnsafeWrap(ReadOnlyMemory<byte> bytes)` to create `ByteString` instances.

C#

```
var data = await File.ReadAllBytesAsync(path);

var payload = new PayloadResponse();
payload.Data = UnsafeByteOperations.UnsafeWrap(data);
```

Bytes are not copied with `UnsafeByteOperations.UnsafeWrap` so they must not be modified while the `ByteString` is in use.

`UnsafeByteOperations.UnsafeWrap` requires Google.Protobuf version 3.15.0 or later.

Read binary payloads

Data can be efficiently read from `ByteString` instances by using `ByteString.Memory` and `ByteString.Span` properties.

C#

```
var byteString = UnsafeByteOperations.UnsafeWrap(new byte[] { 0, 1, 2 });
var data = byteString.Span;

for (var i = 0; i < data.Length; i++)
{
    Console.WriteLine(data[i]);
}
```

These properties allow code to read data directly from a `ByteString` without allocations or copies.

Most .NET APIs have `ReadOnlyMemory<byte>` and `byte[]` overloads, so `ByteString.Memory` is the recommended way to use the underlying data. However, there are circumstances where an app might need to get the data as a byte array. If a byte array is required then the MemoryMarshal.TryGetArray method can be used to get an array from a `ByteString` without allocating a new copy of the data.

C#

```
var byteString = GetByteString();

ByteArrayContent content;
if (MemoryMarshal.TryGetArray(byteString.Memory, out var segment))
{
    // Success. Use the ByteString's underlying array.
    content = new ByteArrayContent(segment.Array, segment.Offset,
segment.Count);
}
else
{
    // TryGetArray didn't succeed. Fall back to creating a copy of the data
with ToByteArray.
    content = new ByteArrayContent(byteString.ToByteArray());
}
```

```
var httpRequest = new HttpRequestMessage();
httpRequest.Content = content;
```

The preceding code:

- Attempts to get an array from `ByteString.Memory` with MemoryMarshal.TryGetArray.
- Uses the `ArraySegment<byte>` if it was successfully retrieved. The segment has a reference to the array, offset and count.
- Otherwise, falls back to allocating a new array with `ByteString.ToByteArray()`.

gRPC services and large binary payloads

gRPC and Protobuf can send and receive large binary payloads. Although binary Protobuf is more efficient than text-based JSON at serializing binary payloads, there are still important performance characteristics to keep in mind when working with large binary payloads.

gRPC is a message-based RPC framework, which means:

- The entire message is loaded into memory before gRPC can send it.
- When the message is received, the entire message is deserialized into memory.

Binary payloads are allocated as a byte array. For example, a 10 MB binary payload allocates a 10 MB byte array. Messages with large binary payloads can allocate byte arrays on the large object heap. Large allocations impact server performance and scalability.

Advice for creating high-performance applications with large binary payloads:

- **Avoid** large binary payloads in gRPC messages. A byte array larger than 85,000 bytes is considered a large object. Keeping below that size avoids allocating on the large object heap.
- **Consider** splitting large binary payloads using gRPC streaming. Binary data is chunked and streamed over multiple messages. For more information on how to stream files, see examples in the grpc-dotnet repository:
  - gRPC streaming file download.
  - gRPC streaming file upload.
- **Consider** not using gRPC for large binary data. In ASP.NET Core, Web APIs can be used alongside gRPC services. An HTTP endpoint can access the request and response stream body directly:
  - Read the request body using minimal web API
  - Return stream response using minimal web API

gRPC and Native AOT

Article • 02/13/2024

By James Newton-King

gRPC supports .NET native ahead-of-time (AOT) in .NET 8. Native AOT enables publishing gRPC client and server apps as small, fast native executables.

> ⚠ **Warning**
>
> In .NET 8, not all ASP.NET Core features are compatible with Native AOT. For more information, see ASP.NET Core and Native AOT compatibility.

Get started

AOT compilation happens when the app is published. Native AOT is enabled with the `PublishAot` option.

1. Add `<PublishAot>true</PublishAot>` to the gRPC client or server app's project file. This will enable Native AOT compilation during publish and enable dynamic code usage analysis during build and editing.

   XML

   ```xml
   <Project Sdk="Microsoft.NET.Sdk.Web">

     <PropertyGroup>
       <TargetFramework>net8.0</TargetFramework>
       <PublishAot>true</PublishAot>
     </PropertyGroup>

     <ItemGroup>
       <PackageReference Include="Grpc.AspNetCore" Version="2.51.0" />
       <PackageReference Include="Google.Protobuf" Version="3.22.0" />
     </ItemGroup>

   </Project>
   ```

   Native AOT can also be enabled by specifying the `-aot` option with the ASP.NET Core gRPC template:

   .NET CLI

```
dotnet new grpc -aot
```

2. Publish the app for a specific runtime identifier (RID) using `dotnet publish -r <RID>`.

The app is available in the publish directory and contains all the code needed to run in it.

Native AOT analysis includes all of the app's code and the libraries the app depends on. Review Native AOT warnings and take corrective steps. It's a good idea to test publishing apps frequently to discover issues early in the development lifecycle.

Optimize publish size

A Native AOT executable contains just the code from external dependencies required to support the app. Unused code is automatically trimmed away.

The publish size of an ASP.NET Core gRPC service can be optimized by creating the host builder with `WebApplication.CreateSlimBuilder()`. This builder provides a minimal list of features required to run an ASP.NET Core app.

C#

```
var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.AddGrpc();

var app = builder.Build();
app.MapGrpcService<GreeterService>();
app.Run();
```

Benefits of using Native AOT

Apps published with Native AOT have:

- Minimized disk footprint
- Reduced startup time
- Reduce memory demand

For more information and examples of the benefits that Native AOT provides, see Benefits of using Native AOT with ASP.NET Core.

Additional resources

- ASP.NET Core support for Native AOT
- Native AOT deployment

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Inter-process communication with gRPC

Article • 07/31/2024

Processes running on the same machine can be designed to communicate with each other. Operating systems provide technologies for enabling fast and efficient inter-process communication (IPC). Popular examples of IPC technologies are Unix domain sockets and Named pipes.

.NET provides support for inter-process communication using gRPC.

Built-in support for Named pipes in ASP.NET Core requires .NET 8 or later.

Get started

IPC calls are sent from a client to a server. To communicate between apps on a machine with gRPC, at least one app must host an ASP.NET Core gRPC server.

An ASP.NET Core gRPC server is usually created from the gRPC template. The project file created by the template uses `Microsoft.NET.SDK.Web` as the SDK:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.47.0" />
    <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
  </ItemGroup>

</Project>
```

The `Microsoft.NET.SDK.Web` SDK value automatically adds a reference to the ASP.NET Core framework. The reference allows the app to use ASP.NET Core types required to host a server.

It's also possible to add a server to existing non-ASP.NET Core projects, such as Windows Services, WPF apps, or WinForms apps. See Host gRPC in non-ASP.NET Core projects for more information.

Inter-process communication (IPC) transports

gRPC calls between a client and server on different machines are usually sent over TCP sockets. TCP is a good choice for communicating across a network or the Internet. However, IPC transports offer advantages when communicating between processes on the same machine:

- Less overhead and faster transfer speeds.
- Integration with OS security features.
- Doesn't use TCP ports, which are a limited resource.

.NET supports multiple IPC transports:

- Unix domain sockets (UDS) is a widely supported IPC technology. UDS is the best choice for building cross-platform apps, and it's usable on Linux, macOS, and Windows 10/Windows Server 2019 or later.
- Named pipes are supported by all versions of Windows. Named pipes integrate well with Windows security, which can control client access to the pipe.
- Additional IPC transports by implementing IConnectionListenerFactory and registering the implementation at app startup.

Depending on the OS, cross-platform apps may choose different IPC transports. An app can check the OS on startup and choose the desired transport for that platform:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    if (OperatingSystem.IsWindows())
    {
        serverOptions.ListenNamedPipe("MyPipeName");
    }
    else
    {
        var socketPath = Path.Combine(Path.GetTempPath(), "socket.tmp");
        serverOptions.ListenUnixSocket(socketPath);
    }

    serverOptions.ConfigureEndpointDefaults(listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
    });
});
```

Security considerations

IPC apps send and receive RPC calls. External communication is a potential attack vector for IPC apps and must be properly secured.

Secure IPC server app against unexpected callers

The IPC server app hosts RPC services for other apps to call. Incoming callers should be authenticated to prevent untrusted clients from making RPC calls to the server.

Transport security is one option for securing a server. IPC transports, such as Unix domain sockets and named pipes, support limiting access based on operating system permissions:

- Named pipes supports securing a pipe with the Windows access control model. Access rights can be configured in .NET when a server is started using the PipeSecurity class.
- Unix domain sockets support securing a socket with file permissions.

Another option for securing an IPC server is to use authentication and authorization built into ASP.NET Core. For example, the server could be configured to require certificate authentication. RPC calls made by client apps without the required certificate fail with an unauthorized response.

Validate the server in the IPC client app

It's important for the client app to validate the identity of the server it is calling. Validation is necessary to protect against a malicious actor from stopping the trusted server, running their own, and accepting incoming data from clients.

Named pipes provides support for getting the account that a server is running under. A client can validate the server was launched by the expected account:

cs

```
internal static bool CheckPipeConnectionOwnership(
    NamedPipeClientStream pipeStream, SecurityIdentifier expectedOwner)
{
    var remotePipeSecurity = pipeStream.GetAccessControl();
    var remoteOwner =
remotePipeSecurity.GetOwner(typeof(SecurityIdentifier));
    return expectedOwner.Equals(remoteOwner);
}
```

Another option for validating the server is to secure its endpoints with HTTPS inside ASP.NET Core. The client can configure `SocketsHttpHandler` to validate the server is

using the expected certificate when the connection is established.

```cs
var socketsHttpHandler = new SocketsHttpHandler()
{
    SslOptions = new SslOptions()
    {
        RemoteCertificateValidationCallback = (sender, certificate, chain,
sslPolicyErrors) =>
        {
            if (sslPolicyErrors != SslPolicyErrors.None)
            {
                return false;
            }

            // Validate server cert thumbprint matches the expected
thumbprint.
        }
    }
};
```

Protect against named pipe privilege escalation

Named pipes supports a feature called impersonation. Using impersonation, the named pipes server can execute code with the privileges of the client user. This is a powerful feature but can allow a low-privilege server to impersonate a high-privilege caller and then run malicious code.

Client's can protect against this attack by not allowing impersonation when connecting to a server. Unless required by a server, a TokenImpersonationLevel value of `None` or `Anonymous` should be used when creating a client connection:

```cs
using var pipeClient = new NamedPipeClientStream(
    serverName: ".", pipeName: "testpipe", PipeDirection.In,
PipeOptions.None, TokenImpersonationLevel.None);
await pipeClient.ConnectAsync();
```

`TokenImpersonationLevel.None` is the default value in `NamedPipeClientStream` constructors that don't have an `impersonationLevel` parameter.

Configure client and server

The client and server must be configured to use an inter-process communication (IPC) transport. For more information about configuring Kestrel and SocketsHttpHandler to use IPC:

- Inter-process communication with gRPC and Unix domain sockets
- Inter-process communication with gRPC and Named pipes

> **Note**
>
> Built-in support for Named pipes in ASP.NET Core requires .NET 8 or later.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Inter-process communication with gRPC and Unix domain sockets

Article • 07/31/2024

By James Newton-King

.NET supports inter-process communication (IPC) using gRPC. For more information about getting started with using gRPC to communicate between processes, see Inter-process communication with gRPC.

Unix domain sockets (UDS) is a widely supported IPC transport that's more efficient than TCP when the client and server are on the same machine. This article discusses how to configure gRPC communication over UDS.

Prerequisites

- .NET 5 or later
- Linux, macOS, or Windows 10/Windows Server 2019 or later

Server configuration

Unix domain sockets are supported by Kestrel, which is configured in `Program.cs`:

C#

```
var socketPath = Path.Combine(Path.GetTempPath(), "socket.tmp");

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenUnixSocket(socketPath, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
    });
});
```

The preceding example:

- Configures Kestrel's endpoints in ConfigureKestrel.
- Calls ListenUnixSocket to listen to a UDS with the specified path.
- Creates a UDS endpoint that isn't configured to use HTTPS. For information about enabling HTTPS, see Kestrel HTTPS endpoint configuration.

Client configuration

`GrpcChannel` supports making gRPC calls over custom transports. When a channel is created, it can be configured with a SocketsHttpHandler that has a custom ConnectCallback. The callback allows the client to make connections over custom transports and then send HTTP requests over that transport.

> **Note**
>
> Some connectivity features of `GrpcChannel`, such as client side load balancing and channel status, can't be used together with Unix domain sockets.

Unix domain sockets connection factory example:

C#

```
public class UnixDomainSocketsConnectionFactory
{
    private readonly EndPoint endPoint;

    public UnixDomainSocketsConnectionFactory(EndPoint endPoint)
    {
        this.endPoint = endPoint;
    }

    public async ValueTask<Stream> ConnectAsync(SocketsHttpConnectionContext
_,
        CancellationToken cancellationToken = default)
    {
        var socket = new Socket(AddressFamily.Unix, SocketType.Stream,
ProtocolType.Unspecified);

        try
        {
            await socket.ConnectAsync(this.endPoint,
cancellationToken).ConfigureAwait(false);
            return new NetworkStream(socket, true);
        }
        catch
        {
            socket.Dispose();
            throw;
        }
    }
}
```

Using the custom connection factory to create a channel:

C#

```
public static readonly string SocketPath = Path.Combine(Path.GetTempPath(),
"socket.tmp");

public static GrpcChannel CreateChannel()
{
    var udsEndPoint = new UnixDomainSocketEndPoint(SocketPath);
    var connectionFactory = new
UnixDomainSocketsConnectionFactory(udsEndPoint);
    var socketsHttpHandler = new SocketsHttpHandler
    {
        ConnectCallback = connectionFactory.ConnectAsync
    };

    return GrpcChannel.ForAddress("http://localhost", new GrpcChannelOptions
    {
        HttpHandler = socketsHttpHandler
    });
}
```

Channels created using the preceding code send gRPC calls over Unix domain sockets.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Inter-process communication with gRPC and Named pipes

Article • 01/10/2024

By James Newton-King

.NET supports inter-process communication (IPC) using gRPC. For more information about getting started with using gRPC to communicate between processes, see Inter-process communication with gRPC.

Named pipes is an IPC transport that is supported on all versions of Windows. Named pipes integrate well with Windows security to control client access to the pipe. This article discusses how to configure gRPC communication over named pipes.

Prerequisites

- .NET 8 or later
- Windows

Server configuration

Named pipes are supported by Kestrel, which is configured in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenNamedPipe("MyPipeName", listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http2;
    });
});
```

The preceding example:

- Configures Kestrel's endpoints in ConfigureKestrel.
- Calls `ListenNamedPipe` to listen to a named pipe with the specified name.
- Creates a named pipe endpoint that isn't configured to use HTTPS. For information about enabling HTTPS, see Kestrel HTTPS endpoint configuration.

Client configuration

`GrpcChannel` supports making gRPC calls over custom transports. When a channel is created, it can be configured with a SocketsHttpHandler that has a custom ConnectCallback. The callback allows the client to make connections over custom transports and then send HTTP requests over that transport.

> **Note**
>
> Some connectivity features of `GrpcChannel`, such as client side load balancing and channel status, can't be used together with named pipes.

Named pipes connection factory example:

C#

```
public class NamedPipesConnectionFactory
{
    private readonly string pipeName;

    public NamedPipesConnectionFactory(string pipeName)
    {
        this.pipeName = pipeName;
    }

    public async ValueTask<Stream> ConnectAsync(SocketsHttpConnectionContext
_,
        CancellationToken cancellationToken = default)
    {
        var clientStream = new NamedPipeClientStream(
            serverName: ".",
            pipeName: this.pipeName,
            direction: PipeDirection.InOut,
            options: PipeOptions.WriteThrough | PipeOptions.Asynchronous,
            impersonationLevel: TokenImpersonationLevel.Anonymous);

        try
        {
            await
clientStream.ConnectAsync(cancellationToken).ConfigureAwait(false);
            return clientStream;
        }
        catch
        {
            clientStream.Dispose();
            throw;
        }
```

```
    }
}
```

Using the custom connection factory to create a channel:

C#

```
public static GrpcChannel CreateChannel()
{
    var connectionFactory = new NamedPipesConnectionFactory("MyPipeName");
    var socketsHttpHandler = new SocketsHttpHandler
    {
        ConnectCallback = connectionFactory.ConnectAsync
    };

    return GrpcChannel.ForAddress("http://localhost", new GrpcChannelOptions
    {
        HttpHandler = socketsHttpHandler
    });
}
```

Channels created using the preceding code send gRPC calls over named pipes.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Code-first gRPC services and clients with .NET

Article • 11/05/2024

By James Newton-King and Marc Gravell

Code-first gRPC uses .NET types to define service and message contracts.

Code-first is a good choice when an entire system uses .NET:

- .NET service and data contract types can be shared between the .NET server and clients.
- Avoids the need to define contracts in `.proto` files and code generation.

Code-first isn't recommended in polyglot systems with multiple languages. .NET service and data contract types can't be used with non-.NET platforms. To call a gRPC service written using code-first, other platforms must create a `.proto` contract that matches the service.

protobuf-net.Grpc

> **Important**
>
> For help with protobuf-net.Grpc, visit the **protobuf-net.Grpc website** or create an issue on the **protobuf-net.Grpc GitHub repository**.

protobuf-net.Grpc is a community project and isn't supported by Microsoft. It adds code-first support to `Grpc.AspNetCore` and `Grpc.Net.Client`. It uses .NET types annotated with attributes to define an app's gRPC services and messages.

The first step to creating a code-first gRPC service is defining the code contract:

- Create a new project that will be shared by the server and client.
- Add a protobuf-net.Grpc package reference.
- Create service and data contract types.

C#

```
using ProtoBuf.Grpc;
using System.Runtime.Serialization;
using System.ServiceModel;
```

```
using System.Threading.Tasks;

namespace Shared.Contracts;

[DataContract]
public class HelloReply
{
    [DataMember(Order = 1)]
    public string Message { get; set; }
}

[DataContract]
public class HelloRequest
{
    [DataMember(Order = 1)]
    public string Name { get; set; }
}

[ServiceContract]
public interface IGreeterService
{
    [OperationContract]
    Task<HelloReply> SayHelloAsync(HelloRequest request,
        CallContext context = default);
}
```

The preceding code:

- Defines `HelloRequest` and `HelloReply` messages.
- Defines the `IGreeterService` contract interface with the unary `SayHelloAsync` gRPC method.

The service contract is implemented on the server and called from the client.

Methods defined on service interfaces must match certain signatures depending on whether they're:

- Unary
- Server streaming
- Client streaming
- Bidirectional streaming

For more information on defining service contracts, see the protobuf-net.Grpc getting started documentation.

Create a code-first gRPC service

To add gRPC code-first service to an ASP.NET Core app:

- Add a protobuf-net.Grpc.AspNetCore package reference.
- Add a reference to the shared code-contract project.

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="protobuf-net.Grpc.AspNetCore"
Version="1.0.152" />
  </ItemGroup>

  <ItemGroup>
      <ProjectReference Include="..\Shared\Shared.Contracts.csproj" />
  </ItemGroup>

</Project>
```

- Create a new `GreeterService.cs` file and implement the `IGreeterService` service interface:

C#

```
using Shared.Contracts;
using ProtoBuf.Grpc;

public class GreeterService : IGreeterService
{
    public Task<HelloReply> SayHelloAsync(HelloRequest request,
CallContext context = default)
    {
        return Task.FromResult(
                new HelloReply
                {
                    Message = $"Hello {request.Name}"
                });
    }
}
```

- Update the `Program.cs` file:

C#

```
using ProtoBuf.Grpc.Server;

var builder = WebApplication.CreateBuilder(args);

// Additional configuration is required to successfully run gRPC on
macOS.
// For instructions on how to configure Kestrel and gRPC clients on
macOS, visit https://go.microsoft.com/fwlink/?linkid=2099682

// Add services to the container.
builder.Services.AddCodeFirstGrpc();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.MapGrpcService<GreeterService>();
app.MapGet("/", () => "Communication with gRPC endpoints must be made
through a gRPC client. To learn how to create a client, visit:
https://go.microsoft.com/fwlink/?linkid=2086909");

app.Run();
```

The preceding highlighted code updates the following:

- `AddCodeFirstGrpc` registers services that enable code-first.
- `MapGrpcService<GreeterService>` adds the code-first service endpoint.

gRPC services implemented with code-first and `.proto` files can co-exist in the same app. All gRPC services use gRPC service configuration.

Create a code-first gRPC client

A code-first gRPC client uses the service contract to call gRPC services.

- In the gRPC client `.csproj` file:
  - Add a protobuf-net.Grpc package reference.
  - Add a Grpc.Net.Client package reference.
  - Add a reference to the shared code-contract project.

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
```

```
  <ItemGroup>
    <PackageReference Include="Grpc.Net.Client" Version="2.52.0" />
    <PackageReference Include="protobuf-net.Grpc" Version="1.0.152" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\Shared\Shared.Contracts.csproj" />
  </ItemGroup>

</Project>
```

- Update the client `program.cs`

C#

```
// See https://aka.ms/new-console-template for more information
using Grpc.Net.Client;
using ProtoBuf.Grpc.Client;
using Shared.Contracts;

namespace GrpcGreeterClient;

internal class Program
{
    private static async Task Main(string[] args)
    {
        using var channel =
GrpcChannel.ForAddress("https://localhost:7184");
        var client = channel.CreateGrpcService<IGreeterService>();

        var reply = await client.SayHelloAsync(
            new HelloRequest { Name = "GreeterClient" });

        Console.WriteLine($"Greeting: {reply.Message}");
        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    }
}
```

The preceding gRPC client `Program.cs` code:

- Creates a gRPC channel.
- Creates a code-first client from the channel with the `CreateGrpcService<IGreeterService>` extension method.
- Calls the gRPC service with `SayHelloAsync`.

A code-first gRPC client is created from a channel. Just like a regular client, a code-first client uses its channel configuration.

View or download sample code (how to download)

Additional resources

- protobuf-net.Grpc website
- protobuf-net.Grpc GitHub repository

gRPC health checks in ASP.NET Core

Article • 07/31/2024

By James Newton-King

The gRPC health checking protocol is a standard for reporting the health of gRPC server apps.

Health checks are exposed by an app as a gRPC service. They're typically used with an external monitoring service to check the status of an app. The service can be configured for various real-time monitoring scenarios:

- Health probes can be used by container orchestrators and load balancers to check an app's status. For example, Kubernetes supports gRPC liveness, readiness and startup probes. Kubernetes can be configured to reroute traffic or restart unhealthy containers based on gRPC health check results.
- Use of memory, disk, and other physical server resources can be monitored for healthy status.
- Health checks can test an app's dependencies, such as databases and external service endpoints, to confirm availability and normal functioning.

Set up gRPC health checks

gRPC ASP.NET Core has built-in support for gRPC health checks with the Grpc.AspNetCore.HealthChecks package. Results from .NET health checks are reported to callers.

To set up gRPC health checks in an app:

- Add a `Grpc.AspNetCore.HealthChecks` package reference.
- Register gRPC health checks service:
  - `AddGrpcHealthChecks` to register services that enable health checks.
  - `MapGrpcHealthChecksService` to add a health checks service endpoint.
- Add health checks by implementing IHealthCheck or using the AddCheck method.

C#

```
using GrpcServiceHC.Services;
using Microsoft.Extensions.Diagnostics.HealthChecks;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();
```

```
builder.Services.AddGrpcHealthChecks()
                .AddCheck("Sample", () => HealthCheckResult.Healthy());

var app = builder.Build();

app.MapGrpcService<GreeterService>();
app.MapGrpcHealthChecksService();

// Code removed for brevity.
```

When health checks is set up:

- The health checks service is added to the server app.
- .NET health checks registered with the app are periodically executed for health results. By default, there's a 5-second delay after app startup, and then health checks are executed every 30 seconds. Health check execution interval can be customized with HealthCheckPublisherOptions.
- Health results determine what the gRPC service reports:
  - `Unknown` is reported when there are no health results.
  - `NotServing` is reported when there are any health results of HealthStatus.Unhealthy.
  - Otherwise, `Serving` is reported.

Health checks service security

gRPC health checks returns health status about an app, which could be sensitive information. Care should be taken to limit access to the gRPC health checks service.

Access to the service can be controlled through standard ASP.NET Core authorization extension methods, such as AllowAnonymous and RequireAuthorization.

For example, if an app has been configured to require authorization by default, configure the gRPC health checks endpoint with `AllowAnonymous` to skip authentication and authorization.

C#

```
app.MapGrpcHealthChecksService().AllowAnonymous();
```

Configure `Grpc.AspNetCore.HealthChecks`

By default, the gRPC health checks service uses all registered health checks to determine health status. gRPC health checks can be customized when registered to use a subset of

health checks. The `MapService` method is used to map health results to service names, along with a predicate for filtering health results:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();
builder.Services.AddGrpcHealthChecks(o =>
{
    o.Services.MapService("", r => r.Tags.Contains("public"));
});

var app = builder.Build();
```

The preceding code overrides the default service (`""`) to only use health results with the "public" tag.

gRPC health checks supports the client specifying a service name argument when checking health. Multiple services are supported by providing a service name to `MapService`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddGrpc();
builder.Services.AddGrpcHealthChecks(o =>
{
    o.Services.MapService("greet.Greeter", r => r.Tags.Contains("greeter"));
    o.Services.MapService("count.Counter", r => r.Tags.Contains("counter"));
});

var app = builder.Build();
```

The service name specified by the client is usually the default (`""`) or a package-qualified name of a service in your app. However, nothing prevents the client using arbitrary values to check app health.

Configure health checks execution interval

Health checks are run immediately when `Check` is called. `Watch` is a streaming method and has a different behavior than `Check`: The long running stream reports health checks results over time by periodically executing IHealthCheckPublisher to gather health results. By default, the publisher:

- Waits 5 seconds after app startup before running health checks.
- Runs health checks every 30 seconds.

Publisher intervals can be configured using HealthCheckPublisherOptions at startup:

C#

```
builder.Services.Configure<HealthCheckPublisherOptions>(options =>
{
    options.Delay = TimeSpan.Zero;
    options.Period = TimeSpan.FromSeconds(10);
});
```

Call gRPC health checks service

The Grpc.HealthCheck package includes a client for gRPC health checks:

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Health.HealthClient(channel);

var response = await client.CheckAsync(new HealthCheckRequest());
var status = response.Status;
```

There are two methods on the `Health` service:

- `Check` is a unary method for getting the current health status. Health checks are executed immediately when `Check` is called. The server returns a `NOT_FOUND` error response if the client requests an unknown service name. This can happen at app startup if health results haven't been published yet.
- `Watch` is a streaming method that reports changes in health status over time. IHealthCheckPublisher is periodically executed to gather health results. The server returns an `Unknown` status if the client requests an unknown service name.

The `Grpc.HealthCheck` client can be used in a client factory approach:

C#

```
builder.Services
    .AddGrpcClient<Health.HealthClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    });
```

In the previous example, a client factory for `Health.HealthClient` instances is registered with the dependency injection system. Then, these instances are injected into services for executing health check calls.

For more information, see gRPC client factory integration in .NET.

Additional resources

- Health checks in ASP.NET Core
- gRPC health checking protocol
- Grpc.AspNetCore.HealthChecks
- Grpc.HealthCheck

Manage Protobuf references with dotnet-grpc

Article • 07/31/2024

`dotnet-grpc` is a .NET Core Global Tool for managing Protobuf (.proto) references within a .NET gRPC project. The tool can be used to add, refresh, remove, and list Protobuf references.

Installation

To install the `dotnet-grpc` .NET Core Global Tool, run the following command:

.NET CLI

```
dotnet tool install -g dotnet-grpc
```

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

Add references

`dotnet-grpc` can be used to add Protobuf references as `<Protobuf />` items to the `.csproj` file:

XML

```
<Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
```

The Protobuf references are used to generate the C# client and/or server assets. The `dotnet-grpc` tool can:

- Create a Protobuf reference from local files on disk.
- Create a Protobuf reference from a remote file specified by a URL.
- Ensure the correct gRPC package dependencies are added to the project.

For example, the `Grpc.AspNetCore` package is added to a web app. `Grpc.AspNetCore` contains gRPC server and client libraries and tooling support. Alternatively, the `Grpc.Net.Client`, `Grpc.Tools` and `Google.Protobuf` packages, which contain only the gRPC client libraries and tooling support, are added to a Console app.

Add file

The `add-file` command is used to add local files on disk as Protobuf references. The file paths provided:

- Can be relative to the current directory or absolute paths.
- May contain wild cards for pattern-based file globbing.

If any files are outside the project directory, a `Link` element is added to display the file under the folder `Protos` in Visual Studio.

Usage

.NET CLI

```
dotnet-grpc add-file [options] <files>...
```

Arguments

Expand table

Argument	Description
files	The protobuf file references. These can be a path to glob for local protobuf files.

Options

Expand table

Short option	Long option	Description
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.
-s	--services	The type of gRPC services that should be generated. If `Default` is specified, `Both` is used for Web projects and `Client` is used for non-

Short option	Long option	Description
		Web projects. Accepted values are `Both`, `Client`, `Default`, `None`, `Server`.
-i	--additional-import-dirs	Additional directories to be used when resolving imports for the protobuf files. This is a semicolon separated list of paths.
	--access	The access modifier to use for the generated C# classes. The default value is `Public`. Accepted values are `Internal` and `Public`.

Add URL

The `add-url` command is used to add a remote file specified by an source URL as Protobuf reference. A file path must be provided to specify where to download the remote file. The file path can be relative to the current directory or an absolute path. If the file path is outside the project directory, a `Link` element is added to display the file under the virtual folder `Protos` in Visual Studio.

Usage

.NET CLI

```
dotnet-grpc add-url [options] <url>
```

Arguments

Expand table

Argument	Description
url	The URL to a remote protobuf file.

Options

Expand table

Short option	Long option	Description
-o	--output	Specifies the download path for the remote protobuf file. This is a

Short option	Long option	Description
		required option.
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.
-s	--services	The type of gRPC services that should be generated. If `Default` is specified, `Both` is used for Web projects and `Client` is used for non-Web projects. Accepted values are `Both`, `Client`, `Default`, `None`, `Server`.
-i	--additional-import-dirs	Additional directories to be used when resolving imports for the protobuf files. This is a semicolon separated list of paths.
	--access	The access modifier to use for the generated C# classes. Default value is `Public`. Accepted values are `Internal` and `Public`.

Remove

The `remove` command is used to remove Protobuf references from the `.csproj` file. The command accepts path arguments and source URLs as arguments. The tool:

- Only removes the Protobuf reference.
- Does not delete the `.proto` file, even if it was originally downloaded from a remote URL.

Usage

.NET CLI

```
dotnet-grpc remove [options] <references>...
```

Arguments

Expand table

Argument	Description
references	The URLs or file paths of the protobuf references to remove.

Options

Expand table

Short option	Long option	Description
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.

Refresh

The `refresh` command is used to update a remote reference with the latest content from the source URL. Both the download file path and the source URL can be used to specify the reference to be updated. Note:

- The hashes of the file contents are compared to determine whether the local file should be updated.
- No timestamp information is compared.

The tool always replaces the local file with the remote file if an update is needed.

Usage

.NET CLI

```
dotnet-grpc refresh [options] [<references>...]
```

Arguments

Expand table

Argument	Description
references	The URLs or file paths to remote protobuf references that should be updated. Leave this argument empty to refresh all remote references.

Options

Expand table

Short option	Long option	Description
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.
	--dry-run	Outputs a list of files that would be updated without downloading any new content.

List

The `list` command is used to display all the Protobuf references in the project file. If all values of a column are default values, the column may be omitted.

Usage

.NET CLI

```
dotnet-grpc list [options]
```

Options

Expand table

Short option	Long option	Description
-p	--project	The path to the project file to operate on. If a file is not specified, the command searches the current directory for one.

Additional resources

- Overview for gRPC on .NET
- gRPC services with C#
- gRPC services with ASP.NET Core

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

Test gRPC services with gRPCurl and gRPCui in ASP.NET Core

Article • 07/31/2024

By James Newton-King

Tooling is available for gRPC that allows developers to test services without building client apps:

- gRPCurl is an open-source command-line tool that provides interaction with gRPC services.
- gRPCui builds on top of gRPCurl and adds an open-source interactive web UI for gRPC.

This article discusses how to:

- Set up gRPC server reflection with a gRPC ASP.NET Core app.
- Interact with gRPC using test tools:
  - Discover and test gRPC services with `grpcurl`.
  - Interact with gRPC services via a browser using `grpcui`.

> **Note**
>
> To learn how to unit test gRPC services, see Test gRPC services in ASP.NET Core.

Set up gRPC reflection

Tooling must know the Protobuf contract of services before it can call them. There are two ways to do this:

- Set up gRPC reflection on the server. Tools, such as gRPCurl, use reflection to automatically discover service contracts.
- Add `.proto` files to the tool manually.

It's easier to use gRPC reflection. gRPC reflection adds a new gRPC service to the app that clients can call to discover services.

gRPC ASP.NET Core has built-in support for gRPC reflection with the Grpc.AspNetCore.Server.Reflection package. To configure reflection in an app:

- Add a `Grpc.AspNetCore.Server.Reflection` package reference.

- Register reflection in `Program.cs`:
  - `AddGrpcReflection` to register services that enable reflection.
  - `MapGrpcReflectionService` to add a reflection service endpoint.

```
C#

builder.Services.AddGrpc();
builder.Services.AddGrpcReflection();

var app = builder.Build();

app.MapGrpcService<GreeterService>();

IWebHostEnvironment env = app.Environment;

if (env.IsDevelopment())
{
    app.MapGrpcReflectionService();
}
```

When gRPC reflection is set up:

- A gRPC reflection service is added to the server app.
- Client apps that support gRPC reflection can call the reflection service to discover services hosted by the server.
- gRPC services are still called from the client. Reflection only enables service discovery and doesn't bypass server-side security. Endpoints protected by authentication and authorization require the caller to pass credentials for the endpoint to be called successfully.

Reflection service security

gRPC reflection returns a list of available APIs, which could contain sensitive information. Care should be taken to limit access to the gRPC reflection service.

gRPC reflection is usually only required in a local development environment. For local development, the reflection service should only be mapped when IsDevelopment returns true:

```
C#

if (env.IsDevelopment())
{
    app.MapGrpcReflectionService();
}
```

Access to the service can be controlled through standard ASP.NET Core authorization extension methods, such as AllowAnonymous and RequireAuthorization.

For example, if an app has been configured to require authorization by default, configuration the gRPC reflection endpoint with `AllowAnonymous` to skip authentication and authorization.

C#

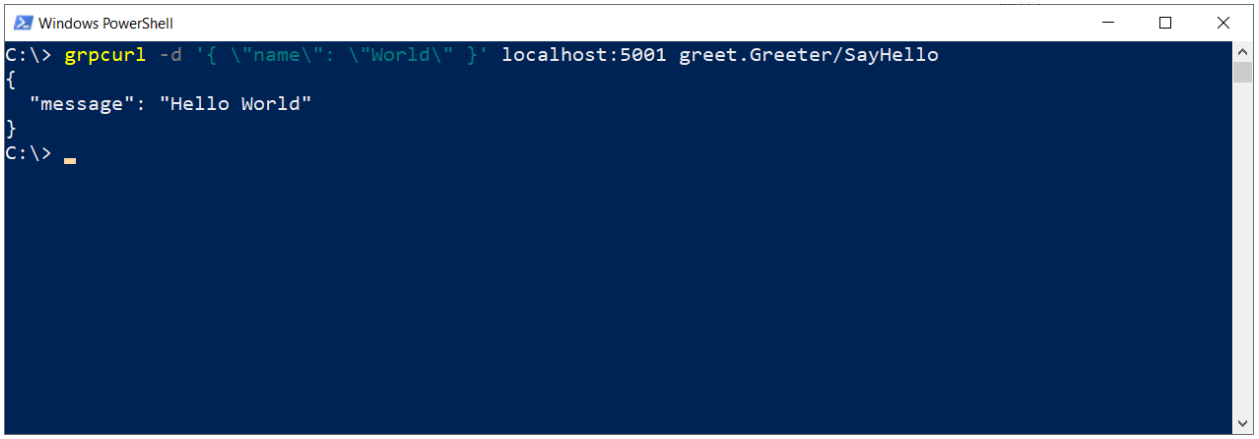
```
if (env.IsDevelopment())
{
    app.MapGrpcReflectionService().AllowAnonymous();
}
```

gRPCurl

gRPCurl is a command-line tool created by the gRPC community. Its features include:

- Calling gRPC services, including streaming services.
- Service discovery using gRPC reflection.
- Listing and describing gRPC services.
- Works with secure (TLS) and insecure (plain-text) servers.

For information about downloading and installing `grpcurl`, see the gRPCurl GitHub homepage.



Use `grpcurl`

The `-help` argument explains `grpcurl` command-line options:

Console

```
$ grpcurl -help
```

Discover services

Use the `describe` verb to view the services defined by the server. Specify `<port>` as the localhost port number of the gRPC server. The port number is randomly assigned when the project is created and set in `Properties/launchSettings.json`:

Console

```
$ grpcurl localhost:<port> describe
greet.Greeter is a service:
service Greeter {
  rpc SayHello ( .greet.HelloRequest ) returns ( .greet.HelloReply );
  rpc SayHellos ( .greet.HelloRequest ) returns ( stream .greet.HelloReply
);
}
grpc.reflection.v1alpha.ServerReflection is a service:
service ServerReflection {
  rpc ServerReflectionInfo ( stream
.grpc.reflection.v1alpha.ServerReflectionRequest ) returns ( stream
.grpc.reflection.v1alpha.ServerReflectionResponse );
}
```

The preceding example:

- Runs the `describe` verb on server `localhost:<port>`. Where `<port>` is randomly assigned when the gRPC server project is created and set in `Properties/launchSettings.json`
- Prints services and methods returned by gRPC reflection.
  - `Greeter` is a service implemented by the app.
  - `ServerReflection` is the service added by the `Grpc.AspNetCore.Server.Reflection` package.

Combine `describe` with a service, method, or message name to view its detail:

PowerShell

```
$ grpcurl localhost:<port> describe greet.HelloRequest
greet.HelloRequest is a message:
message HelloRequest {
  string name = 1;
}
```

Call gRPC services

Call a gRPC service by specifying a service and method name along with a JSON argument that represents the request message. The JSON is converted into Protobuf and sent to the service.

Console

```
$ grpcurl -d '{ \"name\": \"World\" }' localhost:<port>
greet.Greeter/SayHello
{
  "message": "Hello World"
}
```

In the preceding example:

- The `-d` argument specifies a request message with JSON. This argument must come before the server address and method name.
- Calls the `SayHello` method on the `greeter.Greeter` service.
- Prints the response message as JSON.
- Where `<port>` is randomly assigned when the gRPC server project is created and set in `Properties/launchSettings.json`

The preceding example uses `\` to escape the `"` character. Escaping `"` is required in a PowerShell console but must not be used in some consoles. For example, the previous command for a macOS console:

Console

```
$ grpcurl -d '{ "name": "World" }' localhost:<port> greet.Greeter/SayHello
{
  "message": "Hello World"
}
```

gRPCui

gRPCui is an interactive web UI for gRPC. gRPCui builds on top of gRPCurl. gRPCui offers a GUI for discovering and testing gRPC services, similar to HTTP tools such as Swagger UI.

For information about downloading and installing `grpcui`, see the gRPCui GitHub homepage.

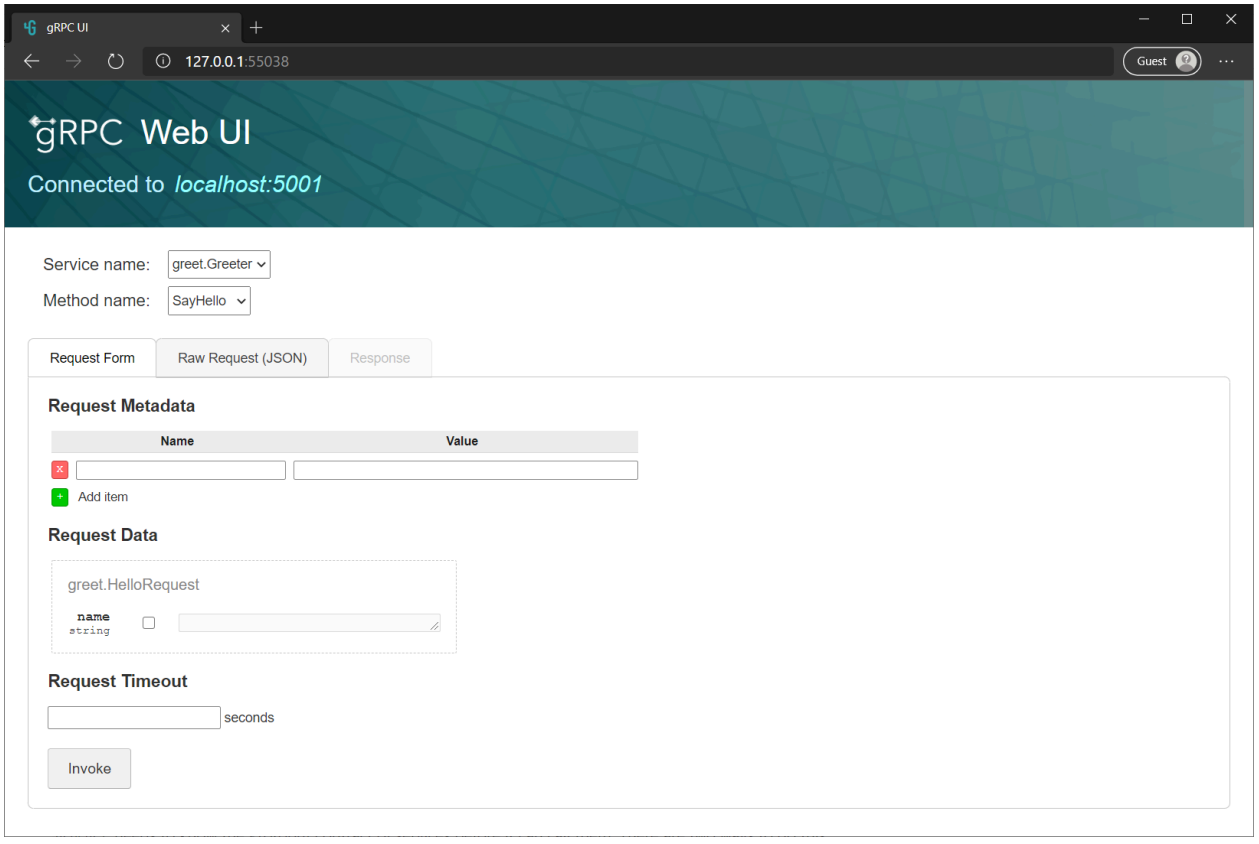
Using `grpcui`

Run `grpcui` with the server address to interact with as an argument:

```
PowerShell

$ grpcui localhost:<port>
gRPC Web UI available at http://127.0.0.1:55038/
```

In the preceding example, specify `<port>` as the localhost port number of the gRPC server. The port number is randomly assigned when the project is created and set in `Properties/launchSettings.json`

The tool launches a browser window with the interactive web UI. gRPC services are automatically discovered using gRPC reflection.



Additional resources

- gRPCurl GitHub homepage
- gRPCui GitHub homepage
- Grpc.AspNetCore.Server.Reflection
- Test gRPC services in ASP.NET Core
- Mock gRPC client in tests

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Migrate gRPC from C-core to gRPC for .NET

Article • 07/31/2024

Due to the implementation of the underlying stack, not all features work in the same way between C-core-based gRPC apps and gRPC for .NET. This document highlights the key differences for migrating between the two stacks.

> **Important**
>
> gRPC C-core is in maintenance mode and **will be deprecated in favor of gRPC for .NET**. gRPC C-core is not recommended for new apps.

Platform support

gRPC C-core and gRPC for .NET have different platform support:

- **gRPC C-core**: A C++ gRPC implementation with its own TLS and HTTP/2 stacks. The `Grpc.Core` package is a .NET wrapper around gRPC C-core and contains a gRPC client and server. It supports .NET Framework, .NET Core, and .NET 5 or later.
- **gRPC for .NET**: Designed for .NET Core 3.x and .NET 5 or later. It uses TLS and HTTP/2 stacks built into modern .NET releases. The `Grpc.AspNetCore` package contains a gRPC server that is hosted in ASP.NET Core and requires .NET Core 3.x or .NET 5 or later. The `Grpc.Net.Client` package contains a gRPC client. The client in `Grpc.Net.Client` has limited support for .NET Framework using WinHttpHandler.

For more information, see gRPC on .NET supported platforms.

Configure server and channel

NuGet packages, configuration, and startup code must be modified when migrating from gRPC C-Core to gRPC for .NET.

gRPC for .NET has separate NuGet packages for its client and server. The packages added depend upon whether an app is hosting gRPC services or calling them:

- `Grpc.AspNetCore`: Services are hosted by ASP.NET Core. For server configuration information, see gRPC services with ASP.NET Core.

- `Grpc.Net.Client`: Clients use `GrpcChannel`, which internally uses networking functionality built into .NET. For client configuration information, see Call gRPC services with the .NET client.

When migration is complete, the `Grpc.Core` package should be removed from the app. `Grpc.Core` contains large native binaries, and removing the package reduces NuGet restore time and app size.

Code generated services and clients

gRPC C-Core and gRPC for .NET share many APIs, and code generated from `.proto` files is compatible with both gRPC implementations. Most clients and service can be migrated from C-Core to gRPC for .NET without changes.

gRPC service implementation lifetime

In the ASP.NET Core stack, gRPC services, by default, are created with a scoped lifetime. In contrast, gRPC C-core by default binds to a service with a singleton lifetime.

A scoped lifetime allows the service implementation to resolve other services with scoped lifetimes. For example, a scoped lifetime can also resolve `DbContext` from the DI container through constructor injection. Using scoped lifetime:

- A new instance of the service implementation is constructed for each request.
- It isn't possible to share state between requests via instance members on the implementation type.
- The expectation is to store shared states in a singleton service in the DI container. The stored shared states are resolved in the constructor of the gRPC service implementation.

For more information on service lifetimes, see Dependency injection in ASP.NET Core.

Add a singleton service

To facilitate the transition from a gRPC C-core implementation to ASP.NET Core, it's possible to change the service lifetime of the service implementation from scoped to singleton. This involves adding an instance of the service implementation to the DI container:

```
C#
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();
    services.AddSingleton(new GreeterService());
}
```

However, a service implementation with a singleton lifetime is no longer able to resolve scoped services through constructor injection.

Configure gRPC services options

In C-core-based apps, settings such as `grpc.max_receive_message_length` and `grpc.max_send_message_length` are configured with `ChannelOption` when constructing the Server instance.

In ASP.NET Core, gRPC provides configuration through the `GrpcServiceOptions` type. For example, a gRPC service's the maximum incoming message size can be configured via `AddGrpc`. The following example changes the default `MaxReceiveMessageSize` of 4 MB to 16 MB:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc(options =>
    {
        options.MaxReceiveMessageSize = 16 * 1024 * 1024; // 16 MB
    });
}
```

For more information on configuration, see gRPC for .NET configuration.

Logging

C-core-based apps rely on the `GrpcEnvironment` to configure the logger for debugging purposes. The ASP.NET Core stack provides this functionality through the Logging API. For example, a logger can be added to the gRPC service via constructor injection:

C#

```
public class GreeterService : Greeter.GreeterBase
{
```

```
    public GreeterService(ILogger<GreeterService> logger)
    {
    }
}
```

For more information on gRPC logging and diagnostics, see Logging and diagnostics in gRPC on .NET.

HTTPS

C-core-based apps configure HTTPS through the Server.Ports property. A similar concept is used to configure servers in ASP.NET Core. For example, Kestrel uses endpoint configuration for this functionality.

gRPC Interceptors

ASP.NET Core middleware offers similar functionalities compared to interceptors in C-core-based gRPC apps. Both are supported by ASP.NET Core gRPC apps, so there's no need to rewrite interceptors.

For more information on how these features compare to each other, see gRPC Interceptors versus Middleware.

Host gRPC in non-ASP.NET Core projects

A C-core-based server can be added to any project type. gRPC for .NET server requires ASP.NET Core. ASP.NET Core is usually available because the project file specifies `Microsoft.NET.SDK.Web` as the SDK.

A gRPC server can be hosted to non-ASP.NET Core projects by adding `<FrameworkReference Include="Microsoft.AspNetCore.App" />` to a project. The framework reference makes ASP.NET Core APIs available and they can be used to start an ASP.NET Core server.

For more information, see Host gRPC in non-ASP.NET Core projects.

Additional resources

- Overview for gRPC on .NET
- gRPC services with C#
- gRPC services with ASP.NET Core

- Create a .NET Core gRPC client and server in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

gRPC for Windows Communication Foundation (WCF) developers

Article • 07/31/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This article provides a summary of why ASP.NET Core gRPC is a good fit for Windows Communication Foundation (WCF) developers who want to migrate to modern architectures and platforms.

Comparison to WCF

Although the implementation and approach are different for gRPC, the experience of developing and consuming services with gRPC should be intuitive for WCF developers. WCF and gRPC are RPC (remote procedure call) frameworks with the same goals:

- Make it possible to code as though the client and server are on the same platform.
- Provide a simplified portable networking API.

Both platforms share the requirement of declaring and implementing an interface, although the process for declaring the interface is different. The many types of RPC calls that gRPC supports map well to the bindings available to WCF services. For more information and examples, see Migrate a WCF solution to gRPC.

Benefits of gRPC

gRPC provides a better framework than other approaches for the following reasons.

Performance

gRPC uses HTTP/2. In contrast to HTTP/1.1, HTTP/2:

- Is a smaller, faster binary protocol.
- Is more efficient for computers to parse.
- Supports multiplexing requests over a single connection. Multiplexing enables multiple requests to be sent over one connection without requests blocking each

other. In HTTP/1.1, the blocking is known as "head-of-line (HOL) blocking."

gRPC uses Protobuf, an efficient binary format, to serialize messages. Protobuf messages are:

- Fast to serialize and deserialize.
- Use less bandwidth than text-based formats.

gRPC is a good solution for mobile devices and networks with bandwidth restrictions.

Interoperability

There are gRPC tools and libraries for all major programming languages and platforms, including .NET, Java, Python, Go, C++, Node.js, Swift, Dart, Ruby, and PHP. Thanks to the Protobuf binary wire format and the efficient code generation for each platform, developers can build cross-platform, performant apps.

Usability and productivity

gRPC is a comprehensive RPC solution. It works consistently across multiple languages and platforms. It also provides excellent tooling, with much of the boilerplate code automatically generated. Like WCF, gRPC automatically generates messages and a strongly typed client. Developer time is freed up to focus on business logic.

Streaming

gRPC has full bidirectional streaming, which provides similar functionality to WCF's full duplex services. gRPC streaming can operate over regular internet connections, load balancers, and service meshes.

Deadlines, timeouts, and cancellation

gRPC allows clients to specify a maximum time for an RPC to finish. If the specified deadline is exceeded, the server can cancel the operation independently of the client. Deadlines and cancellations can be propagated through subsequent gRPC calls to help enforce resource usage limits. Clients can stop operations when a deadline is exceeded, or earlier if necessary. For example, clients can stop operations because of a user interaction.

Security

gRPC can use TLS and HTTP/2 to provide an end-to-end encrypted connection between the client and the server. Support for client certificate authentication further increases security and trust between client and server.

gRPC as a migration path for WCF to .NET Core and .NET 5

.NET Core and .NET 5 marks a shift in the way that Microsoft delivers remote communication solutions to developers who want to deliver services across a range of platforms. .NET Core and .NET 5 support calling WCF services, but won't offer server-side support for hosting WCF.

There are two recommended paths for modernizing WCF apps:

- gRPC is built on modern technologies and has emerged as the most popular choice across the developer community for RPC apps. Starting with .NET Core 3.0, modern .NET platforms have excellent support for gRPC. Migrating WCF services to use gRPC helps provide the RPC features, performance, an interoperability needed in modern apps.

- CoreWCF is a community effort to bring support for hosting WCF services to .NET Core and .NET 5. A preview release is available and the project is working towards being production ready. CoreWCF only supports a subset of WCF's features, and .NET Framework apps that migrate to use it will need code changes and testing to be successful. CoreWCF is a good choice if an app has to maintain compatibility with existing clients that call WCF services.

Get started

For detailed guidance on building gRPC services in ASP.NET Core for WCF developers, see ASP.NET Core gRPC for WCF Developers

Compare gRPC services with HTTP APIs

Article • 07/31/2024

By James Newton-King

This article explains how gRPC services compare to HTTP APIs with JSON (including ASP.NET Core web APIs). The technology used to provide an API for your app is an important choice, and gRPC offers unique benefits compared to HTTP APIs. This article discusses the strengths and weaknesses of gRPC and recommends scenarios for using gRPC over other technologies.

High-level comparison

The following table offers a high-level comparison of features between gRPC and HTTP APIs with JSON.

Expand table

Feature	gRPC	HTTP APIs with JSON
Contract	Required (`.proto`)	Optional (OpenAPI)
Protocol	HTTP/2	HTTP
Payload	Protobuf (small, binary)	JSON (large, human readable)
Prescriptiveness	Strict specification	Loose. Any HTTP is valid.
Streaming	Client, server, bi-directional	Client, server
Browser support	No (requires grpc-web)	Yes
Security	Transport (TLS)	Transport (TLS)
Client code-generation	Yes	OpenAPI + third-party tooling

gRPC strengths

Performance

gRPC messages are serialized using Protobuf, an efficient binary message format. Protobuf serializes very quickly on the server and client. Protobuf serialization results in small message payloads, important in limited bandwidth scenarios like mobile apps.

gRPC is designed for HTTP/2, a major revision of HTTP that provides significant performance benefits over HTTP 1.x:

- Binary framing and compression. HTTP/2 protocol is compact and efficient both in sending and receiving.
- Multiplexing of multiple HTTP/2 calls over a single TCP connection. Multiplexing eliminates head-of-line blocking.

HTTP/2 is not exclusive to gRPC. Many request types, including HTTP APIs with JSON, can use HTTP/2 and benefit from its performance improvements.

Code generation

All gRPC frameworks provide first-class support for code generation. A core file to gRPC development is the .proto file, which defines the contract of gRPC services and messages. From this file, gRPC frameworks generate a service base class, messages, and a complete client.

By sharing the `.proto` file between the server and client, messages and client code can be generated from end to end. Code generation of the client eliminates duplication of messages on the client and server, and creates a strongly-typed client for you. Not having to write a client saves significant development time in applications with many services.

Strict specification

A formal specification for HTTP API with JSON doesn't exist. Developers debate the best format of URLs, HTTP verbs, and response codes.

The gRPC specification is prescriptive about the format a gRPC service must follow. gRPC eliminates debate and saves developer time because gRPC is consistent across platforms and implementations.

Streaming

HTTP/2 provides a foundation for long-lived, real-time communication streams. gRPC provides first-class support for streaming through HTTP/2.

A gRPC service supports all streaming combinations:

- Unary (no streaming)
- Server to client streaming

- Client to server streaming
- Bi-directional streaming

Deadline/timeouts and cancellation

gRPC allows clients to specify how long they are willing to wait for an RPC to complete. The deadline is sent to the server, and the server can decide what action to take if it exceeds the deadline. For example, the server might cancel in-progress gRPC/HTTP/database requests on timeout.

Propagating the deadline and cancellation through child gRPC calls helps enforce resource usage limits.

gRPC recommended scenarios

gRPC is well suited to the following scenarios:

- **Microservices**: gRPC is designed for low latency and high throughput communication. gRPC is great for lightweight microservices where efficiency is critical.
- **Point-to-point real-time communication**: gRPC has excellent support for bi-directional streaming. gRPC services can push messages in real-time without polling.
- **Polyglot environments**: gRPC tooling supports all popular development languages, making gRPC a good choice for multi-language environments.
- **Network constrained environments**: gRPC messages are serialized with Protobuf, a lightweight message format. A gRPC message is always smaller than an equivalent JSON message.
- **Inter-process communication (IPC)**: IPC transports such as Unix domain sockets and named pipes can be used with gRPC to communicate between apps on the same machine. For more information, see Inter-process communication with gRPC.

gRPC weaknesses

Limited browser support

It's impossible to directly call a gRPC service from a browser today. gRPC heavily uses HTTP/2 features and no browser provides the level of control required over web requests to support a gRPC client. For example, browsers do not allow a caller to require that HTTP/2 be used, or provide access to underlying HTTP/2 frames.

gRPC on ASP.NET Core offers two browser-compatible solutions:

- **gRPC-Web** allows browser apps to call gRPC services with the gRPC-Web client and Protobuf. gRPC-Web requires the browser app to generate a gRPC client. gRPC-Web allows browser apps to benefit from the high-performance and low network usage of gRPC.

  .NET has built-in support for gRPC-Web. For more information, see gRPC-Web in ASP.NET Core gRPC apps.

- **gRPC JSON transcoding** allows browser apps to call gRPC services as if they were RESTful APIs with JSON. The browser app doesn't need to generate a gRPC client or know anything about gRPC. RESTful APIs can be automatically created from gRPC services by annotating the `.proto` file with HTTP metadata. Transcoding allows an app to support both gRPC and JSON web APIs without duplicating the effort of building separate services for both.

  .NET has built-in support for creating JSON web APIs from gRPC services. For more information, see gRPC JSON transcoding in ASP.NET Core gRPC apps.

> **Note**
>
> gRPC JSON transcoding requires .NET 7 or later.

Not human readable

HTTP API requests are sent as text and can be read and created by humans.

gRPC messages are encoded with Protobuf by default. While Protobuf is efficient to send and receive, its binary format isn't human readable. Protobuf requires the message's interface description specified in the `.proto` file to properly deserialize. Additional tooling is required to analyze Protobuf payloads on the wire and to compose requests by hand.

Features such as server reflection and the gRPC command line tool exist to assist with binary Protobuf messages. Also, Protobuf messages support conversion to and from JSON. The built-in JSON conversion provides an efficient way to convert Protobuf messages to and from human readable form when debugging.

Alternative framework scenarios

Other frameworks are recommended over gRPC in the following scenarios:

- **Browser accessible APIs**: gRPC isn't fully supported in the browser. gRPC-Web can offer browser support, but it has limitations and introduces a server proxy.
- **Broadcast real-time communication**: gRPC supports real-time communication via streaming, but the concept of broadcasting a message out to registered connections doesn't exist. For example in a chat room scenario where new chat messages should be sent to all clients in the chat room, each gRPC call is required to individually stream new chat messages to the client. SignalR is a useful framework for this scenario. SignalR has the concept of persistent connections and built-in support for broadcasting messages.

Additional resources

- Create a .NET Core gRPC client and server in ASP.NET Core
- Overview for gRPC on .NET
- gRPC services with C#
- Migrate gRPC from C-core to gRPC for .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Troubleshoot gRPC on .NET

Article • 09/27/2024

By James Newton-King

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This document discusses commonly encountered problems when developing gRPC apps on .NET.

Mismatch between client and service SSL/TLS configuration

The gRPC template and samples use Transport Layer Security (TLS) to secure gRPC services by default. gRPC clients need to use a secure connection to call secured gRPC services successfully.

You can verify the ASP.NET Core gRPC service is using TLS in the logs written on app start. The service will be listening on an HTTPS endpoint:

Output

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

The .NET Core client must use `https` in the server address to make calls with a secured connection:

C#

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);
```

All gRPC client implementations support TLS. gRPC clients from other languages typically require the channel configured with `SslCredentials`. `SslCredentials` specifies the certificate that the client will use, and it must be used instead of insecure credentials. For examples of configuring the different gRPC client implementations to use TLS, see gRPC Authentication.

Call a gRPC service with an untrusted/invalid certificate

The .NET gRPC client requires the service to have a trusted certificate. The following error message is returned when calling a gRPC service without a trusted certificate:

> Unhandled exception. System.Net.Http.HttpRequestException: The SSL connection could not be established, see inner exception. ---> System.Security.Authentication.AuthenticationException: The remote certificate is invalid according to the validation procedure.

You may see this error if you are testing your app locally and the ASP.NET Core HTTPS development certificate is not trusted. For instructions to fix this issue, see Trust the ASP.NET Core HTTPS development certificate on Windows and macOS.

If you are calling a gRPC service on another machine and are unable to trust the certificate then the gRPC client can be configured to ignore the invalid certificate. The following code uses HttpClientHandler.ServerCertificateCustomValidationCallback to allow calls without a trusted certificate:

C#

```
var handler = new HttpClientHandler();
handler.ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

var channel = GrpcChannel.ForAddress("https://localhost:5001",
    new GrpcChannelOptions { HttpHandler = handler });
var client = new Greeter.GreeterClient(channel);
```

The gRPC client factory allows calls without a trusted certificate. Use the ConfigurePrimaryHttpMessageHandler extension method to configure the handler on the client:

C#

```
var services = new ServiceCollection();

services
    .AddGrpcClient<Greeter.GreeterClient>(o =>
    {
        o.Address = new Uri("https://localhost:5001");
    })
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        var handler = new HttpClientHandler();
        handler.ServerCertificateCustomValidationCallback =
            HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

        return handler;
    });
```

> ⚠ **Warning**
>
> Untrusted certificates should only be used during app development. Production apps should always use valid certificates.

Call insecure gRPC services with .NET Core client

The .NET gRPC client can call insecure gRPC services by specifing `http` in the server address. For example, `GrpcChannel.ForAddress("http://localhost:5000")`.

There are some additional requirements to call insecure gRPC services depending on the .NET version an app is using:

- .NET 5 or later requires Grpc.Net.Client version 2.32.0 or later.
- .NET Core 3.x requires additional configuration. The app must set the `System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport` switch to `true`. For more information see Asp.Net Core 3.x: Call insecure gRPC services with the .NET client:

> ⓘ **Important**
>
> Insecure gRPC services must be hosted on a HTTP/2-only port. For more information, see **ASP.NET Core protocol negotiation**.

Unable to start ASP.NET Core gRPC app on macOS

Kestrel doesn't support HTTP/2 with TLS on macOS before .NET 8. The ASP.NET Core gRPC template and samples use TLS by default. You'll see the following error message when you attempt to start the gRPC server:

> Unable to bind to https://localhost:5001 on the IPv4 loopback interface: 'HTTP/2 over TLS is not supported on macOS due to missing ALPN support.'.

To work around this issue in .NET 7 and earlier, configure Kestrel and the gRPC client to use HTTP/2 *without* TLS. You should only do this during development. Not using TLS will result in gRPC messages being sent without encryption. For more information see Asp.Net Core 7.0: Unable to start ASP.NET Core gRPC app on macOS.

gRPC C# assets are not code generated from `.proto` files

gRPC code generation of concrete clients and service base classes requires protobuf files and tooling to be referenced from a project. You must include:

- `.proto` files you want to use in the `<Protobuf>` item group. Imported .proto files must be referenced by the project.
- Package reference to the gRPC tooling package Grpc.Tools.

For more information on generating gRPC C# assets, see gRPC services with C#.

An ASP.NET Core web app hosting gRPC services only needs the service base class generated:

XML

```xml
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />
</ItemGroup>
```

A gRPC client app making gRPC calls only needs the concrete client generated:

XML

```xml
<ItemGroup>
  <Protobuf Include="Protos\greet.proto" GrpcServices="Client" />
```

```
</ItemGroup>
```

WPF projects unable to generate gRPC C# assets from `.proto` files

WPF projects have a known issue that prevents gRPC code generation from working correctly. Any gRPC types generated in a WPF project by referencing `Grpc.Tools` and `.proto` files will create compilation errors when used:

> error CS0246: The type or namespace name 'MyGrpcServices' could not be found (are you missing a using directive or an assembly reference?)

You can workaround this issue by:

1. Create a new .NET Core class library project.
2. In the new project, add references to enable C# code generation from .proto files:
   - Add the following package references:
     - Grpc.Tools
     - Grpc.Net.Client
     - Google.Protobuf
   - Add `.proto` files to the `<Protobuf>` item group.
3. In the WPF application, add a reference to the new project.

The WPF application can use the gRPC generated types from the new class library project.

Calling gRPC services hosted in a sub-directory

> **⚠ Warning**
>
> Many third-party gRPC tools don't support services hosted in subdirectories. Consider finding a way to host gRPC as the root directory.

The path component of a gRPC channel's address is ignored when making gRPC calls. For example, `GrpcChannel.ForAddress("https://localhost:5001/ignored_path")` won't use `ignored_path` when routing gRPC calls for the service.

The address path is ignored because gRPC has a standardized, prescriptive address structure. A gRPC address combines the package, service and method names: `https://localhost:5001/PackageName.ServiceName/MethodName`.

There are some scenarios when an app needs to include a path with gRPC calls. For example, when an ASP.NET Core gRPC app is hosted in an IIS directory and the directory needs to be included in the request. When a path is required, it can be added to the gRPC call using the custom `SubdirectoryHandler` specified below:

C#

```
public class SubdirectoryHandler : DelegatingHandler
{
    private readonly string _subdirectory;

    public SubdirectoryHandler(HttpMessageHandler innerHandler, string
subdirectory)
        : base(innerHandler)
    {
        _subdirectory = subdirectory;
    }

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        var old = request.RequestUri;

        var url = $"{old.Scheme}://{old.Host}:{old.Port}";
        url += $"{_subdirectory}{request.RequestUri.AbsolutePath}";
        request.RequestUri = new Uri(url, UriKind.Absolute);

        return base.SendAsync(request, cancellationToken);
    }
}
```

`SubdirectoryHandler` is used when the gRPC channel is created.

C#

```
var handler = new SubdirectoryHandler(new HttpClientHandler(), "/MyApp");

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions { HttpHandler = handler });
var client = new Greeter.GreeterClient(channel);

var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
```

The preceding code:

- Creates a `SubdirectoryHandler` with the path `/MyApp`.
- Configures a channel to use `SubdirectoryHandler`.
- Calls the gRPC service with `SayHelloAsync`. The gRPC call is sent to `https://localhost:5001/MyApp/greet.Greeter/SayHello`.

Alternatively, a client factory can be configured with `SubdirectoryHandler` by using AddHttpMessageHandler.

Configure gRPC client to use HTTP/3

The .NET gRPC client supports HTTP/3 with .NET 6 or later. If the server sends an `alt-svc` response header to the client that indicates the server supports HTTP/3, the client will automatically upgrade its connection to HTTP/3. For more information, see Use HTTP/3 with the ASP.NET Core Kestrel web server.

A DelegatingHandler can be used to force a gRPC client to use HTTP/3. Forcing HTTP/3 avoids the overhead of upgrading the request. Force HTTP/3 with code similar to the following:

C#

```
public class Http3Handler : DelegatingHandler
{
    public Http3Handler() { }
    public Http3Handler(HttpMessageHandler innerHandler) :
base(innerHandler) { }

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.Version = HttpVersion.Version30;
        request.VersionPolicy = HttpVersionPolicy.RequestVersionExact;

        return base.SendAsync(request, cancellationToken);
    }
}
```

`Http3Handler` is used when the gRPC channel is created. The following code creates a channel configured to use `Http3Handler`.

C#

```
var handler = new Http3Handler(new HttpClientHandler());

var channel = GrpcChannel.ForAddress("https://localhost:5001", new
GrpcChannelOptions { HttpHandler = handler });
```

```
var client = new Greeter.GreeterClient(channel);

var reply = await client.SayHelloAsync(
                  new HelloRequest { Name = "GreeterClient" });
```

Alternatively, a client factory can be configured with `Http3Handler` by using AddHttpMessageHandler.

Building gRPC on Alpine Linux

The `Grpc.Tools` package generates .NET types from .proto files using a bundled native binary called `protoc`. Additional steps are required to build gRPC apps on platforms that aren't supported by the native binaries in `Grpc.Tools`, such as Alpine Linux.

Generate code ahead of time

One solution is to generate code ahead of time.

1. Move `.proto` files and the `Grpc.Tools` package reference to a new project.
2. Publish the project as a NuGet package and upload it to a NuGet feed.
3. Update the app to reference the NuGet package.

With the preceding steps, the app no longer requires `Grpc.Tools` to build because code is generated ahead of time.

Customize `Grpc.Tools` native binaries

`Grpc.Tools` supports using custom native binaries. This feature allows gRPC tooling to run in environments its bundled native binaries don't support.

Build or acquire `protoc` and `grpc_csharp_plugin` native binaries and configure `Grpc.Tools` to use them. Configure native binaries by setting the following environment variables:

- `PROTOBUF_PROTOC` - Full path to the protocol buffers compiler
- `GRPC_PROTOC_PLUGIN` - Full path to the grpc_csharp_plugin

For Alpine Linux, there are community-provided packages for the protocol buffers compiler and gRPC plugins at https://pkgs.alpinelinux.org/.

```
sh
```

```
# Build or install the binaries for your architecture.

# For Alpine Linux, the grpc-plugins package can be used.
# See https://pkgs.alpinelinux.org/package/edge/community/x86_64/grpc-
plugins
apk add grpc-plugins  # Alpine Linux specific package installer

# Set environment variables for the built/installed protoc
# and grpc_csharp_plugin binaries
export PROTOBUF_PROTOC=/usr/bin/protoc
export GRPC_PROTOC_PLUGIN=/usr/bin/grpc_csharp_plugin

# When dotnet build runs, the Grpc.Tools NuGet package
# uses the binaries pointed to by the environment variables.
dotnet build
```

For more information about using `Grpc.Tools` with unsupported architectures, see the gRPC build integration documentation.

gRPC call timeout from `HttpClient.Timeout`

HttpClient is configured with a 100 second timeout by default. If a `GrpcChannel` is configured to use an `HttpClient`, long-running gRPC streaming calls are canceled if they don't complete within the timeout limit.

Output

```
System.OperationCanceledException: The request was canceled due to the
configured HttpClient.Timeout of 100 seconds elapsing.
```

There are a couple of ways to fix this error. The first is to configure HttpClient.Timeout to a larger value. Timeout.InfiniteTimeSpan disables the timeout:

C#

```
var handler = new HttpClientHandler();
handler.ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

var httpClient = new HttpClient(handler) { Timeout =
Timeout.InfiniteTimeSpan };
var channel = GrpcChannel.ForAddress("https://localhost:5001",
    new GrpcChannelOptions { HttpClient = httpClient });
var client = new Greeter.GreeterClient(channel);
```

Alternatively, avoid creating `HttpClient` and set `GrpcChannel.HttpHandler` instead:

C#

```
var handler = new HttpClientHandler();
handler.ServerCertificateCustomValidationCallback =
    HttpClientHandler.DangerousAcceptAnyServerCertificateValidator;

var channel = GrpcChannel.ForAddress("https://localhost:5001",
    new GrpcChannelOptions { HttpHandler = handler });
var client = new Greeter.GreeterClient(channel);
```

ASP.NET Core Best Practices

Article • 09/14/2023

By Mike Rousos

This article provides guidelines for maximizing performance and reliability of ASP.NET Core apps.

Cache aggressively

Caching is discussed in several parts of this article. For more information, see Overview of caching in ASP.NET Core.

Understand hot code paths

In this article, a *hot code path* is defined as a code path that is frequently called and where much of the execution time occurs. Hot code paths typically limit app scale-out and performance and are discussed in several parts of this article.

Avoid blocking calls

ASP.NET Core apps should be designed to process many requests simultaneously. Asynchronous APIs allow a small pool of threads to handle thousands of concurrent requests by not waiting on blocking calls. Rather than waiting on a long-running synchronous task to complete, the thread can work on another request.

A common performance problem in ASP.NET Core apps is blocking calls that could be asynchronous. Many synchronous blocking calls lead to Thread Pool starvation and degraded response times.

Do not block asynchronous execution by calling Task.Wait or Task<TResult>.Result. **Do not** acquire locks in common code paths. ASP.NET Core apps perform best when architected to run code in parallel. **Do not** call Task.Run and immediately await it. ASP.NET Core already runs app code on normal Thread Pool threads, so calling `Task.Run` only results in extra unnecessary Thread Pool scheduling. Even if the scheduled code would block a thread, `Task.Run` does not prevent that.

- **Do** make hot code paths asynchronous.
- **Do** call data access, I/O, and long-running operations APIs asynchronously if an asynchronous API is available.

- **Do not** use Task.Run to make a synchronous API asynchronous.
- **Do** make controller/Razor Page actions asynchronous. The entire call stack is asynchronous in order to benefit from async/await patterns.
- **Consider** using message brokers like Azure Service Bus to offload long-running calls

A profiler, such as PerfView, can be used to find threads frequently added to the Thread Pool. The `Microsoft-Windows-DotNETRuntime/ThreadPoolWorkerThread/Start` event indicates a thread added to the thread pool.

Return large collections across multiple smaller pages

A webpage shouldn't load large amounts of data all at once. When returning a collection of objects, consider whether it could lead to performance issues. Determine if the design could produce the following poor outcomes:

- OutOfMemoryException or high memory consumption
- Thread pool starvation (see the following remarks on IAsyncEnumerable<T>)
- Slow response times
- Frequent garbage collection

Do add pagination to mitigate the preceding scenarios. Using page size and page index parameters, developers should favor the design of returning a partial result. When an exhaustive result is required, pagination should be used to asynchronously populate batches of results to avoid locking server resources.

For more information on paging and limiting the number of returned records, see:

- Performance considerations
- Add paging to an ASP.NET Core app

Return `IEnumerable<T>` or `IAsyncEnumerable<T>`

Returning `IEnumerable<T>` from an action results in synchronous collection iteration by the serializer. The result is the blocking of calls and a potential for thread pool starvation. To avoid synchronous enumeration, use `ToListAsync` before returning the enumerable.

Beginning with ASP.NET Core 3.0, `IAsyncEnumerable<T>` can be used as an alternative to `IEnumerable<T>` that enumerates asynchronously. For more information, see Controller

action return types.

Minimize large object allocations

The .NET Core garbage collector manages allocation and release of memory automatically in ASP.NET Core apps. Automatic garbage collection generally means that developers don't need to worry about how or when memory is freed. However, cleaning up unreferenced objects takes CPU time, so developers should minimize allocating objects in hot code paths. Garbage collection is especially expensive on large objects (>= 85,000 bytes). Large objects are stored on the large object heap and require a full (generation 2) garbage collection to clean up. Unlike generation 0 and generation 1 collections, a generation 2 collection requires a temporary suspension of app execution. Frequent allocation and de-allocation of large objects can cause inconsistent performance.

Recommendations:

- **Do** consider caching large objects that are frequently used. Caching large objects prevents expensive allocations.
- **Do** pool buffers by using an ArrayPool<T> to store large arrays.
- **Do not** allocate many, short-lived large objects on hot code paths.

Memory issues, such as the preceding, can be diagnosed by reviewing garbage collection (GC) stats in PerfView and examining:

- Garbage collection pause time.
- What percentage of the processor time is spent in garbage collection.
- How many garbage collections are generation 0, 1, and 2.

For more information, see Garbage Collection and Performance.

Optimize data access and I/O

Interactions with a data store and other remote services are often the slowest parts of an ASP.NET Core app. Reading and writing data efficiently is critical for good performance.

Recommendations:

- **Do** call all data access APIs asynchronously.
- **Do not** retrieve more data than is necessary. Write queries to return just the data that's necessary for the current HTTP request.

- **Do** consider caching frequently accessed data retrieved from a database or remote service if slightly out-of-date data is acceptable. Depending on the scenario, use a MemoryCache or a DistributedCache. For more information, see Response caching in ASP.NET Core.
- **Do** minimize network round trips. The goal is to retrieve the required data in a single call rather than several calls.
- **Do** use no-tracking queries in Entity Framework Core when accessing data for read-only purposes. EF Core can return the results of no-tracking queries more efficiently.
- **Do** filter and aggregate LINQ queries (with `.Where`, `.Select`, or `.Sum` statements, for example) so that the filtering is performed by the database.
- **Do** consider that EF Core resolves some query operators on the client, which may lead to inefficient query execution. For more information, see Client evaluation performance issues.
- **Do not** use projection queries on collections, which can result in executing "N + 1" SQL queries. For more information, see Optimization of correlated subqueries.

The following approaches may improve performance in high-scale apps:

- DbContext pooling
- Explicitly compiled queries

We recommend measuring the impact of the preceding high-performance approaches before committing the code base. The additional complexity of compiled queries may not justify the performance improvement.

Query issues can be detected by reviewing the time spent accessing data with Application Insights or with profiling tools. Most databases also make statistics available concerning frequently executed queries.

Pool HTTP connections with HttpClientFactory

Although HttpClient implements the `IDisposable` interface, it's designed for reuse. Closed `HttpClient` instances leave sockets open in the `TIME_WAIT` state for a short period of time. If a code path that creates and disposes of `HttpClient` objects is frequently used, the app may exhaust available sockets. `HttpClientFactory` was introduced in ASP.NET Core 2.1 as a solution to this problem. It handles pooling HTTP connections to optimize performance and reliability. For more information, see Use HttpClientFactory to implement resilient HTTP requests.

Recommendations:

- **Do not** create and dispose of `HttpClient` instances directly.
- **Do** use HttpClientFactory to retrieve `HttpClient` instances. For more information, see Use HttpClientFactory to implement resilient HTTP requests.

Keep common code paths fast

You want all of your code to be fast. Frequently-called code paths are the most critical to optimize. These include:

- Middleware components in the app's request processing pipeline, especially middleware run early in the pipeline. These components have a large impact on performance.
- Code that's executed for every request or multiple times per request. For example, custom logging, authorization handlers, or initialization of transient services.

Recommendations:

- **Do not** use custom middleware components with long-running tasks.
- **Do** use performance profiling tools, such as Visual Studio Diagnostic Tools or PerfView), to identify hot code paths.

Complete long-running Tasks outside of HTTP requests

Most requests to an ASP.NET Core app can be handled by a controller or page model calling necessary services and returning an HTTP response. For some requests that involve long-running tasks, it's better to make the entire request-response process asynchronous.

Recommendations:

- **Do not** wait for long-running tasks to complete as part of ordinary HTTP request processing.
- **Do** consider handling long-running requests with background services or out of process possibly with an Azure Function and/or using a message broker like Azure Service Bus. Completing work out-of-process is especially beneficial for CPU-intensive tasks.
- **Do** use real-time communication options, such as SignalR, to communicate with clients asynchronously.

Minify client assets

ASP.NET Core apps with complex front-ends frequently serve many JavaScript, CSS, or image files. Performance of initial load requests can be improved by:

- Bundling, which combines multiple files into one.
- Minifying, which reduces the size of files by removing whitespace and comments.

Recommendations:

- **Do** use the bundling and minification guidelines, which mention compatible tools and show how to use ASP.NET Core's `environment` tag to handle both `Development` and `Production` environments.
- **Do** consider other third-party tools, such as Webpack, for complex client asset management.

Compress responses

Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses. For more information, see Response compression.

Use the latest ASP.NET Core release

Each new release of ASP.NET Core includes performance improvements. Optimizations in .NET Core and ASP.NET Core mean that newer versions generally outperform older versions. For example, .NET Core 2.1 added support for compiled regular expressions and benefitted from Span<T>. ASP.NET Core 2.2 added support for HTTP/2. ASP.NET Core 3.0 adds many improvements that reduce memory usage and improve throughput. If performance is a priority, consider upgrading to the current version of ASP.NET Core.

Minimize exceptions

Exceptions should be rare. Throwing and catching exceptions is slow relative to other code flow patterns. Because of this, exceptions shouldn't be used to control normal program flow.

Recommendations:

- **Do not** use throwing or catching exceptions as a means of normal program flow, especially in hot code paths.

- **Do** include logic in the app to detect and handle conditions that would cause an exception.
- **Do** throw or catch exceptions for unusual or unexpected conditions.

App diagnostic tools, such as Application Insights, can help to identify common exceptions in an app that may affect performance.

Avoid synchronous read or write on HttpRequest/HttpResponse body

All I/O in ASP.NET Core is asynchronous. Servers implement the `Stream` interface, which has both synchronous and asynchronous overloads. The asynchronous ones should be preferred to avoid blocking thread pool threads. Blocking threads can lead to thread pool starvation.

Do not do this: The following example uses the ReadToEnd. It blocks the current thread to wait for the result. This is an example of sync over async.

C#

```
public class BadStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public ActionResult<ContosoData> Get()
    {
        var json = new StreamReader(Request.Body).ReadToEnd();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }
}
```

In the preceding code, `Get` synchronously reads the entire HTTP request body into memory. If the client is slowly uploading, the app is doing sync over async. The app does sync over async because Kestrel does **NOT** support synchronous reads.

Do this: The following example uses ReadToEndAsync and does not block the thread while reading.

C#

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
```

```
        var json = await new StreamReader(Request.Body).ReadToEndAsync();

        return JsonSerializer.Deserialize<ContosoData>(json);
    }

}
```

The preceding code asynchronously reads the entire HTTP request body into memory.

> ⚠ **Warning**
>
> If the request is large, reading the entire HTTP request body into memory could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see Avoid reading large request bodies or response bodies into memory in this article.

Do this: The following example is fully asynchronous using a non-buffered request body:

C#

```
public class GoodStreamReaderController : Controller
{
    [HttpGet("/contoso")]
    public async Task<ActionResult<ContosoData>> Get()
    {
        return await JsonSerializer.DeserializeAsync<ContosoData>
(Request.Body);
    }
}
```

The preceding code asynchronously de-serializes the request body into a C# object.

Prefer ReadFormAsync over Request.Form

Use `HttpContext.Request.ReadFormAsync` instead of `HttpContext.Request.Form`. `HttpContext.Request.Form` can be safely read only with the following conditions:

- The form has been read by a call to `ReadFormAsync`, and
- The cached form value is being read using `HttpContext.Request.Form`

Do not do this: The following example uses `HttpContext.Request.Form`. `HttpContext.Request.Form` uses sync over async and can lead to thread pool starvation.

C#

```
public class BadReadController : Controller
{
    [HttpPost("/form-body")]
    public IActionResult Post()
    {
        var form =  HttpContext.Request.Form;

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Do this: The following example uses `HttpContext.Request.ReadFormAsync` to read the form body asynchronously.

C#

```
public class GoodReadController : Controller
{
    [HttpPost("/form-body")]
    public async Task<IActionResult> Post()
    {
       var form = await HttpContext.Request.ReadFormAsync();

        Process(form["id"], form["name"]);

        return Accepted();
    }
```

Avoid reading large request bodies or response bodies into memory

In .NET, every object allocation greater than or equal to 85,000 bytes ends up in the large object heap (LOH). Large objects are expensive in two ways:

- The allocation cost is high because the memory for a newly allocated large object has to be cleared. The CLR guarantees that memory for all newly allocated objects is cleared.
- LOH is collected with the rest of the heap. LOH requires a full garbage collection or Gen2 collection.

This blog post describes the problem succinctly:

> When a large object is allocated, it's marked as Gen 2 object. Not Gen 0 as for small objects. The consequences are that if you run out of memory in LOH, GC cleans up the whole managed heap, not only LOH. So it cleans up Gen 0, Gen 1 and Gen 2 including LOH. This is called full garbage collection and is the most time-consuming garbage collection. For many applications, it can be acceptable. But definitely not for high-performance web servers, where few big memory buffers are needed to handle an average web request (read from a socket, decompress, decode JSON, and more).

Storing a large request or response body into a single `byte[]` or `string`:

- May result in quickly running out of space in the LOH.
- May cause performance issues for the app because of full GCs running.

Working with a synchronous data processing API

When using a serializer/de-serializer that only supports synchronous reads and writes (for example, Json.NET):

- Buffer the data into memory asynchronously before passing it into the serializer/de-serializer.

> ⚠ **Warning**
>
> If the request is large, it could lead to an out of memory (OOM) condition. OOM can result in a Denial Of Service. For more information, see **Avoid reading large request bodies or response bodies into memory** in this article.

ASP.NET Core 3.0 uses System.Text.Json by default for JSON serialization. System.Text.Json:

- Reads and writes JSON asynchronously.
- Is optimized for UTF-8 text.
- Typically is higher performance than `Newtonsoft.Json`.

Do not store IHttpContextAccessor.HttpContext in a field

The IHttpContextAccessor.HttpContext returns the `HttpContext` of the active request when accessed from the request thread. The `IHttpContextAccessor.HttpContext` should **not** be stored in a field or variable.

Do not do this: The following example stores the `HttpContext` in a field and then attempts to use it later.

C#

```
public class MyBadType
{
    private readonly HttpContext _context;
    public MyBadType(IHttpContextAccessor accessor)
    {
        _context = accessor.HttpContext;
    }

    public void CheckAdmin()
    {
        if (!_context.User.IsInRole("admin"))
        {
            throw new UnauthorizedAccessException("The current user isn't an
admin");
        }
    }
}
```

The preceding code frequently captures a null or incorrect `HttpContext` in the constructor.

Do this: The following example:

- Stores the IHttpContextAccessor in a field.
- Uses the `HttpContext` field at the correct time and checks for `null`.

C#

```
public class MyGoodType
{
    private readonly IHttpContextAccessor _accessor;
    public MyGoodType(IHttpContextAccessor accessor)
    {
        _accessor = accessor;
    }

    public void CheckAdmin()
    {
        var context = _accessor.HttpContext;
        if (context != null && !context.User.IsInRole("admin"))
        {
```

```
            throw new UnauthorizedAccessException("The current user isn't an
admin");
        }
    }
}
```

Do not access HttpContext from multiple threads

`HttpContext` is **not** thread-safe. Accessing `HttpContext` from multiple threads in parallel can result in unexpected behavior such as the server to stop responding, crashes, and data corruption.

Do not do this: The following example makes three parallel requests and logs the incoming request path before and after the outgoing HTTP request. The request path is accessed from multiple threads, potentially in parallel.

C#

```
public class AsyncBadSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        var query1 = SearchAsync(SearchEngine.Google, query);
        var query2 = SearchAsync(SearchEngine.Bing, query);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine,
string query)
    {
        var searchResults = _searchService.Empty();
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   HttpContext.Request.Path);
            searchResults = _searchService.Search(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
                                   HttpContext.Request.Path);
        }
```

```
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}",
                             HttpContext.Request.Path);
        }

        return await searchResults;
    }
```

Do this: The following example copies all data from the incoming request before making the three parallel requests.

C#

```
public class AsyncGoodSearchController : Controller
{
    [HttpGet("/search")]
    public async Task<SearchResults> Get(string query)
    {
        string path = HttpContext.Request.Path;
        var query1 = SearchAsync(SearchEngine.Google, query,
                                 path);
        var query2 = SearchAsync(SearchEngine.Bing, query, path);
        var query3 = SearchAsync(SearchEngine.DuckDuckGo, query, path);

        await Task.WhenAll(query1, query2, query3);

        var results1 = await query1;
        var results2 = await query2;
        var results3 = await query3;

        return SearchResults.Combine(results1, results2, results3);
    }

    private async Task<SearchResults> SearchAsync(SearchEngine engine,
string query,
                                                  string path)
    {
        var searchResults = _searchService.Empty();
        try
        {
            _logger.LogInformation("Starting search query from {path}.",
                                   path);
            searchResults = await _searchService.SearchAsync(engine, query);
            _logger.LogInformation("Finishing search query from {path}.",
path);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed query from {path}", path);
        }

        return await searchResults;
```

```
    }
```

Do not use the HttpContext after the request is complete

`HttpContext` is only valid as long as there is an active HTTP request in the ASP.NET Core pipeline. The entire ASP.NET Core pipeline is an asynchronous chain of delegates that executes every request. When the `Task` returned from this chain completes, the `HttpContext` is recycled.

Do not do this: The following example uses `async void` which makes the HTTP request complete when the first `await` is reached:

- Using `async void` is **ALWAYS** a bad practice in ASP.NET Core apps.
- The example code accesses the `HttpResponse` after the HTTP request is complete.
- The late access crashes the process.

C#

```
public class AsyncBadVoidController : Controller
{
    [HttpGet("/async")]
    public async void Get()
    {
        await Task.Delay(1000);

        // The following line will crash the process because of writing
after the
        // response has completed on a background thread. Notice async void
Get()

        await Response.WriteAsync("Hello World");
    }
}
```

Do this: The following example returns a `Task` to the framework, so the HTTP request doesn't complete until the action completes.

C#

```
public class AsyncGoodTaskController : Controller
{
    [HttpGet("/async")]
    public async Task Get()
```

```
    {
        await Task.Delay(1000);

        await Response.WriteAsync("Hello World");
    }
}
```

Do not capture the HttpContext in background threads

Do not do this: The following example shows a closure is capturing the `HttpContext` from the `Controller` property. This is a bad practice because the work item could:

- Run outside of the request scope.
- Attempt to read the wrong `HttpContext`.

C#

```
[HttpGet("/fire-and-forget-1")]
public IActionResult BadFireAndForget()
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        var path = HttpContext.Request.Path;
        Log(path);
    });

    return Accepted();
}
```

Do this: The following example:

- Copies the data required in the background task during the request.
- Doesn't reference anything from the controller.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult GoodFireAndForget()
{
    string path = HttpContext.Request.Path;
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);
```

```
        Log(path);
    });

    return Accepted();
}
```

Background tasks should be implemented as hosted services. For more information, see Background tasks with hosted services.

Do not capture services injected into the controllers on background threads

Do not do this: The following example shows a closure that is capturing the `DbContext` from the `Controller` action parameter. This is a bad practice. The work item could run outside of the request scope. The `ContosoDbContext` is scoped to the request, resulting in an `ObjectDisposedException`.

C#

```
[HttpGet("/fire-and-forget-1")]
public IActionResult FireAndForget1([FromServices]ContosoDbContext context)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        context.Contoso.Add(new Contoso());
        await context.SaveChangesAsync();
    });

    return Accepted();
}
```

Do this: The following example:

- Injects an IServiceScopeFactory in order to create a scope in the background work item. `IServiceScopeFactory` is a singleton.
- Creates a new dependency injection scope in the background thread.
- Doesn't reference anything from the controller.
- Doesn't capture the `ContosoDbContext` from the incoming request.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        await using (var scope = serviceScopeFactory.CreateAsyncScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });

    return Accepted();
}
```

The following highlighted code:

- Creates a scope for the lifetime of the background operation and resolves services from it.
- Uses `ContosoDbContext` from the correct scope.

C#

```
[HttpGet("/fire-and-forget-3")]
public IActionResult FireAndForget3([FromServices]IServiceScopeFactory
                                    serviceScopeFactory)
{
    _ = Task.Run(async () =>
    {
        await Task.Delay(1000);

        await using (var scope = serviceScopeFactory.CreateAsyncScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<ContosoDbContext>();

            context.Contoso.Add(new Contoso());

            await context.SaveChangesAsync();
        }
    });
```

```
    return Accepted();
}
```

Do not modify the status code or headers after the response body has started

ASP.NET Core does not buffer the HTTP response body. The first time the response is written:

- The headers are sent along with that chunk of the body to the client.
- It's no longer possible to change response headers.

Do not do this: The following code tries to add response headers after the response has already started:

C#

```
app.Use(async (context, next) =>
{
    await next();

    context.Response.Headers["test"] = "test value";
});
```

In the preceding code, `context.Response.Headers["test"] = "test value";` will throw an exception if `next()` has written to the response.

Do this: The following example checks if the HTTP response has started before modifying the headers.

C#

```
app.Use(async (context, next) =>
{
    await next();

    if (!context.Response.HasStarted)
    {
        context.Response.Headers["test"] = "test value";
    }
});
```

Do this: The following example uses `HttpResponse.OnStarting` to set the headers before the response headers are flushed to the client.

Checking if the response has not started allows registering a callback that will be invoked just before response headers are written. Checking if the response has not started:

- Provides the ability to append or override headers just in time.
- Doesn't require knowledge of the next middleware in the pipeline.

C#

```
app.Use(async (context, next) =>
{
    context.Response.OnStarting(() =>
    {
        context.Response.Headers["someheader"] = "somevalue";
        return Task.CompletedTask;
    });

    await next();
});
```

Do not call next() if you have already started writing to the response body

Components only expect to be called if it's possible for them to handle and manipulate the response.

Use In-process hosting with IIS

Using in-process hosting, an ASP.NET Core app runs in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter. The loopback adapter is a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the Windows Process Activation Service (WAS).

Projects default to the in-process hosting model in ASP.NET Core 3.0 and later.

For more information, see Host ASP.NET Core on Windows with IIS

Don't assume that HttpRequest.ContentLength is not null

`HttpRequest.ContentLength` is null if the `Content-Length` header is not received. Null in that case means the length of the request body is not known; it doesn't mean the length is zero. Because all comparisons with null (except `==`) return false, the comparison `Request.ContentLength > 1024`, for example, might return `false` when the request body size is more than 1024. Not knowing this can lead to security holes in apps. You might think you're protecting against too-large requests when you aren't.

For more information, see this StackOverflow answer.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Web server implementations in ASP.NET Core

Article • 07/26/2024

By Tom Dykstra, Steve Smith, Stephen Halter, and Chris Ross

An ASP.NET Core app runs with an in-process HTTP server implementation. The server implementation listens for HTTP requests and surfaces them to the app as a set of request features composed into an HttpContext.

Windows

ASP.NET Core ships with the following:

- Kestrel server is the default, cross-platform HTTP server implementation. Kestrel provides the best performance and memory utilization, but it doesn't have some of the advanced features in HTTP.sys. For more information, see Kestrel vs. HTTP.sys in the Windows tab.
- IIS HTTP Server is an in-process server for IIS.
- HTTP.sys server is a Windows-only HTTP server based on the HTTP.sys kernel driver and HTTP Server API.

When using IIS or IIS Express, the app either runs:

- In the same process as the IIS worker process (the in-process hosting model) with the IIS HTTP Server. *In-process* is the recommended configuration.
- In a process separate from the IIS worker process (the out-of-process hosting model) with the Kestrel server.

The ASP.NET Core Module is a native IIS module that handles native IIS requests between IIS and the in-process IIS HTTP Server or Kestrel. For more information, see ASP.NET Core Module (ANCM) for IIS.

Kestrel vs. HTTP.sys

Kestrel has the following advantages over HTTP.sys:

- Better performance and memory utilization.
- Cross platform
- Agility, it's developed and patched independent of the OS.
- Programmatic port and TLS configuration

- Extensibility that allows for protocols like PPv2 and alternate transports.

Http.Sys operates as a shared kernel mode component with the following features that kestrel does not have:

- Port sharing
- Kernel mode windows authentication. Kestrel supports only user-mode authentication.
- Fast proxying via queue transfers
- Direct file transmission
- Response caching

Hosting models

Several hosting models are available:

- Kestrel self-hosting: The Kestrel web server runs without requiring any other external web server such as IIS or HTTP.sys.

- HTTP.sys self-hosting is an alternative to Kestrel. Kestrel is recommended over HTTP.sys unless the app requires features not available in Kestrel.

- IIS in-process hosting: An ASP.NET Core app runs in the same process as its IIS worker process. IIS in-process hosting provides improved performance over IIS out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine. IIS handles process management with the Windows Process Activation Service (WAS).

- IIS out-of-process hosting: ASP.NET Core apps run in a process separate from the IIS worker process, and the module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS). Using a separate process also enables hosting more than one app from the same app pool.

For more information, see the following:

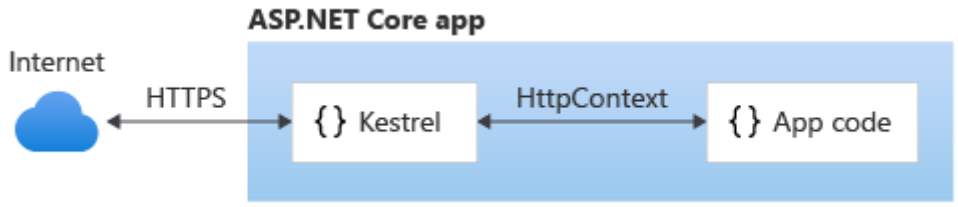
- Kestrel vs. HTTP.sys
- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module (ANCM) for IIS

Kestrel

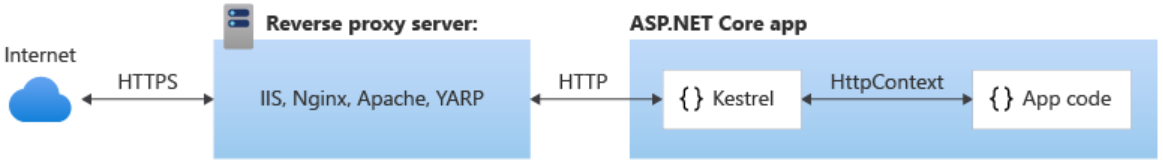
Kestrel server is the default, cross-platform HTTP server implementation. Kestrel provides the best performance and memory utilization, but it doesn't have some of the advanced features in HTTP.sys. For more information, see Kestrel vs. HTTP.sys in this document.

Use Kestrel:

- By itself as an edge server processing requests directly from a network, including the Internet.



- With a *reverse proxy server*, such as Internet Information Services (IIS), Nginx, or Apache. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.



Either hosting configuration—with or without a reverse proxy server—is supported.

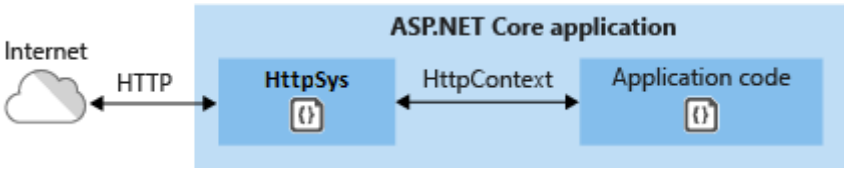
For Kestrel configuration guidance and information on when to use Kestrel in a reverse proxy configuration, see Kestrel web server in ASP.NET Core.

Nginx with Kestrel

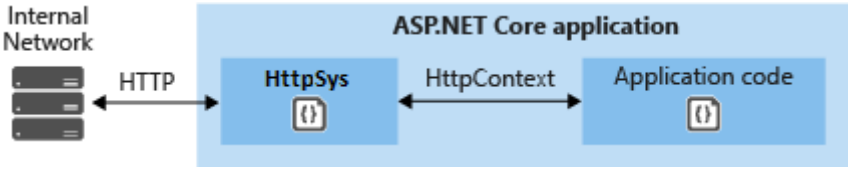
For information on how to use Nginx on Linux as a reverse proxy server for Kestrel, see Host ASP.NET Core on Linux with Nginx.

HTTP.sys

If ASP.NET Core apps are run on Windows, HTTP.sys is an alternative to Kestrel. Kestrel is recommended over HTTP.sys unless the app requires features not available in Kestrel. For more information, see HTTP.sys web server implementation in ASP.NET Core.



HTTP.sys can also be used for apps that are only exposed to an internal network.



For HTTP.sys configuration guidance, see HTTP.sys web server implementation in ASP.NET Core.

ASP.NET Core server infrastructure

The IApplicationBuilder available in the `Startup.Configure` method exposes the ServerFeatures property of type IFeatureCollection. Kestrel and HTTP.sys only expose a single feature each, IServerAddressesFeature, but different server implementations may expose additional functionality.

`IServerAddressesFeature` can be used to find out which port the server implementation has bound at runtime.

Custom servers

If the built-in servers don't meet the app's requirements, a custom server implementation can be created. The Open Web Interface for .NET (OWIN) guide demonstrates how to write a Nowin -based IServer implementation. Only the feature interfaces that the app uses require implementation, though at a minimum IHttpRequestFeature and IHttpResponseFeature must be supported.

Server startup

The server is launched when the Integrated Development Environment (IDE) or editor starts the app:

- Visual Studio : Launch profiles can be used to start the app and server with either IIS Express/ASP.NET Core Module or the console.
- Visual Studio Code : The app and server are started by Omnisharp , which activates the CoreCLR debugger.

- Visual Studio for Mac: The app and server are started by the Mono Soft-Mode Debugger.

When launching the app from a command prompt in the project's folder, dotnet run launches the app and server (Kestrel and HTTP.sys only). The configuration is specified by the `-c|--configuration` option, which is set to either `Debug` (default) or `Release`.

A `launchSettings.json` file provides configuration when launching an app with `dotnet run` or with a debugger built into tooling, such as Visual Studio. If launch profiles are present in a `launchSettings.json` file, use the `--launch-profile {PROFILE NAME}` option with the `dotnet run` command or select the profile in Visual Studio. For more information, see dotnet run and .NET Core distribution packaging.

HTTP/2 support

HTTP/2 is supported with ASP.NET Core in the following deployment scenarios:

- Kestrel
  - Operating system
    - Windows Server 2016/Windows 10 or later†
    - Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
    - macOS 10.15 or later
  - Target framework: .NET Core 2.2 or later
- HTTP.sys
  - Windows Server 2016/Windows 10 or later
  - Target framework: Not applicable to HTTP.sys deployments.
- IIS (in-process)
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Target framework: .NET Core 2.2 or later
- IIS (out-of-process)
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Public-facing edge server connections use HTTP/2, but the reverse proxy connection to Kestrel uses HTTP/1.1.
  - Target framework: Not applicable to IIS out-of-process deployments.

†Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

An HTTP/2 connection must use Application-Layer Protocol Negotiation (ALPN) and TLS 1.2 or later. For more information, see the topics that pertain to your server

deployment scenarios.

Additional resources

- Kestrel web server in ASP.NET Core
- ASP.NET Core Module (ANCM) for IIS
- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service
- Host ASP.NET Core on Linux with Nginx
- HTTP.sys web server implementation in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Kestrel web server in ASP.NET Core

Article • 04/14/2023

By Tom Dykstra, Chris Ross, and Stephen Halter

Kestrel is a cross-platform web server for ASP.NET Core. Kestrel is the recommended server for ASP.NET Core, and it's configured by default in ASP.NET Core project templates.

Kestrel's features include:

- **Cross-platform:** Kestrel is a cross-platform web server that runs on Windows, Linux, and macOS.
- **High performance:** Kestrel is optimized to handle a large number of concurrent connections efficiently.
- **Lightweight:** Optimized for running in resource-constrained environments, such as containers and edge devices.
- **Security hardened:** Kestrel supports HTTPS and is hardened against web server vulnerabilities.
- **Wide protocol support:** Kestrel supports common web protocols, including:
  - HTTP/1.1, HTTP/2 and HTTP/3
  - WebSockets
- **Integration with ASP.NET Core:** Seamless integration with other ASP.NET Core components, such as the middleware pipeline, dependency injection, and configuration system.
- **Flexible workloads**: Kestrel supports many workloads:
  - ASP.NET app frameworks such as Minimal APIs, MVC, Razor pages, SignalR, Blazor, and gRPC.
  - Building a reverse proxy with YARP.
- **Extensibility:** Customize Kestrel through configuration, middleware, and custom transports.
- **Performance diagnostics:** Kestrel provides built-in performance diagnostics features, such as logging and metrics.

Get started

ASP.NET Core project templates use Kestrel by default when not hosted with IIS. In the following template-generated `Program.cs`, the WebApplication.CreateBuilder method calls UseKestrel internally:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

For more information on configuring `WebApplication` and `WebApplicationBuilder`, see Minimal APIs quick reference.

Optional client certificates

For information on apps that must protect a subset of the app with a certificate, see Optional client certificates.

Behavior with debugger attached

The following timeouts and rate limits aren't enforced when a debugger is attached to a Kestrel process:

- KestrelServerLimits.KeepAliveTimeout
- KestrelServerLimits.RequestHeadersTimeout
- KestrelServerLimits.MinRequestBodyDataRate
- KestrelServerLimits.MinResponseDataRate
- IConnectionTimeoutFeature
- IHttpMinRequestBodyDataRateFeature
- IHttpMinResponseDataRateFeature

Additional resources

- Configure endpoints for the ASP.NET Core Kestrel web server
- Source for WebApplication.CreateBuilder method call to UseKestrel
- Configure options for the ASP.NET Core Kestrel web server
- Use HTTP/2 with the ASP.NET Core Kestrel web server
- When to use a reverse proxy with the ASP.NET Core Kestrel web server
- Host filtering with ASP.NET Core Kestrel web server
- Troubleshoot and debug ASP.NET Core projects
- Enforce HTTPS in ASP.NET Core
- Configure ASP.NET Core to work with proxy servers and load balancers
- RFC 9110: HTTP Semantics (Section 7.2: Host and :authority)

- When using UNIX sockets on Linux, the socket isn't automatically deleted on app shutdown. For more information, see this GitHub issue.

> **Note**
>
> As of ASP.NET Core 5.0, Kestrel's libuv transport is obsolete. The libuv transport doesn't receive updates to support new OS platforms, such as Windows ARM64, and will be removed in a future release. Remove any calls to the obsolete **UseLibuv** method and use Kestrel's default Socket transport instead.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Configure endpoints for the ASP.NET Core Kestrel web server

Article • 08/25/2023

Kestrel endpoints provide the infrastructure for listening to incoming requests and routing them to the appropriate middleware. The combination of an address and a protocol defines an endpoint.

- The address specifies the network interface that the server listens on for incoming requests, such as a TCP port.
- The protocol specifies the communication between the client and server, such as HTTP/1.1, HTTP/2, or HTTP/3.
- An endpoint can be secured using the `https` URL scheme or `UseHttps` method.

Endpoints can be configured using URLs, JSON in `appsettings.json`, and code. This article discusses how to use each option to configure an endpoint:

- Configure endpoints
- Configure HTTPS
- Configure HTTP protocols

Default endpoint

New ASP.NET Core projects are configured to bind to a random HTTP port between 5000-5300 and a random HTTPS port between 7000-7300. The selected ports are stored in the generated `Properties/launchSettings.json` file and can be modified by the developer. The `launchSetting.json` file is only used in local development.

If there's no endpoint configuration, then Kestrel binds to `http://localhost:5000`.

Configure endpoints

Kestrel endpoints listen for incoming connections. When an endpoint is created, it must be configured with the address it will listen to. Usually, this is a TCP address and port number.

There are several options for configuring endpoints:

- Configure endpoints with URLs
- Specify ports only

- Configure endpoints in appsettings.json
- Configure endpoints in code

Configure endpoints with URLs

The following sections explain how to configure endpoints using the:

- `ASPNETCORE_URLS` environment variable.
- `--urls` command-line argument.
- `urls` host configuration key.
- UseUrls extension method.
- WebApplication.Urls property.

URL formats

The URLs indicate the IP or host addresses with ports and protocols the server should listen on. The port can be omitted if it's the default for the protocol (typically 80 and 443). URLs can be in any of the following formats.

- IPv4 address with port number

  ```
  http://65.55.39.10:80/
  ```

  `0.0.0.0` is a special case that binds to all IPv4 addresses.

- IPv6 address with port number

  ```
  http://[0:0:0:0:0:ffff:4137:270a]:80/
  ```

  `[::]` is the IPv6 equivalent of IPv4 `0.0.0.0`.

- Wildcard host with port number

  ```
  http://contoso.com:80/
  http://*:80/
  ```

Anything not recognized as a valid IP address or `localhost` is treated as a wildcard that binds to all IPv4 and IPv6 addresses. Some people like to use `*` or `+` to be more explicit. To bind different host names to different ASP.NET Core apps on the same port, use HTTP.sys or a reverse proxy server.

Reverse proxy server examples include IIS, YARP, Nginx, and Apache.

- Host name `localhost` with port number or loopback IP with port number

  ```
  http://localhost:5000/
  http://127.0.0.1:5000/
  http://[::1]:5000/
  ```

  When `localhost` is specified, Kestrel attempts to bind to both IPv4 and IPv6 loopback interfaces. If the requested port is in use by another service on either loopback interface, Kestrel fails to start. If either loopback interface is unavailable for any other reason (most commonly because IPv6 isn't supported), Kestrel logs a warning.

Multiple URL prefixes can be specified by using a semicolon (`;`) delimiter:

```
http://*:5000;http://localhost:5001;https://hostname:5002
```

For more information, see Override configuration.

HTTPS URL prefixes

HTTPS URL prefixes can be used to define endpoints only if a default certificate is provided in the HTTPS endpoint configuration. For example, use KestrelServerOptions configuration or a configuration file, as shown later in this article.

For more information, see Configure HTTPS.

Specify ports only

Apps and containers are often given only a port to listen on, like port 80, without additional constraints like host or path. HTTP_PORTS and HTTPS_PORTS are config keys that specify the listening ports for the Kestrel and HTTP.sys servers. These keys may be specified as environment variables defined with the `DOTNET_` or `ASPNETCORE_` prefixes, or

specified directly through any other config input, such as `appsettings.json`. Each is a semicolon-delimited list of port values, as shown in the following example:

```
ASPNETCORE_HTTP_PORTS=80;8080
ASPNETCORE_HTTPS_PORTS=443;8081
```

The preceding example is shorthand for the following configuration, which specifies the scheme (HTTP or HTTPS) and any host or IP.

```
ASPNETCORE_URLS=http://*:80/;http://*:8080/;https://*:443/;https://*:8081/
```

The HTTP_PORTS and HTTPS_PORTS configuration keys are lower priority and are overridden by URLS or values provided directly in code. Certificates still need to be configured separately via server-specific mechanics for HTTPS.

Configure endpoints in appsettings.json

Kestrel can load endpoints from an IConfiguration instance. By default, Kestrel configuration is loaded from the `Kestrel` section and endpoints are configured in `Kestrel:Endpoints`:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "MyHttpEndpoint": {
        "Url": "http://localhost:8080"
      }
    }
  }
}
```

The preceding example:

- Uses `appsettings.json` as the configuration source. However, any `IConfiguration` source can be used.
- Adds an endpoint named `MyHttpEndpoint` on port 8080.

For more information about configuring endpoints with JSON, see later sections in this article that discuss configuring HTTPS and configuring HTTP protocols in appsettings.json.

Reloading endpoints from configuration

Reloading endpoint configuration when the configuration source changes is enabled by default. It can be disabled using KestrelServerOptions.Configure(IConfiguration, Boolean).

If a change is signaled, the following steps are taken:

- The new configuration is compared to the old one, and any endpoint without configuration changes isn't modified.
- Removed or modified endpoints are given 5 seconds to complete processing requests and shut down.
- New or modified endpoints are started.

Clients connecting to a modified endpoint may be disconnected or refused while the endpoint is restarted.

ConfigurationLoader

KestrelServerOptions.Configure returns a KestrelConfigurationLoader. The loader's Endpoint(String, Action<EndpointConfiguration>) method that can be used to supplement a configured endpoint's settings:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    var kestrelSection = context.Configuration.GetSection("Kestrel");

    serverOptions.Configure(kestrelSection)
        .Endpoint("HTTPS", listenOptions =>
        {
            // ...
        });
});
```

`KestrelServerOptions.ConfigurationLoader` can be directly accessed to continue iterating on the existing loader, such as the one provided by WebApplicationBuilder.WebHost.

- The configuration section for each endpoint is available on the options in the Endpoint method so that custom settings may be read.
- KestrelServerOptions.Configure(IConfiguration) can be called multiple times, but only the last configuration is used unless `Load` is explicitly called on prior instances. The default host doesn't call `Load` so that its default configuration section may be replaced.
- `KestrelConfigurationLoader` mirrors the `Listen` family of APIs from `KestrelServerOptions` as `Endpoint` overloads, so code and config endpoints can be configured in the same place. These overloads don't use names and only consume default settings from configuration.

Configure endpoints in code

KestrelServerOptions provides methods for configuring endpoints in code:

- Listen
- ListenLocalhost
- ListenAnyIP
- ListenUnixSocket
- ListenNamedPipe

When both the `Listen` and UseUrls APIs are used simultaneously, the `Listen` endpoints override the `UseUrls` endpoints.

Bind to a TCP socket

The Listen, ListenLocalhost, and ListenAnyIP methods bind to a TCP socket:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

The preceding example:

- Configures endpoints that listen on port 5000 and 5001.

- Configures HTTPS for an endpoint with the UseHttps extension method on ListenOptions. For more information, see Configure HTTPS in code.

On Windows, self-signed certificates can be created using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

On macOS, Linux, and Windows, certificates can be created using OpenSSL.

Bind to a Unix socket

Listen on a Unix socket with ListenUnixSocket for improved performance with Nginx, as shown in this example:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.ListenUnixSocket("/tmp/kestrel-test.sock");
});
```

- In the Nginx configuration file, set the `server` > `location` > `proxy_pass` entry to `http://unix:/tmp/{KESTREL SOCKET}:/;`. `{KESTREL SOCKET}` is the name of the socket provided to ListenUnixSocket (for example, `kestrel-test.sock` in the preceding example).
- Ensure that the socket is writeable by Nginx (for example, `chmod go+w /tmp/kestrel-test.sock`).

Configure endpoint defaults

ConfigureEndpointDefaults(Action<ListenOptions>) specifies configuration that runs for each specified endpoint. Calling `ConfigureEndpointDefaults` multiple times replaces previous configuration.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureEndpointDefaults(listenOptions =>
    {
        // ...
```

```
    });
});
```

> **Note**
>
> Endpoints created by calling Listen **before** calling ConfigureEndpointDefaults won't have the defaults applied.

Dynamic port binding

When port number `0` is specified, Kestrel dynamically binds to an available port. The following example shows how to determine which port Kestrel bound at runtime:

C#

```
app.Run(async (context) =>
{
    var serverAddressFeature = context.Features.Get<IServerAddressesFeature>
();

    if (serverAddressFeature is not null)
    {
        var listenAddresses = string.Join(", ",
serverAddressFeature.Addresses);

        // ...
    }
});
```

Dynamically binding a port isn't available in some situations:

- KestrelServerOptions.ListenLocalhost
- Binding TCP-based HTTP/1.1 or HTTP/2, and QUIC-based HTTP/3 together.

Configure HTTPS

Kestrel supports securing endpoints with HTTPS. Data sent over HTTPS is encrypted using Transport Layer Security (TLS) to increase the security of data transferred between the client and server.

HTTPS requires a TLS certificate. The TLS certificate is stored on the server, and Kestrel is configured to use it. An app can use the ASP.NET Core HTTPS development certificate in a local development environment. The development certificate isn't installed in

nondevelopment environments. In production, a TLS certificate must be explicitly configured. At a minimum, a default certificate must be provided.

The way HTTPS and the TLS certificate is configured depends on how endpoints are configured:

- If URL prefixes or specify ports only are used to define endpoints, HTTPS can be used only if a default certificate is provided in HTTPS endpoint configuration. A default certificate can be configured with one of the following options:
- Configure HTTPS in appsettings.json
- Configure HTTPS in code

Configure HTTPS in appsettings.json

A default HTTPS app settings configuration schema is available for Kestrel. Configure multiple endpoints, including the URLs and the certificates to use, either from a file on disk or from a certificate store.

Any HTTPS endpoint that doesn't specify a certificate (`HttpsDefaultCert` in the example that follows) falls back to the certificate defined under `Certificates:Default` or the development certificate.

The following example is for `appsettings.json`, but any configuration source can be used:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "Http": {
        "Url": "http://localhost:5000"
      },
      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      },
      "HttpsInlineCertAndKeyFile": {
        "Url": "https://localhost:5002",
        "Certificate": {
          "Path": "<path to .pem/.crt file>",
          "KeyPath": "<path to .key file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      },
```

```
      "HttpsInlineCertStore": {
        "Url": "https://localhost:5003",
        "Certificate": {
          "Subject": "<subject; required>",
          "Store": "<certificate store; required>",
          "Location": "<location; defaults to CurrentUser>",
          "AllowInvalid": "<true or false; defaults to false>"
        }
      },
      "HttpsDefaultCert": {
        "Url": "https://localhost:5004"
      }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "$CREDENTIAL_PLACEHOLDER$"
      }
    }
  }
}
```

> ⚠ **Warning**
>
> In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

Schema notes

- Endpoint names are case-insensitive. For example, `HTTPS` and `Https` are equivalent.
- The `Url` parameter is required for each endpoint. The format for this parameter is the same as the top-level `Urls` configuration parameter except that it's limited to a single value. See URL formats earlier in this article.
- These endpoints replace the ones defined in the top-level `Urls` configuration rather than adding to them. Endpoints defined in code via `Listen` are cumulative with the endpoints defined in the configuration section.
- The `Certificate` section is optional. If the `Certificate` section isn't specified, the defaults defined in `Certificates:Default` are used. If no defaults are available, the development certificate is used. If there are no defaults and the development certificate isn't present, the server throws an exception and fails to start.

- The `Certificate` section supports multiple certificate sources.
- Any number of endpoints may be defined in `Configuration`, as long as they don't cause port conflicts.

Certificate sources

Certificate nodes can be configured to load certificates from a number of sources:

- `Path` and `Password` to load *.pfx* files.
- `Path`, `KeyPath` and `Password` to load *.pem*/*.crt* and *.key* files.
- `Subject` and `Store` to load from the certificate store.

For example, the `Certificates:Default` certificate can be specified as:

JSON

```
"Default": {
  "Subject": "<subject; required>",
  "Store": "<cert store; required>",
  "Location": "<location; defaults to CurrentUser>",
  "AllowInvalid": "<true or false; defaults to false>"
}
```

Configure client certificates in appsettings.json

ClientCertificateMode is used to configure client certificate behavior.

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "MyHttpsEndpoint": {
        "Url": "https://localhost:5001",
        "ClientCertificateMode": "AllowCertificate",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      }
    }
  }
}
```

⚠ **Warning**

In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

The default value is `ClientCertificateMode.NoCertificate`, where Kestrel doesn't request or require a certificate from the client.

For more information, see Configure certificate authentication in ASP.NET Core.

Configure SSL/TLS protocols in appsettings.json

SSL Protocols are protocols used for encrypting and decrypting traffic between two peers, traditionally a client and a server.

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "MyHttpsEndpoint": {
        "Url": "https://localhost:5001",
        "SslProtocols": ["Tls12", "Tls13"],
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "$CREDENTIAL_PLACEHOLDER$"
        }
      }
    }
  }
}
```

⚠ Warning

In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

The default value, `SslProtocols.None`, causes Kestrel to use the operating system defaults to choose the best protocol. Unless you have a specific reason to select a protocol, use the default.

Configure HTTPS in code

When using the `Listen` API, the UseHttps extension method on ListenOptions is available to configure HTTPS.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Loopback, 5000);
    serverOptions.Listen(IPAddress.Loopback, 5001, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
    });
});
```

`ListenOptions.UseHttps` parameters:

- `filename` is the path and file name of a certificate file, relative to the directory that contains the app's content files.
- `password` is the password required to access the X.509 certificate data.
- `configureOptions` is an `Action` to configure the `HttpsConnectionAdapterOptions`. Returns the `ListenOptions`.
- `storeName` is the certificate store from which to load the certificate.
- `subject` is the subject name for the certificate.
- `allowInvalid` indicates if invalid certificates should be considered, such as self-signed certificates.
- `location` is the store location to load the certificate from.
- `serverCertificate` is the X.509 certificate.

For a complete list of `UseHttps` overloads, see UseHttps.

Configure client certificates in code

ClientCertificateMode configures the client certificate requirements.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.ClientCertificateMode =
ClientCertificateMode.AllowCertificate;
    });
});
```

The default value is NoCertificate, where Kestrel doesn't request or require a certificate from the client.

For more information, see Configure certificate authentication in ASP.NET Core.

Configure HTTPS defaults in code

ConfigureHttpsDefaults(Action<HttpsConnectionAdapterOptions>) specifies a configuration `Action` to run for each HTTPS endpoint. Calling `ConfigureHttpsDefaults` multiple times replaces prior `Action` instances with the last `Action` specified.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        // ...
    });
});
```

> ⓘ **Note**
>
> Endpoints created by calling **Listen** **before** calling **ConfigureHttpsDefaults** won't have the defaults applied.

Configure SSL/TLS protocols in code

SSL protocols are protocols used for encrypting and decrypting traffic between two peers, traditionally a client and a server.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.SslProtocols = SslProtocols.Tls13;
    });
});
```

Configure TLS cipher suites filter in code

On Linux, CipherSuitesPolicy can be used to filter TLS handshakes on a per-connection basis:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.ConfigureHttpsDefaults(listenOptions =>
    {
        listenOptions.OnAuthenticate = (context, sslOptions) =>
        {
            sslOptions.CipherSuitesPolicy = new CipherSuitesPolicy(
                new[]
                {
                    TlsCipherSuite.TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,
                    TlsCipherSuite.TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,
                    // ...
                });
        };
    });
});
```

Configure Server Name Indication

Server Name Indication (SNI) can be used to host multiple domains on the same IP address and port. SNI can be used to conserve resources by serving multiple sites from one server.

For SNI to function, the client sends the host name for the secure session to the server during the TLS handshake so that the server can provide the correct certificate. The

client uses the furnished certificate for encrypted communication with the server during the secure session that follows the TLS handshake.

All websites must run on the same Kestrel instance. Kestrel doesn't support sharing an IP address and port across multiple instances without a reverse proxy.

SNI can be configured in two ways:

- Configure a mapping between host names and HTTPS options in Configuration. For example, JSON in the `appsettings.json` file.
- Create an endpoint in code and select a certificate using the host name with the ServerCertificateSelector callback.

Configure SNI in appsettings.json

Kestrel supports SNI defined in configuration. An endpoint can be configured with an `Sni` object that contains a mapping between host names and HTTPS options. The connection host name is matched to the options and they're used for that connection.

The following configuration adds an endpoint named `MySniEndpoint` that uses SNI to select HTTPS options based on the host name:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "MySniEndpoint": {
        "Url": "https://*",
        "SslProtocols": ["Tls11", "Tls12"],
        "Sni": {
          "a.example.org": {
            "Protocols": "Http1AndHttp2",
            "SslProtocols": ["Tls11", "Tls12", "Tls13"],
            "Certificate": {
              "Subject": "<subject; required>",
              "Store": "<certificate store; required>",
            },
            "ClientCertificateMode" : "NoCertificate"
          },
          "*.example.org": {
            "Certificate": {
              "Path": "<path to .pfx file>",
              "Password": "$CREDENTIAL_PLACEHOLDER$"
            }
          },
          "*": {
            // At least one subproperty needs to exist per SNI section or it
            // cannot be discovered via IConfiguration
```

```
            "Protocols": "Http1",
          }
        }
      }
    },
    "Certificates": {
      "Default": {
        "Path": "<path to .pfx file>",
        "Password": "$CREDENTIAL_PLACEHOLDER$"
      }
    }
  }
}
```

> ⚠ **Warning**
>
> In the preceding example, the certificate password is stored in plain-text in `appsettings.json`. The `$CREDENTIAL_PLACEHOLDER$` token is used as a placeholder for the certificate's password. To store certificate passwords securely in development environments, see Protect secrets in development. To store certificate passwords securely in production environments, see Azure Key Vault configuration provider. Development secrets shouldn't be used for production or test.

HTTPS options that can be overridden by SNI:

- `Certificate` configures the certificate source.
- `Protocols` configures the allowed HTTP protocols.
- `SslProtocols` configures the allowed SSL protocols.
- `ClientCertificateMode` configures the client certificate requirements.

The host name supports wildcard matching:

- Exact match. For example, `a.example.org` matches `a.example.org`.
- Wildcard prefix. If there are multiple wildcard matches, then the longest pattern is chosen. For example, `*.example.org` matches `b.example.org` and `c.example.org`.
- Full wildcard. `*` matches everything else, including clients that aren't using SNI and don't send a host name.

The matched SNI configuration is applied to the endpoint for the connection, overriding values on the endpoint. If a connection doesn't match a configured SNI host name, then the connection is refused.

Configure SNI with code

Kestrel supports SNI with several callback APIs:

- `ServerCertificateSelector`
- `ServerOptionsSelectionCallback`
- `TlsHandshakeCallbackOptions`

SNI with `ServerCertificateSelector`

Kestrel supports SNI via the `ServerCertificateSelector` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenAnyIP(5005, listenOptions =>
    {
        listenOptions.UseHttps(httpsOptions =>
        {
            var localhostCert = CertificateLoader.LoadFromStoreCert(
                "localhost", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var exampleCert = CertificateLoader.LoadFromStoreCert(
                "example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var subExampleCert = CertificateLoader.LoadFromStoreCert(
                "sub.example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var certs = new Dictionary<string, X509Certificate2>(
                StringComparer.OrdinalIgnoreCase)
            {
                ["localhost"] = localhostCert,
                ["example.com"] = exampleCert,
                ["sub.example.com"] = subExampleCert
            };

            httpsOptions.ServerCertificateSelector = (connectionContext,
name) =>
            {
                if (name is not null && certs.TryGetValue(name, out var
cert))
                {
                    return cert;
                }

                return exampleCert;
            };
        });
```

```
    });
});
```

SNI with `ServerOptionsSelectionCallback`

Kestrel supports additional dynamic TLS configuration via the `ServerOptionsSelectionCallback` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate and TLS configuration. Default certificates and `ConfigureHttpsDefaults` aren't used with this callback.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenAnyIP(5005, listenOptions =>
    {
        listenOptions.UseHttps(httpsOptions =>
        {
            var localhostCert = CertificateLoader.LoadFromStoreCert(
                "localhost", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var exampleCert = CertificateLoader.LoadFromStoreCert(
                "example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);

            listenOptions.UseHttps((stream, clientHelloInfo, state,
cancellationToken) =>
            {
                if (string.Equals(clientHelloInfo.ServerName, "localhost",
                    StringComparison.OrdinalIgnoreCase))
                {
                    return new ValueTask<SslServerAuthenticationOptions>(
                        new SslServerAuthenticationOptions
                        {
                            ServerCertificate = localhostCert,
                            // Different TLS requirements for this host
                            ClientCertificateRequired = true
                        });
                }

                return new ValueTask<SslServerAuthenticationOptions>(
                    new SslServerAuthenticationOptions
                    {
                        ServerCertificate = exampleCert
                    });
            }, state: null!);
        });
```

```
    });
});
```

SNI with `TlsHandshakeCallbackOptions`

Kestrel supports additional dynamic TLS configuration via the `TlsHandshakeCallbackOptions.OnConnection` callback. The callback is invoked once per connection to allow the app to inspect the host name and select the appropriate certificate, TLS configuration, and other server options. Default certificates and `ConfigureHttpsDefaults` aren't used with this callback.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.ListenAnyIP(5005, listenOptions =>
    {
        listenOptions.UseHttps(httpsOptions =>
        {
            var localhostCert = CertificateLoader.LoadFromStoreCert(
                "localhost", "My", StoreLocation.CurrentUser,
                allowInvalid: true);
            var exampleCert = CertificateLoader.LoadFromStoreCert(
                "example.com", "My", StoreLocation.CurrentUser,
                allowInvalid: true);

            listenOptions.UseHttps(new TlsHandshakeCallbackOptions
            {
                OnConnection = context =>
                {
                    if (string.Equals(context.ClientHelloInfo.ServerName,
"localhost",
                        StringComparison.OrdinalIgnoreCase))
                    {
                        // Different TLS requirements for this host
                        context.AllowDelayedClientCertificateNegotation =
true;

                        return new ValueTask<SslServerAuthenticationOptions>
(
                            new SslServerAuthenticationOptions
                            {
                                ServerCertificate = localhostCert
                            });
                    }

                    return new ValueTask<SslServerAuthenticationOptions>(
                        new SslServerAuthenticationOptions
```

```
                        {
                            ServerCertificate = exampleCert
                        });
                }
            });
        });
    });
});
```

Configure HTTP protocols

Kestrel supports all commonly used HTTP versions. Endpoints can be configured to support different HTTP versions using the HttpProtocols enum, which specifies available HTTP version options.

TLS is required to support more than one HTTP version. The TLS Application-Layer Protocol Negotiation (ALPN) handshake is used to negotiate the connection protocol between the client and the server when an endpoint supports multiple protocols.

Expand table

`HttpProtocols` value	Connection protocol permitted
`Http1`	HTTP/1.1 only. Can be used with or without TLS.
`Http2`	HTTP/2 only. May be used without TLS only if the client supports a Prior Knowledge mode.
`Http3`	HTTP/3 only. Requires TLS. The client may need to be configured to use HTTP/3 only.
`Http1AndHttp2`	HTTP/1.1 and HTTP/2. HTTP/2 requires the client to select HTTP/2 in the TLS Application-Layer Protocol Negotiation (ALPN) handshake; otherwise, the connection defaults to HTTP/1.1.
`Http1AndHttp2AndHttp3`	HTTP/1.1, HTTP/2 and HTTP/3. The first client request normally uses HTTP/1.1 or HTTP/2, and the alt-svc response header prompts the client to upgrade to HTTP/3. HTTP/2 and HTTP/3 requires TLS; otherwise, the connection defaults to HTTP/1.1.

The default protocol value for an endpoint is `HttpProtocols.Http1AndHttp2`.

TLS restrictions for HTTP/2:

- TLS version 1.2 or later
- Renegotiation disabled
- Compression disabled

- Minimum ephemeral key exchange sizes:
  - Elliptic curve Diffie-Hellman (ECDHE) [RFC4492]: 224 bits minimum
  - Finite field Diffie-Hellman (DHE) [`TLS12`]: 2048 bits minimum
- Cipher suite not prohibited.

`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` [`TLS-ECDHE`] with the P-256 elliptic curve [`FIPS186`] is supported by default.

Configure HTTP protocols in appsettings.json

The following `appsettings.json` example establishes the HTTP/1.1 connection protocol for a specific endpoint:

JSON

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsDefaultCert": {
        "Url": "https://localhost:5001",
        "Protocols": "Http1"
      }
    }
  }
}
```

A default protocol can be configured in the `Kestrel:EndpointDefaults` section. The following `appsettings.json` example establishes HTTP/1.1 as the default connection protocol for all endpoints:

JSON

```
{
  "Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1"
    }
  }
}
```

Protocols specified in code override values set by configuration.

Configure HTTP protocols in code

ListenOptions.Protocols is used to specify protocols with the HttpProtocols enum.

The following example configures an endpoint for HTTP/1.1, HTTP/2, and HTTP/3 connections on port 8000. Connections are secured by TLS with a supplied certificate:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2AndHttp3;
    });
});
```

See also

- Kestrel web server in ASP.NET Core
- Configure options for the ASP.NET Core Kestrel web server

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Configure options for the ASP.NET Core Kestrel web server

Article • 07/26/2024

The Kestrel web server has constraint configuration options that are especially useful in Internet-facing deployments. To configure Kestrel configuration options, call ConfigureKestrel in `Program.cs`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    // ...
});
```

Set constraints on the KestrelServerOptions.Limits property. This property holds an instance of the KestrelServerLimits class.

General limits

Keep-alive timeout

KeepAliveTimeout gets or sets the keep-alive timeout:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.KeepAliveTimeout = TimeSpan.FromMinutes(2);
});
```

This timeout is not enforced when a debugger is attached to the Kestrel process.

Maximum client connections

MaxConcurrentConnections gets or sets the maximum number of open connections:

```
C#
```

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentConnections = 100;
});
```

MaxConcurrentUpgradedConnections gets or sets the maximum number of open, upgraded connections:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxConcurrentUpgradedConnections = 100;
});
```

An upgraded connection is one that has been switched from HTTP to another protocol, such as WebSockets. After a connection is upgraded, it isn't counted against the `MaxConcurrentConnections` limit.

Maximum request body size

MaxRequestBodySize gets or sets the maximum allowed size of any request body in bytes.

The recommended approach to override the limit in an ASP.NET Core MVC app is to use the RequestSizeLimitAttribute attribute on an action method:

C#

```
[RequestSizeLimit(100_000_000)]
public IActionResult Get()
```

The following example configures `MaxRequestBodySize` for all requests:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.MaxRequestBodySize = 100_000_000;
});
```

The following example configures `MaxRequestBodySize` for a specific request using IHttpMaxRequestBodySizeFeature in a custom middleware:

C#

```
app.Use(async (context, next) =>
{
    var httpMaxRequestBodySizeFeature =
context.Features.Get<IHttpMaxRequestBodySizeFeature>();

    if (httpMaxRequestBodySizeFeature is not null)
        httpMaxRequestBodySizeFeature.MaxRequestBodySize = 10 * 1024;

    // ...

    await next(context);
});
```

If the app attempts to configure the limit on a request after it starts to read the request, an exception is thrown. Ue the IHttpMaxRequestBodySizeFeature.IsReadOnly property to check if it's safe to set the `MaxRequestBodySize` property.

When an app runs out-of-process behind the ASP.NET Core Module, IIS sets the limit and Kestrel's request body size limit is disabled.

Minimum request body data rate

Kestrel checks every second if data is arriving at the specified rate in bytes/second. If the rate drops below the minimum, the connection is timed out. The grace period is the amount of time Kestrel allows the client to increase its send rate up to the minimum. The rate isn't checked during that time. The grace period helps avoid dropping connections that are initially sending data at a slow rate because of TCP slow-start. A minimum rate also applies to the response.

MinRequestBodyDataRate gets or sets the request body minimum data rate in bytes/second. MinResponseDataRate gets or sets the response minimum data rate in bytes/second.

The following example configures `MinRequestBodyDataRate` and `MinResponseDataRate` for all requests:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
```

```
    serverOptions.Limits.MinRequestBodyDataRate = new MinDataRate(
        bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    serverOptions.Limits.MinResponseDataRate = new MinDataRate(
        bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
});
```

The following example configures `MinRequestBodyDataRate` and `MinResponseDataRate` for a specific request using IHttpMinRequestBodyDataRateFeature and IHttpMinResponseDataRateFeature in a custom middleware:

C#

```
app.Use(async (context, next) =>
{
    var httpMinRequestBodyDataRateFeature = context.Features
        .Get<IHttpMinRequestBodyDataRateFeature>();

    if (httpMinRequestBodyDataRateFeature is not null)
    {
        httpMinRequestBodyDataRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    var httpMinResponseDataRateFeature = context.Features
        .Get<IHttpMinResponseDataRateFeature>();

    if (httpMinResponseDataRateFeature is not null)
    {
        httpMinResponseDataRateFeature.MinDataRate = new MinDataRate(
            bytesPerSecond: 100, gracePeriod: TimeSpan.FromSeconds(10));
    }

    // ...

    await next(context);
});
```

`IHttpMinResponseDataRateFeature` isn't present in HttpContext.Features for HTTP/2 requests. Modifying rate limits on a per-request basis isn't generally supported for HTTP/2 because of the protocol's support for request multiplexing. However, `IHttpMinRequestBodyDataRateFeature` is still present in `HttpContext.Features` for HTTP/2 requests, because the read rate limit can still be *disabled entirely* on a per-request basis by setting IHttpMinResponseDataRateFeature.MinDataRate to `null`, even for an HTTP/2 request. Attempts to read `IHttpMinRequestBodyDataRateFeature.MinDataRate` or attempts to set it to a value other than `null` result in a NotSupportedException for HTTP/2 requests.

Server-wide rate limits configured via KestrelServerOptions.Limits still apply to both HTTP/1.x and HTTP/2 connections.

Request headers timeout

RequestHeadersTimeout gets or sets the maximum amount of time the server spends receiving request headers:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.RequestHeadersTimeout = TimeSpan.FromMinutes(1);
});
```

This timeout is not enforced when a debugger is attached to the Kestrel process.

HTTP/2 limits

The limits in this section are set on KestrelServerLimits.Http2.

Maximum streams per connection

MaxStreamsPerConnection limits the number of concurrent request streams per HTTP/2 connection. Excess streams are refused:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxStreamsPerConnection = 100;
});
```

Header table size

HeaderTableSize limits the size of the header compression tables, in octets, the HPACK encoder and decoder on the server can use. The HPACK decoder decompresses HTTP headers for HTTP/2 connections:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
```

```
    serverOptions.Limits.Http2.HeaderTableSize = 4096;
});
```

Maximum frame size

MaxFrameSize indicates the size of the largest frame payload that is allowed to be received, in octets:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxFrameSize = 16_384;
});
```

Maximum request header size

MaxRequestHeaderFieldSize indicates the size of the maximum allowed size of a request header field sequence. This limit applies to both name and value sequences in their compressed and uncompressed representations:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.MaxRequestHeaderFieldSize = 8192;
});
```

Initial connection window size

InitialConnectionWindowSize indicates how much request body data the server is willing to receive and buffer at a time aggregated across all requests (streams) per connection:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.InitialConnectionWindowSize = 131_072;
});
```

Requests are also limited by InitialStreamWindowSize.

Initial stream window size

InitialStreamWindowSize indicates how much request body data the server is willing to receive and buffer at a time per stream:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.Limits.Http2.InitialStreamWindowSize = 98_304;
});
```

Requests are also limited by InitialConnectionWindowSize.

HTTP/2 keep alive ping configuration

Kestrel can be configured to send HTTP/2 pings to connected clients. HTTP/2 pings serve multiple purposes:

- Keep idle connections alive. Some clients and proxy servers close connections that are idle. HTTP/2 pings are considered as activity on a connection and prevent the connection from being closed as idle.
- Close unhealthy connections. Connections where the client doesn't respond to the keep alive ping in the configured time are closed by the server.

There are two configuration options related to HTTP/2 keep alive pings:

- KeepAlivePingDelay is a TimeSpan that configures the ping interval. The server sends a keep alive ping to the client if it doesn't receive any frames for this period of time. Keep alive pings are disabled when this option is set to TimeSpan.MaxValue.
- KeepAlivePingTimeout is a TimeSpan that configures the ping timeout. If the server doesn't receive any frames, such as a response ping, during this timeout then the connection is closed. Keep alive timeout is disabled when this option is set to TimeSpan.MaxValue.

The following example sets `KeepAlivePingDelay` and `KeepAlivePingTimeout`:

```
C#

builder.WebHost.ConfigureKestrel(serverOptions =>
{
```

```
    serverOptions.Limits.Http2.KeepAlivePingDelay =
TimeSpan.FromSeconds(30);
    serverOptions.Limits.Http2.KeepAlivePingTimeout =
TimeSpan.FromMinutes(1);
});
```

Other options

Synchronous I/O

AllowSynchronousIO controls whether synchronous I/O is allowed for the request and response.

> **⚠ Warning**
>
> A large number of blocking synchronous I/O operations can lead to thread pool starvation, which makes the app unresponsive. Only enable `AllowSynchronousIO` when using a library that doesn't support asynchronous I/O.

The following example enables synchronous I/O:

C#

```
builder.WebHost.ConfigureKestrel(serverOptions =>
{
    serverOptions.AllowSynchronousIO = true;
});
```

For information about other Kestrel options and limits, see:

- KestrelServerOptions
- KestrelServerLimits
- ListenOptions

Behavior with debugger attached

Certain timeouts and rate limits aren't enforced when a debugger is attached to a Kestrel process. For more information, see Behavior with debugger attached.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Diagnostics in Kestrel

Article • 07/26/2024

By Sourabh Shirhatti

This article provides guidance for gathering diagnostics from Kestrel to help troubleshoot issues. Topics covered include:

- **Logging**: Structured logs written to .NET Core logging. ILogger is used by app frameworks to write logs, and by users for their own logging in an app.
- **Metrics**: Representation of data measures over intervals of time, for example, requests per second. Metrics are emitted using `EventCounter` and can be observed using the dotnet-counters command line tool or with Application Insights.
- **DiagnosticSource**: `DiagnosticSource` is a mechanism for production-time logging with rich data payloads for consumption within the process. Unlike logging, which assumes data will leave the process and expects serializable data, `DiagnosticSource` works well with complex data.

Logging

Like most components in ASP.NET Core, Kestrel uses `Microsoft.Extensions.Logging` to emit log information. Kestrel employs the use of multiple categories which allows you to be selective on which logs you listen to.

Expand table

Logging Category Name	Logging Events
`Microsoft.AspNetCore.Server.Kestrel`	`ApplicationError`, `ConnectionHeadResponseBodyWrite`, `ApplicationNeverCompleted`, `RequestBodyStart`, `RequestBodyDone`, `RequestBodyNotEntirelyRead`, `RequestBodyDrainTimedOut`, `ResponseMinimumDataRateNotSatisfied`, `InvalidResponseHeaderRemoved`, `HeartbeatSlow`
`Microsoft.AspNetCore.Server.Kestrel.BadRequests`	`ConnectionBadRequest`, `RequestProcessingError`, `RequestBodyMinimumDataRateNotSatisfied`

Logging Category Name	Logging Events
`Microsoft.AspNetCore.Server.Kestrel.Connections`	`ConnectionAccepted`, `ConnectionStart`, `ConnectionStop`, `ConnectionPause`, `ConnectionResume`, `ConnectionKeepAlive`, `ConnectionRejected`, `ConnectionDisconnect`, `NotAllConnectionsClosedGracefully`, `NotAllConnectionsAborted`, `ApplicationAbortedConnection`
`Microsoft.AspNetCore.Server.Kestrel.Http2`	`Http2ConnectionError`, `Http2ConnectionClosing`, `Http2ConnectionClosed`, `Http2StreamError`, `Http2StreamResetAbort`, `HPackDecodingError`, `HPackEncodingError`, `Http2FrameReceived`, `Http2FrameSending`, `Http2MaxConcurrentStreamsReached`
`Microsoft.AspNetCore.Server.Kestrel.Http3`	`Http3ConnectionError`, `Http3ConnectionClosing`, `Http3ConnectionClosed`, `Http3StreamAbort`, `Http3FrameReceived`, `Http3FrameSending`

Connection logging

Kestrel also supports the ability to emit `Debug` level logs for byte-level communication and can be enabled on a per-endpoint basis. To enable connection logging, see configure endpoints for Kestrel

Metrics

Metrics is a representation of data measures over intervals of time, for example, requests per second. Metrics data allows observation of the state of an app at a high-level. Kestrel metrics are emitted using `EventCounter`.

> ⓘ Note
>
> The `connections-per-second` and `tls-handshakes-per-second` counters are named incorrectly. The counters:
>
> - Do ***not*** always contain the number of new connections or TLS handshakes per second

- Display the number of new connection or TLS handshakes in the last update interval as requested as the consumer of Events via the `EventCounterIntervalSec` argument in the `filterPayload` to `KestrelEventSource`.

We **recommend** consumers of these counters scale the metric value based on the `DisplayRateTimeScale` of one second.

Expand table

Name	Display Name	Description
`connections-per-second`	Connection Rate	The number of new incoming connections per update interval
`total-connections`	Total Connections	The total number of connections
`tls-handshakes-per-second`	TLS Handshake Rate	The number of new TLS handshakes per update interval
`total-tls-handshakes`	Total TLS Handshakes	The total number of TLS handshakes
`current-tls-handshakes`	Current TLS Handshakes	The number of TLS handshakes in process
`failed-tls-handshakes`	Failed TLS Handshakes	The total number of failed TLS handshakes
`current-connections`	Current Connections	The total number of connections, including idle connections
`connection-queue-length`	Connection Queue Length	The total number connections queued to the thread pool. In a healthy system at steady state, this number should always be close to zero
`request-queue-length`	Request Queue Length	The total number requests queued to the thread pool. In a healthy system at steady state, this number should always be close to zero. This metric is unlike the IIS/Http.Sys request queue and cannot be compared
`current-upgraded-requests`	Current Upgraded Requests (WebSockets)	The number of active WebSocket requests

DiagnosticSource

Kestrel emits a `DiagnosticSource` event for HTTP requests rejected at server layer such as malformed requests and protocols violations. As such, these requests never make it into the hosting layer of ASP.NET Core.

Kestrel emits these events with the `Microsoft.AspNetCore.Server.Kestrel.BadRequest` event name and an `IFeatureCollection` as the object payload. The underlying exception can be retrieved by accessing the `IBadRequestExceptionFeature` on the feature collection.

Resolving these events is a two-step process. An observer for `DiagnosticListener` must be created:

C#

```
class BadRequestEventListener : IObserver<KeyValuePair<string, object>>,
IDisposable
{
    private readonly IDisposable _subscription;
    private readonly Action<IBadRequestExceptionFeature> _callback;

    public BadRequestEventListener(DiagnosticListener diagnosticListener,
Action<IBadRequestExceptionFeature> callback)
    {
        _subscription = diagnosticListener.Subscribe(this!, IsEnabled);
        _callback = callback;
    }
    private static readonly Predicate<string> IsEnabled = (provider) =>
provider switch
    {
        "Microsoft.AspNetCore.Server.Kestrel.BadRequest" => true,
        _ => false
    };
    public void OnNext(KeyValuePair<string, object> pair)
    {
        if (pair.Value is IFeatureCollection featureCollection)
        {
            var badRequestFeature =
featureCollection.Get<IBadRequestExceptionFeature>();

            if (badRequestFeature is not null)
            {
                _callback(badRequestFeature);
            }
        }
    }
    public void OnError(Exception error) { }
    public void OnCompleted() { }
```

```
    public virtual void Dispose() => _subscription.Dispose();
}
```

Subscribe to the ASP.NET Core `DiagnosticListener` with the observer. In this example, we create a callback that logs the underlying exception.

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
var diagnosticSource = app.Services.GetRequiredService<DiagnosticListener>
();
using var badRequestListener = new BadRequestEventListener(diagnosticSource,
(badRequestExceptionFeature) =>
{
    app.Logger.LogError(badRequestExceptionFeature.Error, "Bad request
received");
});
app.MapGet("/", () => "Hello world");
app.Run();
```

Behavior with debugger attached

Certain timeouts and rate limits aren't enforced when a debugger is attached to a Kestrel process. For more information, see Behavior with debugger attached.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use HTTP/2 with the ASP.NET Core Kestrel web server

Article • 07/26/2024

HTTP/2 is available for ASP.NET Core apps if the following base requirements are met:

- Operating system
  - Windows Server 2016/Windows 10 or later‡
  - Linux with OpenSSL 1.0.2 or later (for example, Ubuntu 16.04 or later)
  - macOS 10.15 or later
- Target framework: .NET Core 2.2 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

‡Kestrel has limited support for HTTP/2 on Windows Server 2012 R2 and Windows 8.1. Support is limited because the list of supported TLS cipher suites available on these operating systems is limited. A certificate generated using an Elliptic Curve Digital Signature Algorithm (ECDSA) may be required to secure TLS connections.

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

Starting with .NET Core 3.0, HTTP/2 is enabled by default. For more information on configuration, see the Kestrel HTTP/2 limits and ListenOptions.Protocols sections.

Advanced HTTP/2 features

Additional HTTP/2 features in Kestrel support gRPC, including support for response trailers and sending reset frames.

Trailers

HTTP Trailers are similar to HTTP Headers, except they are sent after the response body is sent. For IIS and HTTP.sys, only HTTP/2 response trailers are supported.

C#

```
if (httpContext.Response.SupportsTrailers())
{
    httpContext.Response.DeclareTrailer("trailername");

    // Write body
    httpContext.Response.WriteAsync("Hello world");
```

```
    httpContext.Response.AppendTrailer("trailername", "TrailerValue");
}
```

In the preceding example code:

- `SupportsTrailers` ensures that trailers are supported for the response.
- `DeclareTrailer` adds the given trailer name to the `Trailer` response header. Declaring a response's trailers is optional, but recommended. If `DeclareTrailer` is called, it must be before the response headers are sent.
- `AppendTrailer` appends the trailer.

Reset

Reset allows for the server to reset a HTTP/2 request with a specified error code. A reset request is considered aborted.

C#

```
var resetFeature = httpContext.Features.Get<IHttpResetFeature>();
resetFeature.Reset(errorCode: 2);
```

`Reset` in the preceding code example specifies the `INTERNAL_ERROR` error code. For more information about HTTP/2 error codes, visit the HTTP/2 specification error code section.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use HTTP/3 with the ASP.NET Core Kestrel web server

Article • 09/12/2023

HTTP/3 is an approved standard and the third major version of HTTP. This article discusses the requirements for HTTP/3. HTTP/3 is fully supported in ASP.NET Core 7.0 and later.

> **Important**
>
> Apps configured to take advantage of HTTP/3 should be designed to also support HTTP/1.1 and HTTP/2.

HTTP/3 requirements

HTTP/3 has different requirements depending on the operating system. If the platform that Kestrel is running on doesn't have all the requirements for HTTP/3, then it's disabled, and Kestrel will fall back to other HTTP protocols.

Windows

- Windows 11 Build 22000 or later OR Windows Server 2022.
- TLS 1.3 or later connection.

Linux

- `libmsquic` package installed.

`libmsquic` is published via Microsoft's official Linux package repository at `packages.microsoft.com`. To install this package:

1. Add the `packages.microsoft.com` repository. See Linux Software Repository for Microsoft Products for instructions.
2. Install the `libmsquic` package using the distro's package manager. For example, `apt install libmsquic=1.9*` on Ubuntu.

Note: .NET 6 is only compatible with the 1.9.x versions of libmsquic. Libmsquic 2.x is not compatible due to breaking changes. Libmsquic receives updates to 1.9.x when needed

to incorporate security fixes.

macOS

HTTP/3 isn't currently supported on macOS and may be available in a future release.

Getting started

HTTP/3 is not enabled by default. Add configuration to `Program.cs` to enable HTTP/3.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, options) =>
{
    options.ListenAnyIP(5001, listenOptions =>
    {
        listenOptions.Protocols = HttpProtocols.Http1AndHttp2AndHttp3;
        listenOptions.UseHttps();
    });
});
```

The preceding code configures port 5001 to:

- Use HTTP/3 alongside HTTP/1.1 and HTTP/2 by specifying `HttpProtocols.Http1AndHttp2AndHttp3`.
- Enable HTTPS with `UseHttps`. HTTP/3 requires HTTPS.

Because not all routers, firewalls, and proxies properly support HTTP/3, HTTP/3 should be configured together with HTTP/1.1 and HTTP/2. This can be done by specifying HttpProtocols.Http1AndHttp2AndHttp3 as an endpoint's supported protocols.

For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

Alt-svc

HTTP/3 is discovered as an upgrade from HTTP/1.1 or HTTP/2 via the alt-svc header. That means the first request will normally use HTTP/1.1 or HTTP/2 before switching to HTTP/3. Kestrel automatically adds the `alt-svc` header if HTTP/3 is enabled.

Localhost testing

- Browsers don't allow self-signed certificates on HTTP/3, such as the Kestrel development certificate.

- `HttpClient` can be used for localhost/loopback testing in .NET 6 or later. Extra configuration is required when using `HttpClient` to make an HTTP/3 request:
  - Set HttpRequestMessage.Version to 3.0, or
  - Set HttpRequestMessage.VersionPolicy to HttpVersionPolicy.RequestVersionOrHigher.

HTTP/3 benefits

HTTP/3 uses the same semantics as HTTP/1.1 and HTTP/2: the same request methods, status codes, and message fields apply to all versions. The differences are in the underlying transport. Both HTTP/1.1 and HTTP/2 use TCP as their transport. HTTP/3 uses a new transport technology developed alongside HTTP/3 called QUIC.

HTTP/3 and QUIC have a number of benefits compared to HTTP/1.1 and HTTP/2:

- Faster response time of the first request. QUIC and HTTP/3 negotiates the connection in fewer round-trips between the client and the server. The first request reaches the server faster.
- Improved experience when there is connection packet loss. HTTP/2 multiplexes multiple requests via one TCP connection. Packet loss on the connection affects all requests. This problem is called "head-of-line blocking". Because QUIC provides native multiplexing, lost packets only impact the requests where data has been lost.
- Supports transitioning between networks. This feature is useful for mobile devices where it is common to switch between WIFI and cellular networks as a mobile device changes location. Currently, HTTP/1.1 and HTTP/2 connections fail with an error when switching networks. An app or web browsers must retry any failed HTTP requests. HTTP/3 allows the app or web browser to seamlessly continue when a network changes. Kestrel doesn't support network transitions in .NET 8. It may be available in a future release.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Connection middleware

Article • 07/26/2024

Kestrel supports connection middleware. Connection middleware is software that is assembled into a connection pipeline and runs when Kestrel receives a new connection. Each component:

- Chooses whether to pass the request to the next component in the pipeline.
- Can perform work before and after the next component in the pipeline.

Connection delegates are used to build the connection pipeline. Connection delegates are configured with the ListenOptions.Use method.

Connection middleware is different from ASP.NET Core Middleware. Connection middleware runs per-connection instead of per-request.

Connection logging

Connection logging is connection middleware that is included with ASP.NET Core. Call UseConnectionLogging to emit Debug level logs for byte-level communication on a connection.

Connection logging is helpful for troubleshooting problems in low-level communication, such as during TLS encryption and behind proxies. If `UseConnectionLogging` is placed before `UseHttps`, encrypted traffic is logged. If `UseConnectionLogging` is placed after `UseHttps`, decrypted traffic is logged.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseConnectionLogging();
    });
});
```

Create custom connection middleware

The following example shows a custom connection middleware that can filter TLS handshakes on a per-connection basis for specific ciphers if necessary. The middleware throws NotSupportedException for any cipher algorithm that the app doesn't support. Alternatively, define and compare ITlsHandshakeFeature.CipherAlgorithm to a list of acceptable cipher suites.

No encryption is used with a CipherAlgorithmType.Null cipher algorithm.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel((context, serverOptions) =>
{
    serverOptions.Listen(IPAddress.Any, 8000, listenOptions =>
    {
        listenOptions.UseHttps("testCert.pfx", "testPassword");

        listenOptions.Use((context, next) =>
        {
            var tlsFeature = context.Features.Get<ITlsHandshakeFeature>()!;

            if (tlsFeature.CipherAlgorithm == CipherAlgorithmType.Null)
            {
                throw new NotSupportedException(
                    $"Prohibited cipher: {tlsFeature.CipherAlgorithm}");
            }

            return next();
        });
    });
});
```

See also

- Configure endpoints for the ASP.NET Core Kestrel web server

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

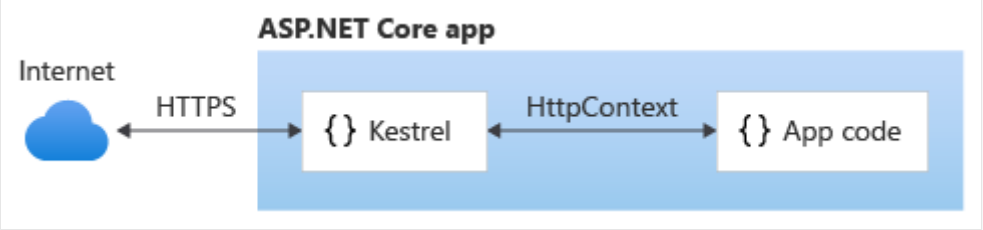
When to use Kestrel with a reverse proxy

Article • 07/26/2024

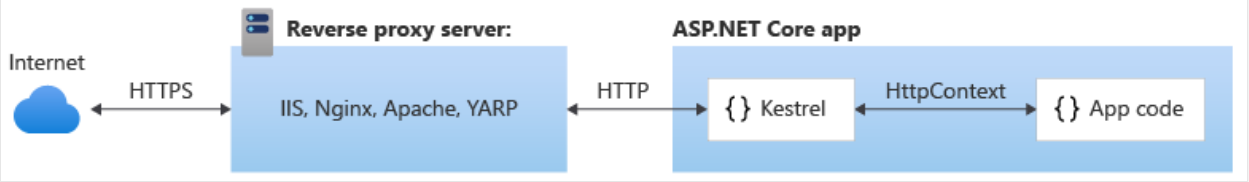
Kestrel can be used by itself or with a *reverse proxy server*. A reverse proxy server receives HTTP requests from the network and forwards them to Kestrel. Examples of a reverse proxy server include:

- Internet Information Services (IIS)
- Nginx
- Apache
- YARP: Yet Another Reverse Proxy

Kestrel used as an edge (Internet-facing) web server:



Kestrel used in a reverse proxy configuration:



Either configuration, with or without a reverse proxy server, is a supported hosting configuration.

When Kestrel is used as an edge server without a reverse proxy server, sharing of the same IP address and port among multiple processes is unsupported. When Kestrel is configured to listen on a port, Kestrel handles all traffic for that port regardless of requests' `Host` headers. A reverse proxy that can share ports can forward requests to Kestrel on a unique IP and port.

Even if a reverse proxy server isn't required, using a reverse proxy server might be a good choice.

A reverse proxy:

- Can limit the exposed public surface area of the apps that it hosts.

- Provides an additional layer of configuration and defense-in-depth cybersecurity.
- Might integrate better with existing infrastructure.
- Simplifies load balancing and secure communication (HTTPS) configuration. Only the reverse proxy server requires the X.509 certificate for the public domain(s). That server can communicate with the app's servers on the internal network using plain HTTP or HTTPS with locally managed certificates. Internal HTTPS increases security but adds significant overhead.

> ⚠ **Warning**
>
> Hosting in a reverse proxy configuration requires **host filtering**.

Additional resources

Configure ASP.NET Core to work with proxy servers and load balancers

Host filtering with ASP.NET Core Kestrel web server

Article • 07/26/2024

While Kestrel supports configuration based on prefixes such as `http://example.com:5000`, Kestrel largely ignores the host name. Host `localhost` is a special case used for binding to loopback addresses. Any host other than an explicit IP address binds to all public IP addresses. `Host` headers aren't validated.

As a workaround, use Host Filtering Middleware. The middleware is added by CreateDefaultBuilder, which calls AddHostFiltering:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Host Filtering Middleware is disabled by default. To enable the middleware, define an `AllowedHosts` key in `appsettings.json`/`appsettings.{Environment}.json`. The value is a semicolon-delimited list of host names without port numbers:

`appsettings.json`:

JSON

```
{
  "AllowedHosts": "example.com;localhost"
}
```

> **Note**
>
> Forwarded Headers Middleware also has an AllowedHosts option. Forwarded Headers Middleware and Host Filtering Middleware have similar functionality for different scenarios. Setting `AllowedHosts` with Forwarded Headers Middleware is

appropriate when the `Host` header isn't preserved while forwarding requests with a reverse proxy server or load balancer. Setting `AllowedHosts` with Host Filtering Middleware is appropriate when Kestrel is used as a public-facing edge server or when the `Host` header is directly forwarded.

For more information on Forwarded Headers Middleware, see Configure ASP.NET Core to work with proxy servers and load balancers.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Request draining with ASP.NET Core Kestrel web server

Article • 07/26/2024

Opening HTTP connections is time consuming. For HTTPS, it's also resource intensive. Therefore, Kestrel tries to reuse connections per the HTTP/1.1 protocol. A request body must be fully consumed to allow the connection to be reused. The app doesn't always consume the request body, such as HTTP POST requests where the server returns a redirect or 404 response. In the HTTP POST redirect case:

- The client may already have sent part of the POST data.
- The server writes the 301 response.
- The connection can't be used for a new request until the POST data from the previous request body has been fully read.
- Kestrel tries to drain the request body. Draining the request body means reading and discarding the data without processing it.

The draining process makes a tradeoff between allowing the connection to be reused and the time it takes to drain any remaining data:

- Draining has a timeout of five seconds, which isn't configurable.
- If all of the data specified by the `Content-Length` or `Transfer-Encoding` header hasn't been read before the timeout, the connection is closed.

Sometimes you may want to terminate the request immediately, before or after writing the response. For example, clients may have restrictive data caps. Limiting uploaded data might be a priority. In such cases to terminate a request, call HttpContext.Abort from a controller, Razor Page, or middleware.

There are caveats to calling `Abort`:

- Creating new connections can be slow and expensive.
- There's no guarantee that the client has read the response before the connection closes.
- Calling `Abort` should be rare and reserved for severe error cases, not common errors.
  - Only call `Abort` when a specific problem needs to be solved. For example, call `Abort` if malicious clients are trying to POST data or when there's a bug in client code that causes large or several requests.
  - Don't call `Abort` for common error situations, such as HTTP 404 (Not Found).

Calling HttpResponse.CompleteAsync before calling `Abort` ensures that the server has completed writing the response. However, client behavior isn't predictable and they may not read the response before the connection is aborted.

This process is different for HTTP/2 because the protocol supports aborting individual request streams without closing the connection. The five-second drain timeout doesn't apply. If there's any unread request body data after completing a response, then the server sends an HTTP/2 RST frame. Additional request body data frames are ignored.

If possible, it's better for clients to use the Expect: 100-continue request header and wait for the server to respond before starting to send the request body. That gives the client an opportunity to examine the response and abort before sending unneeded data.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Host ASP.NET Core on Windows with IIS

Article • 04/26/2024

Internet Information Services (IIS) is a flexible, secure and manageable Web Server for hosting web apps, including ASP.NET Core.

Supported platforms

The following operating systems are supported:

- Windows 7 or later
- Windows Server 2012 R2 or later

Apps published for 32-bit (x86) or 64-bit (x64) deployment are supported. Deploy a 32-bit app with a 32-bit (x86) .NET Core SDK unless the app:

- Requires the larger virtual memory address space available to a 64-bit app.
- Requires the larger IIS stack size.
- Has 64-bit native dependencies.

Install the ASP.NET Core Module/Hosting Bundle

Download the latest installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

For more details instructions on how to install the ASP.NET Core Module, or installing different versions, see Install the .NET Core Hosting Bundle.

To download previous versions of the hosting bundle, see this GitHub issue.

Get started

For getting started with hosting a website on IIS, see our getting started guide.

For getting started with hosting a website on Azure App Services, see our deploying to Azure App Service guide.

Configuration

For configuration guidance, see Advanced configuration.

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module (ANCM) for IIS
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core

Overlapped recycle

In general, we recommend using a pattern like blue-green deployments for zero-downtime deployments. Features like Overlapped Recycle help, but don't guarantee that you can do a zero-downtime deployment. For more information, see this GitHub issue.

Optional client certificates

For information on apps that must protect a subset of the app with a certificate, see Optional client certificates.

Additional resources

- Troubleshoot and debug ASP.NET Core projects
- Overview of ASP.NET Core
- The Official Microsoft IIS Site
- Windows Server technical content library
- HTTP/2 on IIS
- Transform web.config

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core Module (ANCM) for IIS

Article • 04/27/2023

The ASP.NET Core Module (ANCM) is a native IIS module that plugs into the IIS pipeline, allowing ASP.NET Core applications to work with IIS. Run ASP.NET Core apps with IIS by either:

- Hosting an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forwarding web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

There are trade-offs between each of the hosting models. By default, the in-process hosting model is used due to better performance and diagnostics.

For more information and configuration guidance, see the following topics:

- Web server implementations in ASP.NET Core

Install ASP.NET Core Module (ANCM)

The ASP.NET Core Module (ANCM) is installed with the .NET Core Runtime from the .NET Core Hosting Bundle. The ASP.NET Core Module is forward and backward compatible with in-support releases of .NET.

Breaking changes and security advisories are reported on the Announcements repo. Announcements can be limited to a specific version by selecting a **Label** filter.

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

For more information, including installing an earlier version of the module, see Hosting Bundle.

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see Publish an ASP.NET Core app to IIS.

Additional resources

- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service

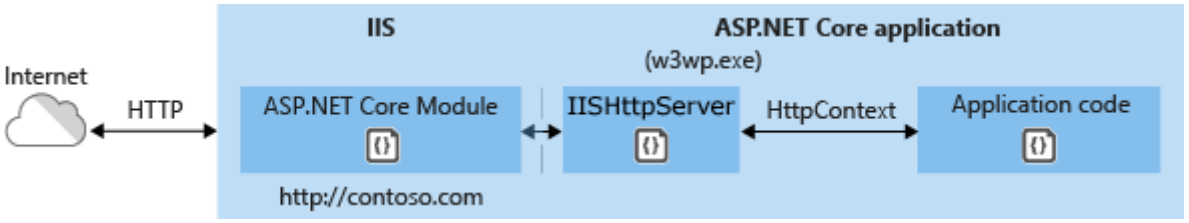
- ASP.NET Core Module reference source [default branch (main)]: Use the **Branch** drop down list to select a specific release (for example, `release/3.1`).
- IIS modules with ASP.NET Core

In-process hosting with IIS and ASP.NET Core

Article • 07/31/2024

In-process hosting runs an ASP.NET Core app in the same process as its IIS worker process. In-process hosting provides improved performance over out-of-process hosting because requests aren't proxied over the loopback adapter, a network interface that returns outgoing network traffic back to the same machine.

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted in-process:



Enable in-process hosting

Since ASP.NET Core 3.0, in-process hosting has been enabled by default for all app deployed to IIS.

To explicitly configure an app for in-process hosting, set the value of the `<AspNetCoreHostingModel>` property to `InProcess` in the project file (`.csproj`):

XML

```
<PropertyGroup>
  <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

General architecture

The general flow of a request is as follows:

1. A request arrives from the web to the kernel-mode HTTP.sys driver.
2. The driver routes the native request to IIS on the website's configured port, usually 80 (HTTP) or 443 (HTTPS).
3. The ASP.NET Core Module receives the native request and passes it to IIS HTTP Server (`IISHttpServer`). IIS HTTP Server is an in-process server implementation for

IIS that converts the request from native to managed.

After the IIS HTTP Server processes the request:

1. The request is sent to the ASP.NET Core middleware pipeline.
2. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic.
3. The app's response is passed back to IIS through IIS HTTP Server.
4. IIS sends the response to the client that initiated the request.

`CreateDefaultBuilder` adds an IServer instance by calling the UseIIS method to boot the CoreCLR and host the app inside of the IIS worker process (`w3wp.exe` or `iisexpress.exe`). Performance tests indicate that hosting a .NET Core app in-process delivers significantly higher request throughput compared to hosting the app out-of-process and proxying requests to Kestrel.

Apps published as a single file executable can't be loaded by the in-process hosting model.

Application configuration

To configure IIS options, include a service configuration for IISServerOptions in `Program.cs`. The following example disables AutomaticAuthentication:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Server.IIS;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

```
builder.Services.Configure<IISServerOptions>(options =>
{
    options.AutomaticAuthentication = false;
});

builder.Services.AddTransient<IClaimsTransformation, MyClaimsTransformation>
();
builder.Services.AddAuthentication(IISServerDefaults.AuthenticationScheme);

builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Expand table

Option	Default	Setting
`AutomaticAuthentication`	`true`	If `true`, IIS Server sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the server only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see Windows Authentication.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`AllowSynchronousIO`	`false`	Whether synchronous I/O is allowed for the

Option	Default	Setting
		`HttpContext.Request` and the `HttpContext.Response`.
`MaxRequestBodySize`	`30000000`	Gets or sets the max request body size for the `HttpRequest`. Note that IIS itself has the limit `maxAllowedContentLength` which will be processed before the `MaxRequestBodySize` set in the `IISServerOptions`. Changing the `MaxRequestBodySize` won't affect the `maxAllowedContentLength`. To increase `maxAllowedContentLength`, add an entry in the `web.config` to set `maxAllowedContentLength` to a higher value. For more details, see Configuration.

Differences between in-process and out-of-process hosting

The following characteristics apply when hosting in-process:

- IIS HTTP Server (`IISHttpServer`) is used instead of Kestrel server. For in-process, CreateDefaultBuilder calls UseIIS to:
  - Register the `IISHttpServer`.
  - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
  - Configure the host to capture startup errors.
- The requestTimeout attribute doesn't apply to in-process hosting.
- Sharing an app pool among apps isn't supported. Use one app pool per app.
- The architecture (bitness) of the app and installed runtime (x64 or x86) must match the architecture of the app pool. For example, apps published for 32-bit (x86) must have 32-bit enabled for their IIS Application Pools. For more information, see the Create the IIS site section.
- Client disconnects are detected. The HttpContext.RequestAborted cancellation token is cancelled when the client disconnects.
- When hosting in-process, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. When transforming claims with an IClaimsTransformation implementation, call AddAuthentication to add authentication services:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Server.IIS;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<IISServerOptions>(options =>
{
    options.AutomaticAuthentication = false;
});

builder.Services.AddTransient<IClaimsTransformation, MyClaimsTransformation>
();
builder.Services.AddAuthentication(IISServerDefaults.AuthenticationScheme);

builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
```

```
app.Run();
```

- Web Package (single-file) deployments aren't supported.

Get timing information

See Get detailed timing information with IHttpSysRequestTimingFeature.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

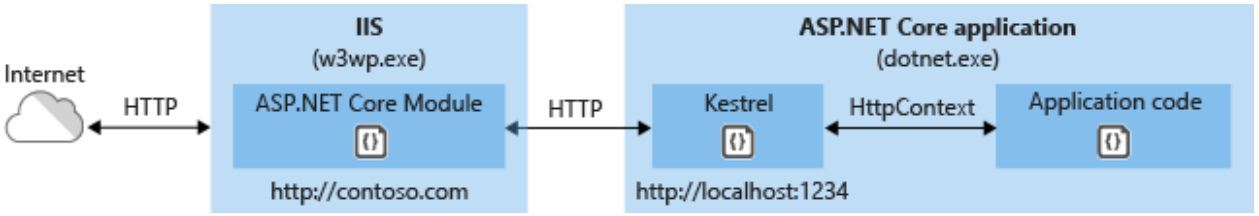
Provide product feedback

Out-of-process hosting with IIS and ASP.NET Core

Article • 07/31/2024

Because ASP.NET Core apps run in a process separate from the IIS worker process, the ASP.NET Core Module handles process management. The module starts the process for the ASP.NET Core app when the first request arrives and restarts the app if it shuts down or crashes. This is essentially the same behavior as seen with apps that run in-process that are managed by the Windows Process Activation Service (WAS).

The following diagram illustrates the relationship between IIS, the ASP.NET Core Module, and an app hosted out-of-process:



1. Requests arrive from the web to the kernel-mode HTTP.sys driver.
2. The driver routes the requests to IIS on the website's configured port. The configured port is usually 80 (HTTP) or 443 (HTTPS).
3. The module forwards the requests to Kestrel on a random port for the app. The random port isn't 80 or 443.

The ASP.NET Core Module specifies the port via an environment variable at startup. The UseIISIntegration extension configures the server to listen on `http://localhost:{PORT}`. Additional checks are performed, and requests that don't originate from the module are rejected. The module doesn't support HTTPS forwarding. Requests are forwarded over HTTP even if received by IIS over HTTPS.

After Kestrel picks up the request from the module, the request is forwarded into the ASP.NET Core middleware pipeline. The middleware pipeline handles the request and passes it on as an `HttpContext` instance to the app's logic. Middleware added by IIS Integration updates the scheme, remote IP, and pathbase to account for forwarding the request to Kestrel. The app's response is passed back to IIS, which forwards it back to the HTTP client that initiated the request.

For ASP.NET Core Module configuration guidance, see ASP.NET Core Module (ANCM) for IIS.

For more information on hosting, see Host in ASP.NET Core.

Application configuration

Enable the IISIntegration components

When building a host in `CreateHostBuilder` (`Program.cs`), call CreateDefaultBuilder to enable IIS integration:

```
C#

public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        ...
```

For more information on `CreateDefaultBuilder`, see .NET Generic Host in ASP.NET Core.

Out-of-process hosting model

To configure IIS options, include a service configuration for IISOptions in ConfigureServices. The following example prevents the app from populating `HttpContext.Connection.ClientCertificate`:

```
C#

services.Configure<IISOptions>(options =>
{
    options.ForwardClientCertificate = false;
});
```

Expand table

Option	Default	Setting
`AutomaticAuthentication`	`true`	If `true`, IIS Integration Middleware sets the `HttpContext.User` authenticated by Windows Authentication. If `false`, the middleware only provides an identity for `HttpContext.User` and responds to challenges when explicitly requested by the `AuthenticationScheme`. Windows Authentication must be enabled in IIS for `AutomaticAuthentication` to function. For more information, see the Windows Authentication topic.
`AuthenticationDisplayName`	`null`	Sets the display name shown to users on login pages.
`ForwardClientCertificate`	`true`	If `true` and the `MS-ASPNETCORE-CLIENTCERT` request header is present, the `HttpContext.Connection.ClientCertificate`

Option	Default	Setting
		is populated.

Proxy server and load balancer scenarios

The IIS Integration Middleware and the ASP.NET Core Module are configured to forward the:

- Scheme (HTTP/HTTPS).
- Remote IP address where the request originated.

The IIS Integration Middleware configures Forwarded Headers Middleware.

Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Out-of-process hosting model

To configure an app for out-of-process hosting, set the value of the `<AspNetCoreHostingModel>` property to `OutOfProcess` in the project file (`.csproj`):

XML

```
<PropertyGroup>
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>
</PropertyGroup>
```

In-process hosting is set with `InProcess`, which is the default value.

The value of `<AspNetCoreHostingModel>` is case insensitive, so `inprocess` and `outofprocess` are valid values.

Kestrel server is used instead of IIS HTTP Server (`IISHttpServer`).

For out-of-process, CreateDefaultBuilder calls UseIISIntegration to:

- Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
- Configure the host to capture startup errors.

Process name

`Process.GetCurrentProcess().ProcessName` reports `w3wp`/`iisexpress` (in-process) or `dotnet` (out-of-process).

Many native modules, such as Windows Authentication, remain active. To learn more about IIS modules active with the ASP.NET Core Module, see IIS modules with ASP.NET Core.

The ASP.NET Core Module can also:

- Set environment variables for the worker process.
- Log stdout output to file storage for troubleshooting startup issues.
- Forward Windows authentication tokens.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

The .NET Core Hosting Bundle

Article • 07/31/2024

The .NET Core Hosting bundle is an installer for the .NET Core Runtime and the ASP.NET Core Module. The bundle allows ASP.NET Core apps to run with IIS.

Install the .NET Core Hosting Bundle

> **Important**
>
> If the Hosting Bundle is installed before IIS, the bundle installation must be repaired. Run the Hosting Bundle installer again after installing IIS.
>
> If the Hosting Bundle is installed after installing the 64-bit (x64) version of .NET Core, SDKs might appear to be missing (**No .NET Core SDKs were detected**). To resolve the problem, see **Troubleshoot and debug ASP.NET Core projects**.

Breaking changes and security advisories are reported on the Announcements repo. Announcements can be limited to a specific version by selecting a **Label** filter.

Direct download

Download the installer using the following links:

- Current version:.NET Core Hosting Bundle installer (direct download)
- Previous and pre-release versions

Visual C++ Redistributable Requirement

On older versions of Windows, for example Windows Server 2012 R2, install the Visual Studio C++ 2015, 2017, 2019 Redistributable. Otherwise, a confusing error message in the Windows Event Log reports that `The data is the error.`

Current x64 VS C++ redistributable Current x86 VS C++ redistributable

Earlier versions of the installer

To obtain an earlier version of the installer:

1. Navigate to the Download .NET Core page.
2. Select the desired .NET Core version.
3. In the **Run apps - Runtime** column, find the row of the .NET Core runtime version desired.
4. Download the installer using the **Hosting Bundle** link.

> ⚠ **Warning**
>
> Some installers contain release versions that have reached their end of life (EOL) and are no longer supported by Microsoft. For more information, see the support policy.
>
> The ASP.NET Core Module is forward and backward compatible with in-support releases of .NET.

Options

1. The following parameters are available when running the installer from an administrator command shell:

   - `OPT_NO_ANCM=1`: Skip installing the ASP.NET Core Module.
   - `OPT_NO_RUNTIME=1`: Skip installing the .NET Core runtime. Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_SHAREDFX=1`: Skip installing the ASP.NET Shared Framework (ASP.NET runtime). Used when the server only hosts self-contained deployments (SCD).
   - `OPT_NO_X86=1`: Skip installing x86 runtimes. Use this parameter when you know that you won't be hosting 32-bit apps. If there's any chance that you will host both 32-bit and 64-bit apps in the future, don't use this parameter and install both runtimes.
   - `OPT_NO_SHARED_CONFIG_CHECK=1`: Disable the check for using an IIS Shared Configuration when the shared configuration (`applicationHost.config`) is on the same machine as the IIS installation. *Only available for ASP.NET Core 2.2 or later Hosting Bundler installers.* For more information, see Advanced configuration.

> ⓘ **Note**
>
> For information on IIS Shared Configuration, see ASP.NET Core Module with IIS Shared Configuration.

> **Note**
>
> When running the Hosting Bundle installer with options set, the value for each option is saved in the registry. Subsequent installs from the same Major.Minor version band use the same options, unless another set of options is explicitly passed from the command line. If the first install of the hosting bundle has no options passed, each option gets a default value of `0` written in to the registry. A value of `0` implies that the option is off, meaning the user is not opting out of the given component.

Restart IIS

After the Hosting Bundle is installed, a manual IIS restart may be required. For example, the `dotnet` CLI tooling (command) might not exist on the PATH for running IIS worker processes.

To manually restart IIS, stop the Windows Process Activation Service (WAS) and then restart the World Wide Web Publishing Service (W3SVC) and any dependent services. Execute the following commands in an elevated command shell:

Console

```
net stop was /y
net start w3svc
```

Module version and Hosting Bundle installer logs

To determine the version of the installed ASP.NET Core Module:

1. On the hosting system, navigate to `%PROGRAMFILES%\IIS\Asp.Net Core Module\V2`.
2. Locate the `aspnetcorev2.dll` file.
3. Right-click the file and select **Properties** from the contextual menu.
4. Select the **Details** tab. The **File version** and **Product version** represent the installed version of the module.

The Hosting Bundle installer logs for the module are found at `C:\Users\%UserName%\AppData\Local\Temp`. The file is named

`dd_DotNetCoreWinSvrHosting__{TIMESTAMP}_000_AspNetCoreModule_x64.log`, where the placeholder `{TIMESTAMP}` is the timestamp of the file.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

`web.config` file

Article • 07/31/2024

The `web.config` is a file that is read by IIS and the ASP.NET Core Module to configure an app hosted with IIS.

`web.config` file location

In order to set up the ASP.NET Core Module correctly, the `web.config` file must be present at the content root path (typically the app base path) of the deployed app. This is the same location as the website physical path provided to IIS. The `web.config` file is required at the root of the app to enable the publishing of multiple apps using Web Deploy.

Sensitive files exist on the app's physical path, such as `{ASSEMBLY}.runtimeconfig.json`, `{ASSEMBLY}.xml` (XML Documentation comments), and `{ASSEMBLY}.deps.json`, where the placeholder `{ASSEMBLY}` is the assembly name. When the `web.config` file is present and the site starts normally, IIS doesn't serve these sensitive files if they're requested. If the `web.config` file is missing, incorrectly named, or unable to configure the site for normal startup, IIS may serve sensitive files publicly.

The `web.config` file must be present in the deployment at all times, correctly named, and able to configure the site for normal start up. Never remove the `web.config` file from a production deployment.

If a `web.config` file isn't present in the project, the file is created with the correct `processPath` and `arguments` to configure the ASP.NET Core Module and moved to published output.

If a `web.config` file is present in the project, the file is transformed with the correct `processPath` and `arguments` to configure the ASP.NET Core Module and moved to published output. The transformation doesn't modify IIS configuration settings in the file.

The `web.config` file may provide additional IIS configuration settings that control active IIS modules. For information on IIS modules that are capable of processing requests with ASP.NET Core apps, see the IIS modules topic.

Creating, transforming, and publishing the `web.config` file is handled by an MSBuild target (`_TransformWebConfig`) when the project is published. This target is present in the

Web SDK targets (`Microsoft.NET.Sdk.Web`). The SDK is set at the top of the project file:

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">
```

To prevent the Web SDK from transforming the `web.config` file, use the `<IsTransformWebConfigDisabled>` property in the project file:

```xml
<PropertyGroup>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

When disabling the Web SDK from transforming the file, the `processPath` and `arguments` should be manually set by the developer. For more information, see ASP.NET Core Module (ANCM) for IIS.

Configuration of ASP.NET Core Module with `web.config`

The ASP.NET Core Module is configured with the `aspNetCore` section of the `system.webServer` node in the site's `web.config` file.

The following `web.config` file is published for a framework-dependent deployment and configures the ASP.NET Core Module to handle site requests:

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*"
modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet"
                  arguments=".\MyApp.dll"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
```

```
  </location>
</configuration>
```

The following `web.config` is published for a self-contained deployment:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <handlers>
        <add name="aspNetCore" path="*" verb="*"
modules="AspNetCoreModuleV2" resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath=".\MyApp.exe"
                  stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout"
                  hostingModel="inprocess" />
    </system.webServer>
  </location>
</configuration>
```

The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the app.

When an app is deployed to Azure App Service, the `stdoutLogFile` path is set to `\\?\%home%\LogFiles\stdout`. The path saves stdout logs to the `LogFiles` folder, which is a location automatically created by the service.

For information on IIS sub-application configuration, see Advanced configuration.

Attributes of the `aspNetCore` element

Expand table

Attribute	Description	Default
`arguments`	Optional string attribute. Arguments to the executable specified in `processPath`.	
`disableStartUpErrorPage`	Optional Boolean attribute. If true, the **502.5 - Process Failure** page is suppressed, and the 502 status code	`false`

Attribute	Description	Default
	page configured in the `web.config` takes precedence.	
`forwardWindowsAuthToken`	Optional Boolean attribute. If true, the token is forwarded to the child process listening on `%ASPNETCORE_PORT%` as a header 'MS-ASPNETCORE-WINAUTHTOKEN' per request. It's the responsibility of that process to call CloseHandle on this token per request.	`true`
`hostingModel`	Optional string attribute. Specifies the hosting model as in-process (`InProcess`/`inprocess`) or out-of-process (`OutOfProcess`/`outofprocess`).	`OutOfProcess`/`outofprocess` when not present
`processesPerApplication`	Optional integer attribute. Specifies the number of instances of the process specified in the `processPath` setting that can be spun up per app. †For in-process hosting, the value is limited to `1`. Setting `processesPerApplication` is discouraged. This attribute will be removed in a future release.	Default: `1` Min: `1` Max: `100`†
`processPath`	Required string attribute. Path to the executable that launches a process listening for HTTP requests. Relative paths are supported. If the path begins with `.`, the path is considered to be relative to the site root.	
`rapidFailsPerMinute`	Optional integer attribute. Specifies the number of times the process specified in `processPath` is allowed to crash per minute. If this limit is exceeded, the module stops	Default: `10` Min: `0` Max: `100`

Attribute	Description	Default
	launching the process for the remainder of the minute. Not supported with in-process hosting.	
`requestTimeout`	Optional timespan attribute. Specifies the duration for which the ASP.NET Core Module waits for a response from the process listening on %ASPNETCORE_PORT%. In versions of the ASP.NET Core Module that shipped with the release of ASP.NET Core 2.1 or later, the `requestTimeout` is specified in hours, minutes, and seconds. Doesn't apply to in-process hosting. For in-process hosting, the module waits for the app to process the request. Valid values for minutes and seconds segments of the string are in the range 0-59. Use of `60` in the value for minutes or seconds results in a *500 - Internal Server Error*.	Default: `00:02:00` Min: `00:00:00` Max: `360:00:00`
`shutdownTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to gracefully shutdown when the `app_offline.htm` file is detected.	Default: `10` Min: `0` Max: `600`
`startupTimeLimit`	Optional integer attribute. Duration in seconds that the module waits for the executable to start a process listening on the port. If this time limit is exceeded, the module kills the process. When hosting *in-process*: The process is **not** restarted and does **not** use the `rapidFailsPerMinute` setting. When hosting *out-of-process*: The module attempts to relaunch the	Default: `120` Min: `0` Max: `3600`

Attribute	Description	Default
	process when it receives a new request and continues to attempt to restart the process on subsequent incoming requests unless the app fails to start `rapidFailsPerMinute` number of times in the last rolling minute. A value of 0 (zero) is **not** considered an infinite timeout.	
`stdoutLogEnabled`	Optional Boolean attribute. If true, `stdout` and `stderr` for the process specified in `processPath` are redirected to the file specified in `stdoutLogFile`.	`false`
`stdoutLogFile`	Optional string attribute. Specifies the relative or absolute file path for which `stdout` and `stderr` from the process specified in `processPath` are logged. Relative paths are relative to the root of the site. Any path starting with `.` are relative to the site root and all other paths are treated as absolute paths. Any folders provided in the path are created by the module when the log file is created. Using underscore delimiters, a timestamp, process ID, and file extension (`.log`) are added to the last segment of the `stdoutLogFile` path. If `.\logs\stdout` is supplied as a value, an example stdout log is saved as `stdout_20180205194132_1934.log` in the `logs` folder when saved on February 5, 2018 at 19:41:32 with a process ID of 1934.	`aspnetcore-stdout`

Set environment variables

Environment variables can be specified for the process in the `processPath` attribute. Specify an environment variable with the `<environmentVariable>` child element of an `<environmentVariables>` collection element. Environment variables set in this section take precedence over system environment variables.

The following example sets two environment variables in `web.config`. `ASPNETCORE_ENVIRONMENT` configures the app's environment to `Development`. A developer may temporarily set this value in the `web.config` file in order to force the Developer Exception Page to load when debugging an app exception. `CONFIG_DIR` is an example of a user-defined environment variable, where the developer has written code that reads the value on startup to form a path for loading the app's configuration file.

XML

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="inprocess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
/>
    <environmentVariable name="CONFIG_DIR" value="f:\application_config" />
  </environmentVariables>
</aspNetCore>
```

> **Note**
>
> An alternative to setting the environment directly in `web.config` is to include the `<EnvironmentName>` property in the publish profile (.pubxml) or project file. This approach sets the environment in `web.config` when the project is published:
>
> XML
>
> ```
> <PropertyGroup>
>   <EnvironmentName>Development</EnvironmentName>
> </PropertyGroup>
> ```

> **⚠ Warning**
>
> Only set the `ASPNETCORE_ENVIRONMENT` environment variable to `Development` on staging and testing servers that aren't accessible to untrusted networks, such as the Internet.

Configuration of IIS with `web.config`

IIS configuration is influenced by the `<system.webServer>` section of `web.config` for IIS scenarios that are functional for ASP.NET Core apps with the ASP.NET Core Module. For example, IIS configuration is functional for dynamic compression. If IIS is configured at the server level to use dynamic compression, the `<urlCompression>` element in the app's `web.config` file can disable it for an ASP.NET Core app.

For more information, see the following topics:

- Configuration reference for <system.webServer>
- ASP.NET Core Module (ANCM) for IIS
- IIS modules with ASP.NET Core

To set environment variables for individual apps running in isolated app pools (supported for IIS 10.0 or later), see the *`AppCmd.exe` command* section of the Environment Variables <environmentVariables> topic in the IIS reference documentation.

Configuration sections of `web.config`

Configuration sections of ASP.NET 4.x apps in `web.config` aren't used by ASP.NET Core apps for configuration:

- `<system.web>`
- `<appSettings>`
- `<connectionStrings>`
- `<location>`

ASP.NET Core apps are configured using other configuration providers. For more information, see Configuration.

Transform web.config

If you need to transform `web.config` on publish, see Transform web.config. You might need to transform `web.config` on publish to set environment variables based on the configuration, profile, or environment.

Additional resources

- IIS <system.webServer>
- IIS modules with ASP.NET Core
- Transform web.config

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Development-time IIS support in Visual Studio for ASP.NET Core

Article • 07/31/2024

By Sourabh Shirhatti

This article describes Visual Studio support for debugging ASP.NET Core apps running with IIS on Windows Server. This topic walks through enabling this scenario and setting up a project.

Prerequisites

- Visual Studio for Windows
- **ASP.NET and web development** workload
- **.NET Core cross-platform development** workload
- X.509 security certificate (for HTTPS support)

Enable IIS

1. In Windows, navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Select the **Internet Information Services** checkbox. Select **OK**.

The IIS installation may require a system restart.

Configure IIS

IIS must have a website configured with the following:

- **Host name**: Typically, the **Default Web Site** is used with a **Host name** of `localhost`. However, any valid IIS website with a unique host name works.
- **Site Binding**
  - For apps that require HTTPS, create a binding to port 443 with a certificate. Typically, the **IIS Express Development Certificate** is used, but any valid certificate works.
  - For apps that use HTTP, confirm the existence of a binding to port 80 or create a binding to port 80 for a new site.
  - Use a single binding for either HTTP or HTTPS. **Binding to both HTTP and HTTPS ports simultaneously isn't supported.**

Configure the project

HTTPS redirection

For a new project that requires HTTPS, select the checkbox to **Configure for HTTPS** in the **Create a new ASP.NET Core Web Application** window. Selecting the checkbox adds HTTPS Redirection and HSTS Middleware to the app when it's created.

For an existing project that requires HTTPS, use HTTPS Redirection and HSTS Middleware in `Startup.Configure`. For more information, see Enforce HTTPS in ASP.NET Core.

For a project that uses HTTP, HTTPS Redirection and HSTS Middleware aren't added to the app. No app configuration is required.

IIS launch profile

Create a new launch profile to add development-time IIS support:

1. Right-click the project in **Solution Explorer**. Select **Properties**. Open the **Debug** tab.

2. For **Profile**, select the **New** button. Name the profile "IIS" in the popup window. Select **OK** to create the profile.

3. For the **Launch** setting, select **IIS** from the list.

4. Select the checkbox for **Launch browser** and provide the endpoint URL.

   When the app requires HTTPS, use an HTTPS endpoint (`https://`). For HTTP, use an HTTP (`http://`) endpoint.

   Provide the same host name and port as the IIS configuration specified earlier uses, typically `localhost`.

   Provide the name of the app at the end of the URL.

   For example, `https://localhost/WebApplication1` (HTTPS) or `http://localhost/WebApplication1` (HTTP) are valid endpoint URLs.

5. In the **Environment variables** section, select the **Add** button. Provide an environment variable with a **Name** of `ASPNETCORE_ENVIRONMENT` and a **Value** of `Development`.

6. In the **Web Server Settings** area, set the **App URL** to the same value used for the **Launch browser** endpoint URL.

7. For the **Hosting Model** setting in Visual Studio 2019 or later, select **Default** to use the hosting model used by the project. If the project sets the `<AspNetCoreHostingModel>` property in its project file, the value of the property (`InProcess` or `OutOfProcess`) is used. If the property isn't present, the default hosting model of the app is used, which is in-process. If the app requires an explicit hosting model setting different from the app's normal hosting model, set the **Hosting Model** to either `In Process` or `Out Of Process` as needed.

8. Save the profile.

When not using Visual Studio, manually add a launch profile to the launchSettings.json file in the *Properties* folder. The following example configures the profile to use the HTTPS protocol:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iis": {
      "applicationUrl": "https://localhost/WebApplication1",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS": {
      "commandName": "IIS",
      "launchBrowser": true,
      "launchUrl": "https://localhost/WebApplication1",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Confirm that the `applicationUrl` and `launchUrl` endpoints match and use the same protocol as the IIS binding configuration, either HTTP or HTTPS.

Run the project

Run Visual Studio as an administrator:

- Confirm that the build configuration drop-down list is set to **Debug**.
- Set the Start Debugging button to the **IIS** profile and select the button to start the app.

Visual Studio may prompt a restart if not running as an administrator. If prompted, restart Visual Studio.

If an untrusted development certificate is used, the browser may require you to create an exception for the untrusted certificate.

> **Note**
>
> Debugging a Release build configuration with Just My Code and compiler optimizations results in a degraded experience. For example, break points aren't hit.

Additional resources

- Getting Started with the IIS Manager in IIS
- Enforce HTTPS in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

IIS modules with ASP.NET Core

Article • 07/31/2024

Some of the native IIS modules and all of the IIS managed modules aren't able to process requests for ASP.NET Core apps. In many cases, ASP.NET Core offers an alternative to the scenarios addressed by IIS native and managed modules.

Native modules

The table indicates native IIS modules that are functional with ASP.NET Core apps and the ASP.NET Core Module.

Expand table

Module	Functional with ASP.NET Core apps	ASP.NET Core Option
Anonymous Authentication `AnonymousAuthenticationModule`	Yes	
Basic Authentication `BasicAuthenticationModule`	Yes	
Client Certification Mapping Authentication `CertificateMappingAuthenticationModule`	Yes	
CGI `CgiModule`	No	
Configuration Validation `ConfigurationValidationModule`	Yes	
HTTP Errors `CustomErrorModule`	No	Status Code Pages Middleware
Custom Logging `CustomLoggingModule`	Yes	
Default Document `DefaultDocumentModule`	No	Default Files Middleware
Digest Authentication `DigestAuthenticationModule`	Yes	
Directory Browsing `DirectoryListingModule`	No	Directory Browsing Middleware

Module	Functional with ASP.NET Core apps	ASP.NET Core Option
Dynamic Compression `DynamicCompressionModule`	Yes	Response Compression Middleware
Failed Requests Tracing `FailedRequestsTracingModule`	Yes	ASP.NET Core Logging
File Caching `FileCacheModule`	No	Response Caching Middleware
HTTP Caching `HttpCacheModule`	No	Response Caching Middleware
HTTP Logging `HttpLoggingModule`	Yes	ASP.NET Core Logging
HTTP Redirection `HttpRedirectionModule`	Yes	URL Rewriting Middleware
HTTP Tracing `TracingModule`	Yes	
IIS Client Certificate Mapping Authentication `IISCertificateMappingAuthenticationModule`	Yes	
IP and Domain Restrictions `IpRestrictionModule`	Yes	
ISAPI Filters `IsapiFilterModule`	Yes	Middleware
ISAPI `IsapiModule`	Yes	Middleware
Protocol Support `ProtocolSupportModule`	Yes	
Request Filtering `RequestFilteringModule`	Yes	URL Rewriting Middleware IRule
Request Monitor `RequestMonitorModule`	Yes	
URL Rewriting† `RewriteModule`	Yes	URL Rewriting Middleware
Server-Side Includes `ServerSideIncludeModule`	No	

Module	Functional with ASP.NET Core apps	ASP.NET Core Option
Static Compression `StaticCompressionModule`	No	Response Compression Middleware
Static Content `StaticFileModule`	No	Static File Middleware
Token Caching `TokenCacheModule`	Yes	
URI Caching `UriCacheModule`	Yes	
URL Authorization `UrlAuthorizationModule`	Yes	ASP.NET Core Identity
WebDav `WebDAV`	No	
Windows Authentication `WindowsAuthenticationModule`	Yes	

†The URL Rewrite Module's `isFile` and `isDirectory` match types don't work with ASP.NET Core apps due to the changes in directory structure.

Managed modules

Managed modules are *not* functional with hosted ASP.NET Core apps when the app pool's .NET CLR version is set to **No Managed Code**. ASP.NET Core offers middleware alternatives in several cases.

Expand table

Module	ASP.NET Core Option
AnonymousIdentification	
DefaultAuthentication	
FileAuthorization	
FormsAuthentication	Cookie Authentication Middleware
OutputCache	Response Caching Middleware
Profile	

Module	ASP.NET Core Option
RoleManager	
ScriptModule-4.0	
Session	Session Middleware
UrlAuthorization	
UrlMappingsModule	URL Rewriting Middleware
UrlRoutingModule-4.0	ASP.NET Core Identity
WindowsAuthentication	

IIS Manager application changes

When using IIS Manager to configure settings, the *web.config* file of the app is changed. If deploying an app and including *web.config*, any changes made with IIS Manager are overwritten by the deployed *web.config* file. If changes are made to the server's *web.config* file, copy the updated *web.config* file on the server to the local project immediately.

Disabling IIS modules

If an IIS module is configured at the server level that must be disabled for an app, an addition to the app's *web.config* file can disable the module. Either leave the module in place and deactivate it using a configuration setting (if available) or remove the module from the app.

Module deactivation

Many modules offer a configuration setting that allows them to be disabled without removing the module from the app. This is the simplest and quickest way to deactivate a module. For example, the HTTP Redirection Module can be disabled with the `<httpRedirect>` element in *web.config*:

XML

```
<configuration>
  <system.webServer>
    <httpRedirect enabled="false" />
```

```
  </system.webServer>
</configuration>
```

For more information on disabling modules with configuration settings, follow the links in the *Child Elements* section of IIS <system.webServer>.

Module removal

If opting to remove a module with a setting in *web.config*, unlock the module and unlock the `<modules>` section of *web.config* first:

1. Unlock the module at the server level. Select the IIS server in the IIS Manager **Connections** sidebar. Open the **Modules** in the **IIS** area. Select the module in the list. In the **Actions** sidebar on the right, select **Unlock**. If the action entry for the module appears as **Lock**, the module is already unlocked, and no action is required. Unlock as many modules as you plan to remove from *web.config* later.

2. Deploy the app without a `<modules>` section in *web.config*. If an app is deployed with a *web.config* containing the `<modules>` section without having unlocked the section first in the IIS Manager, the Configuration Manager throws an exception when attempting to unlock the section. Therefore, deploy the app without a `<modules>` section.

3. Unlock the `<modules>` section of *web.config*. In the **Connections** sidebar, select the website in **Sites**. In the **Management** area, open the **Configuration Editor**. Use the navigation controls to select the `system.webServer/modules` section. In the **Actions** sidebar on the right, select to **Unlock** the section. If the action entry for the module section appears as **Lock Section**, the module section is already unlocked, and no action is required.

4. Add a `<modules>` section to the app's local *web.config* file with a `<remove>` element to remove the module from the app. Add multiple `<remove>` elements to remove multiple modules. If *web.config* changes are made on the server, immediately make the same changes to the project's *web.config* file locally. Removing a module using this approach doesn't affect the use of the module with other apps on the server.

   XML

   ```
   <configuration>
    <system.webServer>
      <modules>
        <remove name="MODULE_NAME" />
      </modules>
   ```

```
  </system.webServer>
</configuration>
```

In order to add or remove modules for IIS Express using *web.config*, modify *applicationHost.config* to unlock the `<modules>` section:

1. Open *{APPLICATION ROOT}\.vs\config\applicationhost.config*.
2. Locate the `<section>` element for IIS modules and change `overrideModeDefault` from `Deny` to `Allow`:

   XML

   ```
   <section name="modules"
            allowDefinition="MachineToApplication"
            overrideModeDefault="Allow" />
   ```

3. Locate the `<location path="" overrideMode="Allow"><system.webServer><modules>` section. For any modules that you wish to remove, set `lockItem` from `true` to `false`. In the following example, the CGI Module is unlocked:

   XML

   ```
   <add name="CgiModule" lockItem="false" />
   ```

4. After the `<modules>` section and individual modules are unlocked, you're free to add or remove IIS modules using the app's *web.config* file for running the app on IIS Express.

An IIS module can also be removed with *Appcmd.exe*. Provide the `MODULE_NAME` and `APPLICATION_NAME` in the command:

Console

```
Appcmd.exe delete module MODULE_NAME /app.name:APPLICATION_NAME
```

For example, remove the `DynamicCompressionModule` from the Default Web Site:

Console

```
%windir%\system32\inetsrv\appcmd.exe delete module DynamicCompressionModule
/app.name:"Default Web Site"
```

Minimum module configuration

The only modules required to run an ASP.NET Core app are the Anonymous Authentication Module and the ASP.NET Core Module.

The URI Caching Module (`UriCacheModule`) allows IIS to cache website configuration at the URL level. Without this module, IIS must read and parse configuration on every request, even when the same URL is repeatedly requested. Parsing the configuration every request results in a significant performance penalty. *Although the URI Caching Module isn't strictly required for a hosted ASP.NET Core app to run, we recommend that the URI Caching Module be enabled for all ASP.NET Core deployments.*

The HTTP Caching Module (`HttpCacheModule`) implements the IIS output cache and also the logic for caching items in the HTTP.sys cache. Without this module, content is no longer cached in kernel mode, and cache profiles are ignored. Removing the HTTP Caching Module usually has adverse effects on performance and resource usage. *Although the HTTP Caching Module isn't strictly required for a hosted ASP.NET Core app to run, we recommend that the HTTP Caching Module be enabled for all ASP.NET Core deployments.*

Additional resources

- Introduction to IIS Architectures: Modules in IIS
- IIS Modules Overview
- Customizing IIS 7.0 Roles and Modules
- IIS <system.webServer>

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

IIS log creation and redirection

Article • 07/31/2024

The ASP.NET Core Module redirects stdout and stderr console output to disk if the `stdoutLogEnabled` and `stdoutLogFile` attributes of the `aspNetCore` element are set. Any folders in the `stdoutLogFile` path are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\{APP POOL NAME}` to provide write permission, where the placeholder `{APP POOL NAME}` is the app pool name).

Logs aren't rotated, unless process recycling/restart occurs. It's the responsibility of the hoster to limit the disk space the logs consume.

Using the stdout log is only recommended for troubleshooting app startup issues when hosting on IIS or when using development-time support for IIS with Visual Studio, not while debugging locally and running the app with IIS Express.

Don't use the stdout log for general app logging purposes. For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

A timestamp and file extension are added automatically when the log file is created. The log file name is composed by appending the timestamp, process ID, and file extension (`.log`) to the last segment of the `stdoutLogFile` path (typically `stdout`) delimited by underscores. If the `stdoutLogFile` path ends with `stdout`, a log for an app with a PID of 1934 created on 2/5/2018 at 19:42:32 has the file name `stdout_20180205194132_1934.log`.

If `stdoutLogEnabled` is false, errors that occur on app startup are captured and emitted to the event log up to 30 KB. After startup, all additional logs are discarded.

The following sample `aspNetCore` element configures stdout logging at the relative path `.\log\`. Confirm that the AppPool user identity has permission to write to the path provided.

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="true"
    stdoutLogFile=".\logs\stdout"
```

```
    hostingModel="inprocess">
</aspNetCore>
```

When publishing an app for Azure App Service deployment, the Web SDK sets the `stdoutLogFile` value to `\\?\%home%\LogFiles\stdout`. The `%home` environment variable is predefined for apps hosted by Azure App Service.

To create logging filter rules, see the Apply log filter rules in code section of the ASP.NET Core logging documentation.

For more information on path formats, see File path formats on Windows systems.

Enhanced diagnostic logs

The ASP.NET Core Module is configurable to provide enhanced diagnostics logs. Add the `<handlerSettings>` element to the `<aspNetCore>` element in `web.config`. Setting the `debugLevel` to `TRACE` exposes a higher fidelity of diagnostic information:

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="debugFile" value=".\logs\aspnetcore-debug.log" />
    <handlerSetting name="debugLevel" value="FILE,TRACE" />
  </handlerSettings>
</aspNetCore>
```

Any folders in the path (`logs` in the preceding example) are created by the module when the log file is created. The app pool must have write access to the location where the logs are written (use `IIS AppPool\{APP POOL NAME}` to provide write permission, where the placeholder `{APP POOL NAME}` is the app pool name).

Debug level (`debugLevel`) values can include both the level and the location.

Levels (in order from least to most verbose):

- ERROR
- WARNING
- INFO
- TRACE

Locations (multiple locations are permitted):

- CONSOLE
- EVENTLOG
- FILE

The handler settings can also be provided via environment variables:

- `ASPNETCORE_MODULE_DEBUG_FILE`: Path to the debug log file. (Default: `aspnetcore-debug.log`)
- `ASPNETCORE_MODULE_DEBUG`: Debug level setting.

> ⚠ **Warning**
>
> Do **not** leave debug logging enabled in the deployment for longer than required to troubleshoot an issue. The size of the log isn't limited. Leaving the debug log enabled can exhaust the available disk space and crash the server or app service.

See Configuration of ASP.NET Core Module with web.config for an example of the `aspNetCore` element in the `web.config` file.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Troubleshoot ASP.NET Core on Azure App Service and IIS

Article • 07/23/2024

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, the ASP.NET Core project default server is Kestrel. Visual studio can be configured to use IIS Express. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally with IIS Express can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:

    - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
    - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
    - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module (ANCM) for IIS.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

This error also may occur when the .NET Core Hosting Bundle isn't installed or is corrupted. Installing or repairing the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express) may fix the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
```

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not
found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error, either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file

deployment model. Use **one** of the following approaches in the app's project file to fix this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.

2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:

    - If deploying a 32-bit (x86) app, set the value to `True`.
    - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Failed to start application (ErrorCode '0x800701b1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x800701b1'.
```

The app failed to start because a Windows Service failed to load.

One common service that needs to be enabled is the "null" service. The following command enables the `null` Windows Service:

Windows Command Prompt

```
sc.exe start null
```

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

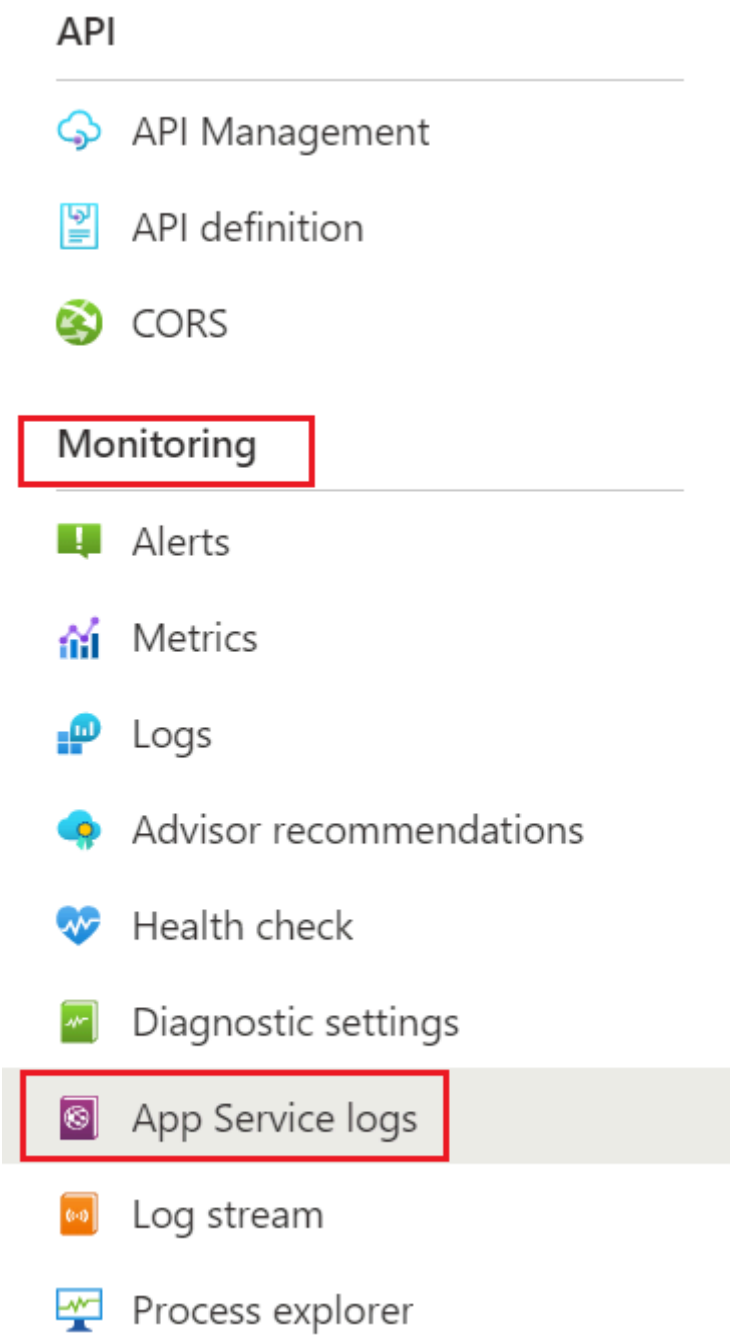
> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Azure App Services Log stream

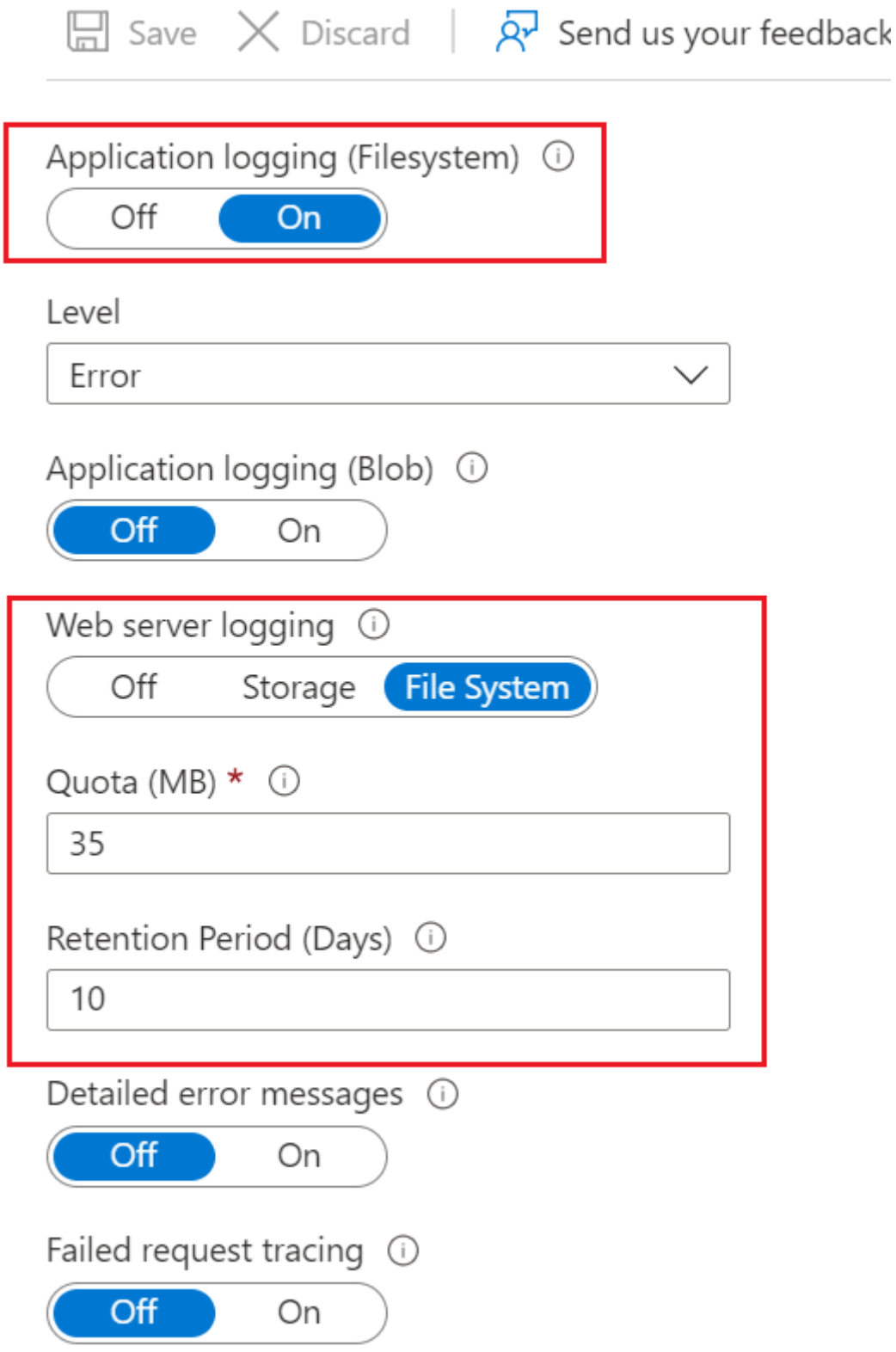
The Azure App Services Log streams logging information as it occurs. To view streaming logs:

1. In the Azure portal, open the app in **App Services**.

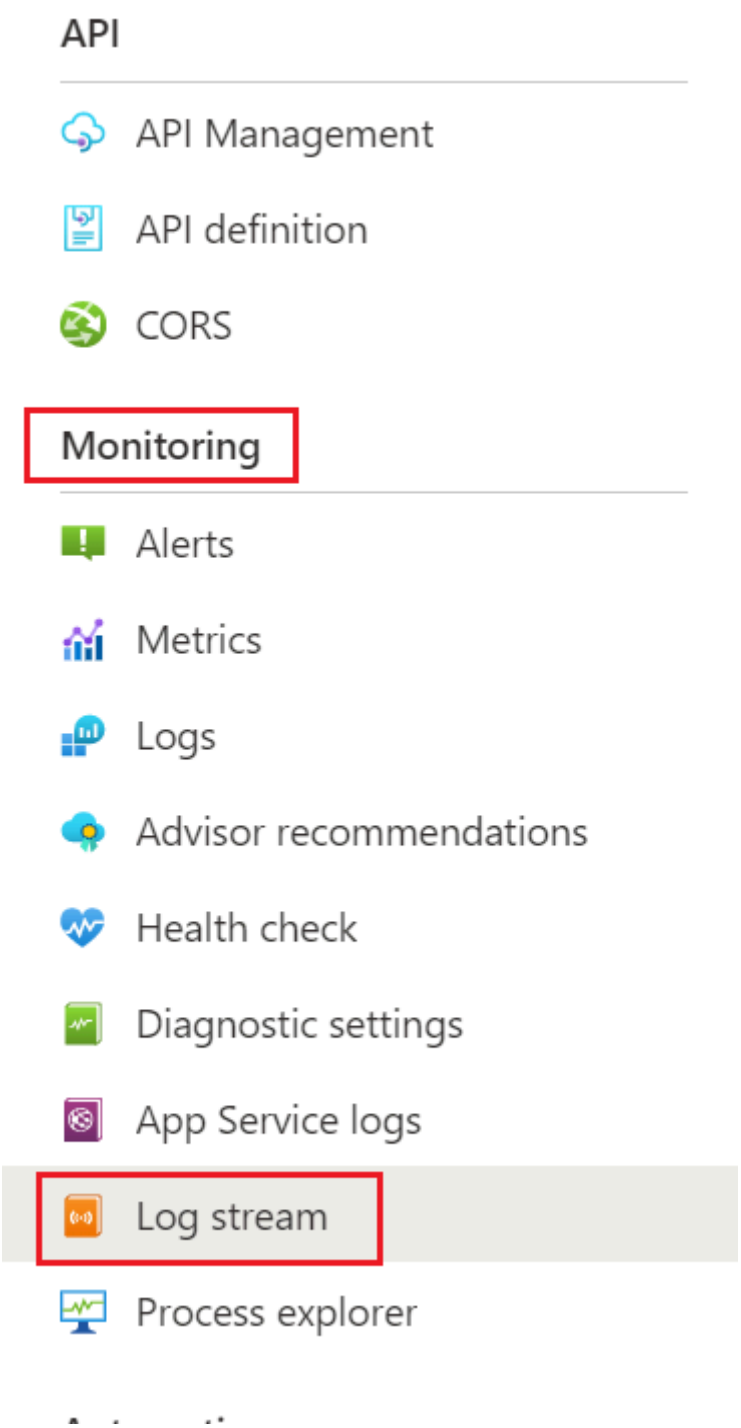
2. In the left pane, navigate to **Monitoring** > **App Service Logs**.



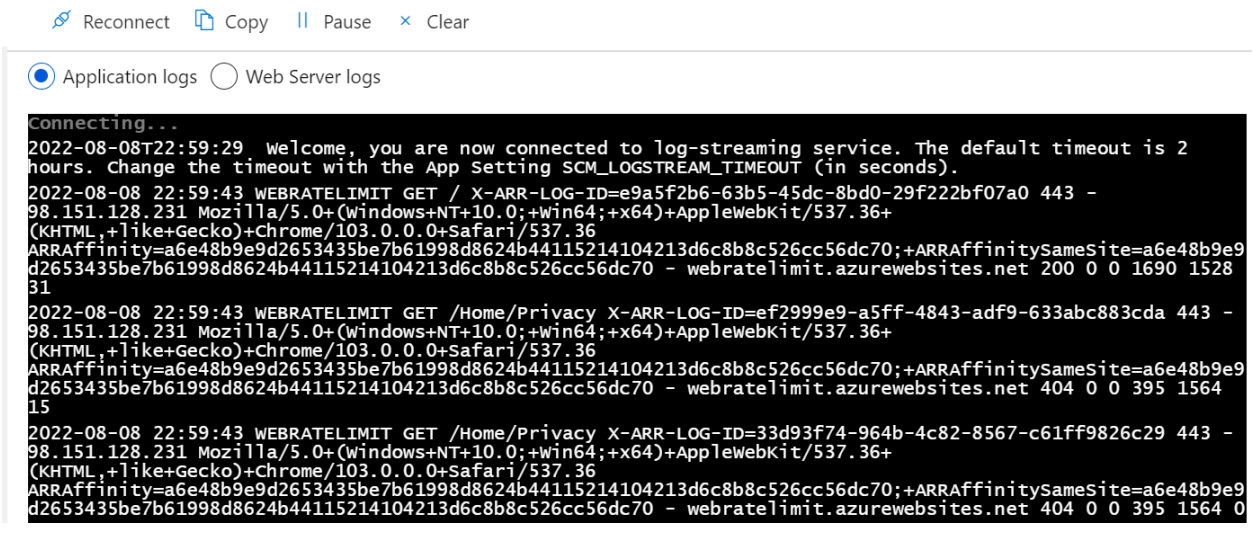
3. Select **File System** for **Web Server Logging**. Optionally enable **Application logging**.



4. In the left pane, navigate to **Monitoring** > **Log stream**, and then select **Application logs** or **Web Server Logs**.



The following images shows the application logs output:



Streaming logs have some latency and might not display immediately.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the `eventlog.xml` file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:

- If the app is a framework-dependent deployment:

  .NET CLI

  ```
  dotnet .\{ASSEMBLY NAME}.dll
  ```

- If the app is a self-contained deployment:

  Console

  ```
  {ASSEMBLY NAME}.exe
  ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:

  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

- If the app is a self-contained deployment:

  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.
4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module (ANCM) for IIS.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:

- Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
- Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
  a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
  b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
  c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see Log creation and redirection with the ASP.NET Core Module.

> ⚠ **Warning**
>
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or nonresponsive app (Azure App Service)

When an app responds slowly or doesn't respond to a request, see Troubleshoot slow web app performance issues in Azure App Service.

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.

5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`).

`stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.

4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module (ANCM) for IIS.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

XML

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see Log creation and redirection with the ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

XML

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
/>
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or non-responding app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.

2. Run the EnableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> ⚠ **Warning**
>
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App doesn't respond, fails during startup, or runs normally

When an app stops responding but doesn't crash, fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Debug .NET and ASP.NET Core source code with Visual Studio
- Troubleshoot and debug ASP.NET Core projects
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module (ANCM) for IIS

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common error troubleshooting for Azure App Service and IIS with ASP.NET Core

Article • 07/26/2024

This topic describes the most common errors and provides troubleshooting advice when hosting ASP.NET Core apps on Azure Apps Service and IIS.

See Troubleshoot ASP.NET Core on Azure App Service and IIS information on common app startup errors and instructions on how to diagnose errors.

Collect the following information:

- Browser behavior such as status code and error message.
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS

    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.

- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**

ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and

Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.

- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure

- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely

- **ASP.NET Core Module stdout Log:** The log file is created but empty.

- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED **--OR--** Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied **--OR--** 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. **--OR--** This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  XML

  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not

found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.

- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Downloads page and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure

- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module (ANCM) for IIS.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure **--OR--** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Advanced configuration of the ASP.NET Core Module and IIS

Article • 10/11/2023

This article covers advanced configuration options and scenarios for the ASP.NET Core Module and IIS.

Modify the stack size

Only applies when using the in-process hosting model.

Configure the managed stack size using the `stackSize` setting in bytes in the `web.config` file. The default size is 1,048,576 bytes (1 MB). The following example changes the stack size to 2 MB (2,097,152 bytes):

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="stackSize" value="2097152" />
  </handlerSettings>
</aspNetCore>
```

Disallow rotation on config

The `disallowRotationOnConfigChange` setting is intended for blue/green scenarios where a change to global config should not cause all sites to recycle. When this flag is true, only changes relevant to the site itself will cause it to recycle. For example, a site recycles if its *web.config* changes or something changes that is relevant to the site's path from IIS's perspective. But a general change to *applicationHost.config* would not cause an app to recycle. The following example sets this setting to true:

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
```

```
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="disallowRotationOnConfigChange" value="true" />
  </handlerSettings>
</aspNetCore>
```

This setting corresponds to the API ApplicationPoolRecycling.DisallowRotationOnConfigChange

Reduce 503 likelihood during app recycle

By default, there is a one second delay between when IIS is notified of a recycle or shutdown and when ANCM tells the managed server to initiate shutdown. The delay is configurable via the `ANCM_shutdownDelay` environment variable or by setting the `shutdownDelay` handler setting. Both values are in milliseconds. The delay is primarily to reduce the likelihood of a race where:

- IIS hasn't started queuing requests to go to the new app.
- ANCM starts rejecting new requests that come into the old app.

This setting doesn't mean incoming requests will be delayed by this amount. The setting indicates that the old app instance will start shutting down after the timeout occurs. Slower machines or machines with heavier CPU usage might need to adjust this value to reduce 503 likelihood.

The following example sets the delay to 5 seconds:

XML

```
<aspNetCore processPath="dotnet"
    arguments=".\MyApp.dll"
    stdoutLogEnabled="false"
    stdoutLogFile="\\?\%home%\LogFiles\stdout"
    hostingModel="inprocess">
  <handlerSettings>
    <handlerSetting name="shutdownDelay" value="5000" />
  </handlerSettings>
</aspNetCore>
```

Proxy configuration uses HTTP protocol and a pairing token

Only applies to out-of-process hosting.

The proxy created between the ASP.NET Core Module and Kestrel uses the HTTP protocol. There's no risk of eavesdropping the traffic between the module and Kestrel from a location off of the server.

A pairing token is used to guarantee that the requests received by Kestrel were proxied by IIS and didn't come from some other source. The pairing token is created and set into an environment variable (`ASPNETCORE_TOKEN`) by the module. The pairing token is also set into a header (`MS-ASPNETCORE-TOKEN`) on every proxied request. IIS Middleware checks each request it receives to confirm that the pairing token header value matches the environment variable value. If the token values are mismatched, the request is logged and rejected. The pairing token environment variable and the traffic between the module and Kestrel aren't accessible from a location off of the server. Without knowing the pairing token value, a cyberattacker can't submit requests that bypass the check in the IIS Middleware.

ASP.NET Core Module with an IIS Shared Configuration

The ASP.NET Core Module installer runs with the privileges of the `TrustedInstaller` account. Because the local system account doesn't have modify permission for the share path used by the IIS Shared Configuration, the installer throws an access denied error when attempting to configure the module settings in the `applicationHost.config` file on the share.

When using an IIS Shared Configuration on the same machine as the IIS installation, run the ASP.NET Core Hosting Bundle installer with the `OPT_NO_SHARED_CONFIG_CHECK` parameter set to `1`:

```
Console

dotnet-hosting-{VERSION}.exe OPT_NO_SHARED_CONFIG_CHECK=1
```

When the path to the shared configuration isn't on the same machine as the IIS installation, follow these steps:

1. Disable the IIS Shared Configuration.
2. Run the installer.
3. Export the updated `applicationHost.config` file to the file share.
4. Re-enable the IIS Shared Configuration.

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including middleware used in authentication. Even if Data Protection APIs aren't called by user code, data protection should be configured with a deployment script or in user code to create a persistent cryptographic key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the Data Protection key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection under IIS to persist the key ring, use **one** of the following approaches:

- **Create Data Protection Registry keys**

  Data Protection keys used by ASP.NET Core apps are stored in the registry external to the apps. To persist the keys for a given app, create Registry keys for the app pool.

  For standalone, non-webfarm IIS installations, the Data Protection Provision-AutoGenKeys.ps1 PowerShell script can be used for each app pool used with an ASP.NET Core app. This script creates a Registry key in the HKLM registry that's accessible only to the worker process account of the app's app pool. Keys are encrypted at rest using DPAPI with a machine-wide key.

  In web farm scenarios, an app can be configured to use a UNC path to store its Data Protection key ring. By default, the keys aren't encrypted. Ensure that the file permissions for the network share are limited to the Windows account that the app runs under. An X509 certificate can be used to protect keys at rest. Consider a mechanism to allow users to upload certificates. Place certificates into the user's trusted certificate store and ensure they're available on all machines where the user's app runs. For more information, see Configure ASP.NET Core Data Protection.

- **Configure the IIS Application Pool to load the user profile**

  This setting is in the **Process Model** section under the **Advanced Settings** for the app pool. Set **Load User Profile** to `True`. When set to `True`, keys are stored in the user profile directory and protected using DPAPI with a key specific to the user

account. Keys are persisted to the `%LOCALAPPDATA%/ASP.NET/DataProtection-Keys` folder.

The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:

1. Navigate to the `%windir%/system32/inetsrv/config` folder.
2. Open the `applicationHost.config` file.
3. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.
4. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

- **Use the file system as a key ring store**

  Adjust the app code to use the file system as a key ring store. Use an X509 certificate to protect the key ring and ensure the certificate is a trusted certificate. If the certificate is self-signed, place the certificate in the Trusted Root store.

  When using IIS in a web farm:
  - Use a file share that all machines can access.
  - Deploy an X509 certificate to each machine. Configure Data Protection in code.

- **Set a machine-wide policy for Data Protection**

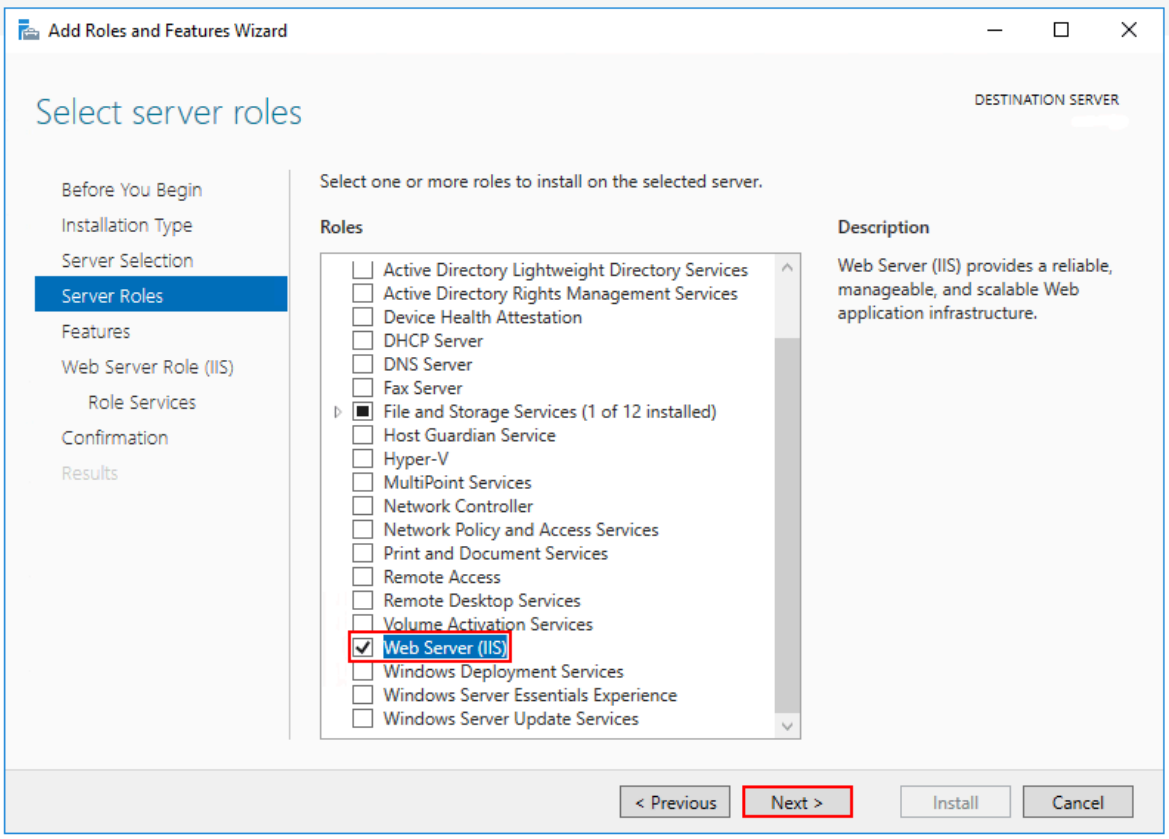
  The Data Protection system has limited support for setting a default machine-wide policy for all apps that consume the Data Protection APIs. For more information, see ASP.NET Core Data Protection Overview.

IIS configuration

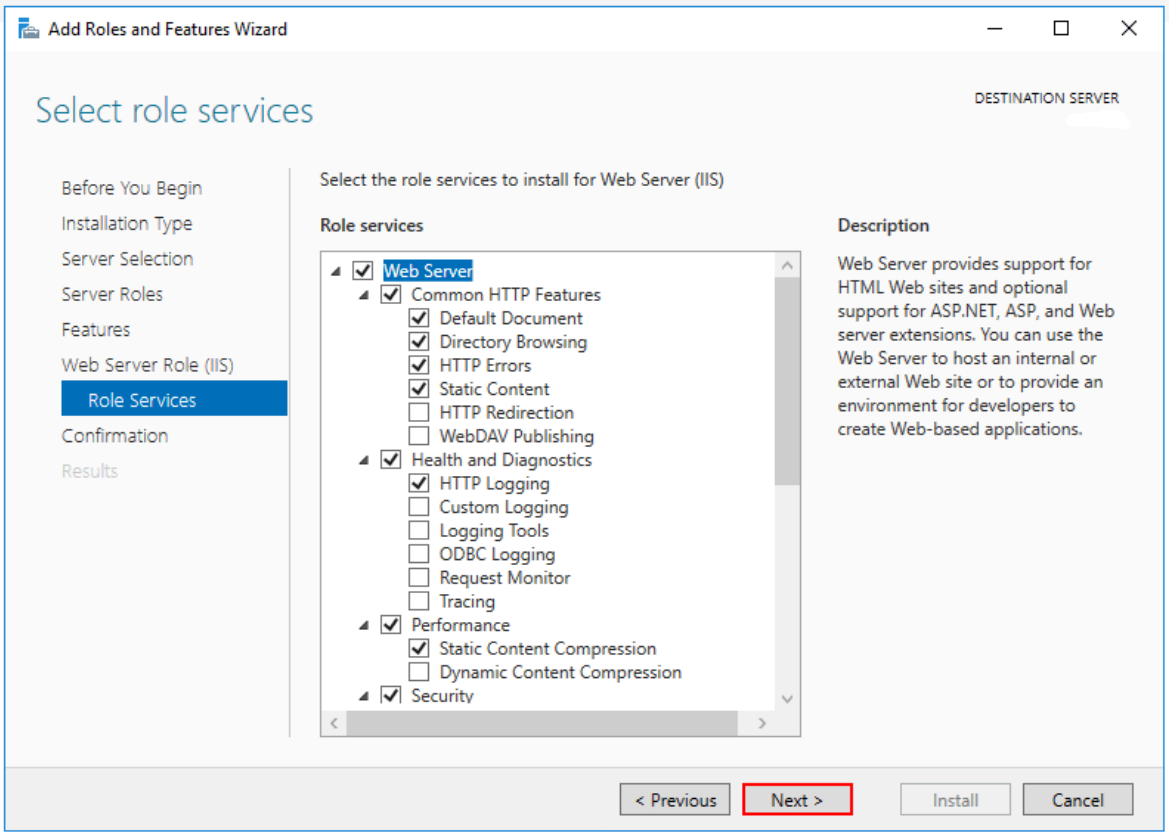
Windows Server operating systems

Enable the **Web Server (IIS)** server role and establish role services.

1. Use the **Add Roles and Features** wizard from the **Manage** menu or the link in **Server Manager**. On the **Server Roles** step, check the box for **Web Server (IIS)**.



2. After the **Features** step, the **Role services** step loads for Web Server (IIS). Select the IIS role services desired or accept the default role services provided.



Windows Authentication (Optional)

To enable Windows Authentication, expand the following nodes: **Web Server** > **Security**. Select the **Windows Authentication** feature. For more information, see

Windows Authentication <windowsAuthentication> and Configure Windows authentication.

WebSockets (Optional)
WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **Web Server** > **Application Development**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

3. Proceed through the **Confirmation** step to install the web server role and services. A server/IIS restart isn't required after installing the **Web Server (IIS)** role.

Windows desktop operating systems

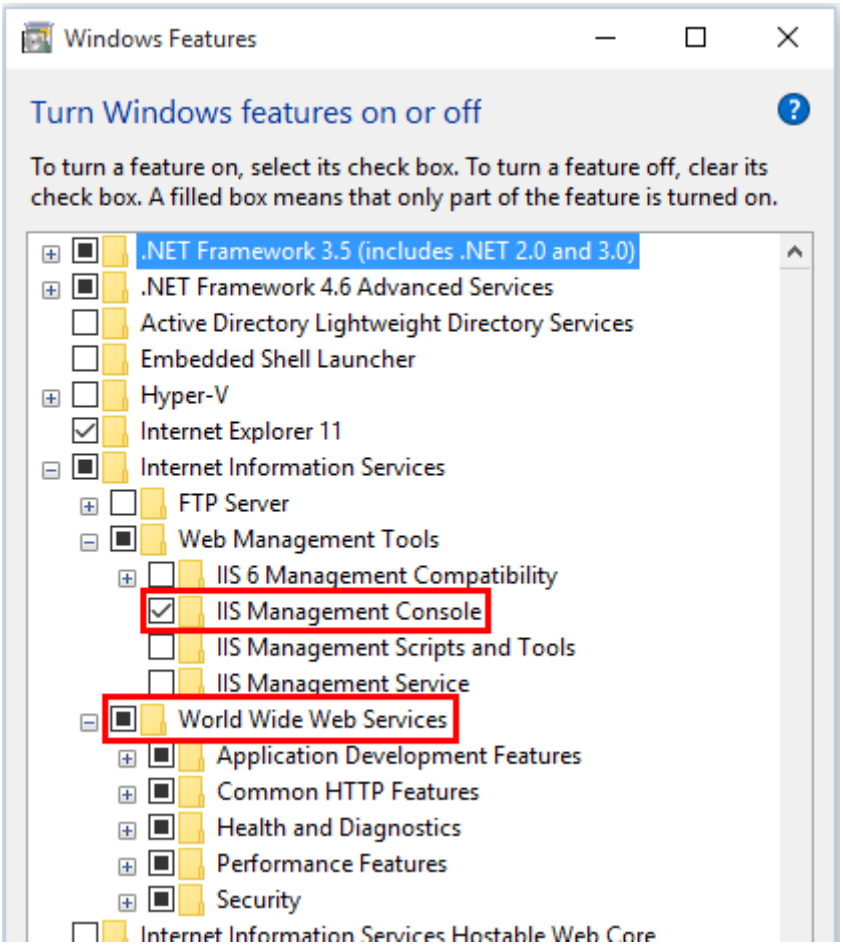
Enable the **IIS Management Console** and **World Wide Web Services**.

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open the **Internet Information Services** node. Open the **Web Management Tools** node.
3. Check the box for **IIS Management Console**.
4. Check the box for **World Wide Web Services**.
5. Accept the default features for **World Wide Web Services** or customize the IIS features.

   Windows Authentication (Optional)
   To enable Windows Authentication, expand the following nodes: **World Wide Web Services** > **Security**. Select the **Windows Authentication** feature. For more information, see Windows Authentication <windowsAuthentication> and Configure Windows authentication.

   WebSockets (Optional)
   WebSockets is supported with ASP.NET Core 1.1 or later. To enable WebSockets, expand the following nodes: **World Wide Web Services** > **Application Development Features**. Select the **WebSocket Protocol** feature. For more information, see WebSockets.

6. If the IIS installation requires a restart, restart the system.



Virtual Directories

IIS Virtual Directories aren't supported with ASP.NET Core apps. An app can be hosted as a sub-application.

Sub-applications

An ASP.NET Core app can be hosted as an IIS sub-application (sub-app). The sub-app's path becomes part of the root app's URL.

Static asset links within the sub-app should use tilde-slash (`~/`) notation in MVC and Razor Pages. Tilde-slash notation triggers a Tag Helper to prepend the sub-app's pathbase to the rendered relative link. For a sub-app at `/subapp_path`, an image linked with `src="~/image.png"` is rendered as `src="/subapp_path/image.png"`. The root app's Static File Middleware doesn't process the static file request. The request is processed by the sub-app's Static File Middleware.

> ⓘ Note

Razor components (`.razor`) shouldn't use tilde-slash notation. For more information, see the **Blazor app base path documentation**.

If a static asset's `src` attribute is set to an absolute path (for example, `src="/image.png"`), the link is rendered without the sub-app's pathbase. The root app's Static File Middleware attempts to serve the asset from the root app's web root, which results in a *404 - Not Found* response unless the static asset is available from the root app.

To host an ASP.NET Core app as a sub-app under another ASP.NET Core app:

1. Establish an app pool for the sub-app. Set the **.NET CLR Version** to **No Managed Code** because the Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process, not the desktop CLR (.NET CLR).
2. Add the root site in IIS Manager with the sub-app in a folder under the root site.
3. Right-click the sub-app folder in IIS Manager and select **Convert to Application**.
4. In the **Add Application** dialog, use the **Select** button for the **Application Pool** to assign the app pool that you created for the sub-app. Select **OK**.

The assignment of a separate app pool to the sub-app is a requirement when using the in-process hosting model.

For more information on the in-process hosting model and configuring the ASP.NET Core Module, see ASP.NET Core Module (ANCM) for IIS.

Application Pools

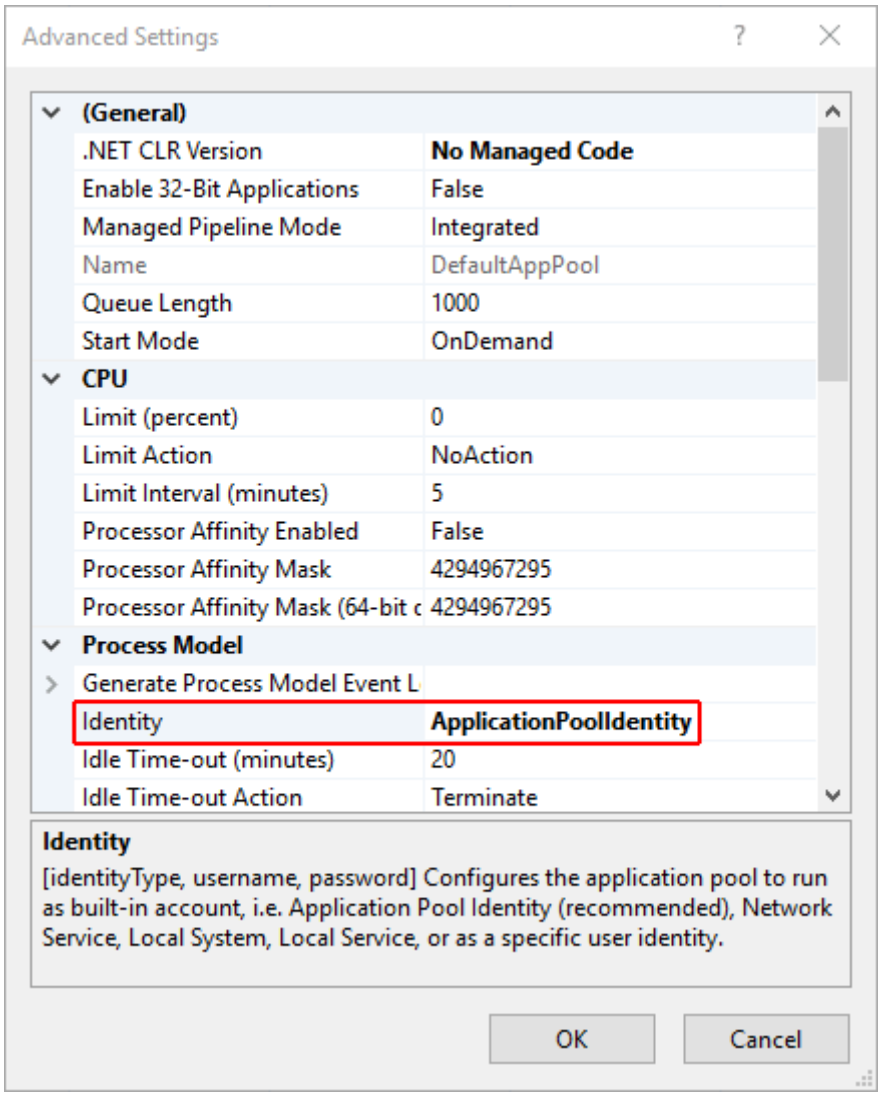
App pool isolation is determined by the hosting model:

- In-process hosting: Apps are required to run in separate app pools.
- Out-of-process hosting: We recommend isolating the apps from each other by running each app in its own app pool.

The IIS **Add Website** dialog defaults to a single app pool per app. When a **Site name** is provided, the text is automatically transferred to the **Application pool** textbox. A new app pool is created using the site name when the site is added.

Application Pool Identity

An app pool identity account allows an app to run under a unique account without having to create and manage domains or local accounts. On IIS 8.0 or later, the IIS Admin Worker Process (WAS) creates a virtual account with the name of the new app pool and runs the app pool's worker processes under this account by default. In the IIS Management Console under **Advanced Settings** for the app pool, ensure that the **Identity** is set to use `ApplicationPoolIdentity`:



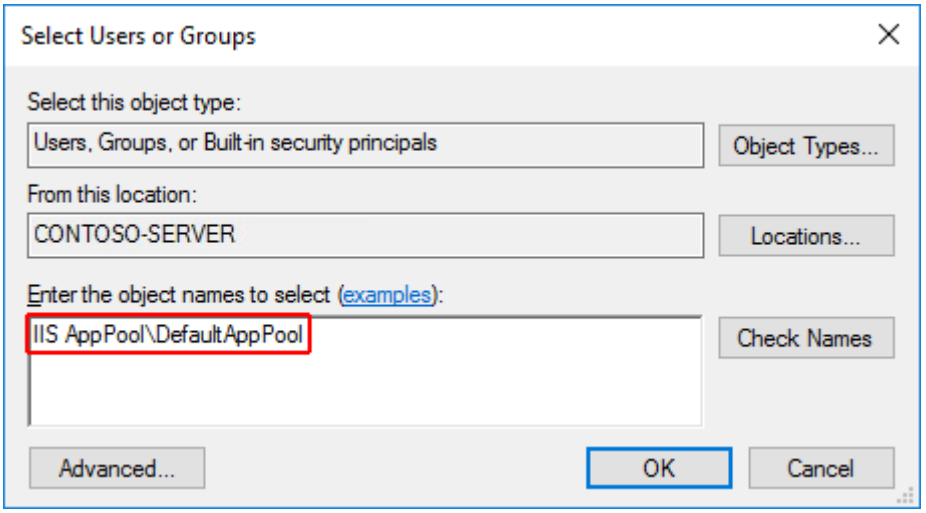
The IIS management process creates a secure identifier with the name of the app pool in the Windows Security System. Resources can be secured using this identity. However, this identity isn't a real user account and doesn't show up in the Windows User Management Console.

If the IIS worker process requires elevated access to the app, modify the Access Control List (ACL) for the directory containing the app:

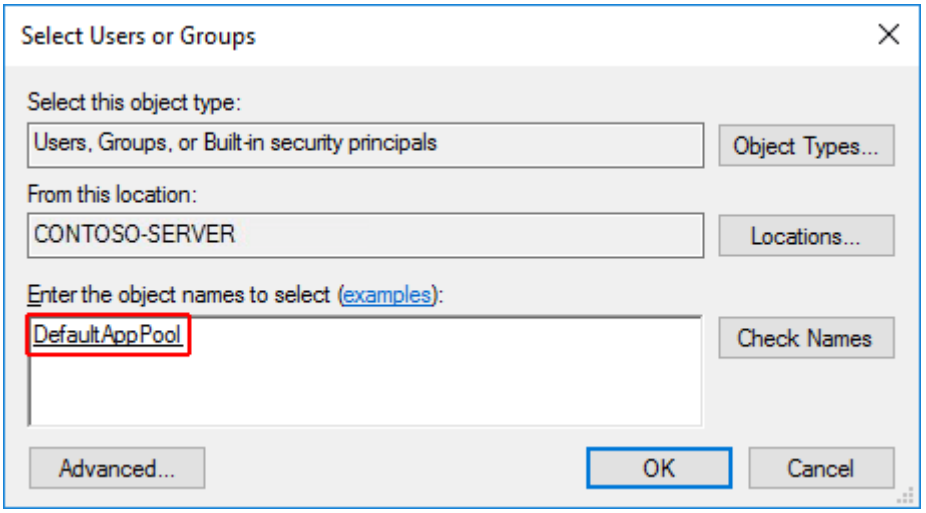
1. Open Windows Explorer and navigate to the directory.
2. Right-click on the directory and select **Properties**.
3. Under the **Security** tab, select the **Edit** button and then the **Add** button.

4. Select the **Locations** button and make sure the system is selected.

5. Enter `IIS AppPool\{APP POOL NAME}` format, where the placeholder `{APP POOL NAME}` is the app pool name, in **Enter the object names to select** area. Select the **Check Names** button. For the *DefaultAppPool* check the names using `IIS AppPool\DefaultAppPool`. When the **Check Names** button is selected, a value of `DefaultAppPool` is indicated in the object names area. It isn't possible to enter the app pool name directly into the object names area. Use the `IIS AppPool\{APP POOL NAME}` format, where the placeholder `{APP POOL NAME}` is the app pool name, when checking for the object name.



6. Select **OK**.



7. Read & execute permissions should be granted by default. Provide additional permissions as needed.

Access can also be granted at a command prompt using the **ICACLS** tool. Using the *DefaultAppPool* as an example, the following command is used to grant read and execute permissions to the `MyWebApp` folder, subfolders, and files:

```
```

Console

```
ICACLS C:\sites\MyWebApp /grant "IIS AppPool\DefaultAppPool:(OI)(CI)RX"
```

For more information, see the icacls topic.

HTTP/2 support

HTTP/2 is supported with ASP.NET Core in the following IIS deployment scenarios:

- In-process
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - TLS 1.2 or later connection
- Out-of-process
  - Windows Server 2016/Windows 10 or later; IIS 10 or later
  - Public-facing edge server connections use HTTP/2, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.
  - TLS 1.2 or later connection

For an in-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`. For an out-of-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/1.1`.

For more information on the in-process and out-of-process hosting models, see ASP.NET Core Module (ANCM) for IIS.

HTTP/2 is enabled by default. Connections fall back to HTTP/1.1 if an HTTP/2 connection isn't established. For more information on HTTP/2 configuration with IIS deployments, see HTTP/2 on IIS.

CORS preflight requests

This section only applies to ASP.NET Core apps that target the .NET Framework.

For an ASP.NET Core app that targets the .NET Framework, OPTIONS requests aren't passed to the app by default in IIS. To learn how to configure the app's IIS handlers in `web.config` to pass OPTIONS requests, see Enable cross-origin requests in ASP.NET Web API 2: How CORS Works.

Application Initialization Module and Idle Timeout

When hosted in IIS by the ASP.NET Core Module version 2:

- Application Initialization Module: App's hosted in-process or out-of-process can be configured to start automatically on a worker process restart or server restart.
- Idle Timeout: App's hosted in-process can be configured not to time out during periods of inactivity.

Application Initialization Module

Applies to apps hosted in-process and out-of-process.

IIS Application Initialization is an IIS feature that sends an HTTP request to the app when the app pool starts or is recycled. The request triggers the app to start. By default, IIS issues a request to the app's root URL (`/`) to initialize the app (see the additional resources for more details on configuration).

Confirm that the IIS Application Initialization role feature in enabled:

On Windows 7 or later desktop systems when using IIS locally:

1. Navigate to **Control Panel** > **Programs** > **Programs and Features** > **Turn Windows features on or off** (left side of the screen).
2. Open **Internet Information Services** > **World Wide Web Services** > **Application Development Features**.
3. Select the checkbox for **Application Initialization**.

On Windows Server 2008 R2 or later:

1. Open the **Add Roles and Features Wizard**.
2. In the **Select role services** panel, open the **Application Development** node.
3. Select the checkbox for **Application Initialization**.

Use either of the following approaches to enable the Application Initialization Module for the site:

- Using IIS Manager:

    1. Select **Application Pools** in the **Connections** panel.
    2. Right-click the app's app pool in the list and select **Advanced Settings**.

3. The default **Start Mode** is `OnDemand`. Set the **Start Mode** to `AlwaysRunning`. Select **OK**.
4. Open the **Sites** node in the **Connections** panel.
5. Right-click the app and select **Manage Website** > **Advanced Settings**.
6. The default **Preload Enabled** setting is `False`. Set **Preload Enabled** to `True`. Select **OK**.

- Using `web.config`, add the `<applicationInitialization>` element with `doAppInitAfterRestart` set to `true` to the `<system.webServer>` elements in the app's `web.config` file:

```
XML
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <location path="." inheritInChildApplications="false">
    <system.webServer>
      <applicationInitialization doAppInitAfterRestart="true" />
    </system.webServer>
  </location>
</configuration>
```

Idle Timeout

Only applies to apps hosted in-process.

To prevent the app from idling, set the app pool's idle timeout using IIS Manager:

1. Select **Application Pools** in the **Connections** panel.
2. Right-click the app's app pool in the list and select **Advanced Settings**.
3. The default **Idle Time-out (minutes)** is `20` minutes. Set the **Idle Time-out (minutes)** to `0` (zero). Select **OK**.
4. Recycle the worker process.

To prevent apps hosted out-of-process from timing out, use either of the following approaches:

- Ping the app from an external service in order to keep it running.
- If the app only hosts background services, avoid IIS hosting and use a Windows Service to host the ASP.NET Core app.

Application Initialization Module and Idle Timeout additional resources

- IIS 8.0 Application Initialization
- Application Initialization <applicationInitialization>.
- Process Model Settings for an Application Pool <processModel>.

Module, schema, and configuration file locations

Module

IIS (x86/amd64):

- `%windir%\System32\inetsrv\aspnetcore.dll`
- `%windir%\SysWOW64\inetsrv\aspnetcore.dll`
- `%ProgramFiles%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll`
- `%ProgramFiles(x86)%\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll`

IIS Express (x86/amd64):

- `%ProgramFiles%\IIS Express\aspnetcore.dll`
- `%ProgramFiles(x86)%\IIS Express\aspnetcore.dll`
- `%ProgramFiles%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll`
- `%ProgramFiles(x86)%\IIS Express\Asp.Net Core Module\V2\aspnetcorev2.dll`

Schema

IIS

- `%windir%\System32\inetsrv\config\schema\aspnetcore_schema.xml`
- `%windir%\System32\inetsrv\config\schema\aspnetcore_schema_v2.xml`

IIS Express

- `%ProgramFiles%\IIS Express\config\schema\aspnetcore_schema.xml`
- `%ProgramFiles%\IIS Express\config\schema\aspnetcore_schema_v2.xml`

Configuration

IIS

- `%windir%\System32\inetsrv\config\applicationHost.config`

IIS Express

- Visual Studio: `{APPLICATION ROOT}\.vs\config\applicationHost.config`
- *iisexpress.exe* CLI: `%USERPROFILE%\Documents\IISExpress\config\applicationhost.config`

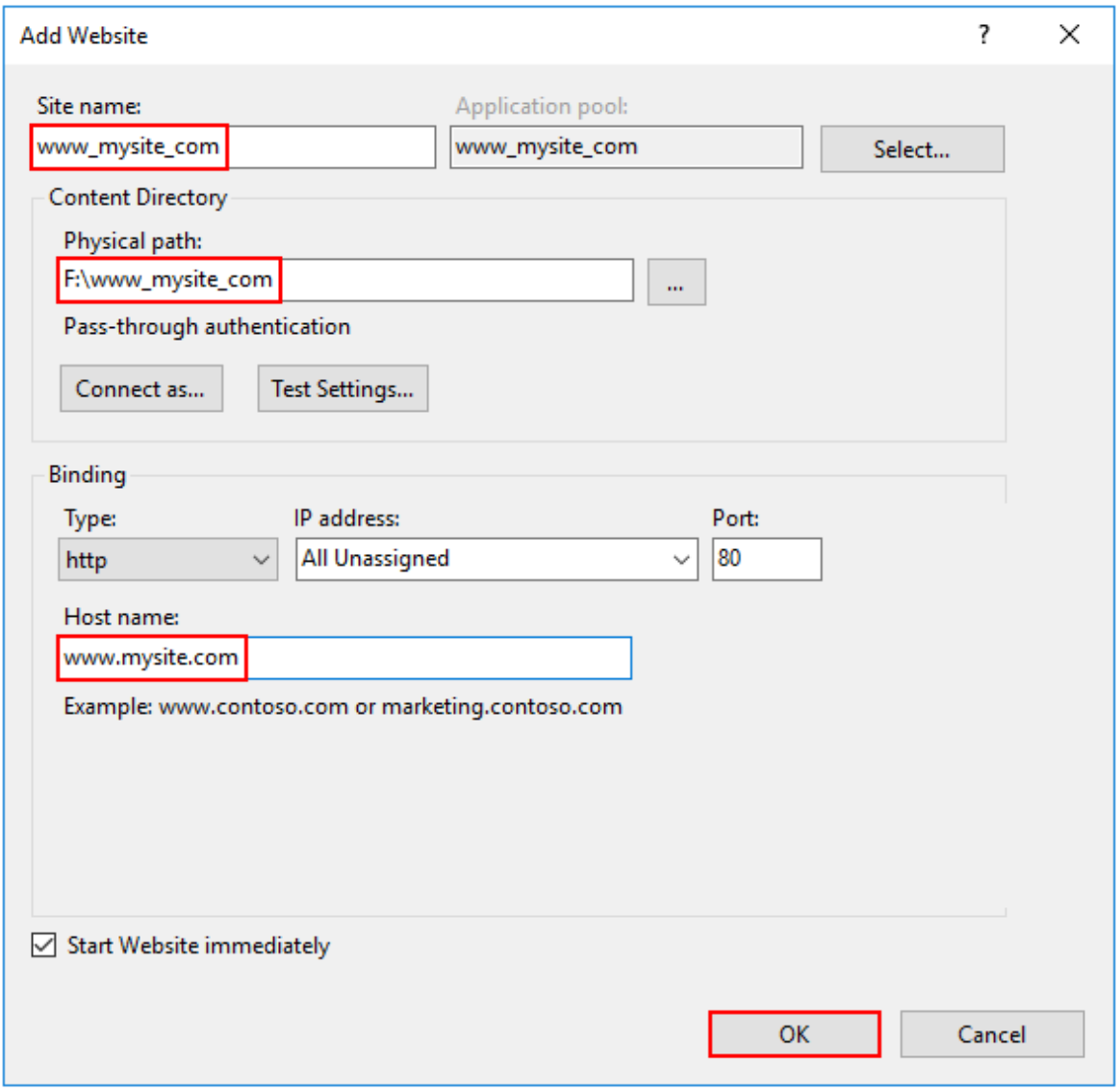
The files can be found by searching for `aspnetcore` in the `applicationHost.config` file.

Install Web Deploy when publishing with Visual Studio

When deploying apps to servers with Web Deploy, install the latest version of Web Deploy on the server. To install Web Deploy, see IIS Downloads: Web Deploy.

Create the IIS site

1. On the hosting system, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see ASP.NET Core directory structure.
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.
3. Provide a **Site name** and set the **Physical path** to the app's deployment folder. Provide the **Binding** configuration and create the website by selecting **OK**:

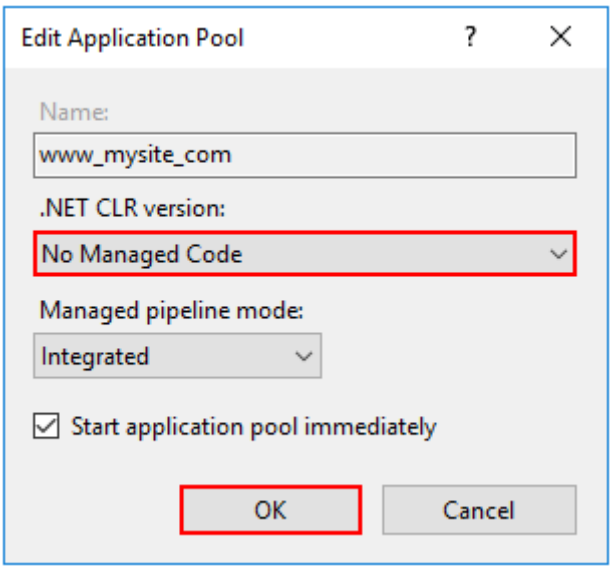


> ⚠ **Warning**
>
> Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See RFC 9110: HTTP Semantics (Section 7.2: Host and :authority) for more information.

4. Under the server's node, select **Application Pools**.

5. Right-click the site's app pool and select **Basic Settings** from the contextual menu.

6. In the **Edit Application Pool** window, set the **.NET CLR version** to **No Managed Code**:



ASP.NET Core runs in a separate process and manages the runtime. ASP.NET Core doesn't rely on loading the desktop CLR (.NET CLR). The Core Common Language Runtime (CoreCLR) for .NET Core is booted to host the app in the worker process. Setting the **.NET CLR version** to **No Managed Code** is optional but recommended.

- For a 32-bit (x86) self-contained deployment published with a 32-bit SDK that uses the in-process hosting model, enable the Application Pool for 32-bit. In IIS Manager, navigate to **Application Pools** in the **Connections** sidebar. Select the app's Application Pool. In the **Actions** sidebar, select **Advanced Settings**. Set **Enable 32-Bit Applications** to `True`.

- For a 64-bit (x64) self-contained deployment that uses the in-process hosting model, disable the app pool for 32-bit (x86) processes. In IIS Manager, navigate to **Application Pools** in the **Connections** sidebar. Select the app's Application Pool. In the **Actions** sidebar, select **Advanced Settings**. Set **Enable 32-Bit Applications** to `False`.

7. Confirm the process model identity has the proper permissions.

   If the default identity of the app pool (**Process Model** > **Identity**) is changed from **ApplicationPoolIdentity** to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

Windows Authentication configuration (Optional)
For more information, see Configure Windows authentication.

Shadow copy

Shadow copying app assemblies to the ASP.NET Core Module (ANCM) for IIS can provide a better end user experience than stopping the app by deploying an app offline file.

When an ASP.NET Core app is running on Windows, the binaries are locked so that they can't be modified or replaced. Shadow copying enables the app assemblies to be updated while the app is running by making a copy of the assemblies.

Shadow copy isn't intended to enable zero-downtime deployment, so its expected that IIS will still recycle the app, and some requests may get an 503 Service Unavailable response. We recommend using a pattern like blue-green deployments or Azure deployment slots for zero-downtime deployments. Shadow copy helps minimize downtime on deployments, but can't completely eliminate it.

Shadow copying is enabled by customizing the ANCM handler settings in `web.config`:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <remove name="aspNetCore"/>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2"
resourceType="Unspecified"/>
    </handlers>
    <aspNetCore processPath="%LAUNCHER_PATH%" arguments="%LAUNCHER_ARGS%"
stdoutLogEnabled="false" stdoutLogFile=".logsstdout">
      <handlerSettings>
        <handlerSetting name="enableShadowCopy" value="true" />
        <!-- Ensure that the IIS ApplicationPool identity has permission to
this directory -->
        <handlerSetting name="shadowCopyDirectory"
value="../ShadowCopyDirectory/" />
      </handlerSettings>
    </aspNetCore>
  </system.webServer>
</configuration>
```

Transform web.config

Article • 07/31/2024

By Vijay Ramakrishnan

Transformations to the *web.config* file can be applied automatically when an app is published based on:

- Build configuration
- Profile
- Environment
- Custom

These transformations occur for either of the following *web.config* generation scenarios:

- Generated automatically by the `Microsoft.NET.Sdk.Web` SDK.
- Provided by the developer in the content root of the app.

Build configuration

Build configuration transforms are run first.

Include a *web.{CONFIGURATION}.config* file for each build configuration (Debug|Release) requiring a *web.config* transformation.

In the following example, a configuration-specific environment variable is set in *web.Release.config*:

XML

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-
Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Configuration_Specific"
                               value="Configuration_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
```

```
  </location>
</configuration>
```

The transform is applied when the configuration is set to *Release*:

.NET CLI

```
dotnet publish --configuration Release
```

The MSBuild property for the configuration is `$(Configuration)`.

Profile

Profile transformations are run second, after Build configuration transforms.

Include a *web.{PROFILE}.config* file for each profile configuration requiring a *web.config* transformation.

In the following example, a profile-specific environment variable is set in *web.FolderProfile.config* for a folder publish profile:

XML

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-
Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Profile_Specific"
                               value="Profile_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the profile is *FolderProfile*:

.NET CLI

```
dotnet publish --configuration Release /p:PublishProfile=FolderProfile
```

The MSBuild property for the profile name is `$(PublishProfile)`.

If no profile is passed, the default profile name is **FileSystem** and *web.FileSystem.config* is applied if the file is present in the app's content root.

Environment

Environment transformations are run third, after Build configuration and Profile transforms.

Include a *web.{ENVIRONMENT}.config* file for each environment requiring a *web.config* transformation.

In the following example, an environment-specific environment variable is set in *web.Production.config* for the Production environment:

XML

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-
Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Environment_Specific"
                               value="Environment_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the environment is *Production*:

.NET CLI

```
dotnet publish --configuration Release /p:EnvironmentName=Production
```

The MSBuild property for the environment is `$(EnvironmentName)`.

When publishing from Visual Studio and using a publish profile, see Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment.

The `ASPNETCORE_ENVIRONMENT` environment variable is automatically added to the *web.config* file when the environment name is specified.

Custom

Custom transformations are run last, after Build configuration, Profile, and Environment transforms.

Include a *{CUSTOM_NAME}.transform* file for each custom configuration requiring a *web.config* transformation.

In the following example, a custom transform environment variable is set in *custom.transform*:

XML

```
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-
Transform">
  <location>
    <system.webServer>
      <aspNetCore>
        <environmentVariables xdt:Transform="InsertIfMissing">
          <environmentVariable name="Custom_Specific"
                               value="Custom_Specific_Value"
                               xdt:Locator="Match(name)"
                               xdt:Transform="InsertIfMissing" />
        </environmentVariables>
      </aspNetCore>
    </system.webServer>
  </location>
</configuration>
```

The transform is applied when the `CustomTransformFileName` property is passed to the dotnet publish command:

.NET CLI

```
dotnet publish --configuration Release
/p:CustomTransformFileName=custom.transform
```

The MSBuild property for the profile name is `$(CustomTransformFileName)`.

Prevent web.config transformation

To prevent transformations of the *web.config* file, set the MSBuild property `$(IsWebConfigTransformDisabled)`:

.NET CLI

```
dotnet publish /p:IsWebConfigTransformDisabled=true
```

Additional resources

- Web.config Transformation Syntax for Web Application Project Deployment
- Web.config Transformation Syntax for Web Project Deployment Using Visual Studio

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use ASP.NET Core with HTTP/2 on IIS

Article • 07/31/2024

By Justin Kotalik

HTTP/2 is supported with ASP.NET Core in the following IIS deployment scenarios:

- Windows Server 2016 or later / Windows 10 or later
- IIS 10 or later
- TLS 1.2 or later connection
- When hosting out-of-process: Public-facing edge server connections use HTTP/2, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.

For an in-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`. For an out-of-process deployment when an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/1.1`.

For more information on the in-process and out-of-process hosting models, see ASP.NET Core Module (ANCM) for IIS.

HTTP/2 is enabled by default for HTTPS/TLS connections. Connections fall back to HTTP/1.1 if an HTTP/2 connection isn't established. For more information on HTTP/2 configuration with IIS deployments, see HTTP/2 on IIS.

Advanced HTTP/2 features to support gRPC

Additional HTTP/2 features in IIS support gRPC, including support for response trailers and sending reset frames.

Requirements to run gRPC on IIS:

- In-process hosting.
- Windows 11 Build 22000 or later, Windows Server 2022 Build 20348 or later.
- TLS 1.2 or later connection.

Trailers

HTTP Trailers are similar to HTTP Headers, except they are sent after the response body is sent. For IIS and HTTP.sys, only HTTP/2 response trailers are supported.

```
C#
```

```
if (httpContext.Response.SupportsTrailers())
{
    httpContext.Response.DeclareTrailer("trailername");

    // Write body
    httpContext.Response.WriteAsync("Hello world");

    httpContext.Response.AppendTrailer("trailername", "TrailerValue");
}
```

In the preceding example code:

- `SupportsTrailers` ensures that trailers are supported for the response.
- `DeclareTrailer` adds the given trailer name to the `Trailer` response header. Declaring a response's trailers is optional, but recommended. If `DeclareTrailer` is called, it must be before the response headers are sent.
- `AppendTrailer` appends the trailer.

Reset

Reset allows for the server to reset a HTTP/2 request with a specified error code. A reset request is considered aborted.

C#

```
var resetFeature = httpContext.Features.Get<IHttpResetFeature>();
resetFeature.Reset(errorCode: 2);
```

`Reset` in the preceding code example specifies the `INTERNAL_ERROR` error code. For more information about HTTP/2 error codes, visit the HTTP/2 specification error code section.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Use ASP.NET Core with HTTP/3 on IIS

Article • 07/31/2024

By Chris Ross

HTTP/3 is fully supported with ASP.NET Core in the following IIS scenarios:

- In-process
- Out-of-Process. In Out-of-Process, IIS responds to the client using HTTP/3, but the reverse proxy connection to the Kestrel server uses HTTP/1.1.

For more information on the in-process and out-of-process hosting models, see ASP.NET Core Module (ANCM) for IIS.

The following requirements also need to be met:

- Windows Server 2022 / Windows 11 or later
- An `https` url binding is used.
- The EnableHttp3 registry key is set.

For an in-process deployment when an HTTP/3 connection is established, HttpRequest.Protocol reports `HTTP/3`. For an out-of-process deployment when an HTTP/3 connection is established, HttpRequest.Protocol reports `HTTP/1.1` because that is how IIS proxies the requests to Kestrel.

Alt-Svc

HTTP/3 is discovered as an upgrade from HTTP/1.1 or HTTP/2 via the alt-svc header. That means the first request will normally use HTTP/1.1 or HTTP/2 before switching to HTTP/3. IIS doesn't automatically add the `alt-svc` header, it must be added by the application. The following code is a middleware example that adds the `alt-svc` response header.

C#

```
app.Use((context, next) =>
{
    context.Response.Headers.AltSvc = "h3=\":443\"";
    return next(context);
});
```

Place the preceding code early in the request pipeline.

IIS also supports sending an AltSvc HTTP/2 protocol message rather than a response header to notify the client that HTTP/3 is available. See the EnableAltSvc registry key. Note this requires netsh sslcert bindings that use host names rather than IP addresses.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

HTTP.sys web server implementation in ASP.NET Core

Article • 10/31/2023

By Tom Dykstra and Chris Ross

HTTP.sys is a web server for ASP.NET Core that only runs on Windows. HTTP.sys is an alternative to Kestrel server and offers some features that Kestrel doesn't provide.

> **Important**
>
> HTTP.sys isn't compatible with the **ASP.NET Core Module** and can't be used with IIS or IIS Express.

HTTP.sys supports the following features:

- Windows Authentication
- Port sharing
- HTTPS with SNI
- HTTP/2 over TLS (Windows 10 or later)
- Direct file transmission
- Response caching
- WebSockets (Windows 8 or later)

Supported Windows versions:

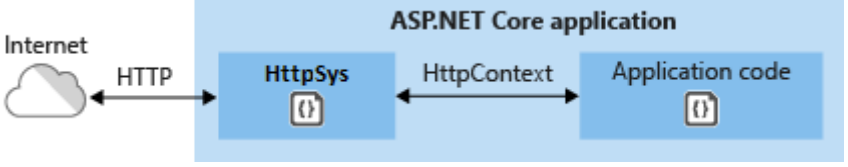
- Windows 7 or later
- Windows Server 2008 R2 or later

View or download sample code (how to download)

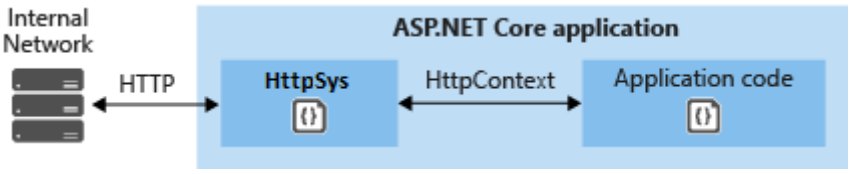
When to use HTTP.sys

HTTP.sys is useful for deployments where:

- There's a need to expose the server directly to the Internet without using IIS.



- An internal deployment requires a feature not available in Kestrel. For more information, see Kestrel vs. HTTP.sys



HTTP.sys is mature technology that protects against many types of attacks and provides the robustness, security, and scalability of a full-featured web server. IIS itself runs as an HTTP listener on top of HTTP.sys.

HTTP/2 support

HTTP/2 is enabled for ASP.NET Core apps when the following base requirements are met:

- Windows Server 2016/Windows 10 or later
- Application-Layer Protocol Negotiation (ALPN) connection
- TLS 1.2 or later connection

If an HTTP/2 connection is established, HttpRequest.Protocol reports `HTTP/2`.

HTTP/2 is enabled by default. If an HTTP/2 connection isn't established, the connection falls back to HTTP/1.1. In a future release of Windows, HTTP/2 configuration flags will be available, including the ability to disable HTTP/2 with HTTP.sys.

HTTP/3 support

HTTP/3 is enabled for ASP.NET Core apps when the following base requirements are met:

- Windows Server 2022/Windows 11 or later
- An `https` url binding is used.
- The EnableHttp3 registry key is set.

The preceding Windows 11 Build versions may require the use of a Windows Insider build.

HTTP/3 is discovered as an upgrade from HTTP/1.1 or HTTP/2 via the `alt-svc` header. That means the first request will normally use HTTP/1.1 or HTTP/2 before switching to HTTP/3. Http.Sys doesn't automatically add the `alt-svc` header, it must be added by the

application. The following code is a middleware example that adds the `alt-svc` response header.

```
C#
```

```
app.Use((context, next) =>
{
    context.Response.Headers.AltSvc = "h3=\":443\"";
    return next(context);
});
```

Place the preceding code early in the request pipeline.

Http.Sys also supports sending an AltSvc HTTP/2 protocol message rather than a response header to notify the client that HTTP/3 is available. See the EnableAltSvc registry key. This requires netsh sslcert bindings that use host names rather than IP addresses.

Kernel mode authentication with Kerberos

HTTP.sys delegates to kernel mode authentication with the Kerberos authentication protocol. User mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

Support for kernel-mode response buffering

In some scenarios, high volumes of small writes with high latency can cause significant performance impact to `HTTP.sys`. This impact is due to the lack of a Pipe buffer in the `HTTP.sys` implementation. To improve performance in these scenarios, support for response buffering is included in `HTTP.sys`. Enable buffering by setting HttpSysOptions.EnableKernelResponseBuffering to `true`.

Response buffering should be enabled by an app that does synchronous I/O, or asynchronous I/O with no more than one outstanding write at a time. In these scenarios, response buffering can significantly improve throughput over high-latency connections.

Apps that use asynchronous I/O and that may have more than one write outstanding at a time should ***not*** use this flag. Enabling this flag can result in higher CPU and memory usage by HTTP.Sys.

How to use HTTP.sys

Configure the ASP.NET Core app to use HTTP.sys

Call the UseHttpSys extension method when building the host, specifying any required HttpSysOptions. The following example sets options to their default values:

C#

```
using Microsoft.AspNetCore.Hosting.Server;
using Microsoft.AspNetCore.Hosting.Server.Features;
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys(options =>
{
    options.AllowSynchronousIO = false;
    options.Authentication.Schemes = AuthenticationSchemes.None;
    options.Authentication.AllowAnonymous = true;
    options.MaxConnections = null;
    options.MaxRequestBodySize = 30_000_000;
    options.UrlPrefixes.Add("http://localhost:5005");
});

builder.Services.AddRazorPages();

var app = builder.Build();
```

Additional HTTP.sys configuration is handled through registry settings.

For more information about HTTP.sys options, see HttpSysOptions.

MaxRequestBodySize

The maximum allowed size of any request body in bytes. When set to `null`, the maximum request body size is unlimited. This limit has no effect on upgraded connections, which are always unlimited.

The recommended method to override the limit in an ASP.NET Core MVC app for a single `IActionResult` is to use the RequestSizeLimitAttribute attribute on an action method:

C#

```
[RequestSizeLimit(100000000)]
```

```
public IActionResult MyActionMethod()
```

An exception is thrown if the app attempts to configure the limit on a request after the app has started reading the request. An `IsReadOnly` property can be used to indicate if the `MaxRequestBodySize` property is in a read-only state, meaning it's too late to configure the limit.

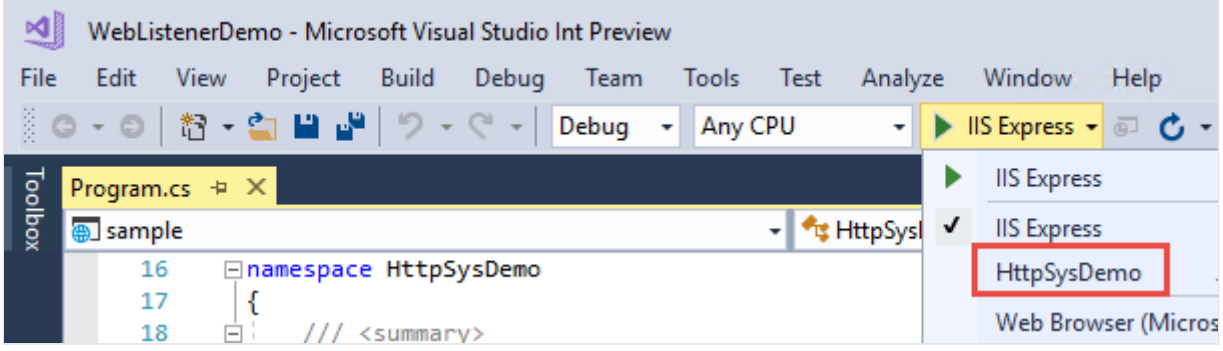
If the app should override MaxRequestBodySize per-request, use the IHttpMaxRequestBodySizeFeature:

C#

```
app.Use((context, next) =>
{
    context.Features.GetRequiredFeature<IHttpMaxRequestBodySizeFeature>()
                                            .MaxRequestBodySize = 10 *
1024;

    var server = context.RequestServices
        .GetRequiredService<IServer>();
    var serverAddressesFeature = server.Features

.GetRequiredFeature<IServerAddressesFeature>();

    var addresses = string.Join(", ", serverAddressesFeature.Addresses);

    var loggerFactory = context.RequestServices
        .GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    logger.LogInformation("Addresses: {addresses}", addresses);

    return next(context);
});
```

If using Visual Studio, make sure the app isn't configured to run IIS or IIS Express.

In Visual Studio, the default launch profile is for IIS Express. To run the project as a console app, manually change the selected profile, as shown in the following screenshot:



Configure Windows Server

1. Determine the ports to open for the app and use Windows Firewall or the New-NetFirewallRule PowerShell cmdlet to open firewall ports to allow traffic to reach HTTP.sys. In the following commands and app configuration, port 443 is used.

2. When deploying to an Azure VM, open the ports in the Network Security Group. In the following commands and app configuration, port 443 is used.

3. Obtain and install X.509 certificates, if required.

   On Windows, create self-signed certificates using the New-SelfSignedCertificate PowerShell cmdlet. For an unsupported example, see UpdateIISExpressSSLForChrome.ps1.

   Install either self-signed or CA-signed certificates in the server's **Local Machine** > **Personal** store.

4. If the app is a framework-dependent deployment, install .NET Core, .NET Framework, or both (if the app is a .NET Core app targeting the .NET Framework).

   - **.NET Core**: If the app requires .NET Core, obtain and run the **.NET Core Runtime** installer from .NET Core Downloads. Don't install the full SDK on the server.
   - **.NET Framework**: If the app requires .NET Framework, see the .NET Framework installation guide. Install the required .NET Framework. The installer for the latest .NET Framework is available from the .NET Core Downloads page.

   If the app is a self-contained deployment, the app includes the runtime in its deployment. No framework installation is required on the server.

5. Configure URLs and ports in the app.

   By default, ASP.NET Core binds to `http://localhost:5000`. To configure URL prefixes and ports, options include:

   - UseUrls
   - `urls` command-line argument
   - `ASPNETCORE_URLS` environment variable
   - UrlPrefixes

   The following code example shows how to use UrlPrefixes with the server's local IP address `10.0.0.4` on port 443:

   ```
   ```

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys(options =>
{
    options.UrlPrefixes.Add("https://10.0.0.4:443");
});

builder.Services.AddRazorPages();

var app = builder.Build();
```

An advantage of `UrlPrefixes` is that an error message is generated immediately for improperly formatted prefixes.

The settings in `UrlPrefixes` override `UseUrls`/`urls`/`ASPNETCORE_URLS` settings. Therefore, an advantage of `UseUrls`, `urls`, and the `ASPNETCORE_URLS` environment variable is that it's easier to switch between Kestrel and HTTP.sys.

HTTP.sys recognizes two types of wild cards in URL prefixes:

- `*` is a *weak binding*, also known as a *fallback binding*. If the URL prefix is `http://*:5000`, and something else is bound to port 5000, this binding won't be used.
- `+` is a *strong binding*. If the URL prefix is `http://+:5000`, this binding will be used before other port 5000 bindings.

For more information, see UrlPrefix Strings.

> ⚠ **Warning**
>
> Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings create app security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names or IP addresses rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) isn't a security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 9110: Section 7.2: Host and :authority.

Apps and containers are often given only a port to listen on, like port 80, without additional constraints like host or path. HTTP_PORTS and HTTPS_PORTS are config keys that specify the listening ports for the Kestrel and HTTP.sys servers. These keys may be specified as environment variables defined with the `DOTNET_` or

`ASPNETCORE_` prefixes, or specified directly through any other config input, such as `appsettings.json`. Each is a semicolon-delimited list of port values, as shown in the following example:

```
ASPNETCORE_HTTP_PORTS=80;8080
ASPNETCORE_HTTPS_PORTS=443;8081
```

The preceding example is shorthand for the following configuration, which specifies the scheme (HTTP or HTTPS) and any host or IP.

```
ASPNETCORE_URLS=http://*:80/;http://*:8080/;https://*:443/;https://*:80
81/
```

The HTTP_PORTS and HTTPS_PORTS configuration keys are lower priority and are overridden by URLS or values provided directly in code. Certificates still need to be configured separately via server-specific mechanics for HTTPS.

These configuration keys are equivalent to top-level wildcard bindings. They're convenient for development and container scenarios, but avoid wildcards when running on a machine that may also host other services.

6. Preregister URL prefixes on the server.

   The built-in tool for configuring HTTP.sys is *netsh.exe*. *netsh.exe* is used to reserve URL prefixes and assign X.509 certificates. The tool requires administrator privileges.

   Use the *netsh.exe* tool to register URLs for the app:

   Console

   ```
   netsh http add urlacl url=<URL> user=<USER>
   ```

   - `<URL>`: The fully qualified Uniform Resource Locator (URL). Don't use a wildcard binding. Use a valid hostname or local IP address. *The URL must include a trailing slash.*
   - `<USER>`: Specifies the user or user-group name.

   In the following example, the local IP address of the server is `10.0.0.4`:

Console

```
netsh http add urlacl url=https://10.0.0.4:443/ user=Users
```

When a URL is registered, the tool responds with `URL reservation successfully added`.

To delete a registered URL, use the `delete urlacl` command:

Console

```
netsh http delete urlacl url=<URL>
```

7. Register X.509 certificates on the server.

   Use the *netsh.exe* tool to register certificates for the app:

   Console

   ```
   netsh http add sslcert ipport=<IP>:<PORT> certhash=<THUMBPRINT> appid="
   {<GUID>}"
   ```

   - `<IP>`: Specifies the local IP address for the binding. Don't use a wildcard binding. Use a valid IP address.
   - `<PORT>`: Specifies the port for the binding.
   - `<THUMBPRINT>`: The X.509 certificate thumbprint.
   - `<GUID>`: A developer-generated GUID to represent the app for informational purposes.

   For reference purposes, store the GUID in the app as a package tag:

   - In Visual Studio:
     - Open the app's project properties by right-clicking on the app in **Solution Explorer** and selecting **Properties**.
     - Select the **Package** tab.
     - Enter the GUID that you created in the **Tags** field.
   - When not using Visual Studio:
     - Open the app's project file.
     - Add a `<PackageTags>` property to a new or existing `<PropertyGroup>` with the GUID that you created:

       XML

```
<PropertyGroup>
  <PackageTags>00001111-aaaa-2222-bbbb-
3333cccc4444</PackageTags>
</PropertyGroup>
```

In the following example:

- The local IP address of the server is `10.0.0.4`.
- An online random GUID generator provides the `appid` value.

Console

```
netsh http add sslcert
    ipport=10.0.0.4:443
    certhash=b66ee04419d4ee37464ab8785ff02449980eae10
    appid="{00001111-aaaa-2222-bbbb-3333cccc4444}"
```

When a certificate is registered, the tool responds with `SSL Certificate successfully added`.

To delete a certificate registration, use the `delete sslcert` command:

Console

```
netsh http delete sslcert ipport=<IP>:<PORT>
```

Reference documentation for *netsh.exe*:

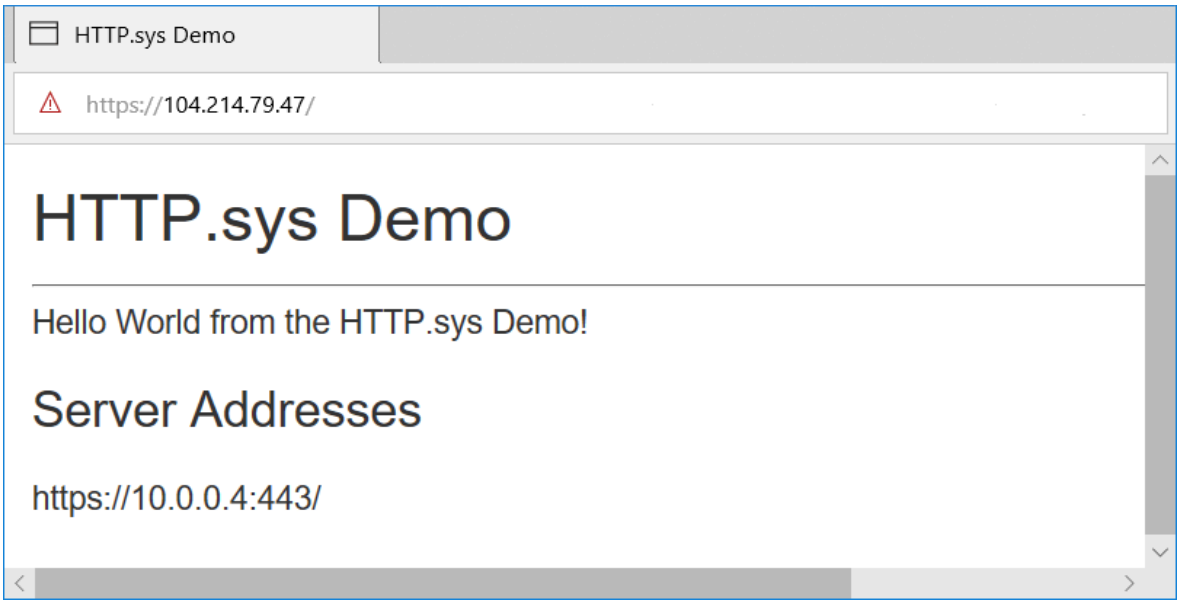
- Netsh Commands for Hypertext Transfer Protocol (HTTP)
- UrlPrefix Strings

8. Run the app.

   Administrator privileges aren't required to run the app when binding to localhost using HTTP (not HTTPS) with a port number greater than 1024. For other configurations (for example, using a local IP address or binding to port 443), run the app with administrator privileges.

   The app responds at the server's public IP address. In this example, the server is reached from the Internet at its public IP address of `104.214.79.47`.

   A development certificate is used in this example. The page loads securely after bypassing the browser's untrusted certificate warning.



Proxy server and load balancer scenarios

For apps hosted by HTTP.sys that interact with requests from the Internet or a corporate network, additional configuration might be required when hosting behind proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Get detailed timing information with IHttpSysRequestTimingFeature

IHttpSysRequestTimingFeature provides detailed timing information for requests:

- Timestamps are obtained using QueryPerformanceCounter.
- The timestamp frequency can be obtained via QueryPerformanceFrequency.
- The index of the timing can be cast to HttpSysRequestTimingType to know what the timing represents.
- The value may be 0 if the timing isn't available for the current request.
- Requires Windows 10 version 2004, Windows Server 2022, or later.

C#

```
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys();

var app = builder.Build();
```

```
app.Use((context, next) =>
{
    var feature =
context.Features.GetRequiredFeature<IHttpSysRequestTimingFeature>();

    var loggerFactory =
context.RequestServices.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    var timestamps = feature.Timestamps;

    for (var i = 0; i < timestamps.Length; i++)
    {
        var timestamp = timestamps[i];
        var timingType = (HttpSysRequestTimingType)i;

        logger.LogInformation("Timestamp {timingType}: {timestamp}",
                                            timingType, timestamp);
    }

    return next(context);
});

app.MapGet("/", () => Results.Ok());

app.Run();
```

IHttpSysRequestTimingFeature.TryGetTimestamp retrieves the timestamp for the provided timing type:

C#

```
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys();

var app = builder.Build();

app.Use((context, next) =>
{
    var feature =
context.Features.GetRequiredFeature<IHttpSysRequestTimingFeature>();

    var loggerFactory =
context.RequestServices.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    var timingType = HttpSysRequestTimingType.RequestRoutingEnd;

    if (feature.TryGetTimestamp(timingType, out var timestamp))
    {
```

```
        logger.LogInformation("Timestamp {timingType}: {timestamp}",
                                              timingType, timestamp);
    }
    else
    {
        logger.LogInformation("Timestamp {timingType}: not available for the
"
                                               + "current request",
timingType);
    }

    return next(context);
});

app.MapGet("/", () => Results.Ok());

app.Run();
```

IHttpSysRequestTimingFeature.TryGetElapsedTime gives the elapsed time between two specified timings:

C#

```
using Microsoft.AspNetCore.Http.Features;
using Microsoft.AspNetCore.Server.HttpSys;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseHttpSys();

var app = builder.Build();

app.Use((context, next) =>
{
    var feature =
context.Features.GetRequiredFeature<IHttpSysRequestTimingFeature>();

    var loggerFactory =
context.RequestServices.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger("Sample");

    var startingTimingType = HttpSysRequestTimingType.RequestRoutingStart;
    var endingTimingType = HttpSysRequestTimingType.RequestRoutingEnd;

    if (feature.TryGetElapsedTime(startingTimingType, endingTimingType, out
var elapsed))
    {
        logger.LogInformation(
            "Elapsed time {startingTimingType} to {endingTimingType}:
{elapsed}",
            startingTimingType,
            endingTimingType,
            elapsed);
```

```
    }
    else
    {
        logger.LogInformation(
            "Elapsed time {startingTimingType} to {endingTimingType}:"
            + " not available for the current request.",
            startingTimingType,
            endingTimingType);
    }

    return next(context);
});

app.MapGet("/", () => Results.Ok());

app.Run();
```

Advanced HTTP/2 features to support gRPC

Additional HTTP/2 features in HTTP.sys support gRPC, including support for response trailers and sending reset frames.

Requirements to run gRPC with HTTP.sys:

- Windows 11 Build 22000 or later, Windows Server 2022 Build 20348 or later.
- TLS 1.2 or later connection.

Trailers

HTTP Trailers are similar to HTTP Headers, except they are sent after the response body is sent. For IIS and HTTP.sys, only HTTP/2 response trailers are supported.

C#

```
if (httpContext.Response.SupportsTrailers())
{
    httpContext.Response.DeclareTrailer("trailername");

    // Write body
    httpContext.Response.WriteAsync("Hello world");

    httpContext.Response.AppendTrailer("trailername", "TrailerValue");
}
```

In the preceding example code:

- `SupportsTrailers` ensures that trailers are supported for the response.

- `DeclareTrailer` adds the given trailer name to the `Trailer` response header. Declaring a response's trailers is optional, but recommended. If `DeclareTrailer` is called, it must be before the response headers are sent.
- `AppendTrailer` appends the trailer.

Reset

Reset allows for the server to reset a HTTP/2 request with a specified error code. A reset request is considered aborted.

C#

```
var resetFeature = httpContext.Features.Get<IHttpResetFeature>();
resetFeature.Reset(errorCode: 2);
```

`Reset` in the preceding code example specifies the `INTERNAL_ERROR` error code. For more information about HTTP/2 error codes, visit the HTTP/2 specification error code section.

Tracing

For information about how to get traces from HTTP.sys, see HTTP.sys Manageability Scenarios.

Additional resources

- Enable Windows Authentication with HTTP.sys
- HTTP Server API
- aspnet/HttpSysServer GitHub repository (source code)
- The host
- Troubleshoot and debug ASP.NET Core projects

.NET Hot Reload support for ASP.NET Core

Article • 04/26/2023

.NET Hot Reload applies code changes, including changes to stylesheets, to a running app without restarting the app and without losing app state. Hot Reload is supported for all ASP.NET Core 6.0 and later projects.

Generally, updated code is rerun to take effect with the following conditions:

- Some startup logic is only run once:
  - Middleware, unless the code update is to an inline middleware delegate.
  - Configured services.
  - Route creation and configuration, unless the code update is to a route handler delegate (for example, `OnInitialized`).
- In Blazor apps, the framework triggers a Razor component render automatically.
- In MVC and Razor Pages apps, Hot Reload triggers a browser refresh automatically.
- Removing a Razor component parameter attribute doesn't cause the component to rerender. The app must be restarted.

For more information on supported scenarios, see Supported code changes (C# and Visual Basic).

Blazor WebAssembly

Blazor WebAssembly Hot Reload supports the following code changes:

- New types.
- Nested classes.
- Most changes to method bodies, such as adding, removing, and editing variables, expressions, and statements.
- Changes to the bodies of lambda expressions and local functions.
- Adding static and instance methods to existing types.
- Adding static and instance fields, events, and properties to existing types.
- Adding static lambdas to existing methods.
- Adding lambdas that capture `this` to existing methods that already captured `this` previously.

Note that when an attribute is removed that previously set the value of a component parameter, the component is disposed and re-initialized to set the removed parameter

back to its default value.

The following code changes aren't supported for Blazor WebAssembly apps:

- Adding a new await operator or yield keyword expression.
- Changing the names of method parameters.

.NET CLI

Hot Reload is activated using the dotnet watch command:

.NET CLI

```
dotnet watch
```

To force the app to rebuild and restart, use the keyboard combination `Ctrl`+`R` in the command shell.

When an unsupported code edit is made, called a *rude edit*, `dotnet watch` asks you if you want to restart the app:

- **Yes**: Restarts the app.
- **No**: Doesn't restart the app and leaves the app running without the changes applied.
- **Always**: Restarts the app as needed when rude edits occur.
- **Never**: Doesn't restart the app and avoids future prompts.

To disable support for Hot Reload, pass the `--no-hot-reload` option to the `dotnet watch` command:

.NET CLI

```
dotnet watch --no-hot-reload
```

Disable Hot Reload

The following setting in `Properties/launchSettings.json` disables Hot Reload:

JSON

```
"hotReloadEnabled" : false
```

Additional resources

For more information, see the following resources in the Visual Studio documentation:

- YouTube video .NET 6 Hot Reload in Visual Studio 2022, VS Code, and NOTEPAD?!?
- Introducing the .NET Hot Reload experience for editing code at runtime
- Write and debug running code with Hot Reload in Visual Studio
- Updates for Blazor & Razor editors + Hot Reload for ASP.NET
- Test Execution with Hot Reload

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

How to use dev tunnels in Visual Studio 2022 with ASP.NET Core apps

Article • 11/06/2024

The *dev tunnels* feature of Visual Studio 2022 enables ad-hoc connections between machines that can't directly connect to each other. A URL is created that enables any device with an internet connection to connect to an ASP.NET Core project while it runs on localhost.

Use cases

Some of the scenarios that dev tunnels enable:

- Test a web app on other devices, like mobile phones and tablets.
- Test an app with external services. For instance, test and debug Power Platform connectors, Azure Communication Services APIs, or Twilio webhooks.
- Make an app temporarily available to others over the internet, for a presentation or to invite others to review your work on a web app or API.
- As an alternative to other port-forwarding solutions.

Prerequisites

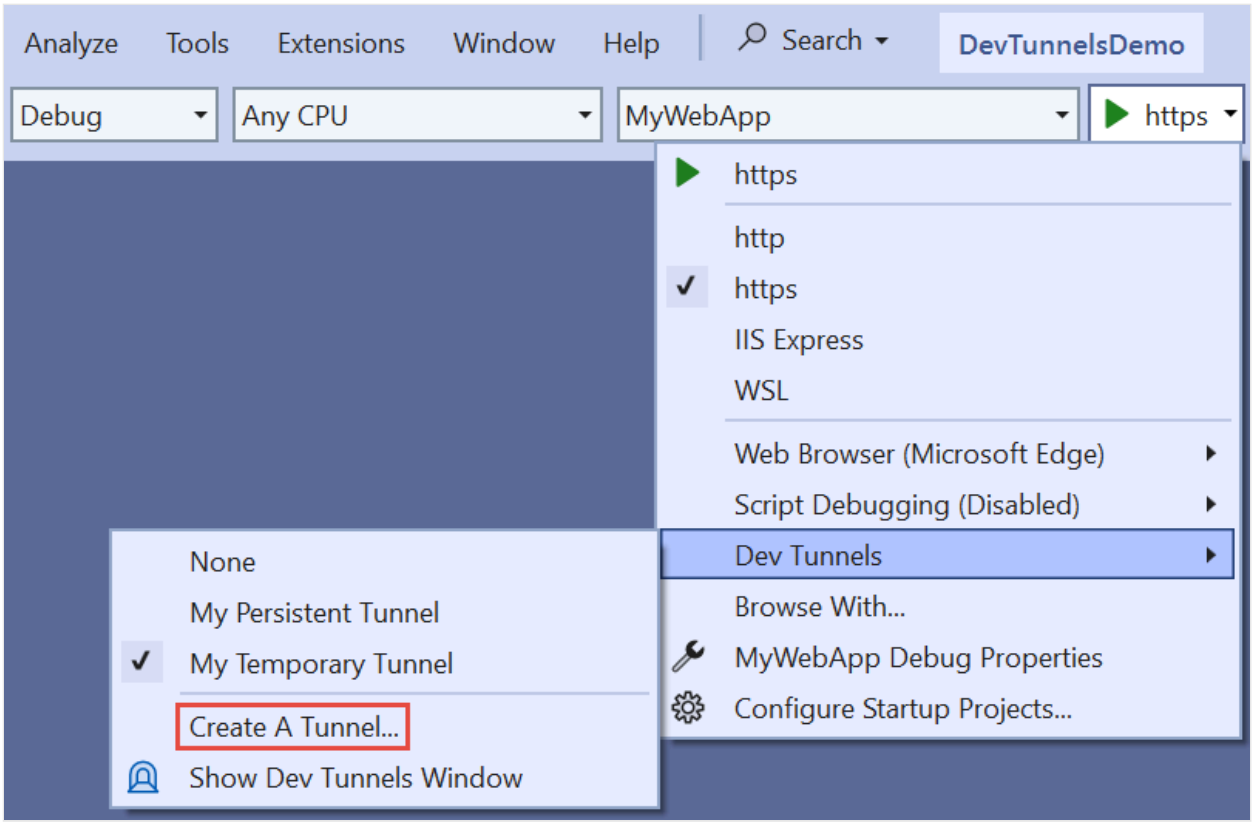
- Visual Studio 2022 version 17.6 or later with the **ASP.NET and web development** workload installed. You need to be signed in to Visual Studio to create and use dev tunnels. The feature isn't available in Visual Studio for Mac.
- One or more ASP.NET Core projects. This article uses a solution with two sample projects to demonstrate the feature.

Create a tunnel

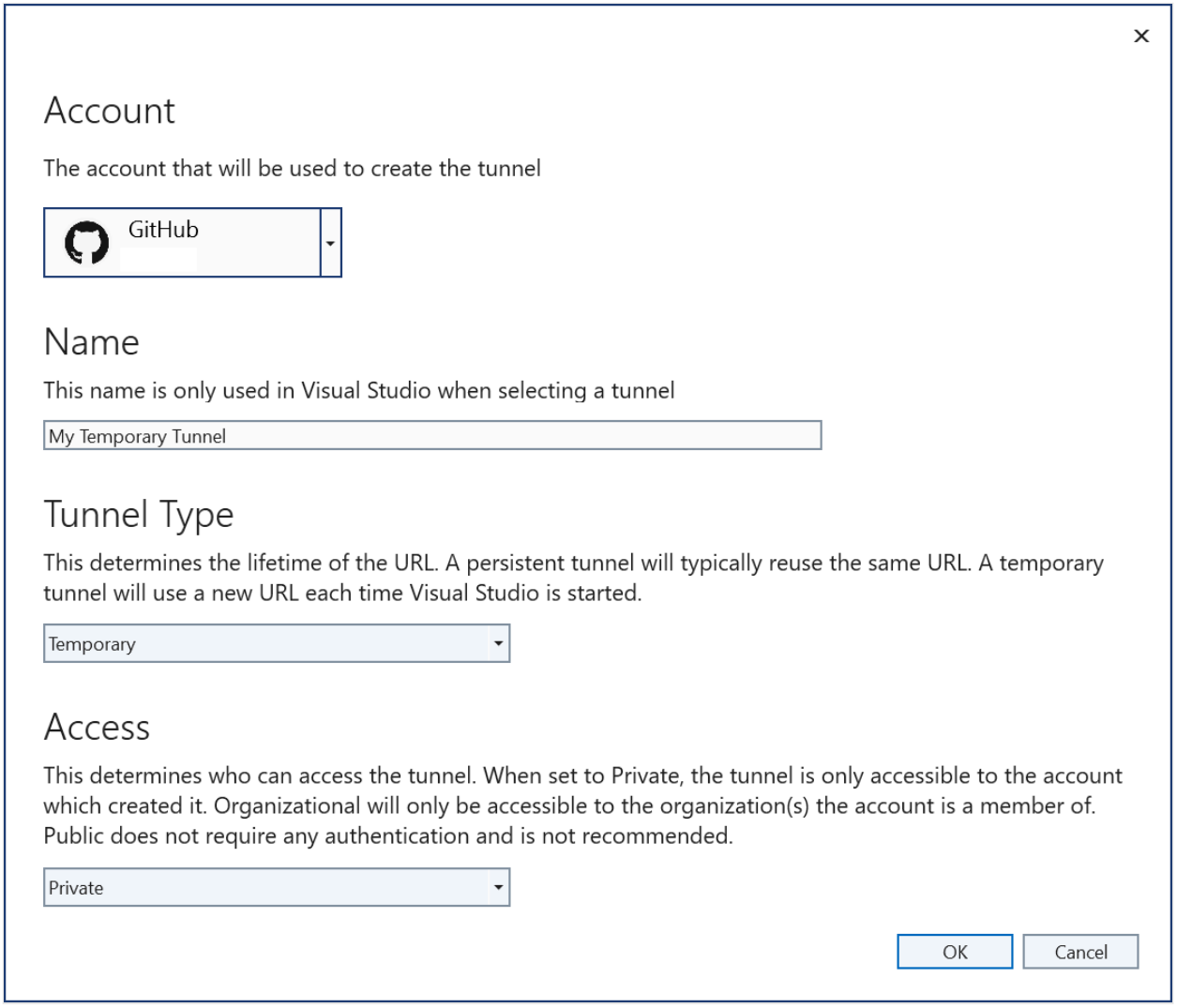
To create a tunnel:

In Visual Studio 2022, open an ASP.NET Core web project, or a solution with at least one web project set as a startup project.

In the debug dropdown, select **Dev Tunnels** > **Create A Tunnel**.



The tunnel creation dialog opens.

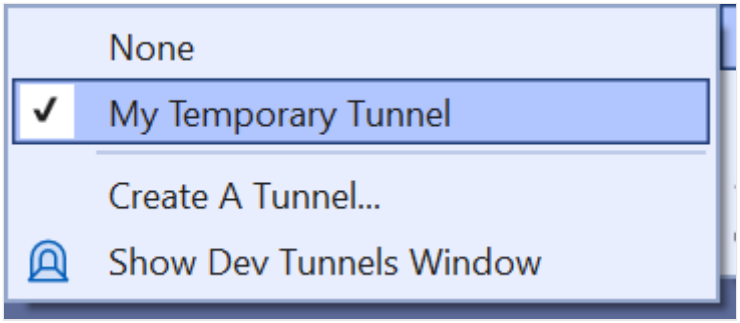


- Select the account to use to create the tunnel. Account types that can be used to create tunnels include Azure, Microsoft Account (MSA), and GitHub.
- Enter a name for the tunnel. This name identifies the tunnel in the Visual Studio UI.
- Choose the tunnel type, Persistent or Temporary:
  - A temporary tunnel gets a new URL each time Visual Studio is started.
  - A persistent tunnel gets the same URL each time Visual Studio is started. For more information, see Persistent vs. temporary tunnels later in this article.
- Choose the authentication that is required for access to the tunnel. The following options are available:
  - Private: The tunnel is accessible only to the account that created it.
  - Organization: The tunnel is accessible to accounts in the same organization as the one that created it. If this option is selected for a personal Microsoft account (MSA), the effect is the same as selecting Private. Organization support for Github accounts isn't supported.
  - Public: No authentication required. Choose this option only if it's safe to make the web app or API accessible to anyone on the internet.
- Select **OK**.

Visual Studio displays confirmation of tunnel creation:

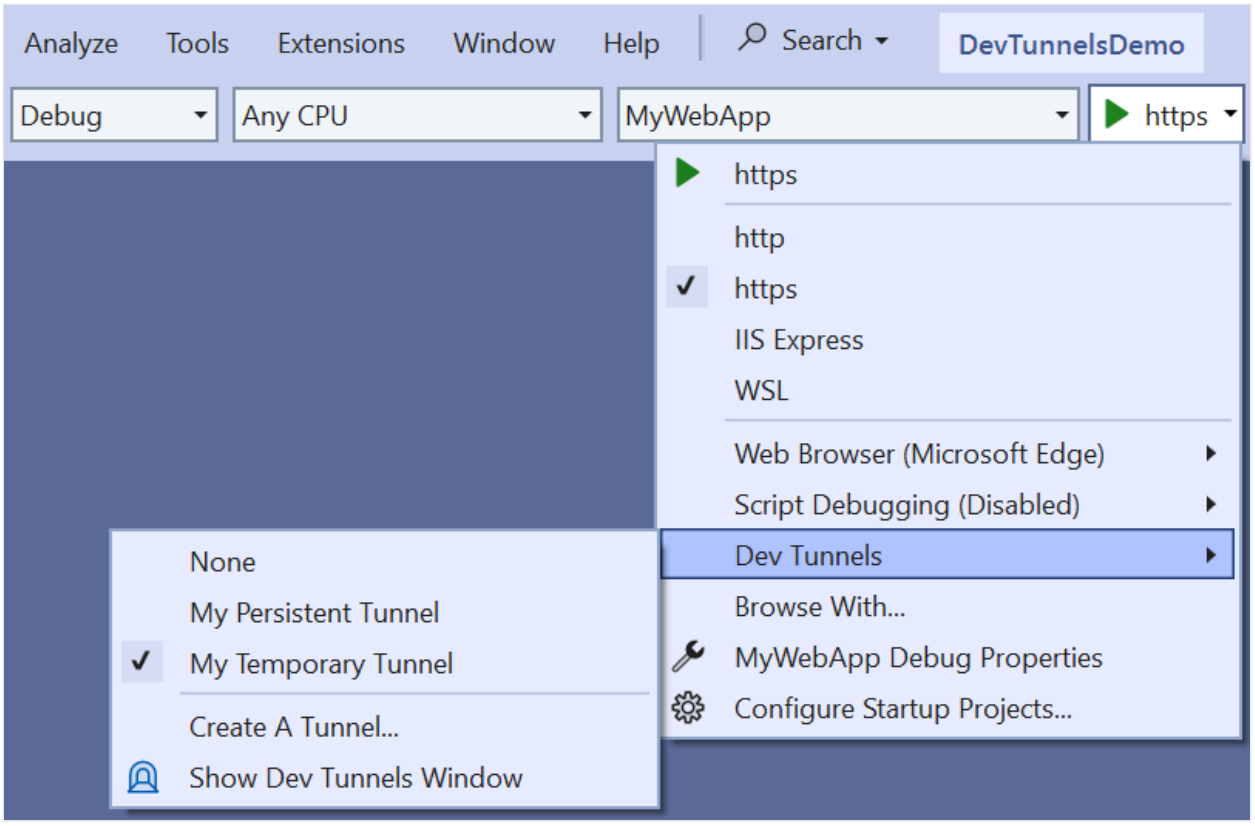


The tunnel appears in the debug dropdown **Dev Tunnels** flyout:



Specify the active tunnel

A project or solution can have multiple tunnels, but only one at a time is active. The **Dev Tunnels** flyout in the debug dropdown can specify the active tunnel. When there is an active tunnel, it's used for all ASP.NET Core projects that are started in Visual Studio. Once a tunnel is selected as active, it remains active until Visual Studio is closed. In the following illustration, **My Temporary Tunnel** is active:



Choose not to use a tunnel by selecting **None** in the flyout. When Visual Studio is restarted, it defaults back to **None**.

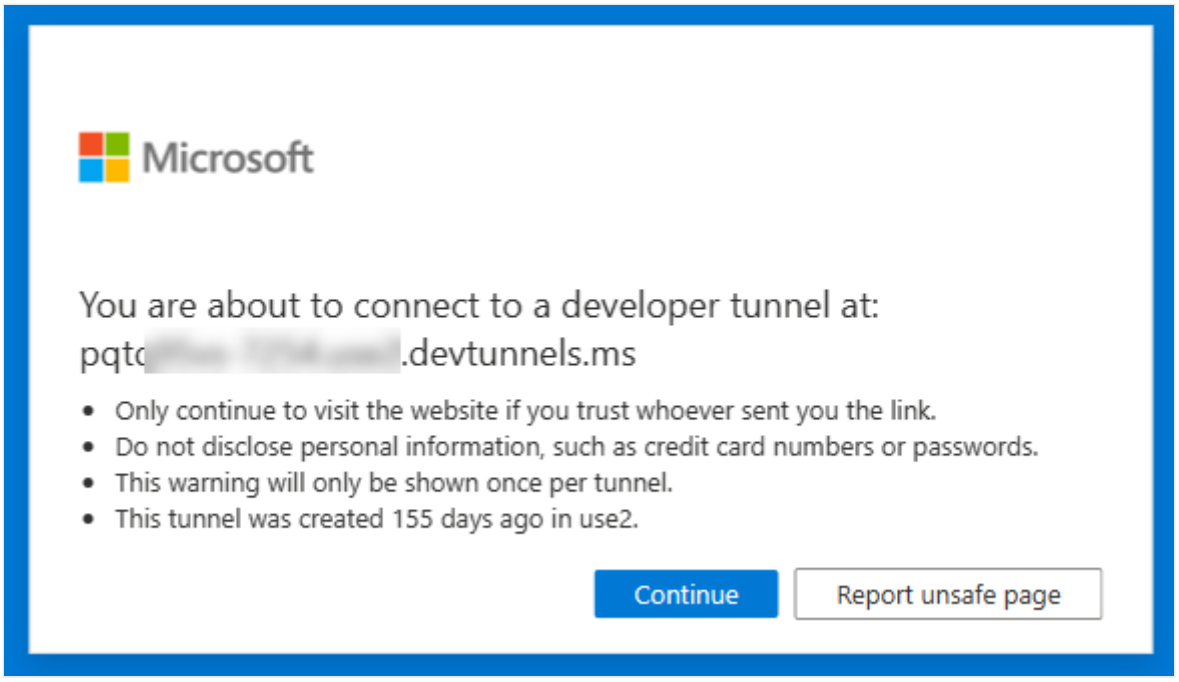
Use a tunnel

When a tunnel is active and Visual Studio runs a web app, the web browser opens to a tunnel URL instead of a localhost URL. The tunnel URL looks like the following example:

```
https://0pbvlk3m-7032.usw2.devtunnels.ms
```

Now any authenticated user can open the same URL on any other internet-connected device. As long as the project continues to run locally, any device with an internet connection can access the web application that is running on a development machine.

For web projects that have browser support, a warning page is shown on the first request sent to the tunnel URL from each device:



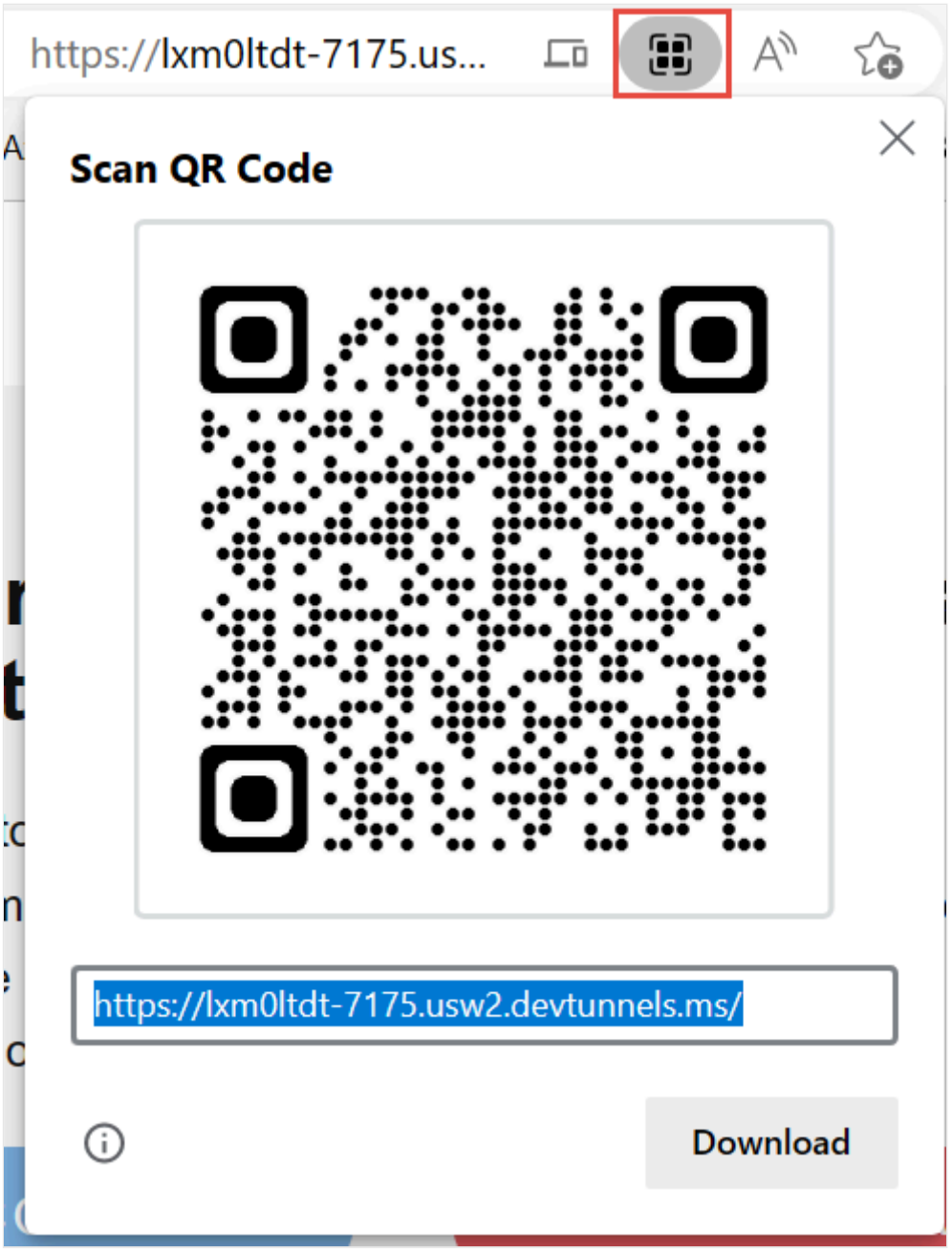
After **Continue** is selected, the request is routed to the local web app. This notification page isn't shown for API requests using dev tunnels.

Use a tunnel to test on a phone or tablet

To test a web app from an external device like a phone or tablet, navigate to the tunnel URL. To make it easier to reproduce the URL on the external device:

- Navigate to the tunnel URL in an Edge browser on the local machine.
- Generate a QR code to the URL in the Edge browser on the local machine:
  - Select the URL bar and the QR code button appears.

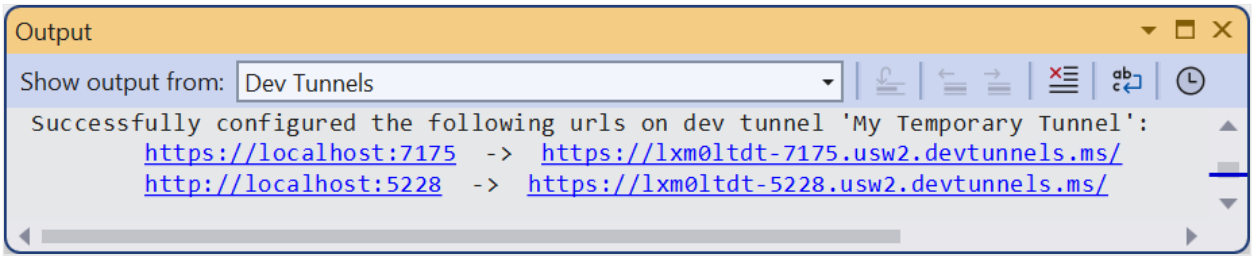
- Select the QR code button to generate and view the QR code.



- Scan this QR code with a phone or tablet to navigate to the URL.

Dev Tunnels output window

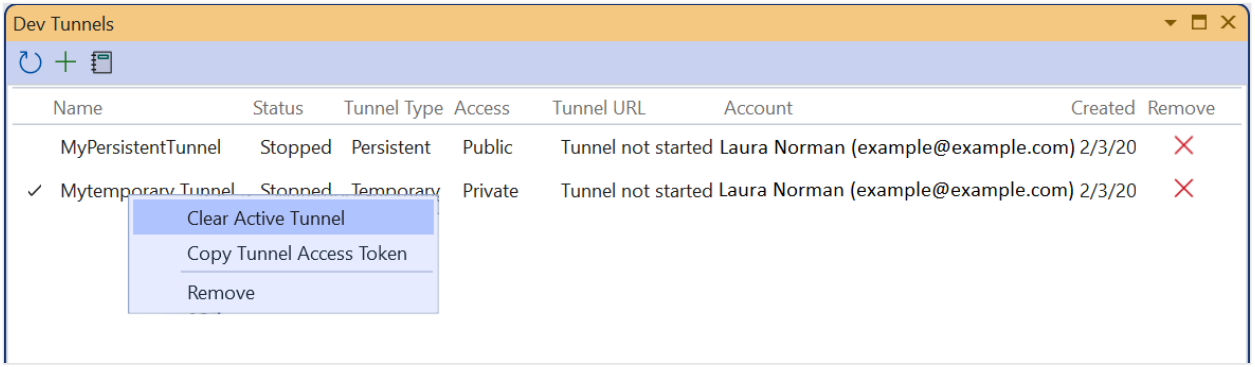
To show the URL of a tunnel of a running project, select **Dev Tunnels** in the **Show output from** dropdown.



This window is especially useful for projects that don't open a browser by default. For example, when working with an Azure Function, this may be the easiest way to discover the public URL that is being used by the dev tunnel.

Dev Tunnels tool window

View and manage dev tunnels in the **Dev Tunnels** tool window:



To open the **Dev Tunnels** window, select the **Show Dev Tunnels Window** menu option in the debug dropdown. Alternatively, select **View** > **Other Windows** > **Dev Tunnels**.

From the **Dev Tunnels** window, create a new tunnel by selecting the green `+` button.

Delete a tunnel by using the red `x` button to the right of the tunnel.

The context menu for a tunnel provides the following options:

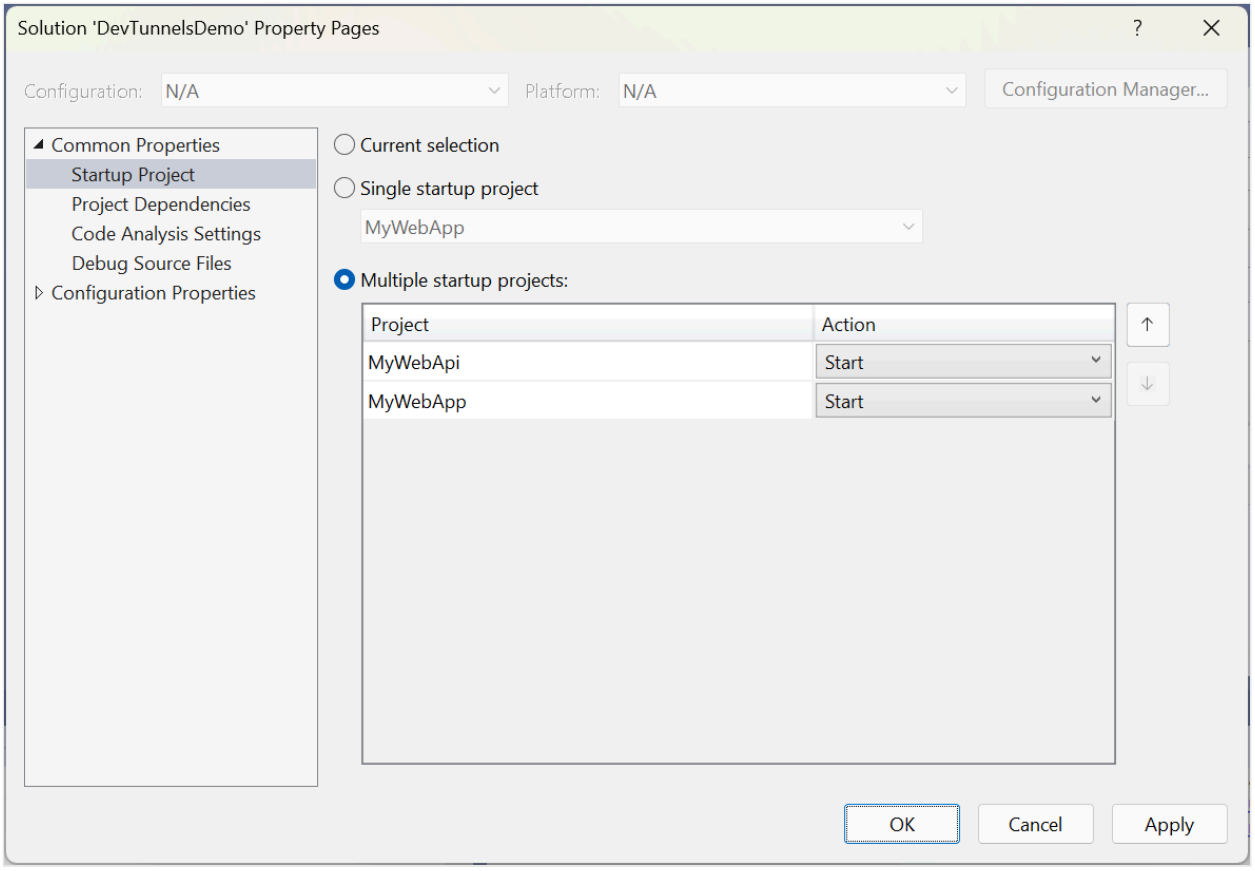
- **Clear Active Tunnel**: Shown when a tunnel is configured as active (indicated by the checkmark on the left hand side), this resets it so the solution is not using a tunnel.
- **Make Active Tunnel**: Shown for tunnels that are not configured as active.
- **Copy Tunnel Access Token**: Provided for scenarios where a tunnel is created with private or organizational access, and the app is a web API. To authenticate for the tunnel, copy and paste the tunnel access token as a header of the form `X-Tunnel-Authorization tunnel <TOKEN>` in the request. If this header is not specified, the request will be blocked because the authentication check failed.
- **Remove**

Tunnel URL environment variables

The dev tunnels feature provides a way to get the tunnel URL of a project programmatically at run time. When an app is launched that uses a tunnel, Visual Studio creates the environment variable `VS_TUNNEL_URL`. The `VS_TUNNEL_URL` value is the URL for the tunnel that is used for the current project. `VS_TUNNEL_URL` can be useful when

integrating the app with an external service, where the tunnel URL needs to be passed to the external service.

If multiple ASP.NET Core projects are configured to start in Visual Studio, the app that is starting up gets an environment variable for any project that started before it. The pattern for this variable name is `VS_TUNNEL_URL_{ProjectName}`, where `{ProjectName}` is the name of the other project. For example, consider this example showing two projects set to start:



Since MyWebApi is above MyWebApp, it's started before the MyWebApp project. When the MyWebApi project is started, it receives its tunnel URL in the `VS_TUNNEL_URL` environment variable. When the MyWebApp project is started, it receives its own tunnel URL in `VS_TUNNEL_URL` and the other project's tunnel URL is provided in the `VS_TUNNEL_URL_MyWebApi` environment variable.

To illustrate, the following highlighted lines of code have been added to the *Program.cs* file in MyWebApp:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
```

```
Console.WriteLine($"Tunnel URL: {Environment.
    GetEnvironmentVariable("VS_TUNNEL_URL")}");
Console.WriteLine($"API project tunnel URL: {Environment.
    GetEnvironmentVariable("VS_TUNNEL_URL_MyWebApi")}");
```

When the web app is started the console output looks like the following example:

Console

```
Tunnel URL: https://lxm0ltdt-7175.usw2.devtunnels.ms/
API project tunnel URL: https://lxm0ltdt-7042.usw2.devtunnels.ms/
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:7175
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5228
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\DevTunnelsDemo\MyWebApp
```

For information about how to set up multiple startup projects, see How to: Set multiple startup projects.

Persistent versus temporary tunnels

A persistent tunnel is one that uses the same URL after exiting and restarting Visual Studio. Having a URL that doesn't change can be useful when integrating a web app with an external service. For example, implementing a GitHub webhook, or developing an API to integrate with a Power Platform app. In such cases, you might need to specify the callback URL to the external service. With a persistent tunnel, the external service URL only needs to be configured once. Using a temporary tunnel, the tunnel URL must be configured each time Visual Studio restarts.

Persistent doesn't mean the tunnel works when Visual Studio isn't open. A tunnel URL connects to the local machine only if the ASP.NET Core project that the tunnel URL connects to is running in Visual Studio.

A temporary tunnel is fine when the dev tunnel URL needs to work for a short time. For example, sharing in-progress work on a web app with others, or testing an app on an external device. In some cases, it might be best to get a new URL each time Visual Studio starts.

See also

The following resources use an early preview version of the dev tunnels feature, so parts of them are out of date:

- Use dev tunnels in Visual Studio to debug your web APIs
- ACS Call Automation and Azure OpenAI Service
- Use Visual Studio dev tunnels to handle Twilio Webhooks.

Use .http files in Visual Studio 2022

Article • 01/13/2025

The Visual Studio 2022 `.http` file editor provides a convenient way to test ASP.NET Core projects, especially API apps. The editor provides a UI that:

- Creates and updates `.http` files.
- Sends HTTP requests specified in `.http` files.
- Displays the responses.

This article contains documentation for:

- The .http file syntax.
- How to create an .http file.
- How to send a request from an .http file.
- Where to find .http file options that can be configured.
- How to create requests in .http files by using the Visual Studio 2022 **Endpoints Explorer**.

The `.http` file format and editor was inspired by the Visual Studio Code REST Client extension. The Visual Studio 2022 `.http` editor recognizes `.rest` as an alternative file extension for the same file format.

Prerequisites

- Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed.

`.http` file syntax

The following sections explain `.http` file syntax.

Requests

The format for an HTTP request is `HTTPMethod URL HTTPVersion`, all on one line, where:

- `HTTPMethod` is the HTTP method to use, for example:
  - OPTIONS
  - GET
  - HEAD

    - POST
    - PUT
    - PATCH
    - DELETE
    - TRACE
    - CONNECT
- `URL` is the URL to send the request to. The URL can include query string parameters. The URL doesn't have to point to a local web project. It can point to any URL that Visual Studio can access.
- `HTTPVersion` is optional and specifies the HTTP version that should be used, that is, `HTTP/1.1`, `HTTP/2`, or `HTTP/3`.

A file can contain multiple requests by using lines with `###` as delimiters. The following example showing three requests in a file illustrates this syntax:

```HTTP
GET https://localhost:7220/weatherforecast

###

GET https://localhost:7220/weatherforecast?date=2023-05-11&location=98006

###

GET https://localhost:7220/weatherforecast HTTP/3

###
```

Request headers

To add one or more headers, add each header on its own line immediately after the request line. Don't include any blank lines between the request line and the first header or between subsequent header lines. The format is `HeaderName: Value`, as shown in the following examples:

```HTTP
GET https://localhost:7220/weatherforecast
Date: Wed, 27 Apr 2023 07:28:00 GMT

###

GET https://localhost:7220/weatherforecast
Cache-Control: max-age=604800
```

```
Age: 100

###
```

> **Important**
>
> When calling an API that authenticates with headers, do not commit any secrets to a source code repository. See the supported methods for storing secrets later in this article, such as ASP.NET Core user secrets, Azure Key Vault and DPAPI encryption.

Request body

Add the request body after a blank line, as shown in the following example:

HTTP

```
POST https://localhost:7220/weatherforecast
Content-Type: application/json
Accept-Language: en-US,en;q=0.5

{
    "date": "2023-05-10",
    "temperatureC": 30,
    "summary": "Warm"
}

###
```

Comments

Lines that start with either `#` or `//` are comments. These lines are ignored when Visual Studio sends HTTP requests.

Variables

A line that starts with `@` defines a variable by using the syntax `@VariableName=Value`.

Variables can be referenced in requests that are defined later in the file. They're referenced by wrapping their names in double curly braces, `{{` and `}}`. The following example shows two variables defined and used in a request:

HTTP

```
@hostname=localhost
@port=44320
GET https://{{hostname}}:{{port}}/weatherforecast
```

Variables can be defined using values of other variables that were defined earlier in the file. The following example uses one variable in the request instead of the two shown in the preceding example:

HTTP

```
@hostname=localhost
@port=44320
@host={{hostname}}:{{port}}
GET https://{{host}}/api/search/tool
```

Environment files

To give variables different values in different environments, create a file named `http-client.env.json`. Locate the file in the same directory as the `.http` file or in one of its parent directories. Here's an example of an environment file:

JSON

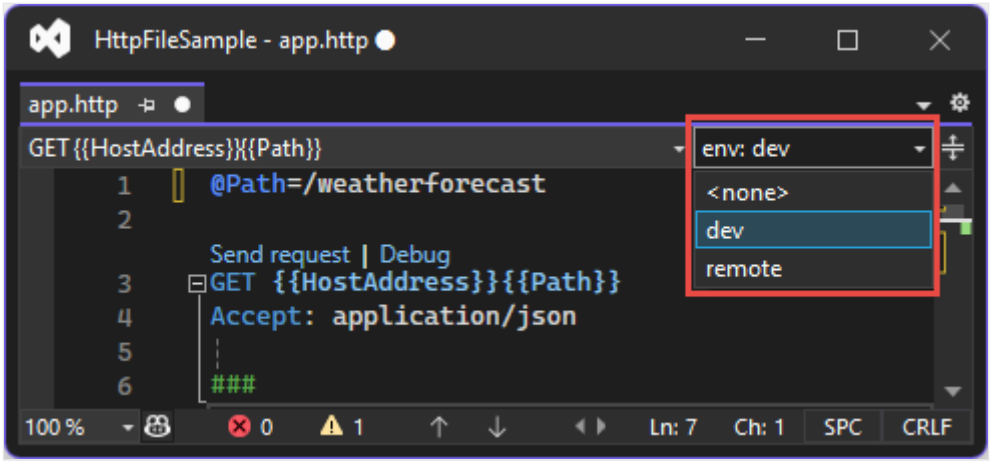
```
{
  "dev": {
    "HostAddress": "https://localhost:44320"
  },
  "remote": {
    "HostAddress": "https://contoso.com"
  }
}
```

The environment file is a JSON file that contains one or more named environments, such as "dev" and "remote" in the preceding example. Each named environment contains one or more variables, such as `HostAddress` in the preceding example. Variables from an environment file are referenced the same way as other variables, as shown in the following example:

HTTP

```
GET {{HostAddress}}/api/search/tool
```

The value that is used for the variable when sending a request is determined by an environment selector dropdown at the upper right corner of the `.http` file editor. The following screenshot shows the selector:



The environment file doesn't have to be in the project folder. Visual Studio looks for an environment file in the folder where the `.http` file exists. If it's not in that folder, Visual Studio looks through the parent directories to find it. When a file named `http-client.env.json` is found, the search ends. The file found nearest to the `.http` file is used.

After creating or editing an `.http` file, you might have to close and reopen the project to see the changes reflected in the environment selector. Press `F6` to select the environment selector.

Visual Studio displays warnings in the following situations:

- The `.http` file references a variable that isn't defined in the `.http` file or in the environment file.
- The environment file contains a variable that isn't referenced in the `.http` file.

A variable defined in an environment file can be the same as one defined in the `.http` file, or it can be different. If a variable is defined in both the `.http` file and the environment file, the value in the `.http` file overrides the value in the environment file.

Shared variables

`$shared` is a special environment name for values that are the same for multiple environments. For example, consider the following environment file (`http-client.env.json`):

JSON

```
{
    "$shared": {
        "HostAddress": "https://localhost:7293"
    },
    "dev1": {
        "username": "dev1user"
    },
    "dev2": {
        "username": "dev2user"
    },
    "staging": {
        "username": "staginguser",
        "HostAddress": "https://staging.contoso.com"
    }
}
```

In the preceding example, the `$shared` environment defines the `HostAddress` variable with the value `localhost:7293`. The `HostAddress` variable with the value `localhost:7293` functions as a default for environments that don't define a `HostAddress`. When the `dev1` or `dev2` environment is defined, the value for `HostAddress` comes from the `$shared` environment because `dev1` and `dev2` don't define a `HostAddress` variable. When the `staging` environment is defined, the value for `HostAddress` is set to `https://staging.contoso.com`, overriding the `$shared` default.

User-specific environment files

A user-specific value is any value that a developer wants to test with but doesn't want to share with the team. The `http-client.env.json` file is checked in to source control by default, therefore, ***DO NOT*** add user-specific values to this file. Rather, add user-specific values in a file named `http-client.env.json.user`. The `http-client.env.json.user` file is located in the same folder as the `http-client.env.json` file. Files that end with `.user` are excluded from source control by default when using Visual Studio source control features.

When the `http-client.env.json` file is loaded, Visual Studio looks for a sibling `http-client.env.json.user` file. If a variable is defined in an environment in both the `http-client.env.json` file and the `http-client.env.json.user` file, the value in the `http-client.env.json.user` file wins.

Here's an example scenario that shows how a user-specific environment file works. Suppose the `.http` file has the following content:

```
HTTP
```

```
GET {{HostAddress}}/{{Path}}
Accept: application/json
```

And suppose the `http-client.env.json` file contains the following content:

JSON

```
{
  "dev": {
    "HostAddress": "https://localhost:7128",
    "Path": "/weatherforecast"
  },
  "remote": {
    "HostAddress": "https://contoso.com",
    "Path": "/weatherforecast"
  }
}
```

And suppose there's a user-specific environment file that contains the following content:

JSON

```
{
  "dev": {
    "Path": "/swagger/index.html"
  }
}
```

When the user selects the "dev" environment, the request is sent to `https://localhost:7128/swagger/index.html` because the `Path` value in the `http-client.env.json.user` file overrides the value from the `http-client.env.json` file.

With the same environment files, suppose the variables are defined in the `.http` file:

HTTP

```
@HostAddress=https://contoso.com
@Path=/weatherforecast

GET {{HostAddress}}/{{Path}}
Accept: application/json
```

In this scenario, the "dev" environment request is sent to `https://contoso.com/weatherforecast` because variable definitions in `.http` files override environment file definitions.

ASP.NET Core user secrets

To get a value from user secrets, use an environment file that is located in the same folder as the ASP.NET Core project. In the environment file, define a variable that has `provider` and `secretName` properties. Set the `provider` value to `AspnetUserSecrets` and set `secretName` to the name of the desired user secret. For example, the following environment file defines a variable named `ApiKeyDev` that gets its value from the `config:ApiKeyDev` user secret:

JSON

```
{
  "dev": {
    "ApiKeyDev": {
      "provider": "AspnetUserSecrets",
      "secretName": "config:ApiKeyDev"
    }
  }
}
```

To use this variable in the `.http` file, reference it like a standard variable. For example:

HTTP

```
GET {{HostAddress}}{{Path}}
X-API-KEY: {{ApiKeyDev}}
```

When the request is sent, the value of the `ApiKeyDev` secret is in the X-API-KEY header.

As you type in the `http` file, the editor shows a completion list for the variable name but doesn't show its value.

Azure Key Vault

Azure Key Vault is one of several key management solutions in Azure that can be used for secrets management. Of the three secrets stores currently supported for `.http` files, Key Vault is the best choice for sharing secrets across different users. The other two options—ASP.NET User Secrets and DPAPI encryption—aren't easily shared.

To use a value from Azure Key Vault, you must be signed into Visual Studio with an account that has access to the desired Key Vault. Define a variable in an environment file with the metadata to access the secret. The variable is named `AKVSecret` in the following example:

JSON

```
{
  "dev": {
    "AKVSecret": {
      "provider": "AzureKeyVault",
      "secretName": "SecretInKeyVault",
      "resourceId": "/subscriptions/3a914c59-8175a9e0e540/resourceGroups/my-key-vault-rg/providers/Microsoft.KeyVault/vaults/my-key-vault-01182024"
    }
  }
}
```

The variable `AKVSecret` pulls its value from Azure Key Vault. The following properties are defined on `AKVSecret`:

Expand table

Name	Description
provider	For Key Vault, always use `AzureKeyVault`.
secretName	Name of the secret to extract.
resourceId	Azure resource ID for the specific Key Vault to access.

The value for the `resourceId` property can be found in the Azure portal. Go to **Settings** > **Properties** to find it. For `secretName`, use the name of the secret that appears on the **Secrets** page in the Azure portal.

For example, the following `.http` file has a request that uses this secret value.

HTTP

```
GET {{HostAddress}}{{Path}}
X-AKV-SECRET: {{akvSecret}}
```

DPAPI encryption

On Windows, there is a Data Protection API (DPAPI) that can be used to encrypt sensitive data. When DPAPI is used to encrypt data, the encrypted values are always machine-specific, and they're also user-specific in `.http` files. These values can't be shared with other users.

To encrypt a value, use the following console application:

C#

```
using System.Security.Cryptography;
using System.Text;

string stringToEncrypt = "Hello, World!";
byte[] encBytes =
ProtectedData.Protect(Encoding.Unicode.GetBytes(stringToEncrypt),
optionalEntropy: null, scope: DataProtectionScope.CurrentUser);
string base64 = Convert.ToBase64String(encBytes);
Console.WriteLine(base64);
```

The preceding console application references the System.Security.Cryptography.ProtectedData NuGet package. To enable the encrypted value to work in the `.http` file, encrypt with the scope set to DataProtectionScope.CurrentUser. The encrypted value is a base64 encoded string that can be copied and pasted into the environment file.

In the environment file, create a variable that has `provider` and `value` properties. Set `provider` to `Encrypted`, and set `value` to the encrypted value. For example, the following environment file defines a variable named `dpapiValue` that gets its value from a string that was encrypted with DPAPI.

JSON

```
{
  "dev": {
    "dpapiValue": {
      "provider": "Encrypted",
      "value":
"AQAAANCMnd8BFdERjHoAwE/Cl+sBAAAA5qwfg4+Bhk2nsy6ujgg3GAAAAAACAAAAAAAQZgAAAAE
AACAAAAAqNXhXc098k1TtKmaI4cUAbJVALMVP1zOR7mhC1RBJegAAAAOgAAAAIAACAAAABKu4E
9WC/zX5LYZZhOS2pukxMTF9R4yS+XA9HoYF98GzAAAAAzFXatt461ZnVeUWgOV8M/DkqNviWUUje
xAXOF/JfpJMw/CdsizQyESus2QjsCtZlAAAAAL7ns3u9mEk6wSMIn+KNsW/vdAw51OaI+HPVrt5v
FvXRilTtvGbU/JnxsoIHj0Z7OOxlwOSg1Qdn60zEqmlFJBg=="
    }
  }
}
```

With the preceding environment file, `dpapiValue` can be used in the `.http` file like any other variable. For example:

HTTP

```
GET {{HostAddress}}{{Path}}
X-DPAPI-Secret: {{dpapiSecret}}
```

When this request is sent, X-DPAPI-Secret has the decrypted secret value.

Environment variables

To get the value of an environment variable, use `$processEnv`. The following example puts the value of the USERNAME environment variable in the X-UserName header.

```
HTTP

GET {{HostAddress}}{{Path}}
X-UserName: {{$processEnv USERNAME}}
```

If you try to use `$processEnv` to access an environment variable that doesn't exist, the `.http` file editor displays an error message.

`.env` files

To get the value of a variable that is defined in a .env file, use `$dotenv`. The `.env` file must be in the project folder. The format for `$dotenv` is the same as for `$processEnv`. For example, if the `.env` file has this content:

```

USERNAME=userFromDotenv
```

And the `.http` file has this content:

```
HTTP

GET {{HostAddress}}{{Path}}
X-UserName: {{$dotEnv USERNAME}}
```

The `X-UserName` header will have "userFromDotenv".

When `$dotenv` is entered in the editor, it shows completions for the variables defined in the `.env` file.

> **Note**

`.env` files might not be excluded from source control by default, so be careful to avoid checking in any secret values.

Random integers

To generate a random integer, use `$randomInt`. The syntax is `{{$randomInt [min max]}}` where the `min` and `max` values are optional.

Dates and times

- `$datetime` generates a `datetime` string in UTC. The syntax is `{{$datetime [format] [offset option]}}` where the format and offset options are optional.
- `$localDatetime` generates a `datetime` string in the local time zone. The syntax is `{{$localDatetime [format] [offset option]}}` where the format and offset options are optional.
- `$timeStamp` generates a `timestamp` in UTC. The `timestamp` is the number of seconds since the Unix Epoch in UTC time. The syntax is `{{$timestamp [offset option]}}` where the offset option is optional.

The `[format]` option is one of `rfc1123`, `iso8601`, or a custom format in quotation marks. For example:

HTTP

```
GET https://httpbin.org/headers
X-CUSTOM: {{$datetime "dd-MM-yyyy"}}
X-ISO8601: {{$datetime iso8601}}
X-ISO8601L: {{$localDatetime iso8601}}
X-RFC1123: {{$datetime rfc1123}}
X-RFC1123L: {{$localDatetime rfc1123}}
```

Here are some sample values that the preceding examples generate:

JSON

```
{
  "headers": {
    "X-Custom": "17-01-2024",
    "X-Iso8601": "2024-01-17T22:59:55.5345770+00:00",
    "X-Iso8601L": "2024-01-17T14:59:55.5345770-08:00",
    "X-Rfc1123": "Wed, 17 Jan 2024 22:59:55 GMT",
    "X-Rfc1123L": "Wed, 17 Jan 2024 14:59:55 -08"
```

```
  }
}
```

The `[offset option]` syntax is in the form `number` `unit` where `number` is an integer and `unit` is one of the following values:

Expand table

`unit`	**Explanation**
`ms`	Milliseconds
`s`	Seconds
`m`	Minutes
`h`	Hours
`d`	Days
`w`	Weeks
`M`	Months
`y`	Years

For example:

HTTP

```
GET https://httpbin.org/headers
X-Custom-Minus-1-Year: {{$datetime "dd-MM-yyyy" -1 y}}
X-RFC1123-Plus-1-Day: {{$datetime rfc1123 1 d}}
X-Timestamp-Plus-1-Year: {{$timestamp 1 y}}
```

Here are some sample values that the preceding examples generate:

JSON

```
{
  "headers": {
    "X-Custom-Minus-1-Year": "17-01-2023",
    "X-Rfc1123-Plus-1-Day": "Thu, 18 Jan 2024 23:02:48 GMT",
    "X-Timestamp-Plus-1-Year": "1737154968"
  }
}
```

Some of the preceding examples use the free open-source website <httpbin.org>. This is a third-party website not affiliated with Microsoft. In these examples it returns a response body with the headers that were sent in the request. For information about other ways to use this resource for API testing, see the httpbin.org site's home page.

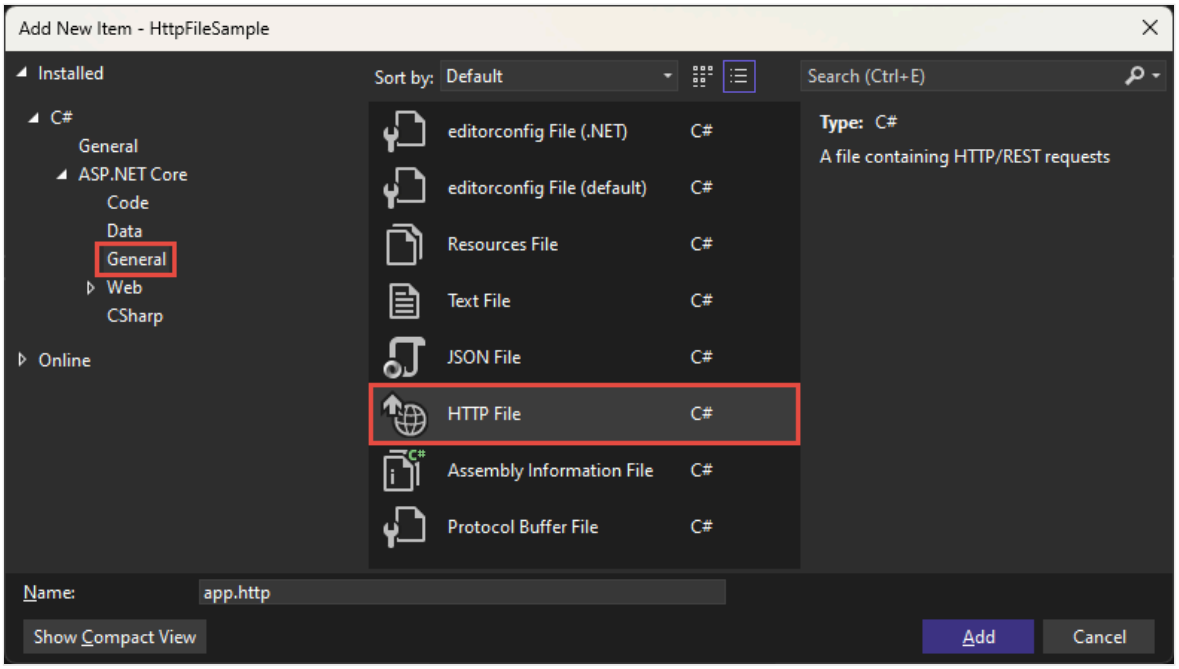
Unsupported syntax

The Visual Studio 2022 `.http` file editor doesn't have all the features that the Visual Studio Code REST Client extension has. The following list includes some of the more significant features available only in the Visual Studio Code extension:

- Request line that spans more than one line
- Named requests
- Specify file path as body of the request
- Mixed format for body when using multipart/form-data
- GraphQL requests
- cURL request
- Copy/paste as cURL
- Request history
- Save response body to file
- Certificate based authentication
- Prompt variables
- Customize response preview
- Per-request settings

Create an `.http` file

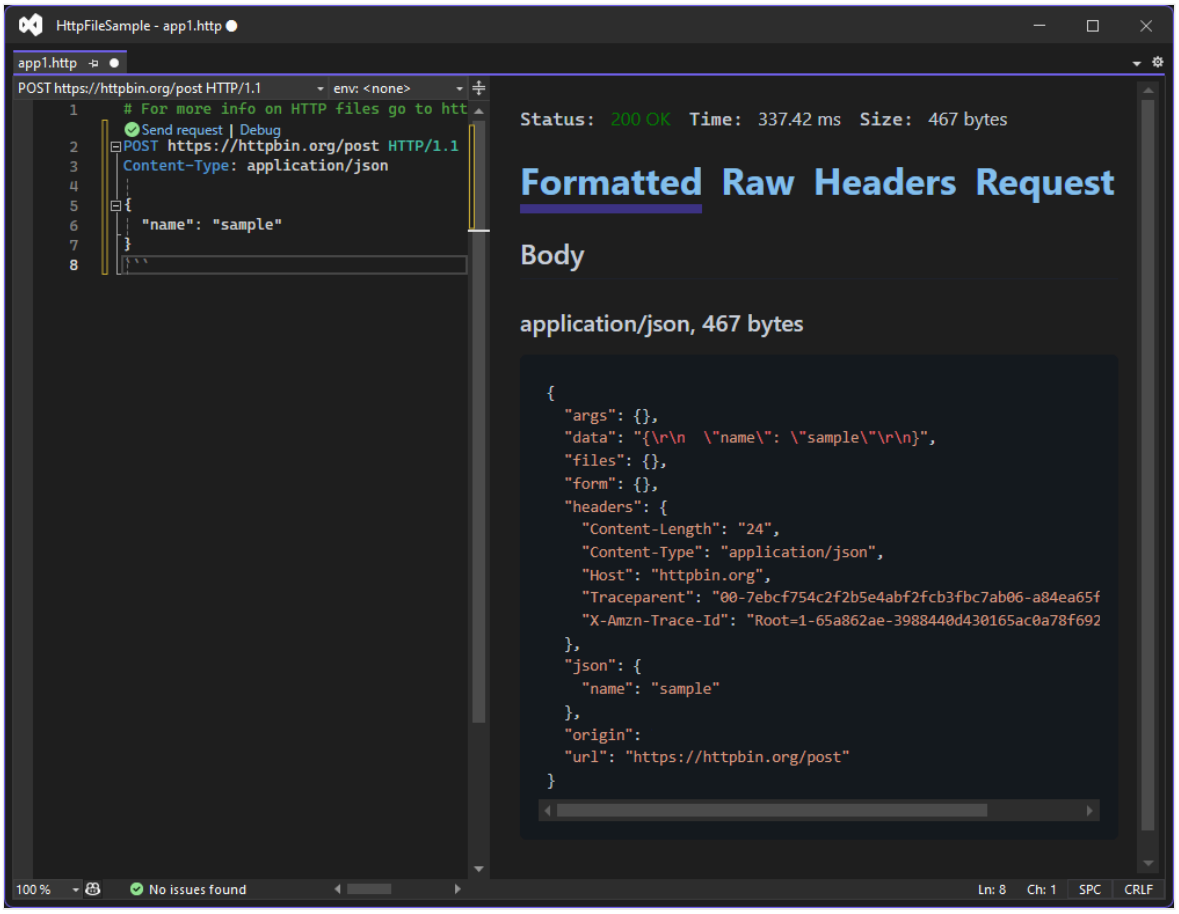
- In **Solution Explorer**, right-click an ASP.NET Core project.
- In the context menu, select **Add** > **New Item**.
- In the **Add New Item** dialog, select **ASP.NET Core** > **General**.
- Select **HTTP File**, and select **Add**.



Send an HTTP request

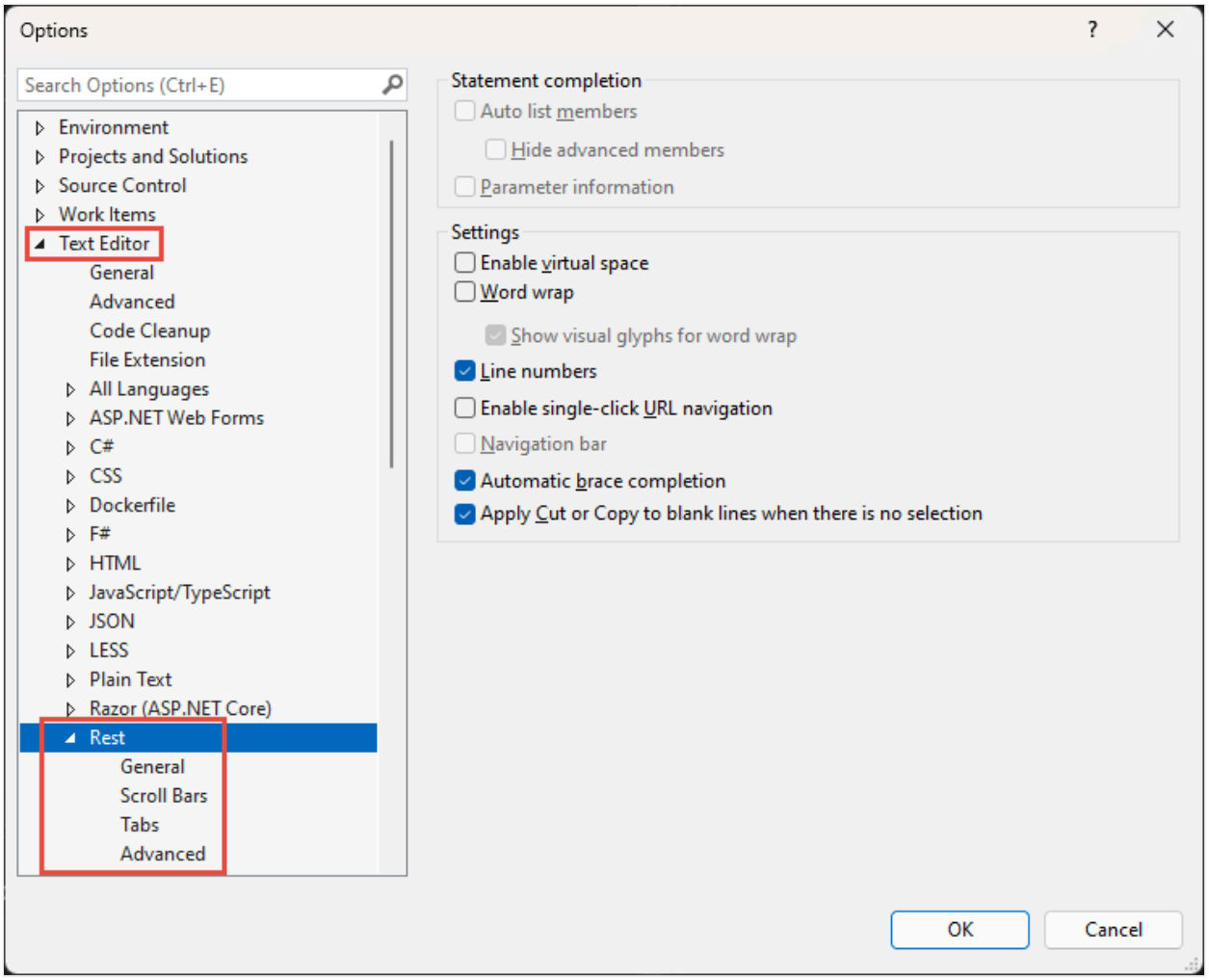
- Add at least one request to an `.http` file and save the file.
- If the request URL points to localhost and the project's port, run the project before trying to send a request to it.
- Select the `Send Request` or `Debug` link that is directly above the request to be sent.

  The request is sent to the specified URL, and the response appears in a separate pane to the right of the editor window.



`.http` file options

Some aspects of `.http` file behavior can be configured. To see what's available, go to **Tools** > **Options** > **Text Editor** > **Rest**. For example, the timeout setting can be configured on the **Advanced** tab. Here's a screenshot of the **Options** dialog:



Use Endpoints Explorer

Endpoints Explorer is a tool window that shows all the endpoints that a web API defines. The tool enables you to send requests to the endpoints by using an `.http` file.

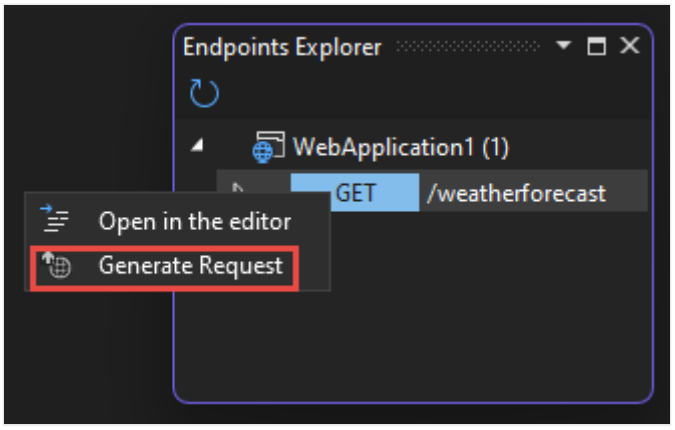
The initial set of endpoints that **Endpoints Explorer** displays are discovered statically. There are some endpoints that can't be discovered statically. For example, endpoints defined in a class library project can't be discovered until runtime. When you run or debug a web API, Visual Studio version 17.11 Preview discovers endpoints dynamically at run time also and adds those to **Endpoints Explorer**.

Open Endpoints Explorer

Select **View** > **Other Windows** > **Endpoints Explorer**.

Add a request to an `.http` file

Right-click a request in **Endpoints Explorer** and select **Generate Request**.



- If an `.http` file with the project name as the file name exists, the request is added to that file.
- Otherwise, an `.http` file is created with the project name as the file name, and the request is added to that file.

The preceding screenshot shows endpoints defined by the minimal API project template. The following example shows the request that is generated for the selected endpoint:

```http
GET {{WebApplication1_HostAddress}}/weatherforecast/
Accept: application/json

###
```

Send the request as described earlier in this article.

See also

- Endpoints Explorer window only recognizes literal strings for routes
- Web API development in Visual Studio 2022
- Visual Studio Code REST Client extension

Test web APIs with the HttpRepl

Article • 07/28/2023

The HTTP Read-Eval-Print Loop (REPL) is:

- A lightweight, cross-platform command-line tool that's supported everywhere .NET Core is supported.
- Used for making HTTP requests to test ASP.NET Core web APIs (and non-ASP.NET Core web APIs) and view their results.
- Capable of testing web APIs hosted in any environment, including localhost and Azure App Service.

The following HTTP verbs are supported:

- DELETE
- GET
- HEAD
- OPTIONS
- PATCH
- POST
- PUT

To follow along, view or download the sample ASP.NET Core web API (how to download).

Prerequisites

- .NET Core 3.1 SDK

Installation

To install the HttpRepl, run the following command:

.NET CLI

```
dotnet tool install -g Microsoft.dotnet-httprepl
```

A .NET Core Global Tool is installed from the Microsoft.dotnet-httprepl NuGet package.

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see **dotnet tool install, --arch option**. For more information, see GitHub issue **dotnet/AspNetCore.Docs #29262**.

On macOS, update the path:

```Bash
export PATH="$HOME/.dotnet/tools:$PATH"
```

Usage

After successful installation of the tool, run the following command to start the HttpRepl:

```Console
httprepl
```

To view the available HttpRepl commands, run one of the following commands:

```Console
httprepl -h
```

```Console
httprepl --help
```

The following output is displayed:

```Console
Usage:
  httprepl [<BASE_ADDRESS>] [options]

Arguments:
  <BASE_ADDRESS> - The initial base address for the REPL.

Options:
```

```
  -h|--help - Show help information.

Once the REPL starts, these commands are valid:

Setup Commands:
Use these commands to configure the tool for your API server

connect        Configures the directory structure and base address of the
api server
set header     Sets or clears a header for all requests. e.g. `set header
content-type application/json`

HTTP Commands:
Use these commands to execute requests against your application.

GET            get - Issues a GET request
POST           post - Issues a POST request
PUT            put - Issues a PUT request
DELETE         delete - Issues a DELETE request
PATCH          patch - Issues a PATCH request
HEAD           head - Issues a HEAD request
OPTIONS        options - Issues a OPTIONS request

Navigation Commands:
The REPL allows you to navigate your URL space and focus on specific APIs
that you are working on.

ls             Show all endpoints for the current path
cd             Append the given directory to the currently selected path, or
move up a path when using `cd ..`

Shell Commands:
Use these commands to interact with the REPL shell.

clear          Removes all text from the shell
echo [on/off]  Turns request echoing on or off, show the request that was
made when using request commands
exit           Exit the shell

REPL Customization Commands:
Use these commands to customize the REPL behavior.

pref [get/set] Allows viewing or changing preferences, e.g. 'pref set
editor.command.default 'C:\\Program Files\\Microsoft VS Code\\Code.exe'`
run            Runs the script at the given path. A script is a set of
commands that can be typed with one command per line
ui             Displays the Swagger UI page, if available, in the default
browser

Use `help <COMMAND>` for more detail on an individual command. e.g. `help
get`.
For detailed tool info, see https://aka.ms/http-repl-doc.
```

The HttpRepl offers command completion. Pressing the `Tab` key iterates through the list of commands that complete the characters or API endpoint that you typed. The following sections outline the available CLI commands.

Connect to the web API

Connect to a web API by running the following command:

```
Console

httprepl <ROOT URI>
```

`<ROOT URI>` is the base URI for the web API. For example:

```
Console

httprepl https://localhost:5001
```

Alternatively, run the following command at any time while the HttpRepl is running:

```
Console

connect <ROOT URI>
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001
```

Manually point to the OpenAPI description for the web API

The connect command above will attempt to find the OpenAPI description automatically. If for some reason it's unable to do so, you can specify the URI of the OpenAPI description for the web API by using the `--openapi` option:

```
Console

connect <ROOT URI> --openapi <OPENAPI DESCRIPTION ADDRESS>
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001 --openapi
/swagger/v1/swagger.json
```

Enable verbose output for details on OpenAPI description searching, parsing, and validation

Specifying the `--verbose` option with the `connect` command will produce more details when the tool searches for the OpenAPI description, parses, and validates it.

```
Console

connect <ROOT URI> --verbose
```

For example:

```
Console

(Disconnected)> connect https://localhost:5001 --verbose
Checking https://localhost:5001/swagger.json... 404 NotFound
Checking https://localhost:5001/swagger/v1/swagger.json... 404 NotFound
Checking https://localhost:5001/openapi.json... Found
Parsing... Successful (with warnings)
The field 'info' in 'document' object is REQUIRED [#/info]
The field 'paths' in 'document' object is REQUIRED [#/paths]
```

Navigate the web API

View available endpoints

To list the different endpoints (controllers) at the current path of the web API address, run the `ls` or `dir` command:

```
Console

https://localhost:5001/> ls
```

The following output format is displayed:

Console

```
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/>
```

The preceding output indicates that there are two controllers available: `Fruits` and `People`. Both controllers support parameterless HTTP GET and POST operations.

Navigating into a specific controller reveals more detail. For example, the following command's output shows the `Fruits` controller also supports HTTP GET, PUT, and DELETE operations. Each of these operations expects an `id` parameter in the route:

Console

```
https://localhost:5001/fruits> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/fruits>
```

Alternatively, run the `ui` command to open the web API's Swagger UI page in a browser. For example:

Console

```
https://localhost:5001/> ui
```

Navigate to an endpoint

To navigate to a different endpoint on the web API, run the `cd` command:

Console

```
https://localhost:5001/> cd people
```

The path following the `cd` command is case insensitive. The following output format is displayed:

Console

```
/people    [get|post]

https://localhost:5001/people>
```

Customize the HttpRepl

The HttpRepl's default colors can be customized. Additionally, a default text editor can be defined. The HttpRepl preferences are persisted across the current session and are honored in future sessions. Once modified, the preferences are stored in the following file:

Windows

%USERPROFILE%\.httpreplprefs

The *.httpreplprefs* file is loaded on startup and not monitored for changes at runtime. Manual modifications to the file take effect only after restarting the tool.

View the settings

To view the available settings, run the `pref get` command. For example:

Console

```
https://localhost:5001/> pref get
```

The preceding command displays the available key-value pairs:

Console

```
colors.json=Green
colors.json.arrayBrace=BoldCyan
colors.json.comma=BoldYellow
colors.json.name=BoldMagenta
colors.json.nameSeparator=BoldWhite
colors.json.objectBrace=Cyan
colors.protocol=BoldGreen
colors.status=BoldYellow
```

Set color preferences

Response colorization is currently supported for JSON only. To customize the default HttpRepl tool coloring, locate the key corresponding to the color to be changed. For instructions on how to find the keys, see the View the settings section. For example, change the `colors.json` key value from `Green` to `White` as follows:

Console

```
https://localhost:5001/people> pref set colors.json White
```

Only the allowed colors may be used. Subsequent HTTP requests display output with the new coloring.

When specific color keys aren't set, more generic keys are considered. To demonstrate this fallback behavior, consider the following example:

- If `colors.json.name` doesn't have a value, `colors.json.string` is used.
- If `colors.json.string` doesn't have a value, `colors.json.literal` is used.
- If `colors.json.literal` doesn't have a value, `colors.json` is used.
- If `colors.json` doesn't have a value, the command shell's default text color (`AllowedColors.None`) is used.

Set indentation size

Response indentation size customization is currently supported for JSON only. The default size is two spaces. For example:

JSON

```
[
  {
    "id": 1,
    "name": "Apple"
  },
  {
    "id": 2,
    "name": "Orange"
  },
  {
    "id": 3,
    "name": "Strawberry"
  }
]
```

To change the default size, set the `formatting.json.indentSize` key. For example, to always use four spaces:

Console

```
pref set formatting.json.indentSize 4
```

Subsequent responses honor the setting of four spaces:

JSON

```
[
    {
        "id": 1,
        "name": "Apple"
    },
    {
        "id": 2,
        "name": "Orange"
    },
    {
        "id": 3,
        "name": "Strawberry"
    }
]
```

Set the default text editor

By default, the HttpRepl has no text editor configured for use. To test web API methods requiring an HTTP request body, a default text editor must be set. The HttpRepl tool launches the configured text editor for the sole purpose of composing the request body. Run the following command to set your preferred text editor as the default:

Console

```
pref set editor.command.default "<EXECUTABLE>"
```

In the preceding command, `<EXECUTABLE>` is the full path to the text editor's executable file. For example, run the following command to set Visual Studio Code as the default text editor:

Windows

Console

```
pref set editor.command.default "C:\Program Files\Microsoft VS
Code\Code.exe"
```

To launch the default text editor with specific CLI arguments, set the `editor.command.default.arguments` key. For example, assume Visual Studio Code is the default text editor and that you always want the HttpRepl to open Visual Studio Code in a new session with extensions disabled. Run the following command:

Console

```
pref set editor.command.default.arguments "--disable-extensions --new-
window"
```

> **Tip**
>
> If your default editor is Visual Studio Code, you'll usually want to pass the `-w` or `--wait` argument to force Visual Studio Code to wait for you to close the file before returning.

Set the OpenAPI Description search paths

By default, the HttpRepl has a set of relative paths that it uses to find the OpenAPI description when executing the `connect` command without the `--openapi` option. These relative paths are combined with the root and base paths specified in the `connect` command. The default relative paths are:

- `swagger.json`
- `swagger/v1/swagger.json`
- `/swagger.json`
- `/swagger/v1/swagger.json`
- `openapi.json`
- `/openapi.json`

To use a different set of search paths in your environment, set the `swagger.searchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.searchPaths
"swagger/v2/swagger.json|swagger/v3/swagger.json"
```

Instead of replacing the default list altogether, the list can also be modified by adding or removing paths.

To add one or more search paths to the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.addToSearchPaths
"openapi/v2/openapi.json|openapi/v3/openapi.json"
```

To remove one or more search paths from the default list, set the `swagger.addToSearchPaths` preference. The value must be a pipe-delimited list of relative paths. For example:

Console

```
pref set swagger.removeFromSearchPaths "swagger.json|/swagger.json"
```

Test HTTP GET requests

Synopsis

Console

```
get <PARAMETER> [-F|--no-formatting] [-h|--header] [--response:body] [--
response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

The following options are available for the `get` command:

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP GET request:

1. Run the `get` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/people> get
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Fri, 21 Jun 2019 03:38:45 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "name": "Scott Hunter"
     },
     {
       "id": 2,
       "name": "Scott Hanselman"
   ```

```
  },
  {
    "id": 3,
    "name": "Scott Guthrie"
  }
]


https://localhost:5001/people>
```

2. Retrieve a specific record by passing a parameter to the `get` command:

```
Console

https://localhost:5001/people> get 2
```

The preceding command displays the following output format:

```
Console

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 21 Jun 2019 06:17:57 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 2,
    "name": "Scott Hanselman"
  }
]


https://localhost:5001/people>
```

Test HTTP POST requests

Synopsis

```
Console

post <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--
no-formatting] [--response] [--response:body] [--response:headers] [-s|--
streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

Indicates that no HTTP request body is needed.

Example

To issue an HTTP POST request:

1. Run the `post` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/people> post -h Content-Type=application/json
   ```

   In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

   JSON

   ```
   {
     "id": 0,
     "name": ""
   }
   ```

   > **Tip**
   >
   > To set the default text editor, see the **Set the default text editor** section.

2. Modify the JSON template to satisfy model validation requirements:

   JSON

   ```
   {
     "id": 0,
     "name": "Scott Addie"
   }
   ```

3. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

   Console

   ```
   HTTP/1.1 201 Created
   Content-Type: application/json; charset=utf-8
   Date: Thu, 27 Jun 2019 21:24:18 GMT
   ```

```
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Test HTTP PUT requests

Synopsis

Console

```
put <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Example

To issue an HTTP PUT request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
   ```

```
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `put` command on an endpoint that supports it:

```
Console

https://localhost:5001/fruits> put 2 -h Content-Type=application/json
```

In the preceding command, the `Content-Type` HTTP request header is set to indicate a request body media type of JSON. The default text editor opens a *.tmp* file with a JSON template representing the HTTP request body. For example:

```
JSON

{
  "id": 0,
  "name": ""
}
```

> **Tip**
>
> To set the default text editor, see the **Set the default text editor** section.

3. Modify the JSON template to satisfy model validation requirements:

```
JSON

{
  "id": 2,
  "name": "Cherry"
}
```

4. Save the *.tmp* file, and close the text editor. The following output appears in the command shell:

```
Console
```

```
[main 2019-06-28T17:27:01.805Z] update#setState idle
HTTP/1.1 204 No Content
Date: Fri, 28 Jun 2019 17:28:21 GMT
Server: Kestrel
```

5. *Optional*: Issue a `get` command to see the modifications. For example, if you typed "Cherry" in the text editor, a `get` returns the following output:

Console

```
https://localhost:5001/fruits> get
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Sat, 22 Jun 2019 00:08:20 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Cherry"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]


https://localhost:5001/fruits>
```

Test HTTP DELETE requests

Synopsis

Console

```
delete <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Example

To issue an HTTP DELETE request:

1. *Optional*: Run the `get` command to view the data before modifying it:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:07:32 GMT
   Server: Kestrel
   Transfer-Encoding: chunked
   ```

```
[
  {
    "id": 1,
    "data": "Apple"
  },
  {
    "id": 2,
    "data": "Orange"
  },
  {
    "id": 3,
    "data": "Strawberry"
  }
]
```

2. Run the `delete` command on an endpoint that supports it:

   Console

   ```
   https://localhost:5001/fruits> delete 2
   ```

   The preceding command displays the following output format:

   Console

   ```
   HTTP/1.1 204 No Content
   Date: Fri, 28 Jun 2019 17:36:42 GMT
   Server: Kestrel
   ```

3. *Optional*: Issue a `get` command to see the modifications. In this example, a `get` returns the following output:

   Console

   ```
   https://localhost:5001/fruits> get
   HTTP/1.1 200 OK
   Content-Type: application/json; charset=utf-8
   Date: Sat, 22 Jun 2019 00:16:30 GMT
   Server: Kestrel
   Transfer-Encoding: chunked

   [
     {
       "id": 1,
       "data": "Apple"
     },
     {
       "id": 3,
       "data": "Strawberry"
     }
   ```

```
]


https://localhost:5001/fruits>
```

Test HTTP PATCH requests

Synopsis

Console

```
patch <PARAMETER> [-c|--content] [-f|--file] [-h|--header] [--no-body] [-F|--no-formatting] [--response] [--response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

  Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't

exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

- `-c|--content`

  Provides an inline HTTP request body. For example, `-c "{\"id\":2,\"name\":\"Cherry\"}"`.

- `-f|--file`

  Provides a path to a file containing the HTTP request body. For example, `-f "C:\request.json"`.

- `--no-body`

  Indicates that no HTTP request body is needed.

Test HTTP HEAD requests

Synopsis

Console

```
head <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Test HTTP OPTIONS requests

Synopsis

Console

```
options <PARAMETER> [-F|--no-formatting] [-h|--header] [--response] [--
response:body] [--response:headers] [-s|--streaming]
```

Arguments

`PARAMETER`

The route parameter, if any, expected by the associated controller action method.

Options

- `-F|--no-formatting`

  A flag whose presence suppresses HTTP response formatting.

- `-h|--header`

Sets an HTTP request header. The following two value formats are supported:
  - `{header}={value}`
  - `{header}:{value}`

- `--response:body`

  Specifies a file to which the HTTP response body should be written. For example, `--response:body "C:\response.json"`. The file is created if it doesn't exist.

- `--response:headers`

  Specifies a file to which the HTTP response headers should be written. For example, `--response:headers "C:\response.txt"`. The file is created if it doesn't exist.

- `-s|--streaming`

  A flag whose presence enables streaming of the HTTP response.

Set HTTP request headers

To set an HTTP request header, use one of the following approaches:

- Set inline with the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> post -h Content-Type=application/json
  ```

  With the preceding approach, each distinct HTTP request header requires its own `-h` option.

- Set before sending the HTTP request. For example:

  Console

  ```
  https://localhost:5001/people> set header Content-Type application/json
  ```

  When setting the header before sending a request, the header remains set for the duration of the command shell session. To clear the header, provide an empty value. For example:

  Console

```
https://localhost:5001/people> set header Content-Type
```

Test secured endpoints

The HttpRepl supports the testing of secured endpoints in the following ways:

- Via the default credentials of the logged in user.
- Through the use of HTTP request headers.

Default credentials

Consider a web API you're testing that's hosted in IIS and secured with Windows authentication. You want the credentials of the user running the tool to flow across to the HTTP endpoints being tested. To pass the default credentials of the logged in user:

1. Set the `httpClient.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

Default proxy credentials

Consider a scenario in which the web API you're testing is behind a proxy secured with Windows authentication. You want the credentials of the user running the tool to flow to the proxy. To pass the default credentials of the logged in user:

1. Set the `httpClient.proxy.useDefaultCredentials` preference to `true`:

   Console

   ```
   pref set httpClient.proxy.useDefaultCredentials true
   ```

2. Exit and restart the tool before sending another request to the web API.

HTTP request headers

Examples of supported authentication and authorization schemes include:

- basic authentication
- JWT bearer tokens
- digest authentication

For example, you can send a bearer token to an endpoint with the following command:

Console

```
set header Authorization "bearer <TOKEN VALUE>"
```

To access an Azure-hosted endpoint or to use the Azure REST API, you need a bearer token. Use the following steps to obtain a bearer token for your Azure subscription via the Azure CLI. The HttpRepl sets the bearer token in an HTTP request header. A list of Azure App Service Web Apps is retrieved.

1. Sign in to Azure:

   Azure CLI

   ```
   az login
   ```

2. Get your subscription ID with the following command:

   Azure CLI

   ```
   az account show --query id
   ```

3. Copy your subscription ID and run the following command:

   Azure CLI

   ```
   az account set --subscription "<SUBSCRIPTION ID>"
   ```

4. Get your bearer token with the following command:

   Azure CLI

   ```
   az account get-access-token --query accessToken
   ```

5. Connect to the Azure REST API via the HttpRepl:

   Console

```
httprepl https://management.azure.com
```

6. Set the `Authorization` HTTP request header:

```
Console

https://management.azure.com/> set header Authorization "bearer <ACCESS
TOKEN>"
```

7. Navigate to the subscription:

```
Console

https://management.azure.com/> cd subscriptions/<SUBSCRIPTION ID>
```

8. Get a list of your subscription's Azure App Service Web Apps:

```
Console

https://management.azure.com/subscriptions/{SUBSCRIPTION ID}> get
providers/Microsoft.Web/sites?api-version=2016-08-01
```

The following response is displayed:

```
Console

HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Length: 35948
Content-Type: application/json; charset=utf-8
Date: Thu, 19 Sep 2019 23:04:03 GMT
Expires: -1
Pragma: no-cache
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
x-ms-correlation-request-id: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx</em>
x-ms-original-request-ids: <em>xxxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxxx;xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx</em>
x-ms-ratelimit-remaining-subscription-reads: 11999
x-ms-request-id: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
x-ms-routing-request-id: WESTUS:xxxxxxxxxxxxxxxx:xxxxxxxx-xxxx-xxxx-
xxxx-xxxxxxxxxx
{
  "value": [
    <AZURE RESOURCES LIST>
```

```
  ]
}
```

Toggle HTTP request display

By default, display of the HTTP request being sent is suppressed. It's possible to change the corresponding setting for the duration of the command shell session.

Enable request display

View the HTTP request being sent by running the `echo on` command. For example:

Console

```
https://localhost:5001/people> echo on
Request echoing is on
```

Subsequent HTTP requests in the current session display the request headers. For example:

Console

```
https://localhost:5001/people> post

[main 2019-06-28T18:50:11.930Z] update#setState idle
Request to https://localhost:5001...

POST /people HTTP/1.1
Content-Length: 41
Content-Type: application/json
User-Agent: HTTP-REPL

{
  "id": 0,
  "name": "Scott Addie"
}

Response from https://localhost:5001...

HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Date: Fri, 28 Jun 2019 18:50:21 GMT
Location: https://localhost:5001/people/4
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 4,
```

```
  "name": "Scott Addie"
}


https://localhost:5001/people>
```

Disable request display

Suppress display of the HTTP request being sent by running the `echo off` command. For example:

Console

```
https://localhost:5001/people> echo off
Request echoing is off
```

Run a script

If you frequently execute the same set of HttpRepl commands, consider storing them in a text file. Commands in the file take the same form as commands executed manually on the command line. The commands can be executed in a batched fashion using the `run` command. For example:

1. Create a text file containing a set of newline-delimited commands. To illustrate, consider a *people-script.txt* file containing the following commands:

   text

   ```
   set base https://localhost:5001
   ls
   cd People
   ls
   get 1
   ```

2. Execute the `run` command, passing in the text file's path. For example:

   Console

   ```
   https://localhost:5001/> run C:\http-repl-scripts\people-script.txt
   ```

   The following output appears:

   Console

```
https://localhost:5001/> set base https://localhost:5001
Using OpenAPI description at
https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]

https://localhost:5001/> cd People
/People    [get|post]

https://localhost:5001/People> ls
.      [get|post]
..     []
{id}   [get|put|delete]

https://localhost:5001/People> get 1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Date: Fri, 12 Jul 2019 19:20:10 GMT
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "name": "Scott Hunter"
}


https://localhost:5001/People>
```

Clear the output

To remove all output written to the command shell by the HttpRepl tool, run the `clear` or `cls` command. To illustrate, imagine the command shell contains the following output:

Console

```
httprepl https://localhost:5001
(Disconnected)> set base "https://localhost:5001"
Using OpenAPI description at https://localhost:5001/swagger/v1/swagger.json

https://localhost:5001/> ls
.        []
Fruits   [get|post]
People   [get|post]
```

```
https://localhost:5001/>
```

Run the following command to clear the output:

Console

```
https://localhost:5001/> clear
```

After running the preceding command, the command shell contains only the following output:

Console

```
https://localhost:5001/>
```

Additional resources

- REST API requests
- HttpRepl GitHub repository
- Configure Visual Studio to launch HttpRepl
- Configure Visual Studio Code to launch HttpRepl
- Configure Visual Studio for Mac to launch HttpRepl

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

HttpRepl telemetry

Article • 04/11/2023

The HttpRepl includes a telemetry feature that collects usage data. It's important that the HttpRepl team understands how the tool is used so it can be improved.

How to opt out

The HttpRepl telemetry feature is enabled by default. To opt out of the telemetry feature, set the `DOTNET_HTTPREPL_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

Disclosure

The HttpRepl displays text similar to the following when you first run the tool. Text may vary slightly depending on the version of the tool you're running. This "first run" experience is how Microsoft notifies you about data collection.

Console

```
Telemetry
---------
The .NET tools collect usage data in order to help us improve your
experience. It is collected by Microsoft and shared with the community. You
can opt-out of telemetry by setting the DOTNET_HTTPREPL_TELEMETRY_OPTOUT
environment variable to '1' or 'true' using your favorite shell.
```

To suppress the "first run" experience text, set the `DOTNET_HTTPREPL_SKIP_FIRST_TIME_EXPERIENCE` environment variable to `1` or `true`.

Data points

The telemetry feature doesn't:

- Collect personal data, such as usernames, email addresses, or URLs.
- Scan your HTTP requests or responses.

The data is sent securely to Microsoft servers and held under restricted access.

Protecting your privacy is important to us. If you suspect the telemetry feature is collecting sensitive data or the data is being insecurely or inappropriately handled, take one of the following actions:

- File an issue in the dotnet/httprepl repository.
- Send an email to dotnet@microsoft.com for investigation.

The telemetry feature collects the following data.

Expand table

.NET SDK versions	Data
>=5.0	Timestamp of invocation.
>=5.0	Three-octet IP address used to determine the geographical location.
>=5.0	Operating system and version.
>=5.0	Runtime ID (RID) the tool is running on.
>=5.0	Whether the tool is running in a container.
>=5.0	Hashed Media Access Control (MAC) address: a cryptographically (SHA256) hashed and unique ID for a machine.
>=5.0	Kernel version.
>=5.0	HttpRepl version.
>=5.0	Whether the tool was started with `help`, `run`, or `connect` arguments. Actual argument values aren't collected.
>=5.0	Command invoked (for example, `get`) and whether it succeeded.
>=5.0	For the `connect` command, whether the `root`, `base`, or `openapi` arguments were supplied. Actual argument values aren't collected.
>=5.0	For the `pref` command, whether a `get` or `set` was issued and which preference was accessed. If not a well-known preference, the name is hashed. The value isn't collected.
>=5.0	For the `set header` command, the header name being set. If not a well-known header, the name is hashed. The value isn't collected.
>=5.0	For the `connect` command, whether a special case for `dotnet new webapi` was used and, whether it was bypassed via preference.
>=5.0	For all HTTP commands (for example, GET, POST, PUT), whether each of the options was specified. The values of the options aren't collected.

Additional resources

- .NET Core SDK telemetry
- .NET CLI telemetry data

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Unit and integration tests in Minimal API apps

Article • 07/26/2024

By Fiyaz Bin Hasan, and Rick Anderson

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test, choose the unit test.

In discussions of integration tests, the tested project is frequently called the ***System Under Test***, or "**SUT**" for short. "SUT" is used throughout this article to refer to the ASP.NET Core app being tested.

Don't write integration tests for every permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with

databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes or mocks result in faster test execution.

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (`.deps`) from the SUT into the test project's `bin` directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.

- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

Separate unit tests from integration tests into different projects. Separating the tests:

- Helps ensure that infrastructure testing components aren't accidentally included in the unit tests.
- Allows control over which set of tests are run.

The sample code on GitHub provides an example of unit and integration tests on a Minimal API app.

IResult implementation types

Public IResult implementation types in the Microsoft.AspNetCore.Http.HttpResults namespace can be used to unit test minimal route handlers when using named methods instead of lambdas.

The following code uses the NotFound<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsNotFoundIfNotExists()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var notFoundResult = (NotFound) result.Result;

    Assert.NotNull(notFoundResult);
}
```

The following code uses the Ok<TValue> class:

C#

```
[Fact]
public async Task GetTodoReturnsTodoFromDatabase()
{
    // Arrange
    await using var context = new MockDb().CreateDbContext();

    context.Todos.Add(new Todo
    {
        Id = 1,
        Title = "Test title",
        Description = "Test description",
        IsDone = false
    });

    await context.SaveChangesAsync();

    // Act
    var result = await TodoEndpointsV1.GetTodo(1, context);

    //Assert
    Assert.IsType<Results<Ok<Todo>, NotFound>>(result);

    var okResult = (Ok<Todo>)result.Result;

    Assert.NotNull(okResult.Value);
    Assert.Equal(1, okResult.Value.Id);
}
```

Additional Resources

- Basic authentication tests is not a .NET repository but was written by a member of the .NET team. It provides examples of basic authentication testing.
- View or download sample code
- Authentication and authorization in minimal APIs
- Use port tunneling Visual Studio to debug web APIs
- Test controller logic in ASP.NET Core
- Razor Pages unit tests in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Test Razor components in ASP.NET Core Blazor

Article • 11/12/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Egil Hansen

Testing Razor components is an important aspect of releasing stable and maintainable Blazor apps.

To test a Razor component, the *component under test* (CUT) is:

- Rendered with relevant input for the test.
- Depending on the type of test performed, possibly subject to interaction or modification. For example, event handlers can be triggered, such as an `onclick` event for a button.
- Inspected for expected values. A test passes when one or more inspected values matches the expected values for the test.

Test approaches

Two common approaches for testing Razor components are end-to-end (E2E) testing and unit testing:

- **Unit testing**: Unit tests are written with a unit testing library that provides:
  - Component rendering.
  - Inspection of component output and state.
  - Triggering of event handlers and life cycle methods.
  - Assertions that component behavior is correct.

  bUnit is an example of a library that enables Razor component unit testing.

- **E2E testing**: A test runner runs a Blazor app containing the CUT and automates a browser instance. The testing tool inspects and interacts with the CUT through the browser. Playwright for .NET is an example of an E2E testing framework that can be used with Blazor apps.

In unit testing, only the Razor component (Razor/C#) is involved. External dependencies, such as services and JS interop, must be mocked. In E2E testing, the Razor component and all of its auxiliary infrastructure are part of the test, including CSS, JS, and the DOM and browser APIs.

Test scope describes how extensive the tests are. Test scope typically has an influence on the speed of the tests. Unit tests run on a subset of the app's subsystems and usually execute in milliseconds. E2E tests, which test a broad group of the app's subsystems, can take several seconds to complete.

Unit testing also provides access to the instance of the CUT, allowing for inspection and verification of the component's internal state. This normally isn't possible in E2E testing.

With regard to the component's environment, E2E tests must make sure that the expected environmental state has been reached before verification starts. Otherwise, the result is unpredictable. In unit testing, the rendering of the CUT and the life cycle of the test are more integrated, which improves test stability.

E2E testing involves launching multiple processes, network and disk I/O, and other subsystem activity that often lead to poor test reliability. Unit tests are typically insulated from these sorts of issues.

The following table summarizes the difference between the two testing approaches.

Expand table

Capability	Unit testing	E2E testing
Test scope	Razor component (Razor/C#) only	Razor component (Razor/C#) with CSS/JS
Test execution time	Milliseconds	Seconds
Access to the component instance	Yes	No
Sensitive to the environment	No	Yes
Reliability	More reliable	Less reliable

Choose the most appropriate test approach

Consider the scenario when choosing the type of testing to perform. Some considerations are described in the following table.

Expand table

Scenario	Suggested approach	Remarks
Component without JS interop logic	Unit testing	When there's no dependency on JS interop in a Razor component, the component can be tested without access to JS or the DOM API. In this scenario, there are no disadvantages to choosing unit testing.
Component with simple JS interop logic	Unit testing	It's common for components to query the DOM or trigger animations through JS interop. Unit testing is usually preferred in this scenario, since it's straightforward to mock the JS interaction through the IJSRuntime interface.
Component that depends on complex JS code	Unit testing and separate JS testing	If a component uses JS interop to call a large or complex JS library but the interaction between the Razor component and JS library is simple, then the best approach is likely to treat the component and JS library or code as two separate parts and test each individually. Test the Razor component with a unit testing library, and test the JS with a JS testing library.
Component with logic that depends on JS manipulation of the browser DOM	E2E testing	When a component's functionality is dependent on JS and its manipulation of the DOM, verify both the JS and Blazor code together in an E2E test. This is the approach that the Blazor framework developers have taken with Blazor's browser rendering logic, which has tightly-coupled C# and JS code. The C# and JS code must work together to correctly render Razor components in a browser.
Component that depends on 3rd party class library with hard-to-mock dependencies	E2E testing	When a component's functionality is dependent on a 3rd party class library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component.

Test components with bUnit

There's no official Microsoft testing framework for Blazor, but the community-driven project bUnit provides a convenient way to unit test Razor components.

> **Note**
>
> bUnit is a third-party testing library and isn't supported or maintained by Microsoft.

bUnit works with general-purpose testing frameworks, such as MSTest, NUnit, and xUnit. These testing frameworks make bUnit tests look and feel like regular unit tests. bUnit tests integrated with a general-purpose testing framework are ordinarily executed with:

- Visual Studio's Test Explorer.
- dotnet test CLI command in a command shell.
- An automated DevOps testing pipeline.

> **Note**
>
> Test concepts and test implementations across different test frameworks are similar but not identical. Refer to the test framework's documentation for guidance.

The following demonstrates the structure of a bUnit test on the `Counter` component in an app based on a Blazor project template. The `Counter` component displays and increments a counter based on the user selecting a button in the page:

razor

```razor
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

The following bUnit test verifies that the CUT's counter is incremented correctly when the button is selected:

razor

```razor
@code {
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
```

```
        using var ctx = new TestContext();
        var cut = ctx.Render(@<Counter />);
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElm.MarkupMatches("Current count: 1");
    }
}
```

Tests can also be written in a C# class file:

C#

```
public class CounterTests
{
    [Fact]
    public void CounterShouldIncrementWhenClicked()
    {
        // Arrange
        using var ctx = new TestContext();
        var cut = ctx.RenderComponent<Counter>();
        var paraElm = cut.Find("p");

        // Act
        cut.Find("button").Click();

        // Assert
        var paraElmText = paraElm.TextContent;
        paraElmText.MarkupMatches("Current count: 1");
    }
}
```

The following actions take place at each step of the test:

- *Arrange*: The `Counter` component is rendered using bUnit's `TestContext`. The CUT's paragraph element (`<p>`) is found and assigned to `paraElm`. In Razor syntax, a component can be passed as a RenderFragment to bUnit.
- *Act*: The button's element (`<button>`) is located and selected by calling `Click`, which should increment the counter and update the content of the paragraph tag (`<p>`). The paragraph element text content is obtained by calling `TextContent`.
- *Assert*: `MarkupMatches` is called on the text content to verify that it matches the expected string, which is `Current count: 1`.

> **Note**
>
> The `MarkupMatches` assert method differs from a regular string comparison assertion (for example, `Assert.Equal("Current count: 1", paraElmText);`). `MarkupMatches` performs a semantic comparison of the input and expected HTML markup. A semantic comparison is aware of HTML semantics, meaning things like insignificant whitespace is ignored. This results in more stable tests. For more information, see **Customizing the Semantic HTML Comparison**.

Additional resources

- Getting Started with bUnit: bUnit instructions include guidance on creating a test project, referencing testing framework packages, and building and running tests.
- Blazor Testing from A to Z - Egil Hansen - Swetugg Stockholm 2024 (Swetugg)

Razor Pages unit tests in ASP.NET Core

Article • 09/17/2023

ASP.NET Core supports unit tests of Razor Pages apps. Tests of the data access layer (DAL) and page models help ensure:

- Parts of a Razor Pages app work independently and together as a unit during app construction.
- Classes and methods have limited scopes of responsibility.
- Additional documentation exists on how the app should behave.
- Regressions, which are errors brought about by updates to the code, are found during automated building and deployment.

This topic assumes that you have a basic understanding of Razor Pages apps and unit tests. If you're unfamiliar with Razor Pages apps or test concepts, see the following topics:

- Introduction to Razor Pages in ASP.NET Core
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Unit testing C# in .NET Core using dotnet test and xUnit

View or download sample code (how to download)

The sample project is composed of two apps:

App	Project folder	Description
Message app	*src/RazorPagesTestSample*	Allows a user to add a message, delete one message, delete all messages, and analyze messages (find the average number of words per message).
Test app	*tests/RazorPagesTestSample.Tests*	Used to unit test the DAL and Index page model of the message app.

The tests can be run using the built-in test features of an IDE, such as Visual Studio. If using Visual Studio Code or the command line, execute the following command at a command prompt in the *tests/RazorPagesTestSample.Tests* folder:

.NET CLI

```
dotnet test
```

Message app organization

The message app is a Razor Pages message system with the following characteristics:

- The Index page of the app (`Pages/Index.cshtml` and `Pages/Index.cshtml.cs`) provides a UI and page model methods to control the addition, deletion, and analysis of messages (find the average number of words per message).
- A message is described by the `Message` class (`Data/Message.cs`) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.
- The app contains a DAL in its database context class, `AppDbContext` (`Data/AppDbContext.cs`). The DAL methods are marked `virtual`, which allows mocking the methods for use in the tests.
- If the database is empty on app startup, the message store is initialized with three messages. These *seeded messages* are also used in tests.

†The EF topic, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the sample app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic in ASP.NET Core (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the *tests/RazorPagesTestSample.Tests* folder.

Test app folder	Description
UnitTests	• `DataAccessLayerTest.cs` contains the unit tests for the DAL. • `IndexPageTests.cs` contains the unit tests for the Index page model.
Utilities	Contains the `TestDbContextOptions` method used to create new database context options for each DAL unit test so that the database is reset to its baseline condition for each test.

The test framework is xUnit. The object mocking framework is Moq.

Unit tests of the data access layer (DAL)

The message app has a DAL with four methods contained in the `AppDbContext` class (`src/RazorPagesTestSample/Data/AppDbContext.cs`). Each method has one or two unit tests in the test app.

DAL method	Function
`GetMessagesAsync`	Obtains a `List<Message>` from the database sorted by the `Text` property.
`AddMessageAsync`	Adds a `Message` to the database.
`DeleteAllMessagesAsync`	Deletes all `Message` entries from the database.
`DeleteMessageAsync`	Deletes a single `Message` from the database by `Id`.

Unit tests of the DAL require DbContextOptions when creating a new `AppDbContext` for each test. One approach to creating the `DbContextOptions` for each test is to use a DbContextOptionsBuilder:

C#

```
var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
    .UseInMemoryDatabase("InMemoryDb");

using (var db = new AppDbContext(optionsBuilder.Options))
{
    // Use the db here in the unit test.
}
```

The problem with this approach is that each test receives the database in whatever state the previous test left it. This can be problematic when trying to write atomic unit tests that don't interfere with each other. To force the `AppDbContext` to use a new database context for each test, supply a `DbContextOptions` instance that's based on a new service provider. The test app shows how to do this using its `Utilities` class method `TestDbContextOptions` (`tests/RazorPagesTestSample.Tests/Utilities/Utilities.cs`):

C#

```
public static DbContextOptions<AppDbContext> TestDbContextOptions()
{
    // Create a new service provider to create a new in-memory database.
    var serviceProvider = new ServiceCollection()
        .AddEntityFrameworkInMemoryDatabase()
        .BuildServiceProvider();
```

```
    // Create a new options instance using an in-memory database and
    // IServiceProvider that the context should resolve all of its
    // services from.
    var builder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb")
        .UseInternalServiceProvider(serviceProvider);

    return builder.Options;
}
```

Using the `DbContextOptions` in the DAL unit tests allows each test to run atomically with a fresh database instance:

C#

```
using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
{
    // Use the db here in the unit test.
}
```

Each test method in the `DataAccessLayerTest` class (`UnitTests/DataAccessLayerTest.cs`) follows a similar Arrange-Act-Assert pattern:

1. Arrange: The database is configured for the test and/or the expected outcome is defined.
2. Act: The test is executed.
3. Assert: Assertions are made to determine if the test result is a success.

For example, the `DeleteMessageAsync` method is responsible for removing a single message identified by its `Id` (`src/RazorPagesTestSample/Data/AppDbContext.cs`):

C#

```
public async virtual Task DeleteMessageAsync(int id)
{
    var message = await Messages.FindAsync(id);

    if (message != null)
    {
        Messages.Remove(message);
        await SaveChangesAsync();
    }
}
```

There are two tests for this method. One test checks that the method deletes a message when the message is present in the database. The other method tests that the database

doesn't change if the message `Id` for deletion doesn't exist. The `DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound` method is shown below:

C#

```
[Fact]
public async Task DeleteMessageAsync_MessageIsDeleted_WhenMessageIsFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var seedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(seedMessages);
        await db.SaveChangesAsync();
        var recId = 1;
        var expectedMessages =
            seedMessages.Where(message => message.Id != recId).ToList();

        // Act
        await db.DeleteMessageAsync(recId);

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

First, the method performs the Arrange step, where preparation for the Act step takes place. The seeding messages are obtained and held in `seedMessages`. The seeding messages are saved into the database. The message with an `Id` of `1` is set for deletion. When the `DeleteMessageAsync` method is executed, the expected messages should have all of the messages except for the one with an `Id` of `1`. The `expectedMessages` variable represents this expected outcome.

C#

```
// Arrange
var seedMessages = AppDbContext.GetSeedingMessages();
await db.AddRangeAsync(seedMessages);
await db.SaveChangesAsync();
var recId = 1;
var expectedMessages =
    seedMessages.Where(message => message.Id != recId).ToList();
```

The method acts: The `DeleteMessageAsync` method is executed passing in the `recId` of `1`:

C#

```
// Act
await db.DeleteMessageAsync(recId);
```

Finally, the method obtains the `Messages` from the context and compares it to the `expectedMessages` asserting that the two are equal:

C#

```
// Assert
var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

In order to compare that the two `List<Message>` are the same:

- The messages are ordered by `Id`.
- Message pairs are compared on the `Text` property.

A similar test method, `DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound` checks the result of attempting to delete a message that doesn't exist. In this case, the expected messages in the database should be equal to the actual messages after the `DeleteMessageAsync` method is executed. There should be no change to the database's content:

C#

```
[Fact]
public async Task
DeleteMessageAsync_NoMessageIsDeleted_WhenMessageIsNotFound()
{
    using (var db = new AppDbContext(Utilities.TestDbContextOptions()))
    {
        // Arrange
        var expectedMessages = AppDbContext.GetSeedingMessages();
        await db.AddRangeAsync(expectedMessages);
        await db.SaveChangesAsync();
        var recId = 4;

        // Act
        try
        {
            await db.DeleteMessageAsync(recId);
        }
        catch
        {
```

```
            // recId doesn't exist
        }

        // Assert
        var actualMessages = await db.Messages.AsNoTracking().ToListAsync();
        Assert.Equal(
            expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
            actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
    }
}
```

Unit tests of the page model methods

Another set of unit tests is responsible for tests of page model methods. In the message app, the Index page models are found in the `IndexModel` class in `src/RazorPagesTestSample/Pages/Index.cshtml.cs`.

Page model method	Function
`OnGetAsync`	Obtains the messages from the DAL for the UI using the `GetMessagesAsync` method.
`OnPostAddMessageAsync`	If the ModelState is valid, calls `AddMessageAsync` to add a message to the database.
`OnPostDeleteAllMessagesAsync`	Calls `DeleteAllMessagesAsync` to delete all of the messages in the database.
`OnPostDeleteMessageAsync`	Executes `DeleteMessageAsync` to delete a message with the `Id` specified.
`OnPostAnalyzeMessagesAsync`	If one or more messages are in the database, calculates the average number of words per message.

The page model methods are tested using seven tests in the `IndexPageTests` class (`tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs`). The tests use the familiar Arrange-Assert-Act pattern. These tests focus on:

- Determining if the methods follow the correct behavior when the ModelState is invalid.
- Confirming the methods produce the correct IActionResult.
- Checking that property value assignments are made correctly.

This group of tests often mock the methods of the DAL to produce expected data for the Act step where a page model method is executed. For example, the `GetMessagesAsync` method of the `AppDbContext` is mocked to produce output. When a

page model method executes this method, the mock returns the result. The data doesn't come from the database. This creates predictable, reliable test conditions for using the DAL in the page model tests.

The `OnGetAsync_PopulatesThePageModel_WithAListOfMessages` test shows how the `GetMessagesAsync` method is mocked for the page model:

```
C#

var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
var expectedMessages = AppDbContext.GetSeedingMessages();
mockAppDbContext.Setup(
    db => db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
var pageModel = new IndexModel(mockAppDbContext.Object);
```

When the `OnGetAsync` method is executed in the Act step, it calls the page model's `GetMessagesAsync` method.

Unit test Act step (`tests/RazorPagesTestSample.Tests/UnitTests/IndexPageTests.cs`):

```
C#

// Act
await pageModel.OnGetAsync();
```

`IndexPage` page model's `OnGetAsync` method (`src/RazorPagesTestSample/Pages/Index.cshtml.cs`):

```
C#

public async Task OnGetAsync()
{
    Messages = await _db.GetMessagesAsync();
}
```

The `GetMessagesAsync` method in the DAL doesn't return the result for this method call. The mocked version of the method returns the result.

In the `Assert` step, the actual messages (`actualMessages`) are assigned from the `Messages` property of the page model. A type check is also performed when the messages are assigned. The expected and actual messages are compared by their `Text` properties. The test asserts that the two `List<Message>` instances contain the same messages.

C#

```
// Assert
var actualMessages = Assert.IsAssignableFrom<List<Message>>
(pageModel.Messages);
Assert.Equal(
    expectedMessages.OrderBy(m => m.Id).Select(m => m.Text),
    actualMessages.OrderBy(m => m.Id).Select(m => m.Text));
```

Other tests in this group create page model objects that include the DefaultHttpContext, the ModelStateDictionary, an ActionContext to establish the `PageContext`, a `ViewDataDictionary`, and a `PageContext`. These are useful in conducting tests. For example, the message app establishes a `ModelState` error with AddModelError to check that a valid PageResult is returned when `OnPostAddMessageAsync` is executed:

C#

```
[Fact]
public async Task
OnPostAddMessageAsync_ReturnsAPageResult_WhenModelStateIsInvalid()
{
    // Arrange
    var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>()
        .UseInMemoryDatabase("InMemoryDb");
    var mockAppDbContext = new Mock<AppDbContext>(optionsBuilder.Options);
    var expectedMessages = AppDbContext.GetSeedingMessages();
    mockAppDbContext.Setup(db =>
db.GetMessagesAsync()).Returns(Task.FromResult(expectedMessages));
    var httpContext = new DefaultHttpContext();
    var modelState = new ModelStateDictionary();
    var actionContext = new ActionContext(httpContext, new RouteData(), new
PageActionDescriptor(), modelState);
    var modelMetadataProvider = new EmptyModelMetadataProvider();
    var viewData = new ViewDataDictionary(modelMetadataProvider,
modelState);
    var tempData = new TempDataDictionary(httpContext,
Mock.Of<ITempDataProvider>());
    var pageContext = new PageContext(actionContext)
    {
        ViewData = viewData
    };
    var pageModel = new IndexModel(mockAppDbContext.Object)
    {
        PageContext = pageContext,
        TempData = tempData,
        Url = new UrlHelper(actionContext)
    };
    pageModel.ModelState.AddModelError("Message.Text", "The Text field is
required.");

    // Act
```

```
        var result = await pageModel.OnPostAddMessageAsync();

        // Assert
        Assert.IsType<PageResult>(result);
    }
```

Additional resources

- Unit testing C# in .NET Core using dotnet test and xUnit
- Test controller logic in ASP.NET Core
- Unit Test Your Code (Visual Studio)
- Integration tests in ASP.NET Core
- xUnit.net
- Getting started with xUnit.net: Using .NET Core with the .NET SDK command line
- Moq
- Moq Quickstart

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Unit test controller logic in ASP.NET Core

Article • 09/27/2022

By Steve Smith

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Unit testing controllers

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app.

View or download sample code (how to download)

The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

C#

```
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new
StormSessionViewModel()
```

```
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.
- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

C#

```
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

C#

```
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
  - The `Add` method on the repository is called.
  - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

C#

```
[Fact]
public async Task
IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task
IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);
```

```
    // Assert
    var redirectToActionResult = Assert.IsType<RedirectToActionResult>
(result);
    Assert.Null(redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
    mockRepo.Verify();
}
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> **Note**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (`Controllers/SessionController.cs`):

C#

```
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }
```

```
    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

C#

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task
IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
```

```
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var contentResult = Assert.IsType<ContentResult>(result);
    Assert.Equal("Session not found.", contentResult.Content);
}

[Fact]
public async Task IndexReturnsViewResultWithStormSessionViewModel()
{
    // Arrange
    int testSessionId = 1;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSessions().FirstOrDefault(
            s => s.Id == testSessionId));
    var controller = new SessionController(mockRepo.Object);

    // Act
    var result = await controller.Index(testSessionId);

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsType<StormSessionViewModel>(
        viewResult.ViewData.Model);
    Assert.Equal("Test One", model.Name);
    Assert.Equal(2, model.DateCreated.Day);
    Assert.Equal(testSessionId, model.Id);
}
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

C#

```
[HttpGet("forsession/{sessionId}")]
public async Task<IActionResult> ForSession(int sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);
    if (session == null)
    {
        return NotFound(sessionId);
```

```
        }

        var result = session.Ideas.Select(idea => new IdeaDTO()
        {
            Id = idea.Id,
            Name = idea.Name,
            Description = idea.Description,
            DateCreated = idea.DateCreated
        }).ToList();

        return Ok(result);
    }

    [HttpPost("create")]
    public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        var session = await _sessionRepository.GetByIdAsync(model.SessionId);
        if (session == null)
        {
            return NotFound(model.SessionId);
        }

        var idea = new Idea()
        {
            DateCreated = DateTimeOffset.Now,
            Description = model.Description,
            Name = model.Name
        };
        session.AddIdea(idea);

        await _sessionRepository.UpdateAsync(session);

        return Ok(session);
    }
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).

- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

C#

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

C#

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
```

```
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState`:

C#

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null`, so the mock repository is configured to return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

C#

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
```

```
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

C#

```
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

ActionResult<T> (ActionResult<TValue>) can return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

C#

```
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int
sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

C#

```
[Fact]
public async Task
ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
```

```
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await
controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

C#

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

C#

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>>
CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id
}, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

C#

```
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
```

```
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

C#

```
[Fact]
public async Task
CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

C#

```
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>
(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>
(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnValue.Ideas.LastOrDefault().Description);
}
```

Additional resources

- Integration tests in ASP.NET Core
- Create and run unit tests with Visual Studio
- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)

- JustMockLite: A mocking framework for .NET developers. (*Not maintained or supported by Microsoft.*)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Integration tests in ASP.NET Core

Article • 02/29/2024

By Jos van der Til, Martin Costello, and Javier Calvarro Nelson.

Integration tests ensure that an app's components function correctly at a level that includes the app's supporting infrastructure, such as the database, file system, and network. ASP.NET Core supports integration tests using a unit test framework with a test web host and an in-memory test server.

This article assumes a basic understanding of unit tests. If unfamiliar with test concepts, see the Unit Testing in .NET Core and .NET Standard article and its linked content.

View or download sample code (how to download)

The sample app is a Razor Pages app and assumes a basic understanding of Razor Pages. If you're unfamiliar with Razor Pages, see the following articles:

- Introduction to Razor Pages
- Get started with Razor Pages
- Razor Pages unit tests

For testing SPAs, we recommend a tool such as Playwright for .NET, which can automate a browser.

Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test, choose the unit test.

In discussions of integration tests, the tested project is frequently called the ***System Under Test***, or "**SUT**" for short. "SUT" is used throughout this article to refer to the ASP.NET Core app being tested.

Don't write integration tests for every permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes or mocks result in faster test execution.

ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (`.deps`) from the SUT into the test project's `bin` directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

Separate unit tests from integration tests into different projects. Separating the tests:

- Helps ensure that infrastructure testing components aren't accidentally included in the unit tests.
- Allows control over which set of tests are run.

There's virtually no difference between the configuration for tests of Razor Pages apps and MVC apps. The only difference is in how the tests are named. In a Razor Pages app, tests of page endpoints are usually named after the page model class (for example, `IndexPageTests` to test component integration for the Index page). In an MVC app, tests are usually organized by controller classes and named after the controllers they test (for example, `HomeControllerTests` to test component integration for the Home controller).

Test app prerequisites

The test project must:

- Reference the Microsoft.AspNetCore.Mvc.Testing package.
- Specify the Web SDK in the project file (`<Project Sdk="Microsoft.NET.Sdk.Web">`).

These prerequisites can be seen in the sample app. Inspect the `tests/RazorPagesProject.Tests/RazorPagesProject.Tests.csproj` file. The sample app

uses the xUnit test framework and the AngleSharp parser library, so the sample app also references:

- AngleSharp
- xunit
- xunit.runner.visualstudio

In apps that use xunit.runner.visualstudio version 2.4.2 or later, the test project must reference the Microsoft.NET.Test.Sdk package.

Entity Framework Core is also used in the tests. See the project file in GitHub.

SUT environment

If the SUT's environment isn't set, the environment defaults to Development.

Basic tests with the default WebApplicationFactory

Expose the implicitly defined `Program` class to the test project by doing one of the following:

- Expose internal types from the web app to the test project. This can be done in the SUT project's file (`.csproj`):

  XML

  ```
  <ItemGroup>
       <InternalsVisibleTo Include="MyTestProject" />
  </ItemGroup>
  ```

- Make the Program class public using a partial class declaration:

  diff

  ```
  var builder = WebApplication.CreateBuilder(args);
  // ... Configure services, routes, etc.
  app.Run();
  + public partial class Program { }
  ```

  The sample app uses the `Program` partial class approach.

WebApplicationFactory<TEntryPoint> is used to create a TestServer for the integration tests. `TEntryPoint` is the entry point class of the SUT, usually `Program.cs`.

Test classes implement a *class fixture* interface (IClassFixture) to indicate the class contains tests and provide shared object instances across the tests in the class.

The following test class, `BasicTests`, uses the `WebApplicationFactory` to bootstrap the SUT and provide an HttpClient to a test method, `Get_EndpointsReturnSuccessAndCorrectContentType`. The method verifies the response status code is successful (200-299) and the `Content-Type` header is `text/html; charset=utf-8` for several app pages.

CreateClient() creates an instance of `HttpClient` that automatically follows redirects and handles cookies.

C#

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly WebApplicationFactory<Program> _factory;

    public BasicTests(WebApplicationFactory<Program> factory)
    {
        _factory = factory;
    }

    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    [InlineData("/Privacy")]
    [InlineData("/Contact")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string
url)
    {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

By default, non-essential cookies aren't preserved across requests when the General Data Protection Regulation consent policy is enabled. To preserve non-essential cookies, such as those used by the TempData provider, mark them as essential in your tests. For instructions on marking a cookie as essential, see Essential cookies.

AngleSharp vs `Application Parts` for antiforgery checks

This article uses the AngleSharp parser to handle the antiforgery checks by loading pages and parsing the HTML. For testing the endpoints of controller and Razor Pages views at a lower-level, without caring about how they render in the browser, consider using `Application Parts`. The Application Parts approach injects a controller or Razor Page into the app that can be used to make JSON requests to get the required values. For more information, see the blog Integration Testing ASP.NET Core Resources Protected with Antiforgery Using Application Parts and associated GitHub repo by Martin Costello.

Customize WebApplicationFactory

Web host configuration can be created independently of the test classes by inheriting from WebApplicationFactory<TEntryPoint> to create one or more custom factories:

1. Inherit from `WebApplicationFactory` and override ConfigureWebHost. The IWebHostBuilder allows the configuration of the service collection with IWebHostBuilder.ConfigureServices

```
C#

public class CustomWebApplicationFactory<TProgram>
    : WebApplicationFactory<TProgram> where TProgram : class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            var dbContextDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));

            services.Remove(dbContextDescriptor);

            var dbConnectionDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbConnection));
```

```
            services.Remove(dbConnectionDescriptor);

            // Create open SqliteConnection so EF won't automatically
close it.
            services.AddSingleton<DbConnection>(container =>
            {
                var connection = new
SqliteConnection("DataSource=:memory:");
                connection.Open();

                return connection;
            });

            services.AddDbContext<ApplicationDbContext>((container,
options) =>
            {
                var connection =
container.GetRequiredService<DbConnection>();
                options.UseSqlite(connection);
            });
        });

        builder.UseEnvironment("Development");
    }
}
```

Database seeding in the sample app is performed by the `InitializeDbForTests` method. The method is described in the Integration tests sample: Test app organization section.

The SUT's database context is registered in `Program.cs`. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Program.cs` code is executed. To use a different database for the tests than the app's database, the app's database context must be replaced in `builder.ConfigureServices`.

The sample app finds the service descriptor for the database context and uses the descriptor to remove the service registration. The factory then adds a new `ApplicationDbContext` that uses an in-memory database for the tests..

To connect to a different database, change the `DbConnection`. To use a SQL Server test database:

- Reference the Microsoft.EntityFrameworkCore.SqlServer NuGet package in the project file.
- Call `UseInMemoryDatabase`.

2. Use the custom `CustomWebApplicationFactory` in test classes. The following example uses the factory in the `IndexPageTests` class:

```
C#

public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<Program>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<Program>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<Program> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new
WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });
    }
```

The sample app's client is configured to prevent the `HttpClient` from following redirects. As explained later in the Mock authentication section, this permits tests to check the result of the app's first response. The first response is a redirect in many of these tests with a `Location` header.

3. A typical test uses the `HttpClient` and helper methods to process the request and the response:

```
C#

[Fact]
public async Task Post_DeleteAllMessagesHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),

(IHtmlButtonElement)content.QuerySelector("button[id='deleteAllBtn']"))
;

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
```

```
        Assert.Equal("/", response.Headers.Location.OriginalString);
    }
```

Any POST request to the SUT must satisfy the antiforgery check that's automatically made by the app's data protection antiforgery system. In order to arrange for a test's POST request, the test app must:

1. Make a request for the page.
2. Parse the antiforgery cookie and request validation token from the response.
3. Make the POST request with the antiforgery cookie and request validation token in place.

The `SendAsync` helper extension methods (`Helpers/HttpClientExtensions.cs`) and the `GetDocumentAsync` helper method (`Helpers/HtmlHelpers.cs`) in the sample app use the AngleSharp parser to handle the antiforgery check with the following methods:

- `GetDocumentAsync`: Receives the HttpResponseMessage and returns an `IHtmlDocument`. `GetDocumentAsync` uses a factory that prepares a *virtual response* based on the original `HttpResponseMessage`. For more information, see the AngleSharp documentation.
- `SendAsync` extension methods for the `HttpClient` compose an HttpRequestMessage and call SendAsync(HttpRequestMessage) to submit requests to the SUT. Overloads for `SendAsync` accept the HTML form (`IHtmlFormElement`) and the following:
  - Submit button of the form (`IHtmlElement`)
  - Form values collection (`IEnumerable<KeyValuePair<string, string>>`)
  - Submit button (`IHtmlElement`) and form values (`IEnumerable<KeyValuePair<string, string>>`)

AngleSharp is a **third-party parsing library used for demonstration purposes** in this article and the sample app. AngleSharp isn't supported or required for integration testing of ASP.NET Core apps. Other parsers can be used, such as the Html Agility Pack (HAP). Another approach is to write code to handle the antiforgery system's request verification token and antiforgery cookie directly. See AngleSharp vs Application Parts for antiforgery checks in this article for more information.

The EF-Core in-memory database provider can be used for limited and basic testing, however the ***SQLite provider* is the recommended choice for in-memory testing**.

See Extend Startup with startup filters which shows how to configure middleware using IStartupFilter, which is useful when a test requires a custom service or middleware.

Customize the client with WithWebHostBuilder

When additional configuration is required within a test method, WithWebHostBuilder creates a new `WebApplicationFactory` with an IWebHostBuilder that is further customized by configuration.

The sample code calls `WithWebHostBuilder` to replace configured services with test stubs. For more information and example usage, see Inject mock services in this article.

The `Post_DeleteMessageHandler_ReturnsRedirectToRoot` test method of the sample app demonstrates the use of `WithWebHostBuilder`. This test performs a record delete in the database by triggering a form submission in the SUT.

Because another test in the `IndexPageTests` class performs an operation that deletes all of the records in the database and may run before the `Post_DeleteMessageHandler_ReturnsRedirectToRoot` method, the database is reseeded in this test method to ensure that a record is present for the SUT to delete. Selecting the first delete button of the `messages` form in the SUT is simulated in the request to the SUT:

C#

```
[Fact]
public async Task Post_DeleteMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    using (var scope = _factory.Services.CreateScope())
    {
        var scopedServices = scope.ServiceProvider;
        var db = scopedServices.GetRequiredService<ApplicationDbContext>();

        Utilities.ReinitializeDbForTests(db);
    }

    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("form[id='messages']")
            .QuerySelector("div[class='panel-body']")
            .QuerySelector("button"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
```

```
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

Client options

See the WebApplicationFactoryClientOptions page for defaults and available options when creating `HttpClient` instances.

Create the `WebApplicationFactoryClientOptions` class and pass it to the CreateClient() method:

C#

```
public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<Program>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<Program>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<Program> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new
WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });
    }
```

NOTE: To avoid HTTPS redirection warnings in logs when using HTTPS Redirection Middleware, set `BaseAddress = new Uri("https://localhost")`

Inject mock services

Services can be overridden in a test with a call to ConfigureTestServices on the host builder. To scope the overridden services to the test itself, the WithWebHostBuilder method is used to retrieve a host builder. This can be seen in the following tests:

- Get_QuoteService_ProvidesQuoteInPage
- Get_GithubProfilePageCanGetAGithubUser
- Get_SecurePageIsReturnedForAnAuthenticatedUser

The sample SUT includes a scoped service that returns a quote. The quote is embedded in a hidden field on the Index page when the Index page is requested.

`Services/IQuoteService.cs`:

C#

```
public interface IQuoteService
{
    Task<string> GenerateQuote();
}
```

`Services/QuoteService.cs`:

C#

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Dr. Who: Planet of Evil
// https://www.bbc.co.uk/programmes/p00pyrx6
public class QuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Come on, Sarah. We've an appointment in London, " +
            "and we're already 30,000 years late.");
    }
}
```

`Program.cs`:

C#

```
services.AddScoped<IQuoteService, QuoteService>();
```

`Pages/Index.cshtml.cs`:

C#

```
public class IndexModel : PageModel
{
    private readonly ApplicationDbContext _db;
    private readonly IQuoteService _quoteService;

    public IndexModel(ApplicationDbContext db, IQuoteService quoteService)
    {
        _db = db;
        _quoteService = quoteService;
    }
```

```
    [BindProperty]
    public Message Message { get; set; }

    public IList<Message> Messages { get; private set; }

    [TempData]
    public string MessageAnalysisResult { get; set; }

    public string Quote { get; private set; }

    public async Task OnGetAsync()
    {
        Messages = await _db.GetMessagesAsync();

        Quote = await _quoteService.GenerateQuote();
    }
```

`Pages/Index.cs`:

CSHTML

```
<input id="quote" type="hidden" value="@Model.Quote">
```

The following markup is generated when the SUT app is run:

HTML

```
<input id="quote" type="hidden" value="Come on, Sarah. We&#x27;ve an
appointment in
    London, and we&#x27;re already 30,000 years late.">
```

To test the service and quote injection in an integration test, a mock service is injected into the SUT by the test. The mock service replaces the app's `QuoteService` with a service provided by the test app, called `TestQuoteService`:

`IntegrationTests.IndexPageTests.cs`:

C#

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Pyramids of Mars
// https://www.bbc.co.uk/programmes/p00pys55
public class TestQuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult(
            "Something's interfering with time, Mr. Scarman, " +
            "and time is my business.");
```

```
    }
}
```

`ConfigureTestServices` is called, and the scoped service is registered:

C#

```
[Fact]
public async Task Get_QuoteService_ProvidesQuoteInPage()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddScoped<IQuoteService, TestQuoteService>();
            });
        })
        .CreateClient();

    //Act
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);
    var quoteElement = content.QuerySelector("#quote");

    // Assert
    Assert.Equal("Something's interfering with time, Mr. Scarman, " +
        "and time is my business.", quoteElement.Attributes["value"].Value);
}
```

The markup produced during the test's execution reflects the quote text supplied by `TestQuoteService`, thus the assertion passes:

HTML

```
<input id="quote" type="hidden" value="Something&#x27;s interfering with
time,
    Mr. Scarman, and time is my business.">
```

Mock authentication

Tests in the `AuthTests` class check that a secure endpoint:

- Redirects an unauthenticated user to the app's sign in page.
- Returns content for an authenticated user.

In the SUT, the `/SecurePage` page uses an AuthorizePage convention to apply an AuthorizeFilter to the page. For more information, see Razor Pages authorization conventions.

C#

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/SecurePage");
});
```

In the `Get_SecurePageRedirectsAnUnauthenticatedUser` test, a WebApplicationFactoryClientOptions is set to disallow redirects by setting AllowAutoRedirect to `false`:

C#

```
[Fact]
public async Task Get_SecurePageRedirectsAnUnauthenticatedUser()
{
    // Arrange
    var client = _factory.CreateClient(
        new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });

    // Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.StartsWith("http://localhost/Identity/Account/Login",
        response.Headers.Location.OriginalString);
}
```

By disallowing the client to follow the redirect, the following checks can be made:

- The status code returned by the SUT can be checked against the expected HttpStatusCode.Redirect result, not the final status code after the redirect to the sign in page, which would be HttpStatusCode.OK.
- The `Location` header value in the response headers is checked to confirm that it starts with `http://localhost/Identity/Account/Login`, not the final sign in page response, where the `Location` header wouldn't be present.

The test app can mock an AuthenticationHandler<TOptions> in ConfigureTestServices in order to test aspects of authentication and authorization. A minimal scenario returns an

AuthenticateResult.Success:

```
C#

public class TestAuthHandler :
AuthenticationHandler<AuthenticationSchemeOptions>
{
    public TestAuthHandler(IOptionsMonitor<AuthenticationSchemeOptions>
options,
        ILoggerFactory logger, UrlEncoder encoder)
        : base(options, logger, encoder)
    {
    }

    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        var claims = new[] { new Claim(ClaimTypes.Name, "Test user") };
        var identity = new ClaimsIdentity(claims, "Test");
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, "TestScheme");

        var result = AuthenticateResult.Success(ticket);

        return Task.FromResult(result);
    }
}
```

The `TestAuthHandler` is called to authenticate a user when the authentication scheme is set to `TestScheme` where `AddAuthentication` is registered for `ConfigureTestServices`. It's important for the `TestScheme` scheme to match the scheme your app expects. Otherwise, authentication won't work.

```
C#

[Fact]
public async Task Get_SecurePageIsReturnedForAnAuthenticatedUser()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddAuthentication(defaultScheme: "TestScheme")
                    .AddScheme<AuthenticationSchemeOptions, TestAuthHandler>
(
                        "TestScheme", options => { });
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false,
```

```
        });

    client.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue(scheme: "TestScheme");

    //Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

For more information on `WebApplicationFactoryClientOptions`, see the Client options section.

Basic tests for authentication middleware

See this GitHub repository for basic tests of authentication middleware. It contains a test server that's specific to the test scenario.

Set the environment

Set the environment in the custom application factory:

C#

```
public class CustomWebApplicationFactory<TProgram>
    : WebApplicationFactory<TProgram> where TProgram : class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            var dbContextDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));

            services.Remove(dbContextDescriptor);

            var dbConnectionDescriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbConnection));

            services.Remove(dbConnectionDescriptor);

            // Create open SqliteConnection so EF won't automatically close
it.
            services.AddSingleton<DbConnection>(container =>
            {
```

```
                var connection = new
SqliteConnection("DataSource=:memory:");
                connection.Open();

                return connection;
            });

            services.AddDbContext<ApplicationDbContext>((container, options)
=>
            {
                var connection = container.GetRequiredService<DbConnection>
();
                options.UseSqlite(connection);
            });
        });

        builder.UseEnvironment("Development");
    }
}
```

How the test infrastructure infers the app content root path

The `WebApplicationFactory` constructor infers the app content root path by searching for a WebApplicationFactoryContentRootAttribute on the assembly containing the integration tests with a key equal to the `TEntryPoint` assembly `System.Reflection.Assembly.FullName`. In case an attribute with the correct key isn't found, `WebApplicationFactory` falls back to searching for a solution file (*.sln*) and appends the `TEntryPoint` assembly name to the solution directory. The app root directory (the content root path) is used to discover views and content files.

Disable shadow copying

Shadow copying causes the tests to execute in a different directory than the output directory. If your tests rely on loading files relative to `Assembly.Location` and you encounter issues, you might have to disable shadow copying.

To disable shadow copying when using xUnit, create a `xunit.runner.json` file in your test project directory, with the correct configuration setting :

JSON

```
{
  "shadowCopy": false
```

```
}
```

Disposal of objects

After the tests of the `IClassFixture` implementation are executed, TestServer and HttpClient are disposed when xUnit disposes of the WebApplicationFactory. If objects instantiated by the developer require disposal, dispose of them in the `IClassFixture` implementation. For more information, see Implementing a Dispose method.

Integration tests sample

The sample app is composed of two apps:

Expand table

App	Project directory	Description
Message app (the SUT)	`src/RazorPagesProject`	Allows a user to add, delete one, delete all, and analyze messages.
Test app	`tests/RazorPagesProject.Tests`	Used to integration test the SUT.

The tests can be run using the built-in test features of an IDE, such as Visual Studio. If using Visual Studio Code or the command line, execute the following command at a command prompt in the `tests/RazorPagesProject.Tests` directory:

Console

```
dotnet test
```

Message app (SUT) organization

The SUT is a Razor Pages message system with the following characteristics:

- The Index page of the app (`Pages/Index.cshtml` and `Pages/Index.cshtml.cs`) provides a UI and page model methods to control the addition, deletion, and analysis of messages (average words per message).
- A message is described by the `Message` class (`Data/Message.cs`) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.

- The app contains a data access layer (DAL) in its database context class, `AppDbContext` (`Data/AppDbContext.cs`).
- If the database is empty on app startup, the message store is initialized with three messages.
- The app includes a `/SecurePage` that can only be accessed by an authenticated user.

†The EF article, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic (the sample implements the repository pattern).

Test app organization

The test app is a console app inside the `tests/RazorPagesProject.Tests` directory.

Expand table

Test app directory	Description
`AuthTests`	Contains test methods for: • Accessing a secure page by an unauthenticated user. • Accessing a secure page by an authenticated user with a mock AuthenticationHandler<TOptions>. • Obtaining a GitHub user profile and checking the profile's user login.
`BasicTests`	Contains a test method for routing and content type.
`IntegrationTests`	Contains the integration tests for the Index page using custom `WebApplicationFactory` class.
`Helpers/Utilities`	• `Utilities.cs` contains the `InitializeDbForTests` method used to seed the database with test data. • `HtmlHelpers.cs` provides a method to return an AngleSharp `IHtmlDocument` for use by the test methods. • `HttpClientExtensions.cs` provide overloads for `SendAsync` to submit requests to the SUT.

The test framework is xUnit. Integration tests are conducted using the Microsoft.AspNetCore.TestHost, which includes the TestServer. Because the Microsoft.AspNetCore.Mvc.Testing package is used to configure the test host and test server, the `TestHost` and `TestServer` packages don't require direct package references in the test app's project file or developer configuration in the test app.

Integration tests usually require a small dataset in the database prior to the test execution. For example, a delete test calls for a database record deletion, so the database must have at least one record for the delete request to succeed.

The sample app seeds the database with three messages in `Utilities.cs` that tests can use when they execute:

C#

```
public static void InitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.AddRange(GetSeedingMessages());
    db.SaveChanges();
}

public static void ReinitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.RemoveRange(db.Messages);
    InitializeDbForTests(db);
}

public static List<Message> GetSeedingMessages()
{
    return new List<Message>()
    {
        new Message(){ Text = "TEST RECORD: You're standing on my scarf." },
        new Message(){ Text = "TEST RECORD: Would you like a jelly baby?" },
        new Message(){ Text = "TEST RECORD: To the rational mind, " +
            "nothing is inexplicable; only unexplained." }
    };
}
```

The SUT's database context is registered in `Program.cs`. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Program.cs` code is executed. To use a different database for the tests, the app's database context must be replaced in `builder.ConfigureServices`. For more information, see the Customize WebApplicationFactory section.

Additional resources

- Unit tests
- Razor Pages unit tests in ASP.NET Core
- ASP.NET Core Middleware
- Test controller logic in ASP.NET Core
- Basic tests for authentication middleware

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core load/stress testing

Article • 08/03/2023

Load testing and stress testing are important to ensure a web app is performant and scalable. Load and stress testing have different goals even though they often share similar tests.

Load tests: Test whether the app can handle a specified load of users for a certain scenario while still satisfying the response goal. The app is run under normal conditions.

Stress tests: Test app stability when running under extreme conditions, often for a long period of time. The tests place high user load, either spikes or gradually increasing load on the app, or they limit the app's computing resources.

Stress tests determine if an app under stress can recover from failure and gracefully return to expected behavior. Under stress, the app is run at abnormally high stress.

Azure Load Testing is a fully managed load-testing service that enables you to generate high-scale load. The service simulates traffic for apps, regardless of where they're hosted. Azure Load Testing Preview enables you to use existing Apache JMeter scripts to generate high-scale load.

Visual Studio 2019 load testing has been deprecated. The corresponding Azure DevOps cloud-based load testing service has been closed.

Third-party tools

The following list contains third-party web performance tools with various feature sets:

- Apache JMeter
- ApacheBench (ab)
- Gatling
- jmeter-dotnet-dsl
- k6
- Locust
- West Wind WebSurge
- Netling
- Vegeta
- NBomber

Load and stress test with release builds

Load and stress tests should be done in release and production mode and not in debug and development mode. Release configurations are fully optimized with minimal logging. Debug configuration is not optimized. Development mode enables more information logging that can impact performance.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Test ASP.NET Core middleware

Article • 01/11/2024

By Chris Ross

Middleware can be tested in isolation with TestServer. It allows you to:

- Instantiate an app pipeline containing only the components that you need to test.
- Send custom requests to verify middleware behavior.

Advantages:

- Requests are sent in-memory rather than being serialized over the network.
- This avoids additional concerns, such as port management and HTTPS certificates.
- Exceptions in the middleware can flow directly back to the calling test.
- It's possible to customize server data structures, such as HttpContext, directly in the test.

Set up the TestServer

In the test project, create a test:

- Build and start a host that uses TestServer.
- Add any required services that the middleware uses.
- Add a package reference to the project for the Microsoft.AspNetCore.TestHost NuGet package.
- Configure the processing pipeline to use the middleware for the test.

```
C#

[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
```

```
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    ...
}
```

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at **Package consumption workflow (NuGet documentation)**. Confirm correct package versions at **NuGet.org**.

Send requests with HttpClient

Send a request using HttpClient:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();
```

```
    var response = await host.GetTestClient().GetAsync("/");

    ...
}
```

Assert the result. First, make an assertion the opposite of the result that you expect. An initial run with a false positive assertion confirms that the test fails when the middleware is performing correctly. Run the test and confirm that the test fails.

In the following example, the middleware should return a 404 status code (*Not Found*) when the root endpoint is requested. Make the first test run with `Assert.NotEqual( ... );`, which should fail:

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.NotEqual(HttpStatusCode.NotFound, response.StatusCode);
}
```

Change the assertion to test the middleware under normal operating conditions. The final test uses `Assert.Equal( ... );`. Run the test again to confirm that it passes.

C#

```
[Fact]
public async Task MiddlewareTest_ReturnsNotFoundForRequest()
{
    using var host = await new HostBuilder()
```

```
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var response = await host.GetTestClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Send requests with HttpContext

A test app can also send a request using SendAsync(Action<HttpContext>, CancellationToken). In the following example, several checks are made when `https://example.com/A/Path/?and=query` is processed by the middleware:

C#

```
[Fact]
public async Task TestMiddleware_ExpectedResponse()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .ConfigureServices(services =>
                {
                    services.AddMyServices();
                })
                .Configure(app =>
                {
                    app.UseMiddleware<MyMiddleware>();
                });
        })
        .StartAsync();

    var server = host.GetTestServer();
    server.BaseAddress = new Uri("https://example.com/A/Path/");

    var context = await server.SendAsync(c =>
```

```
    {
        c.Request.Method = HttpMethods.Post;
        c.Request.Path = "/and/file.txt";
        c.Request.QueryString = new QueryString("?and=query");
    });

    Assert.True(context.RequestAborted.CanBeCanceled);
    Assert.Equal(HttpProtocol.Http11, context.Request.Protocol);
    Assert.Equal("POST", context.Request.Method);
    Assert.Equal("https", context.Request.Scheme);
    Assert.Equal("example.com", context.Request.Host.Value);
    Assert.Equal("/A/Path", context.Request.PathBase.Value);
    Assert.Equal("/and/file.txt", context.Request.Path.Value);
    Assert.Equal("?and=query", context.Request.QueryString.Value);
    Assert.NotNull(context.Request.Body);
    Assert.NotNull(context.Request.Headers);
    Assert.NotNull(context.Response.Headers);
    Assert.NotNull(context.Response.Body);
    Assert.Equal(404, context.Response.StatusCode);
    Assert.Null(context.Features.Get<IHttpResponseFeature>().ReasonPhrase);
}
```

SendAsync permits direct configuration of an HttpContext object rather than using the HttpClient abstractions. Use SendAsync to manipulate structures only available on the server, such as HttpContext.Items or HttpContext.Features.

As with the earlier example that tested for a *404 - Not Found* response, check the opposite for each `Assert` statement in the preceding test. The check confirms that the test fails correctly when the middleware is operating normally. After you've confirmed that the false positive test works, set the final `Assert` statements for the expected conditions and values of the test. Run it again to confirm that the test passes.

Add request routes

Additional routes can be added by configuration using the test `HttpClient`:

C#

```
    [Fact]
    public async Task TestWithEndpoint_ExpectedResponse ()
    {
        using var host = await new HostBuilder()
            .ConfigureWebHost(webBuilder =>
            {
                webBuilder
                    .UseTestServer()
                    .ConfigureServices(services =>
                    {
                        services.AddRouting();
```

```
                    })
                    .Configure(app =>
                    {
                        app.UseRouting();
                        app.UseMiddleware<MyMiddleware>();
                        app.UseEndpoints(endpoints =>
                        {
                            endpoints.MapGet("/hello", () =>
                                TypedResults.Text("Hello Tests"));
                        });
                    });
            })
            .StartAsync();

        var client = host.GetTestClient();

        var response = await client.GetAsync("/hello");

        Assert.True(response.IsSuccessStatusCode);
        var responseBody = await response.Content.ReadAsStringAsync();
        Assert.Equal("Hello Tests", responseBody);
```

Additional routes can also be added using the approach `server.SendAsync`.

TestServer limitations

TestServer:

- Was created to replicate server behaviors to test middleware.
- Does ***not*** try to replicate all HttpClient behaviors.
- Attempts to give the client access to as much control over the server as possible, and with as much visibility into what's happening on the server as possible. For example it may throw exceptions not normally thrown by `HttpClient` in order to directly communicate server state.
- Doesn't set some transport specific headers by default as those aren't usually relevant to middleware. For more information, see the next section.
- Ignores the `Stream` position passed through StreamContent. HttpClient sends the entire stream from the start position, even when positioning is set. For more information, see this GitHub issue.

Content-Length and Transfer-Encoding headers

TestServer does ***not*** set transport related request or response headers such as Content-Length or Transfer-Encoding. Applications should avoid depending on these headers because their usage varies by client, scenario, and protocol. If `Content-Length`

and `Transfer-Encoding` are necessary to test a specific scenario, they can be specified in the test when composing the HttpRequestMessage or HttpContext. For more information, see the following GitHub issues:

- dotnet/aspnetcore#21677
- dotnet/aspnetcore#18463
- dotnet/aspnetcore#13273

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

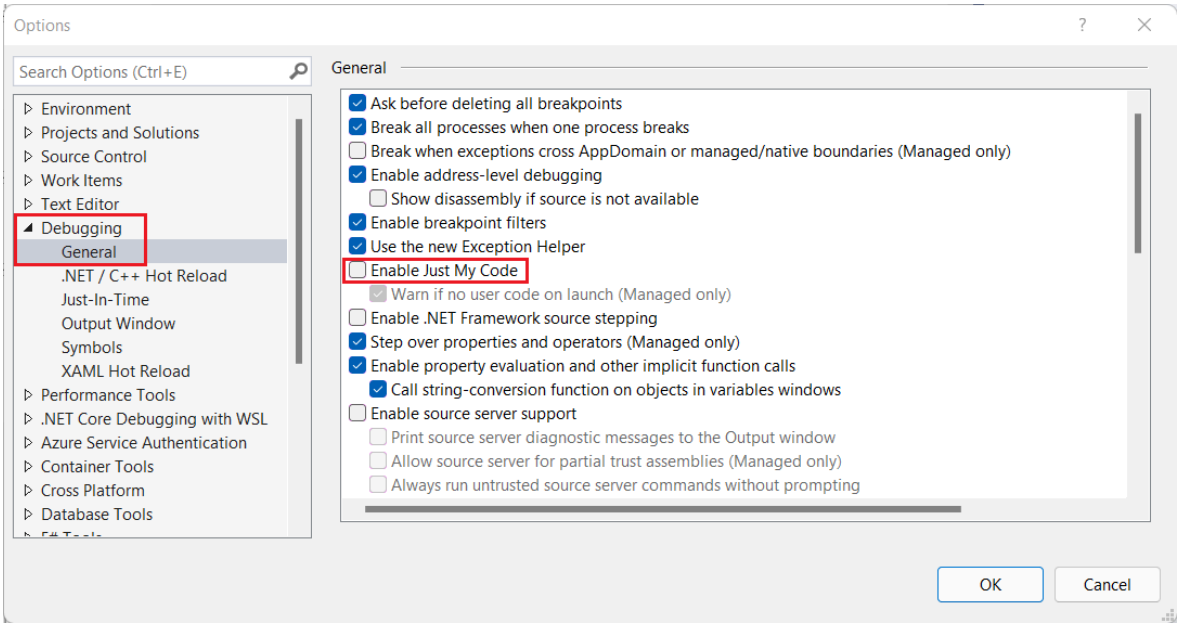
Provide product feedback

Debug .NET and ASP.NET Core source code with Visual Studio

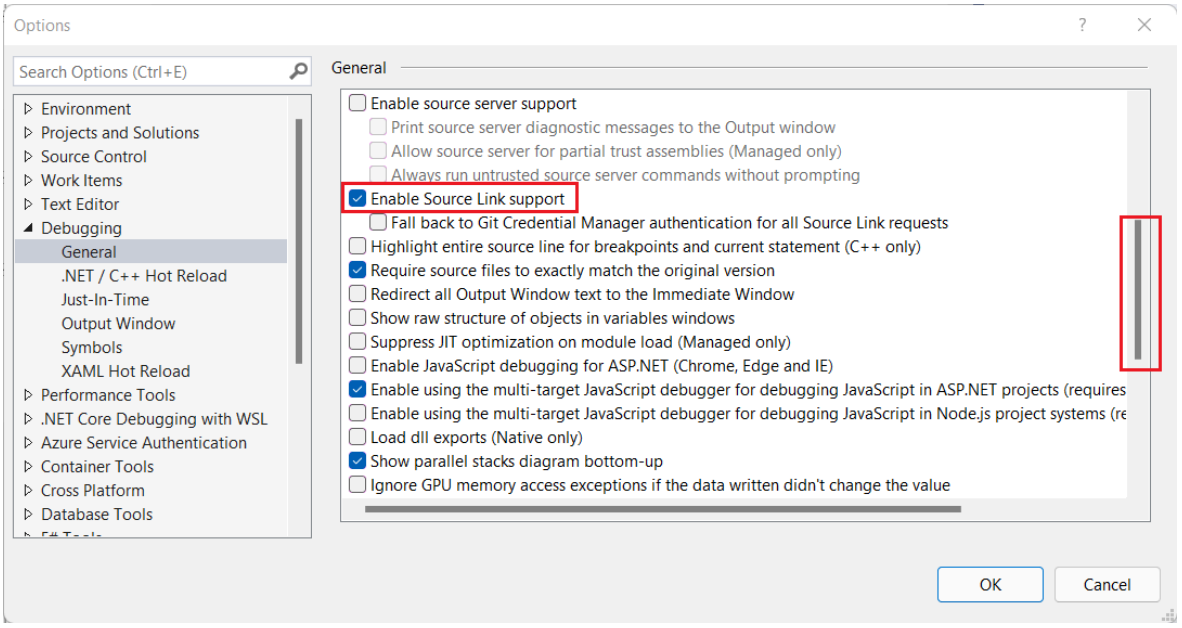
Article • 10/04/2022

To debug .NET and ASP.NET Core source code in Visual Studio:

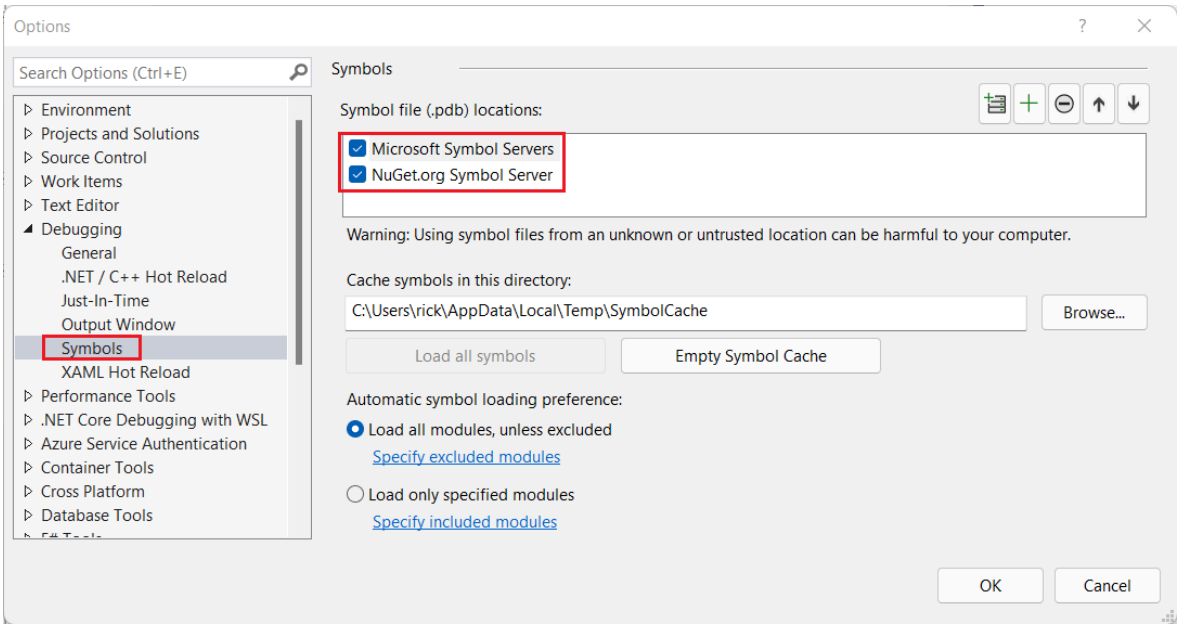
- In **Tools -> Options -> Debugging -> General**, un-check **Enable Just My Code**.



- Verify **Enable Source Link support** is checked.

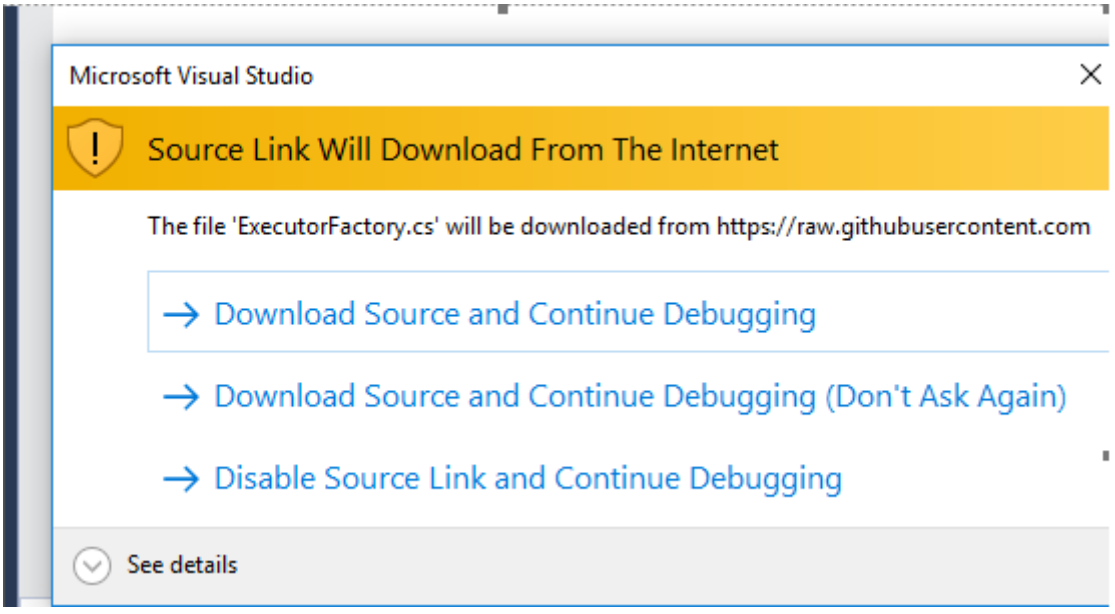


- In **Tool -> Options -> Debugging -> Symbols**, enable **Microsoft Symbol Servers**.



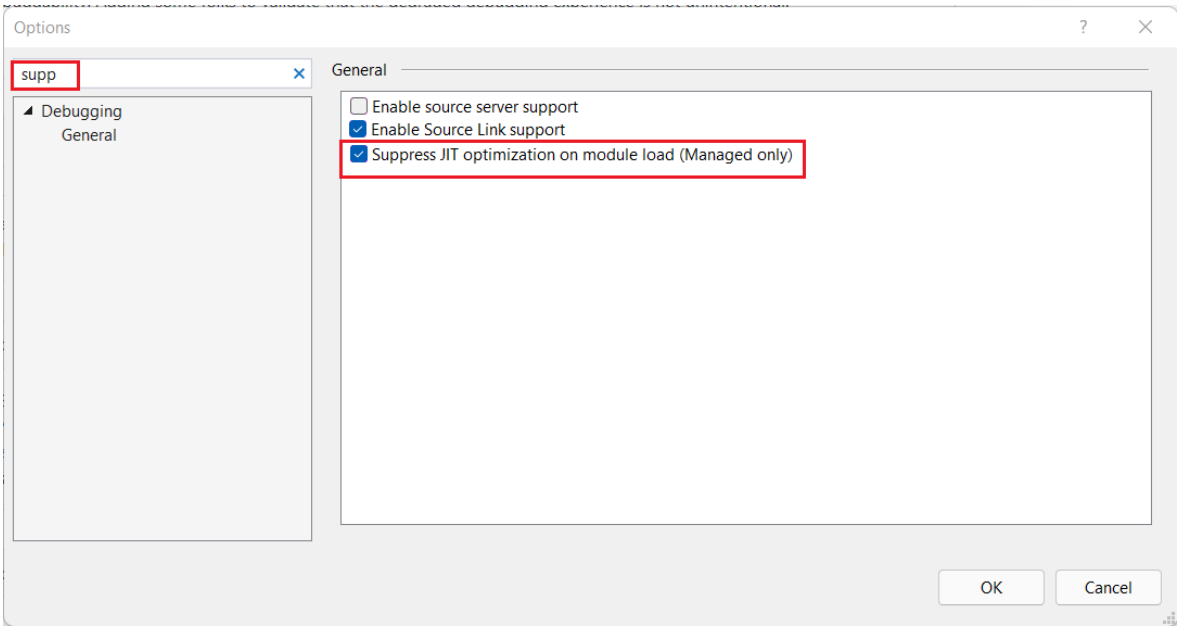
When you step into any .NET or ASP.NET Core code, Visual Studio displays the source code. For example:

- Set a break point in `OnGet` in `Pages/Privacy.cshtml.cs` and select the **Privacy** link.
- Select one of the **Download Source and Continue Debugging** options.



The preceding instructions work for basic stepping into functions, but the optimized .NET code often removes local variables and functions. To disable optimizations and allow better source debugging:

- In **Tools -> Options -> Debugging -> General**, enable **Suppress JIT optimization on module load (Managed only)**:



- Add the environment variable and value `COMPlus_ReadyToRun=0` to the `Properties/launchSettings.json` file:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:10892",
      "sslPort": 44315
    }
  },
  "profiles": {
    "WebApplication18": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:7045;http://localhost:5045",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "COMPlus_ReadyToRun": "0"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "COMPlus_ReadyToRun": "0"
      }
    }
  }
}
```

If you have debugged an app before with the previous version of .NET, delete the `%TEMP%/SymbolCache` directory as it can have old PDBs that are out of date.

Debugging .NET Core on Unix over SSH

- Debugging .NET Core on Unix over SSH
- Debugging ASP Core on Linux with Visual Studio 2017

Additional resources

- JIT Optimization and Debugging
- Limitations of the 'Suppress JIT optimization' option To set `COMPlus_ReadyToRun` to `0`
- .NET Hot Reload support for ASP.NET Core
- Test Execution with Hot Reload
- Debug ASP.NET Core Blazor apps

Remote Debug ASP.NET Core on IIS using an Azure VM from Visual Studio

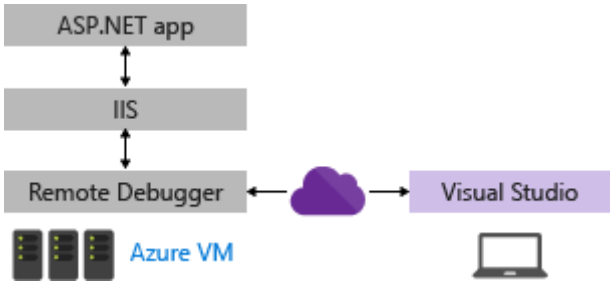
Article • 04/23/2024

This guide explains how to set up and configure a Visual Studio ASP.NET Core app, deploy it to IIS using an Azure VM, and attach the remote debugger from Visual Studio.

For IIS scenarios, Linux isn't supported.

To debug IIS on an Azure VM, follow the steps in this article. Using this method, you can use a customized configuration of IIS, but the setup and deployment steps are more complicated. If you don't need to customize IIS for your scenario, you might choose simpler methods to host and debug the app in Azure App Service instead.

For an Azure VM, you must deploy your app from Visual Studio to Azure and you also need to manually install the IIS role and the remote debugger, as shown in the following illustration.



> ⚠ **Warning**
>
> Be sure to delete the Azure resources that you create when you have completed the steps in this tutorial. That way you can avoid incurring unnecessary charges.

These procedures have been tested on these server configurations:

- Windows Server 2022 and IIS 10
- Windows Server 2019 and IIS 10
- Windows Server 2016 and IIS 10

Prerequisites

Visual Studio 2019 or later versions is required to follow the steps shown in this article.

Network requirements

Debugging between two computers connected through a proxy isn't supported. Debugging over a high latency or low bandwidth connection, such as dialup Internet, or over the Internet across countries/regions isn't recommended and might fail or be unacceptably slow. For a complete list of requirements, see Requirements.

App already running in IIS on the Azure VM?

This article includes steps on setting up a basic configuration of IIS on Windows server and deploying the app from Visual Studio. These steps are included to make sure that the server has the required components installed, that the app can run correctly, and that you're ready to remote debug.

- If your app is running in IIS and you just want to download the remote debugger and start debugging, go to Download and Install the remote tools on Windows Server.
- If you want help ensuring your app is set up, deployed, and running correctly in IIS so that you can debug, follow all the steps in this article.
    - Before you begin, follow all the steps described in Create a Windows Virtual Machine, which includes steps to install the IIS web server.
    - Make sure you open port 80 in the Azure Network security group. When you verify that port 80 is open, also open the correct port for the remote debugger (4026, 4024, or 4022). That way, you don't have to open it later. If you're using Web Deploy, also open port 8172.

Create the ASP.NET Core application on the Visual Studio computer

1. Create a new ASP.NET Core web application.

   In Visual Studio, choose **File** > **Start window** to open the Start window, and then choose **Create a new project**. In the search box, type **web app**, then choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project **MyASPApp**, and then choose **Next**.

   Choose either the recommended target framework or .NET 8, and then choose **Create**. The version must match the version installed on the server.

2. Open the *HomeController.cs* file in the Controllers folder and set a breakpoint in the `return View;` statement in the `Privacy` method.

   In older templates, open the *Privacy.cshtml.cs* file and set a breakpoint in the `OnGet` method.

Update browser security settings on Windows Server

If you're using an older version of Windows Server, you might need to add some domains as trusted sites to enable you to download some of the web server components. Add the trusted sites by going to **Internet Options** > **Security** > **Trusted Sites** > **Sites**. Add the following domains.

- microsoft.com
- go.microsoft.com
- download.microsoft.com
- iis.net

When you download the software, you might get requests to grant permission to load various web site scripts and resources. Some of these resources aren't required, but to simplify the process, select **Add** when prompted.

Install ASP.NET Core on Windows Server

1. Install the .NET Core Hosting Bundle on the hosting system. The bundle installs the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. For more in-depth instructions, see Publishing to IIS.

   For the current .NET Core hosting bundle, install the ASP.NET Core Hosting Bundle.

   > **Note**
   >
   > If you previously installed IIS, the ASP.NET Core IIS Module gets installed with ASP.NET Core. Otherwise, install the ASP.NET Core IIS Module manually.

   For .NET Core 2, install the .NET Core Windows Server Hosting.

   > **Note**

If the system doesn't have an Internet connection, obtain and install the *Microsoft Visual C++ 2015 Redistributable* before installing the .NET Core Windows Server Hosting bundle.

2. Restart the system (or execute **net stop was /y** followed by **net start w3svc** from a command prompt to pick up a change to the system PATH).

Choose a deployment option

If you need help deploying the app to IIS, consider these options:

- Deploy by creating a publish settings file in IIS and importing the settings in Visual Studio. In some scenarios, this approach is a fast way to deploy your app. When you create the publish settings file, permissions are automatically set up in IIS.
- Deploy by publishing to a local folder and copying the output by a preferred method to a prepared app folder on IIS.

(Optional) Deploy using a publish settings file

You can use this option create a publish settings file and import it into Visual Studio.

> **Note**
>
> This deployment method uses Web Deploy, which must be installed on the server. If you want to configure Web Deploy manually instead of importing the settings, you can install Web Deploy 3.6 instead of Web Deploy 3.6 for Hosting Servers. However, if you configure Web Deploy manually, you will need to make sure that an app folder on the server is configured with the correct values and permissions (see Configure ASP.NET Web site).

Configure the ASP.NET Core web site

1. In IIS Manager, in the left pane under **Connections**, select **Application Pools**. Open **DefaultAppPool** and set the **.NET CLR version** to **No Managed Code**. This is required for ASP.NET Core. The Default Web Site uses the DefaultAppPool.

2. Stop and restart the DefaultAppPool.

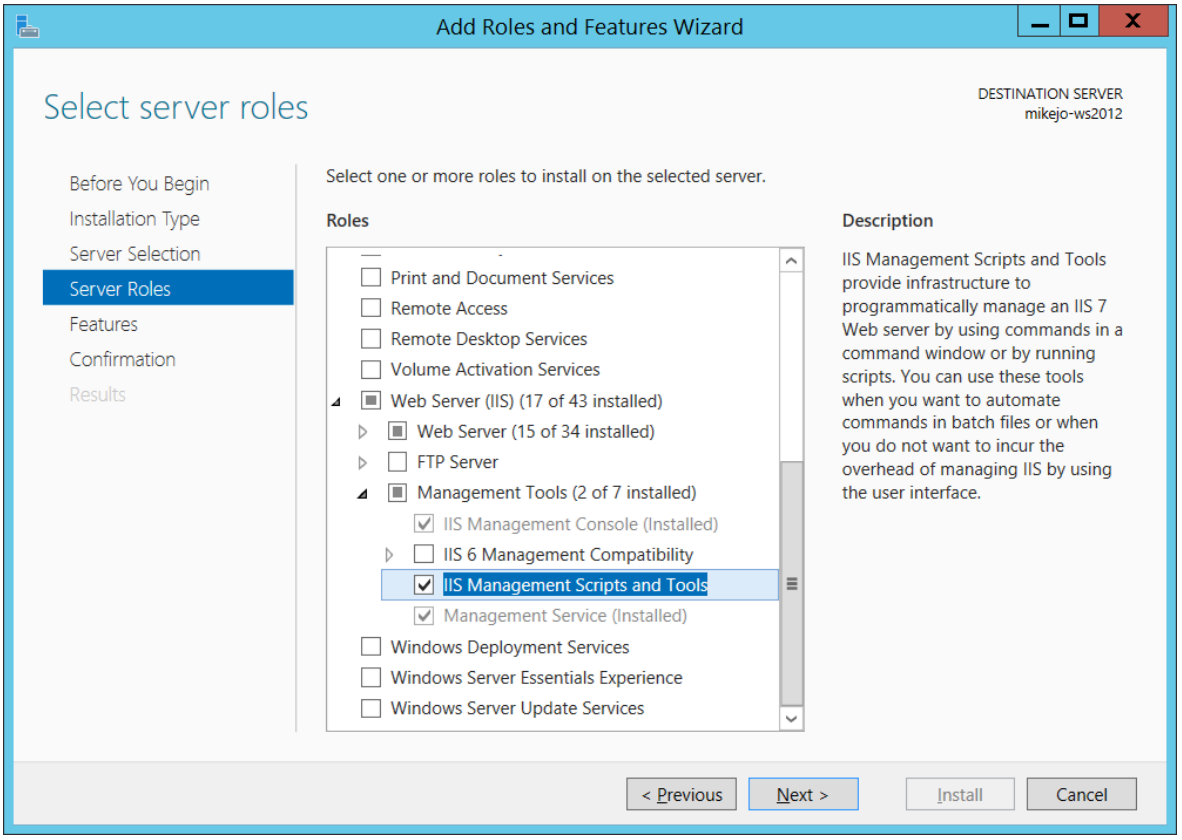
Install and configure Web Deploy on Windows Server

Web Deploy provides additional configuration features that enable the creation of the publish settings file from the UI.

> **Note**
>
> The Web Platform Installer reached End-of-Life on 7/1/22. For more information, see Web Platform Installer - End of support and sunsetting the product/application feed. You can directly install Web Deploy 4.0 to create the publish settings file.

1. If you did not already install **IIS Management Scripts and Tools**, install it now.

   Go to **Select server roles** > **Web Server (IIS)** > **Management Tools**, and then select the **IIS Management Scripts and Tools** role, click **Next**, and then install the role.



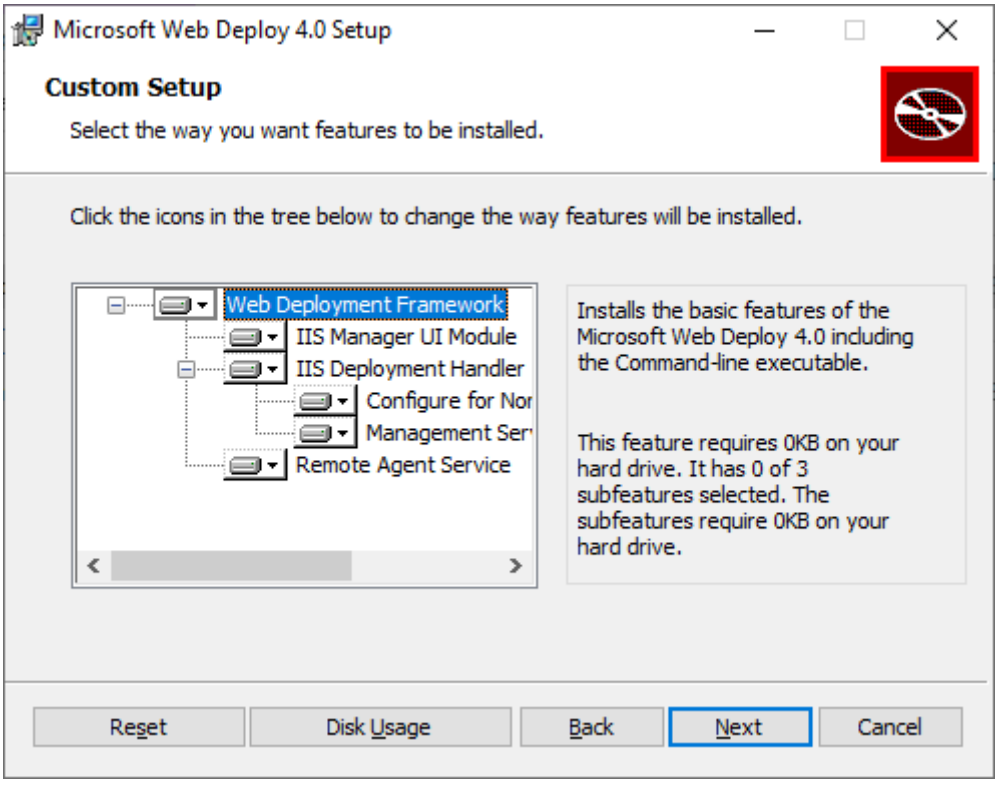
   The scripts and tools are required to enable the generation of the publish settings file.

   Make sure you also install the **Management Service** and **IIS Management Console** (they may be already installed).

2. On Windows Server, download Web Deploy 4.0.

3. Run the Web Deploy installation program, and make sure you select **Complete** installation instead of a typical installation.

With a complete installation, you get the components you need to generate a publish settings file. (If you choose **Custom** instead, you can see the list of components, as shown in the following illustration.)



4. (Optional) Verify that Web Deploy is running correctly by opening **Control Panel** > **System and Security** > **Administrative Tools** > **Services**, and then make sure that:

   - **Web Deployment Agent Service** is running (the service name is different in older versions).
   - **Web Management Service** is running.

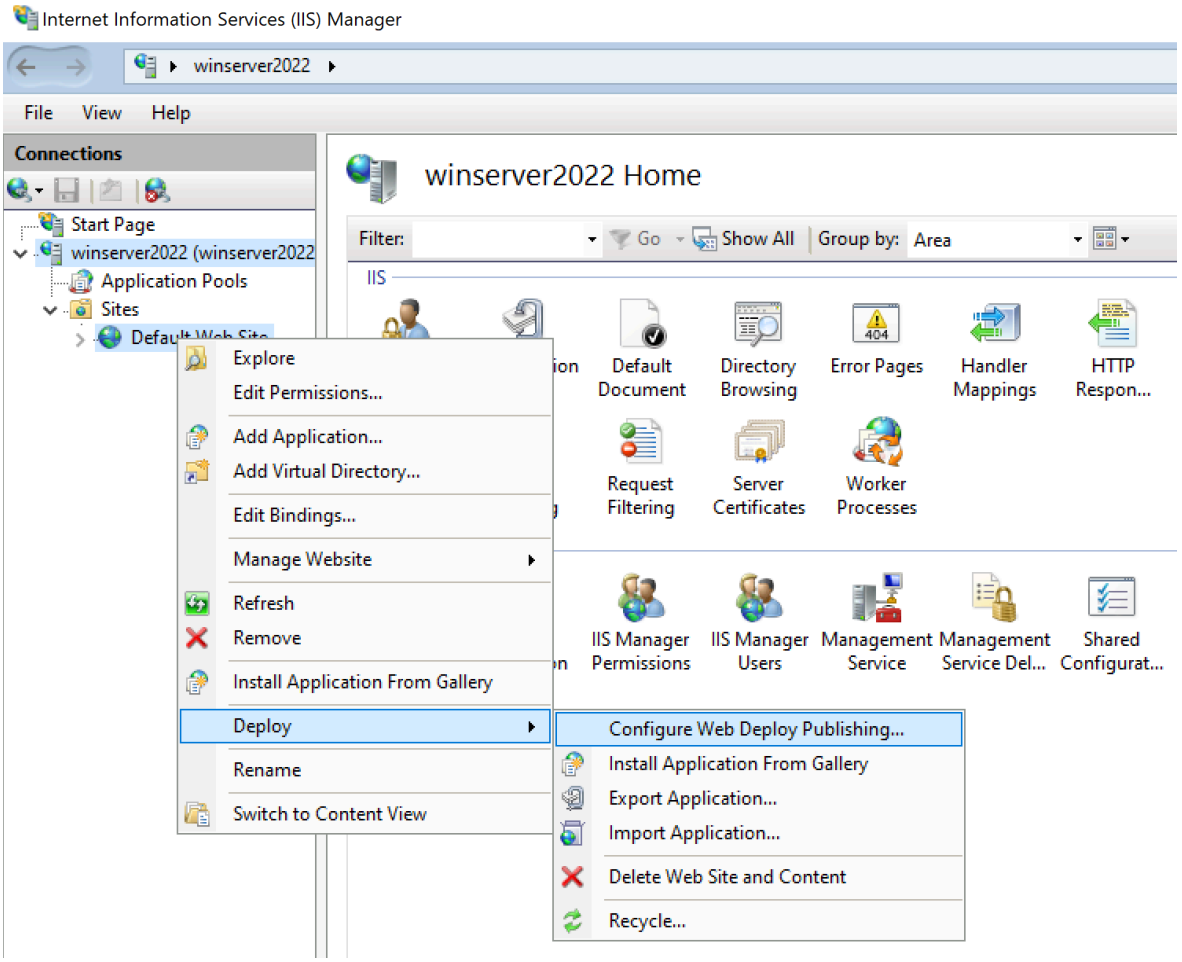
   If one of the agent services is not running, restart the **Web Deployment Agent Service**.

   If the Web Deployment Agent Service is not present at all, go to **Control Panel** > **Programs** > **Uninstall a program**, find **Microsoft Web Deploy <version>**. Choose to **Change** the installation and make sure that you choose **Will be installed to the local hard drive** for the Web Deploy components. Complete the change installation steps.

Create the publish settings file in IIS on Windows Server

1. Close and reopen the IIS Management Console to show updated configuration options in the UI.

2. In IIS, right-click the **Default Web Site**, choose **Deploy** > **Configure Web Deploy Publishing**.



If you don't see the **Deploy** menu, see the preceding section to verify that Web Deploy is running.

3. In the **Configure Web Deploy Publishing** dialog box, examine the settings.

4. Click **Setup**.

   In the **Results** panel, the output shows that access rights are granted to the specified user, and that a file with a *.publishsettings* file extension has been generated in the location shown in the dialog box.

```xml
<?xml version="1.0" encoding="utf-8"?>
<publishData>
  <publishProfile
    publishUrl="https://myhostname:8172/msdeploy.axd"
    msdeploySite="Default Web Site"
    destinationAppUrl="http://myhostname:80/"
    profileName="Default Settings"
    publishMethod="MSDeploy"
```

```
    userName="myhostname\myusername" />
</publishData>
```

Depending on your Windows Server and IIS configuration, you see different values in the XML file. Here are a few details about the values that you see:

- The *msdeploy.axd* file referenced in the `publishUrl` attribute is a dynamically generated HTTP handler file for Web Deploy. (For testing purposes, `http://myhostname:8172` generally works as well.)
- The `publishUrl` port is set to port 8172, which is the default for Web Deploy.
- The `destinationAppUrl` port is set to port 80, which is the default for IIS.
- If, in later steps, you are unable to connect to the remote host from Visual Studio using the host name, test the server's IP address in place of the host name.

> ⓘ **Note**
>
> If you are publishing to IIS running on an Azure VM, you must open an inbound port for Web Deploy and IIS in the Network Security group. For detailed information, see Open ports to a virtual machine.

5. Copy this file to the computer where you are running Visual Studio.

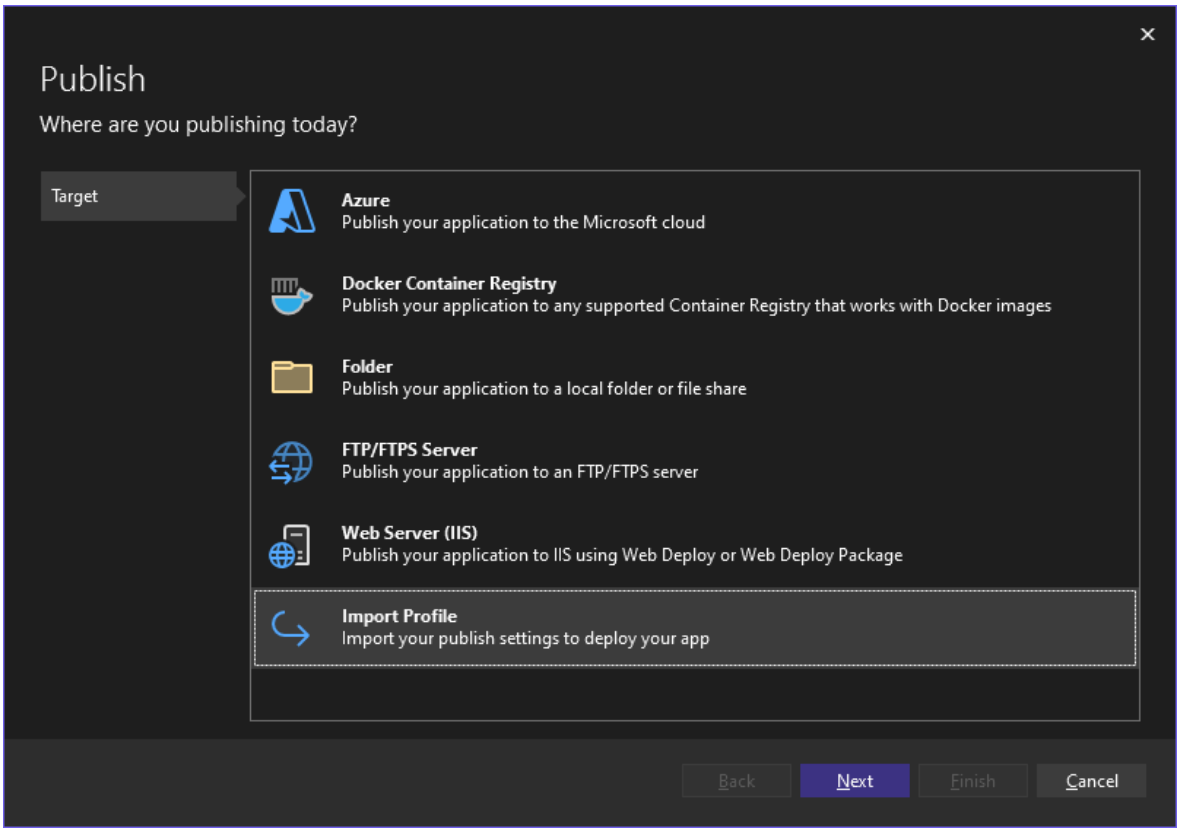
Import the publish settings in Visual Studio and deploy

1. On the computer where you have the ASP.NET project open in Visual Studio, right-click the project in Solution Explorer, and choose **Publish**.

   If you have previously configured any publishing profiles, the **Publish** pane appears. Click **New** or **Create new profile**.

2. Select the option to import a profile.

   In the **Publish** dialog box, click **Import Profile**.



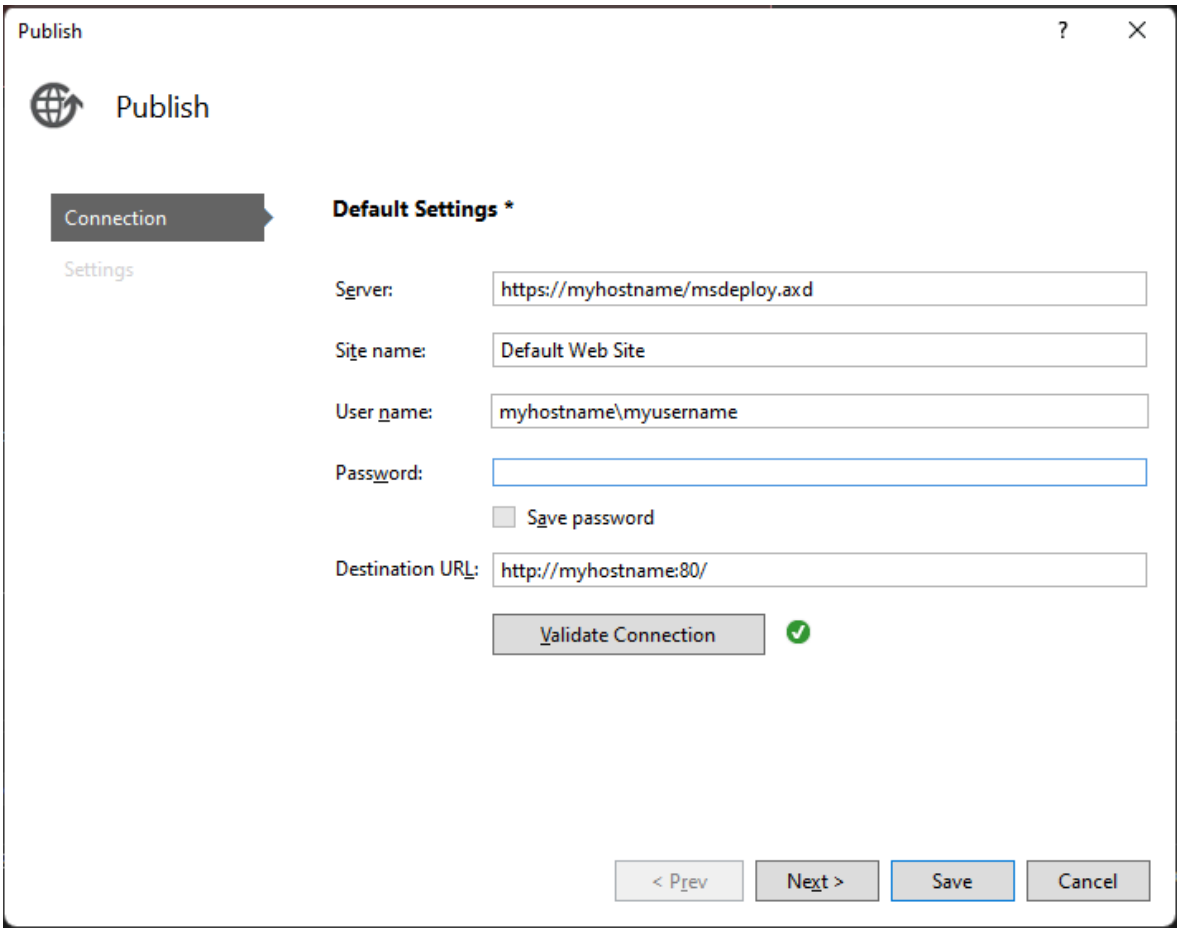
3. Navigate to the location of the publish settings file that you created in the previous section.

4. In the **Import Publish Settings File** dialog, navigate to and select the profile that you created in the previous section, and click **Open**.

   Click **Finish** to save the publishing profile, and then click **Publish**.

   Visual Studio begins the deployment process, and the Output window shows progress and results.

   If you get an any deployment errors, click **More Actions** > **Edit** to edit settings. Modify settings and click **Validate** to test new settings. If the host name is not found, try the IP address instead of the host name in both the **Server** and **Destination URL** fields.



> **Note**
>
> If you restart an Azure VM, the IP address might change.

After the app deploys successfully, it should start automatically.

- If the app doesn't start after deployment, start the app in IIS to verify that it runs correctly.
- For ASP.NET Core, make sure the Application pool field for the **DefaultAppPool** is set to **No Managed Code**.

When you're ready, switch to a debug configuration.

> **Important**
>
> If you choose to debug a Release configuration, you disable debugging in the *web.config* file when you publish.

1. Select **More Options** > **Edit** to edit the profile, and then select **Settings**.
2. Select **Save** and then republish the app.
3. Select a **Debug** configuration, and then select **Remove additional files at destination** under the **File Publish** options.

> ⚠ **Warning**
>
> Using username and password credentials (basic authentication) is not the most secure method of authentication. Whenever possible, use alternative methods. For example, consider publishing to a package from Visual Studio, and then use *WebDeploy.exe* from a command line to deploy the package. With that method, you can use IIS Manager to configure authorized Windows users who can publish to the web server, and run *WebDeploy.exe* under that Windows user account. See Installing and Configuring Web Deploy on IIS 8.0 or Later. If you do use password credentials, be sure to use a strong password, and secure the password from being leaked or shared.

(Optional) Deploy by publishing to a local folder

You can use this option to deploy your app if you want to copy the app to IIS using PowerShell, RoboCopy, or you want to manually copy the files.

Configure the ASP.NET Core Web site on the Windows Server computer

If you're importing publish settings, you can skip this section.

1. Open the **Internet Information Services (IIS) Manager** and go to **Sites**.
2. Right-click the **Default Web Site** node and select **Add Application**.
3. Set the **Alias** field to **MyASPApp** and the Application pool field to **No Managed Code**. Set the **Physical path** to **C:\Publish** (where you later deploy the ASP.NET Core project).
4. With the site selected in the IIS Manager, choose **Edit Permissions**, and make sure that IUSR, IIS_IUSRS, or the user configured for the Application Pool is an authorized user with Read & Execute rights.

   If you don't see one of these users with access, go through steps to add IUSR as a user with Read & Execute rights.

> ⓘ **Important**

For security information related to the built-in accounts, see [Understanding Built-In User and Group Accounts in IIS 7](#).

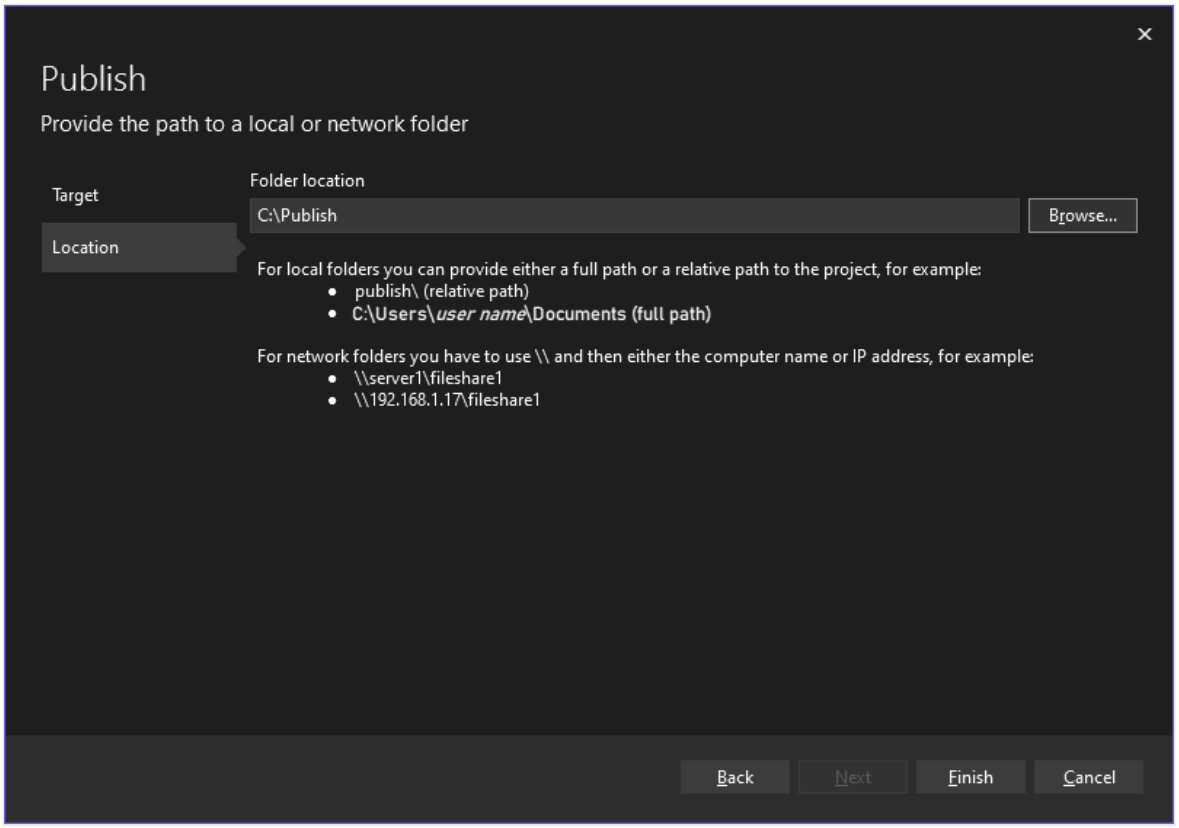
(Optional) Publish and Deploy the app by publishing to a local folder from Visual Studio

If you're not using Web Deploy, you must publish and deploy the app using the file system or other tools. You can start by creating a package using the file system, and then either deploy the package manually or use other tools like PowerShell, Robocopy, or XCopy. In this section, we assume you're manually copying the package if you aren't using Web Deploy.

1. In the **Solution Explorer**, right-click the project node and select **Publish** (for Web Forms, **Publish Web App**).

   If you have previously configured any publishing profiles, the **Publish** pane appears. Click **New profile**.

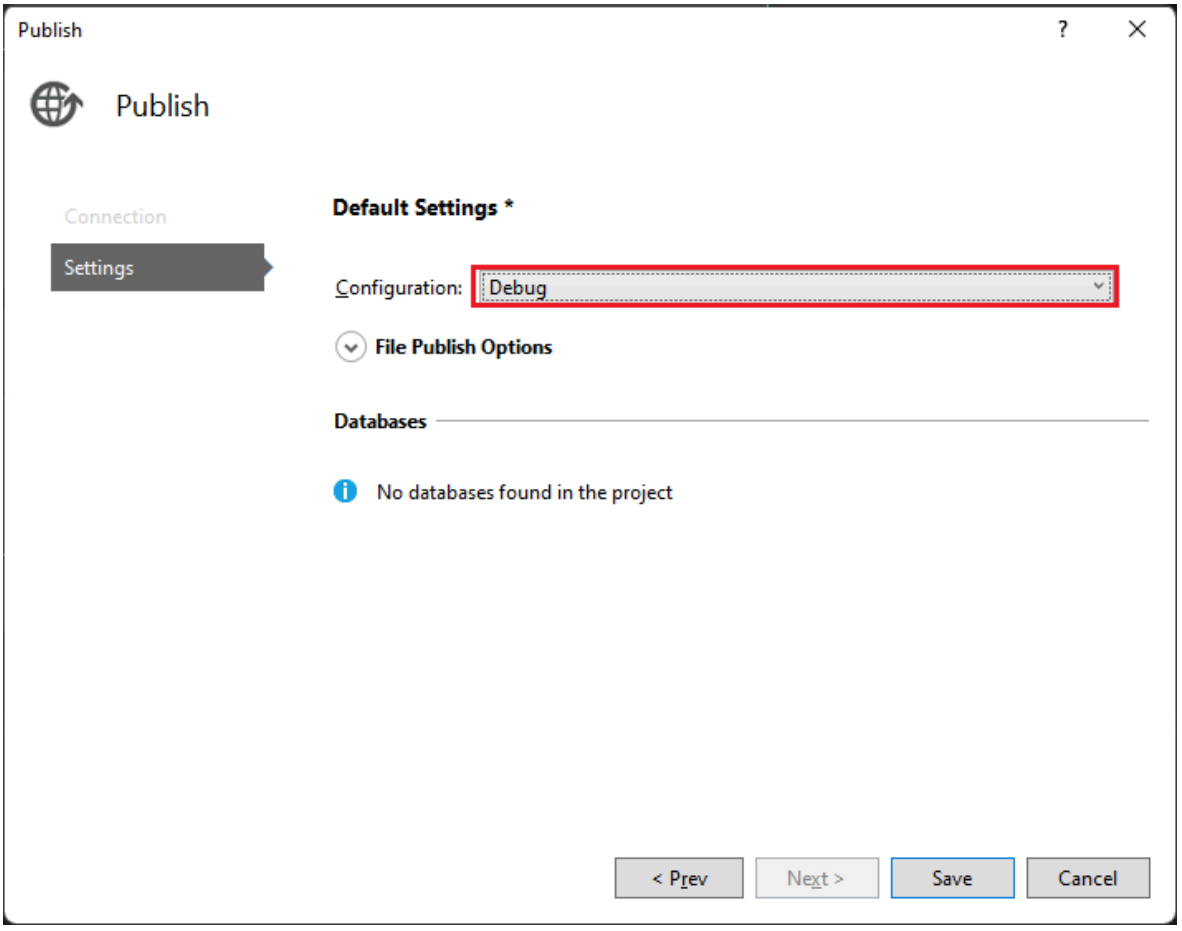
2. In the **Publish** dialog box, select **Folder**, click **Browse**, and create a new folder, **C:\Publish**.

   Click **Finish** to save the publish profile.

3. Switch to a debug configuration.

Choose **Edit** to edit the profile, and then choose **Settings**. Choose a **Debug** configuration, and then choose **Remove additional files at destination** under the **File Publish** options.



> ⓘ **Note**
>
> If you use a Release build, you disable debugging in the *web.config* file when you publish.

4. Click **Publish**.

   The application publishes a **Debug** configuration of the project to the local folder. Progress shows in the Output window.

5. Copy the ASP.NET project directory from the Visual Studio computer to the local directory configured for the ASP.NET app (in this example, **C:\Publish**) on the Windows Server computer. In this tutorial, we assume you are copying manually, but you can use other tools like PowerShell, Xcopy, or Robocopy.

> ⊗ **Caution**

If you need to make changes to the code or rebuild, you must republish and repeat this step. The executable you copied to the remote machine must exactly match your local source and symbols. If you do not do this you will receive a `cannot find or open the PDB file` warning in Visual Studio when you attempt to debug the process.

6. On the Windows Server, verify that you can run the app correctly by opening the app in your browser.

   If the app doesn't run correctly, there may be a mismatch between the version of ASP.NET installed on your server and your Visual Studio machine, or you may have an issue with your IIS or Web site configuration. Recheck earlier steps.

Download and Install the remote tools on Windows Server

Download the version of the remote tools that matches your version of Visual Studio.

On the remote device or server that you want to debug on, rather than the Visual Studio machine, download and install the correct version of the remote tools from the links in the following table.

- Download the most recent update of the remote tools for your version of Visual Studio. Earlier remote tools versions aren't compatible with later Visual Studio versions. (For example, if you're using Visual Studio 2019, download the latest update of the remote tools for Visual Studio 2019. In this scenario, don't download the remote tools for Visual Studio 2022.)
- Download the remote tools with the same architecture as the machine you're installing them on. For example, if you want to debug x86 applications on a remote computer running an x64 operating system, install the x64 remote tools. To debug x86, ARM, or x64 applications on an ARM64 operating system, install the ARM64 remote tools.

Expand table

Version	Link	Notes
Visual Studio 2022	Remote tools	Compatible with all Visual Studio 2022 versions. Download the version matching your device operating system (x86, x64 (AMD64), or ARM64). On older versions of Windows Server, see Unblock the file download for help with downloading the remote tools.

Version	Link	Notes
Visual Studio 2019	Remote tools	Remote tools for Visual Studio 2019 are available from My.VisualStudio.com. If prompted, join the free Visual Studio Dev Essentials program, or sign in with your Visual Studio subscription ID. Download the version matching your device operating system (x86, x64 (AMD64), or ARM64). On older versions of Windows Server, see Unblock the file download for help with downloading the remote tools.
Visual Studio 2017	Remote tools	Remote tools for Visual Studio 2017 are available from My.VisualStudio.com. If prompted, join the free Visual Studio Dev Essentials program, or sign in with your Visual Studio subscription ID. Download the version matching your device operating system (x86, x64 (AMD64), or ARM64). On Windows Server, see Unblock the file download for help with downloading the remote tools.
Visual Studio 2015	Remote tools	Remote tools for Visual Studio 2015 are available from My.VisualStudio.com. If prompted, join the free Visual Studio Dev Essentials program, or sign in with your Visual Studio subscription ID. On Windows Server, see Unblock the file download for help with downloading the remote tools.
Visual Studio 2013	Remote tools	Download page in Visual Studio 2013 documentation
Visual Studio 2012	Remote tools	Download page in Visual Studio 2012 documentation

You can run the remote debugger by copying *msvsmon.exe* to the remote computer, rather than installing the remote tools. However, the Remote Debugger Configuration Wizard (*rdbgwiz.exe*) is available only when you install the remote tools. You may need to use the wizard for configuration if you want to run the remote debugger as a service. For more information, see (Optional) Configure the remote debugger as a service.

> **Note**
>
> - To debug Windows 10 or later apps on ARM devices, use ARM64, which is available with the latest version of the remote tools.
> - To debug Windows 10 apps on Windows RT devices, use ARM, which is available only in the Visual Studio 2015 remote tools download.
> - To debug x64 apps on an ARM64 operating system, run the x64 msvsmon.exe that is installed with the ARM64 remote tools.

Set up the remote debugger on Windows Server

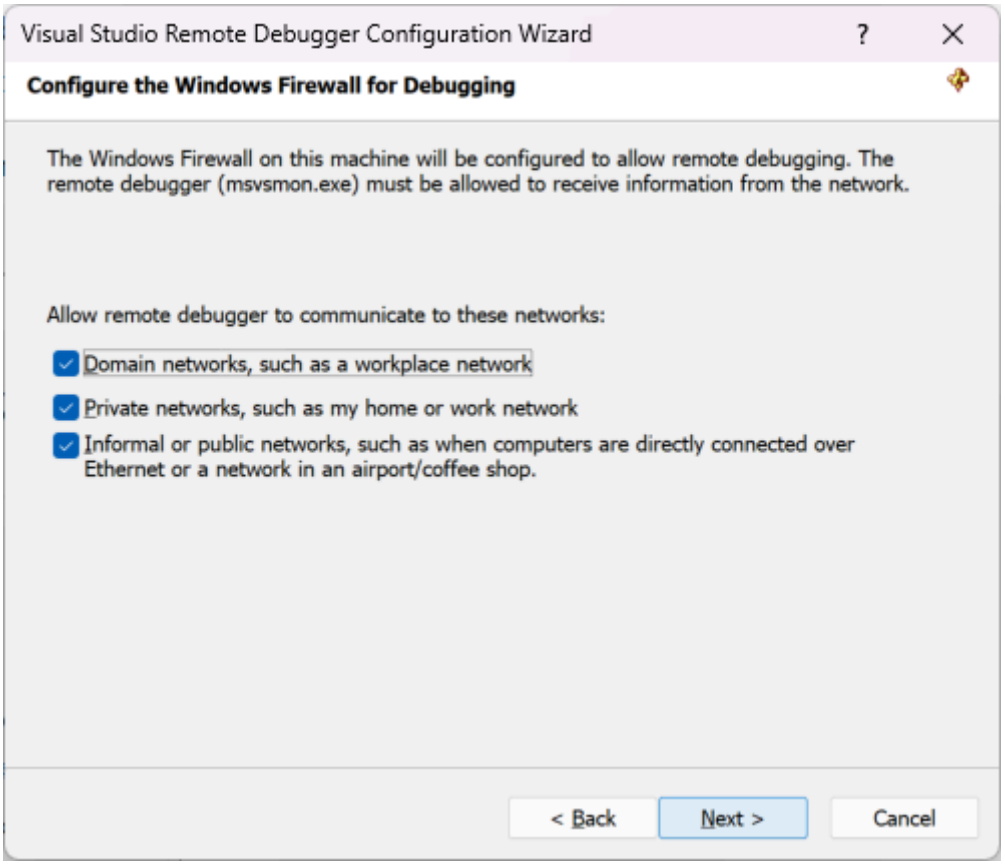
1. On the remote computer, find and start the **Remote Debugger** from the **Start** menu.

   If you don't have administrative permissions on the remote computer, right-click the **Remote Debugger** app and select **Run as administrator**. Otherwise, just start it normally.

   If you are planning to attach to a process which is running as an administrator, or is running under a different user account (such as IIS), right-click the **Remote Debugger** app and select **Run as administrator**. For more information, see Run the remote debugger as an administrator.

2. The first time you start the remote debugger (or before you have configured it), the **Remote Debugging Configuration** wizard appears.

   In most scenarios, choose **Next** until you get to the **Configure the Windows Firewall** page of the wizard.



3. Select at least one network type you want to use the remote tools on. If the computers are connected through a domain, you must choose the first item. If the computers are connected through a workgroup or homegroup, choose the second or third item as appropriate.

Next, select **Finish** to start the remote debugger.

4. When configuration is complete, the **Remote Debugger** window appears.



   The remote debugger is now waiting for a connection. Use the server name and port number shown to set the remote connection configuration in Visual Studio.

To stop the remote debugger, select **File** > **Exit**. You can restart it from the **Start** menu, or from the command line:

Windows Command Prompt

```
<Remote debugger installation directory>\msvsmon.exe
```

> ⓘ **Note**
>
> If you need to add permissions for additional users, change the authentication mode, or port number for the remote debugger, see Configure the remote debugger.

Attach to the ASP.NET Core application from the Visual Studio computer

Starting in Visual Studio 2022 version 17.10 Preview 2, the Attach to Process dialog box has changed. If you need instructions that match the older dialog box, switch to the Visual Studio 2019 view (upper left version selector in the article).

1. On the Visual Studio computer, open the solution that you're trying to debug (**MyASPApp** if you're following all the steps in this article).

2. In Visual Studio, select **Debug** > **Attach to Process** (Ctrl + Alt + P).

   > 💡 **Tip**

> In Visual Studio 2017 and later versions, you can reattach to the same process you previously attached to by using **Debug** > **Reattach to Process...** (Shift + Alt + P).

3. Set the **Connection Type** to **Remote (Windows)**.

   The **Connection Target** option appears.

   Set the **Connection Target** to **<remote computer name>** and press **Enter**.

   Verify that Visual Studio adds the required port to the computer name, which appears in the format: **<remote computer name>:port**

   On Visual Studio 2022, you should see **<remote computer name>:4026**

   The port is required. If you don't see the port number, add it manually.

4. Select **Refresh**.

   You should see some processes appear in the **Available Processes** window.

   If you don't see any processes, try using the IP address instead of the remote computer name (the port is required). You can use `ipconfig` in a command line to get the IPv4 address.

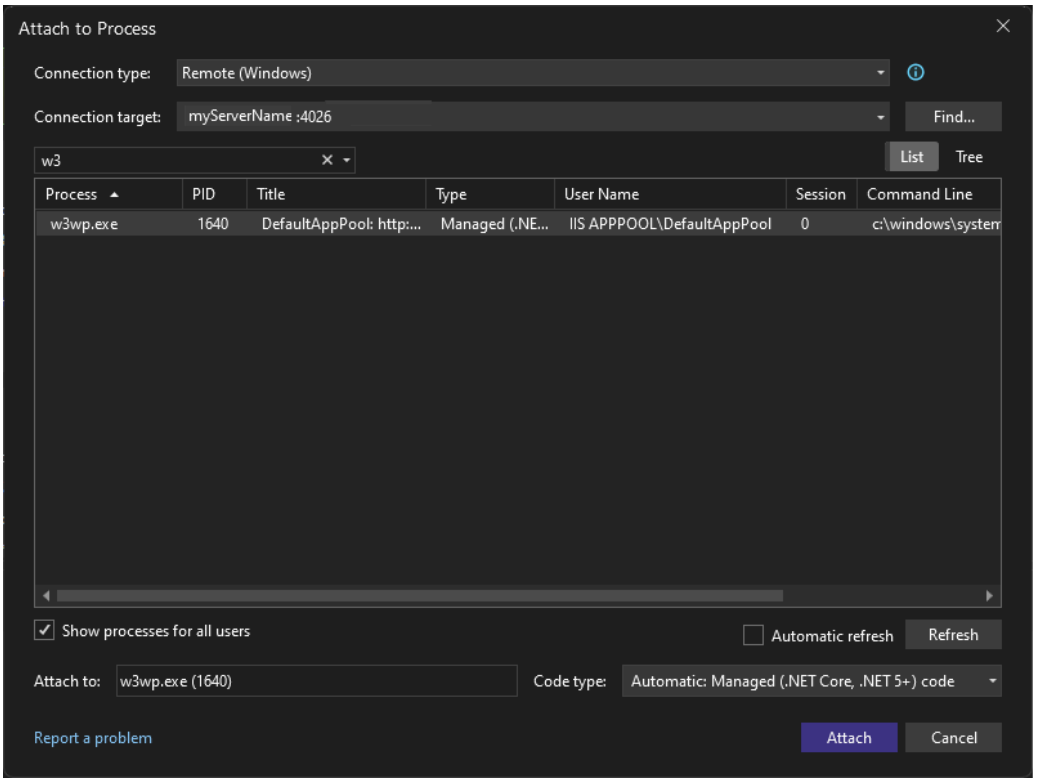
   If you want to use the **Find** button, you might need to open outbound UDP port 3702 on the server.

5. Check **Show processes from all users**.

6. Type the first letter of your process name to quickly find your app.

   - If you're using the in-process hosting model on IIS, select the correct **w3wp.exe** process. Starting in .NET Core 3, this process is the default.
   - Otherwise, select the **dotnet.exe** process. (This is the out-of-process hosting model.)

   If you have multiple processes showing *w3wp.exe* or *dotnet.exe*, check the **User Name** column. In some scenarios, the **User Name** column shows your app pool name, such as **IIS APPPOOL\DefaultAppPool**. If you see the App Pool, but it's not unique, create a new named App Pool for the app instance you want to debug, and then you can find it easily in the **User Name** column.



7. Select **Attach**.

8. Open the remote computer's website. In a browser, go to **http://<remote computer name>**.

   You should see the ASP.NET web page.

9. In the running ASP.NET application, select the link to the **Privacy** page.

   The breakpoint should be hit in Visual Studio.

   If you're unable to attach or hit the breakpoint, see Troubleshoot remote debugging.

Troubleshooting IIS deployment

- If you can't connect to the host using the host name, try the IP address instead.
- Make sure the required ports are open on the remote server.
- For ASP.NET Core, you need to make sure that the Application pool field for the **DefaultAppPool** is set to **No Managed Code**.
- Verify that the version of ASP.NET used in your app is the same as the version you installed on the server. For your app, you can view and set the version in the **Properties** page. To set the app to a different version, that version must be installed.
- If the app tried to open, but you see a certificate warning, choose to trust the site. If you already closed the warning, you can edit the publishing profile, a *.pubxml

file, in your project and add the following element (for test only):
`<AllowUntrustedCertificate>true</AllowUntrustedCertificate>`

- After it's deployed, start the app in IIS to test that it deployed correctly.
- Check the Output window in Visual Studio for status information, and check your error messages.

Open required ports on Windows Server

In most setups, required ports are opened by the installation of ASP.NET and the remote debugger. However, if you're troubleshooting deployment issues and the app is hosted behind a firewall, you might need to verify that the correct ports are open.

On an Azure VM, you must open ports through:

- The Network security group.
- The firewall on Windows Server

Required ports:

- 80 - Required for IIS
- 4026 - Required for remote debugging from Visual Studio 2022 (see Remote Debugger Port Assignments for more information).
- UDP 3702 - (Optional) The Discovery port enables you to the **Find** button when attaching to the remote debugger in Visual Studio. This must be an outbound port (outbound rule).

In addition, these ports should already be opened by the ASP.NET Core installation:

- 8172 - (Optional) Required for Web Deploy to deploy the app from Visual Studio

---

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Debug exceptions in .NET applications using Snapshot Debugger

Article • 09/11/2024

When enabled, Snapshot Debugger automatically collects a debug snapshot of the source code and variables when an exception occurs in your live .NET application. The Snapshot Debugger in Application Insights:

- Monitors system-generated logs from your web app.
- Collects snapshots on your top-throwing exceptions.
- Provides information you need to diagnose issues in production.

Learn more about the Snapshot Debugger and Snapshot Uploader processes.

Supported applications and environments

This section lists the applications and environments that are supported.

Applications

Snapshot collection is available for:

- .NET Framework 4.6.2 and newer versions.
- .NET 6.0 or later on Windows.

Environments

The following environments are supported:

- Azure App Service
- Azure Functions
- Azure Cloud Services running OS family 4 or later
- Azure Service Fabric running on Windows Server 2012 R2 or later
- Azure Virtual Machines and Azure Virtual Machine Scale Sets running Windows Server 2012 R2 or later
- On-premises virtual or physical machines running Windows Server 2012 R2 or later or Windows 8.1 or later

> **Note**

Client applications (for example, WPF, Windows Forms, or UWP) aren't supported.

Prerequisites for using Snapshot Debugger

Packages and configurations

- Include the Snapshot Collector NuGet package in your application.
- Configure collection parameters in ApplicationInsights.config.

Permissions

- Verify you're added to the Application Insights Snapshot Debugger role for the target **Application Insights Snapshot**.

How Snapshot Debugger works

The Snapshot Debugger is implemented as an Application Insights telemetry processor. When your application runs, the Snapshot Debugger telemetry processor is added to your application's system-generated logs pipeline.

> **Important**
>
> Snapshots might contain personal data or other sensitive information in variable and parameter values. Snapshot data is stored in the same region as your Application Insights resource.

Snapshot Debugger process

The Snapshot Debugger process starts and ends with the `TrackException` method. A process snapshot is a suspended clone of the running process, so that your users experience little to no interruption. In a typical scenario:

1. Your application throws the TrackException.
2. The Snapshot Debugger monitors exceptions as they're thrown by subscribing to the AppDomain.CurrentDomain.FirstChanceException event.
3. A counter is incremented for the problem ID.

- When the counter reaches the `ThresholdForSnapshotting` value, the problem ID is added to a collection plan.

> ⓘ **Note**
>
> The `ThresholdForSnapshotting` default minimum value is 1. With this value, your app has to trigger the same exception *twice* before a snapshot is created.

4. The exception event's problem ID is computed and compared against the problem IDs in the collection plan.

5. If there's a match between problem IDs, a **snapshot** of the running process is created.

   - The snapshot is assigned a unique identifier and the exception is stamped with that identifier.

   > ⓘ **Note**
   >
   > The snapshot creation rate is limited by the `SnapshotsPerTenMinutesLimit` setting. By default, the limit is one snapshot every 10 minutes.

6. After the `FirstChanceException` handler returns, the thrown exception is processed as normal.

7. The exception reaches the `TrackException` method again and is reported to Application Insights, along with the snapshot identifier.

> ⓘ **Note**
>
> Set `IsEnabledInDeveloperMode` to `true` if you want to generate snapshots while you debug in Visual Studio.

Snapshot Uploader process

While the Snapshot Debugger process continues to run and serve traffic to users with little interruption, the snapshot is handed off to the Snapshot Uploader process. In a typical scenario, the Snapshot Uploader:

1. Creates a minidump.

2. Uploads the minidump to Application Insights, along with any relevant symbol (*.pdb*) files.

> ⓘ **Note**
>
> No more than 50 snapshots per day can be uploaded.

If you enabled the Snapshot Debugger but you aren't seeing snapshots, see the Troubleshooting guide.

Upgrading Snapshot Debugger

Snapshot Debugger auto-upgrades via the built-in, preinstalled Application Insights site extension.

Manually adding an Application Insights site extension to keep Snapshot Debugger up-to-date is deprecated.

Overhead

The Snapshot Debugger is designed for use in production environments. The default settings include rate limits to minimize the impact on your applications.

However, you may experience small CPU, memory, and I/O overhead associated with the Snapshot Debugger, such as:

- When an exception is thrown in your application
- If the exception handler decides to create a snapshot
- When `TrackException` is called

There is **no additional cost** for storing data captured by Snapshot Debugger.

See example scenarios in which you may experience Snapshot Debugger overhead.

Limitations

This section discusses limitations for the Snapshot Debugger.

- **Data retention**

  Debug snapshots are stored for 15 days. The default data retention policy is set on a per-application basis. If you need to increase this value, you can request an

increase by opening a support case in the Azure portal. For each Application Insights instance, a maximum number of 50 snapshots are allowed per day.

- **Publish symbols**

  The Snapshot Debugger requires symbol files on the production server to:
  - Decode variables
  - Provide a debugging experience in Visual Studio

  By default, Visual Studio 2017 versions 15.2+ publishes symbols for release builds when it publishes to App Service.

  In prior versions, you must add the following line to your publish profile `.pubxml` file so that symbols are published in release mode:

  ```
  XML

  <ExcludeGeneratedDebugSymbol>False</ExcludeGeneratedDebugSymbol>
  ```

  For Azure Compute and other types, make sure that the symbol files are either:
  - In the same folder of the main application `.dll` (typically, `wwwroot/bin`), or
  - Available on the current path.

  For more information on the different symbol options that are available, see the Visual Studio documentation. For best results, we recommend that you use *Full*, *Portable*, or *Embedded*.

- **Optimized builds**

  In some cases, local variables can't be viewed in release builds because of optimizations applied by the JIT compiler.

  However, in App Service, the Snapshot Debugger can deoptimize throwing methods that are part of its collection plan.

  > **Tip**
  >
  > Install the Application Insights Site extension in your instance of App Service to get deoptimization support.

Next steps

Enable the Application Insights Snapshot Debugger for your application:

- Azure App Service
- Azure Functions
- Azure Cloud Services
- Azure Service Fabric
- Azure Virtual Machines and Virtual Machine Scale Sets
- On-premises virtual or physical machines

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Debug live ASP.NET Azure apps using the Snapshot Debugger

Article • 10/20/2022

The Snapshot Debugger takes a snapshot of your in-production apps when code that you're interested in executes. To instruct the debugger to take a snapshot, you set snappoints and logpoints in your code. The debugger lets you see exactly what went wrong, without impacting traffic of your production application. The Snapshot Debugger can help you dramatically reduce the time it takes to resolve issues that occur in production environments.

Snappoints and logpoints are similar to breakpoints, but unlike breakpoints, snappoints don't halt the application when hit. Typically, capturing a snapshot at a snappoint takes 10-20 milliseconds.

In this tutorial, you will:

- ✓ Start the Snapshot Debugger
- ✓ Set a snappoint and view a snapshot
- ✓ Set a logpoint

Prerequisites

- Snapshot Debugger is only available starting in Visual Studio 2017 Enterprise version 15.5 or higher with the **Azure development workload**. (Under the **Individual components** tab, you find it under **Debugging and testing** > **Snapshot debugger**.)

  If it's not already installed, install Visual Studio 2019. If you're updating from a previous Visual Studio installation, run the Visual Studio Installer and check the Snapshot Debugger component in the **ASP.NET and web development workload**.

- Basic or higher Azure App Service plan.

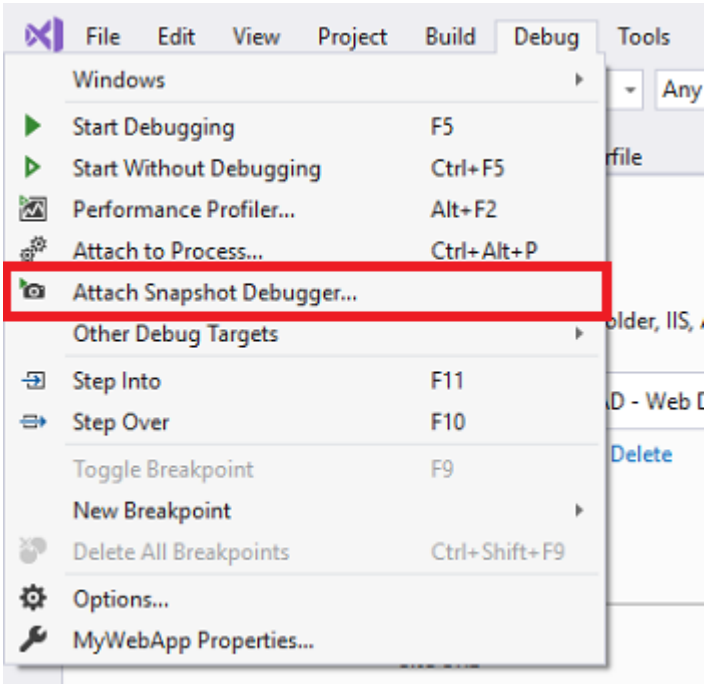
- Snapshot collection is available for the following web apps running in Azure App Service:
  - ASP.NET applications running on .NET Framework 4.6.1 or later.
  - ASP.NET Core applications running on .NET Core 2.0 or later on Windows.

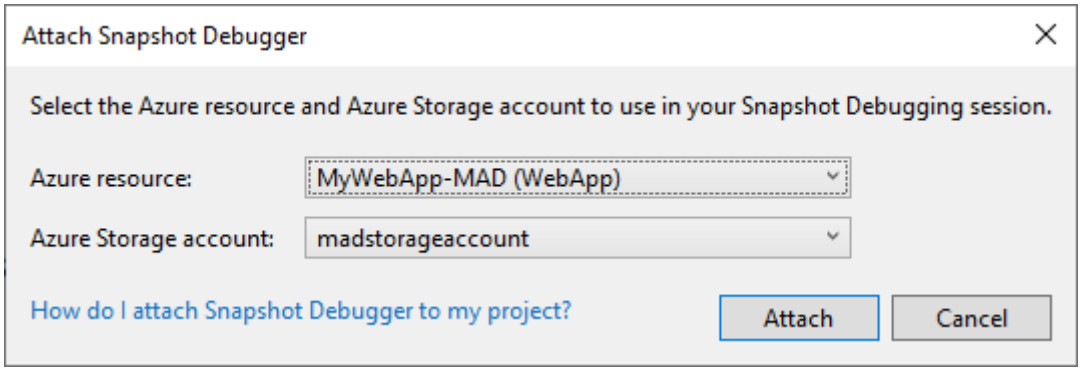
Open your project and start the Snapshot Debugger

1. Open the project you would like to snapshot debug.

   > **ⓘ Important**
   >
   > To snapshot debug, you need to open the *same version of source code* that is published to your Azure App Service.

2. Choose **Debug** > **Attach Snapshot Debugger...**. Select the Azure App Service your project is deployed to and an Azure storage account, and then click **Attach**. Snapshot Debugger also supports Azure Kubernetes Service and Azure Virtual Machines (VM) & Virtual Machine Scale Sets.





> **ⓘ Important**

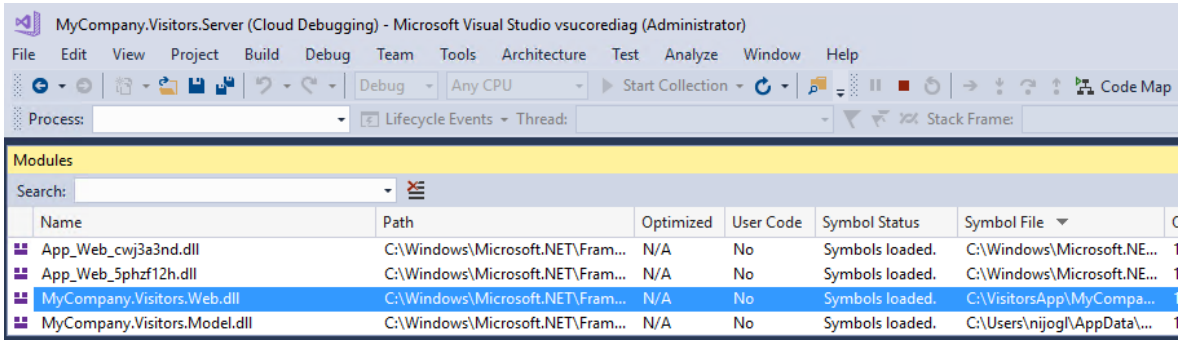
The first time you select **Attach Snapshot Debugger**, you're prompted to install the Snapshot Debugger site extension on your Azure App Service. This installation requires a restart of your Azure App Service.

> **Note**
>
> (Visual Studio 2019 version 16.2 and above) Snapshot Debugger has enabled Azure cloud support. Make sure that both the Azure resource and Azure Storage account you select are from the same cloud. Please contact your Azure administrator if you have questions about your enterprise's Azure compliance configurations.

Visual Studio is now in snapshot debugging mode.

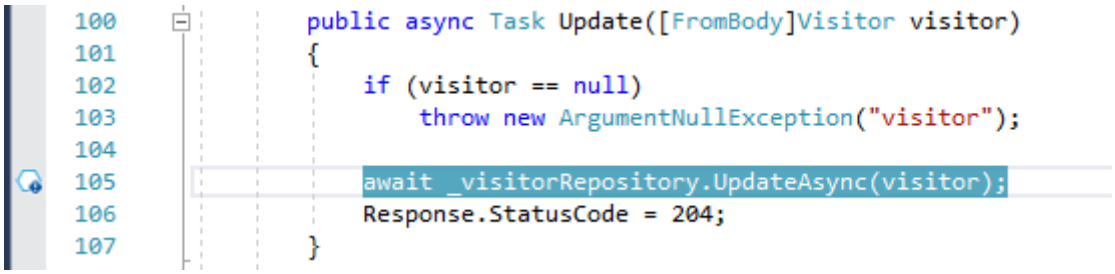


The **Modules** window shows you when all the modules have loaded for the Azure App Service (choose **Debug** > **Windows** > **Modules** to open this window).

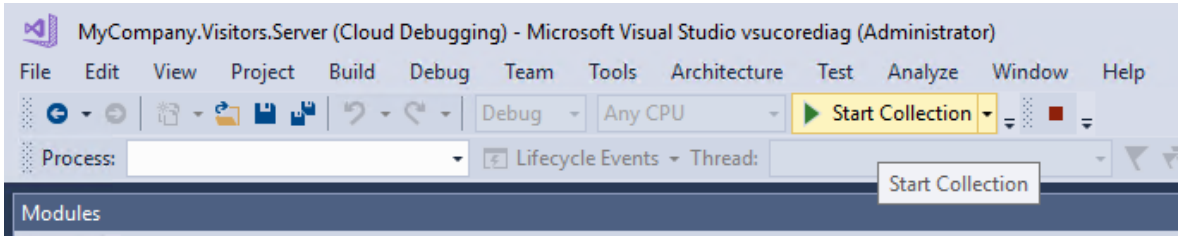


Set a snappoint

1. In the code editor, click the left gutter next to a line of code you're interested in to set a snappoint. Make sure it's code that you know will execute.



2. Click **Start Collection** to turn on the snappoint.



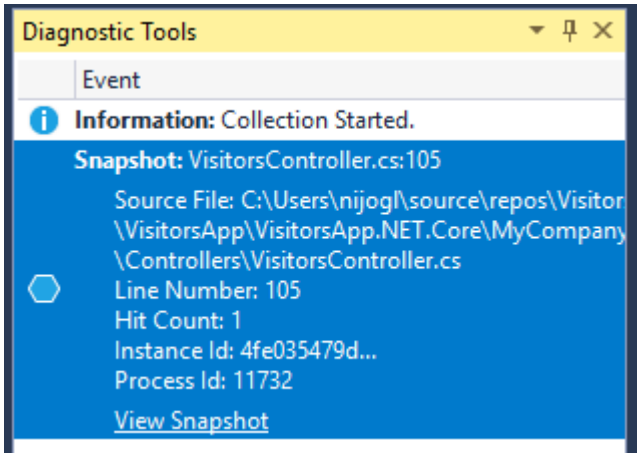
> **Tip**
>
> You can't step when viewing a snapshot, but you can place multiple snappoints in your code to follow execution at different lines of code. If you have multiple snappoints in your code, the Snapshot Debugger makes sure that the corresponding snapshots are from the same end-user session. The Snapshot Debugger does this even if there are many users hitting your app.

Take a snapshot

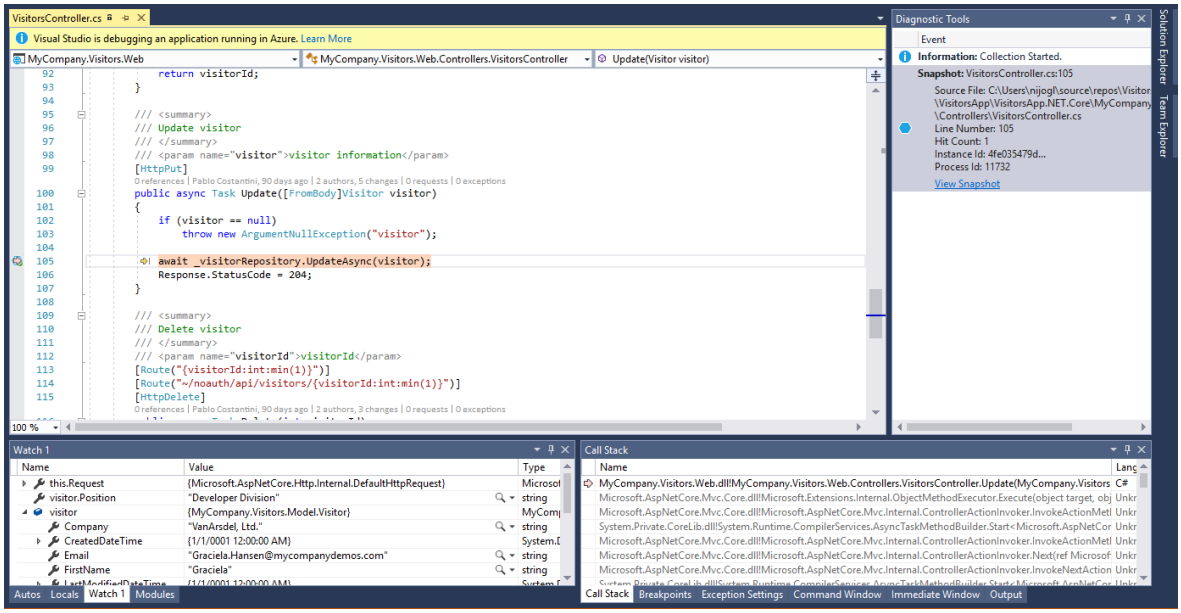
Once a snappoint is set, you can either manually generate a snapshot by going to the browser view of your web site and running the line of code marked or wait for your users to generate one from their usage of the site.

Inspect snapshot data

1. When the snappoint is hit, a snapshot appears in the Diagnostic Tools window. To open this window, choose **Debug** > **Windows** > **Show Diagnostic Tools**.



2. Double-click the snappoint to open the snapshot in the code editor.



From this view, you can hover over variables to view DataTips, use the **Locals**, **Watches**, and **Call Stack** windows, and also evaluate expressions.

The website itself is still live and end users aren't affected. Only one snapshot is captured per snappoint by default: after a snapshot is captured the snappoint turns off. If you want to capture another snapshot at the snappoint, you can turn the snappoint back on by clicking **Update Collection**.

You can also add more snappoints to your app and turn them on with the **Update Collection** button.

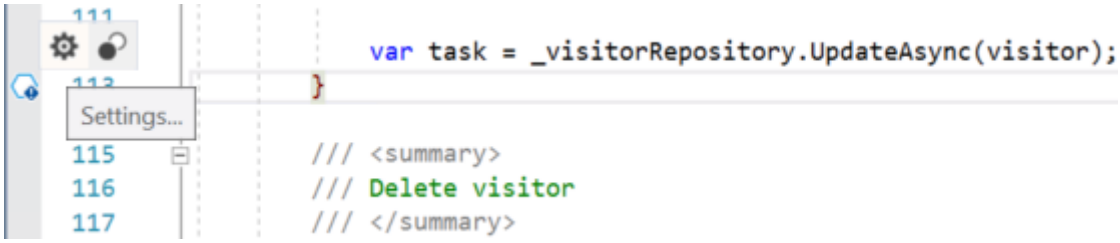
Need help? See the Troubleshooting and known issues and FAQ for snapshot debugging pages.

Set a conditional snappoint

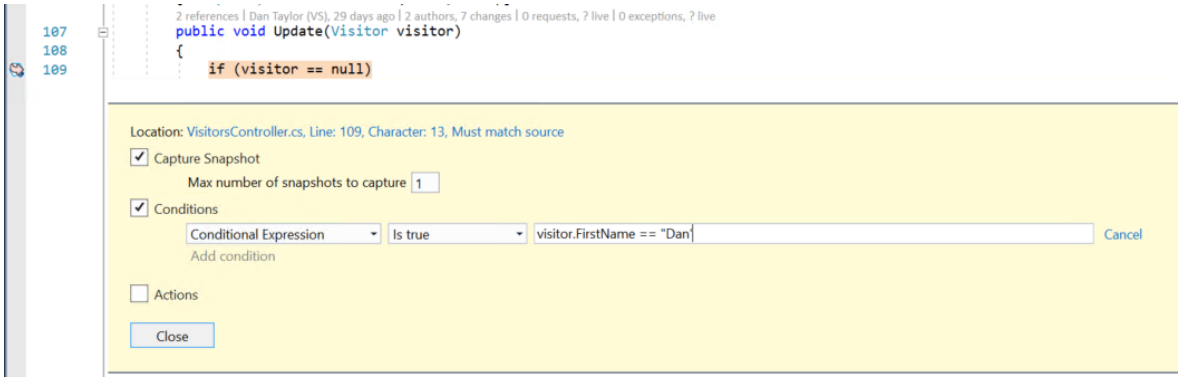
If it's difficult to recreate a particular state in your app, consider using a conditional snappoint. Conditional snappoints help you control when to take a snapshot such as when a variable contains a particular value that you want to inspect. You can set conditions using expressions, filters, or hit counts.

To create a conditional snappoint

1. Right-click a snappoint icon (the hollow sphere) and choose **Settings**.



2. In the snappoint settings window, type an expression.



In the preceding illustration, the snapshot is only taken for the snappoint when `visitor.FirstName == "Dan"`.

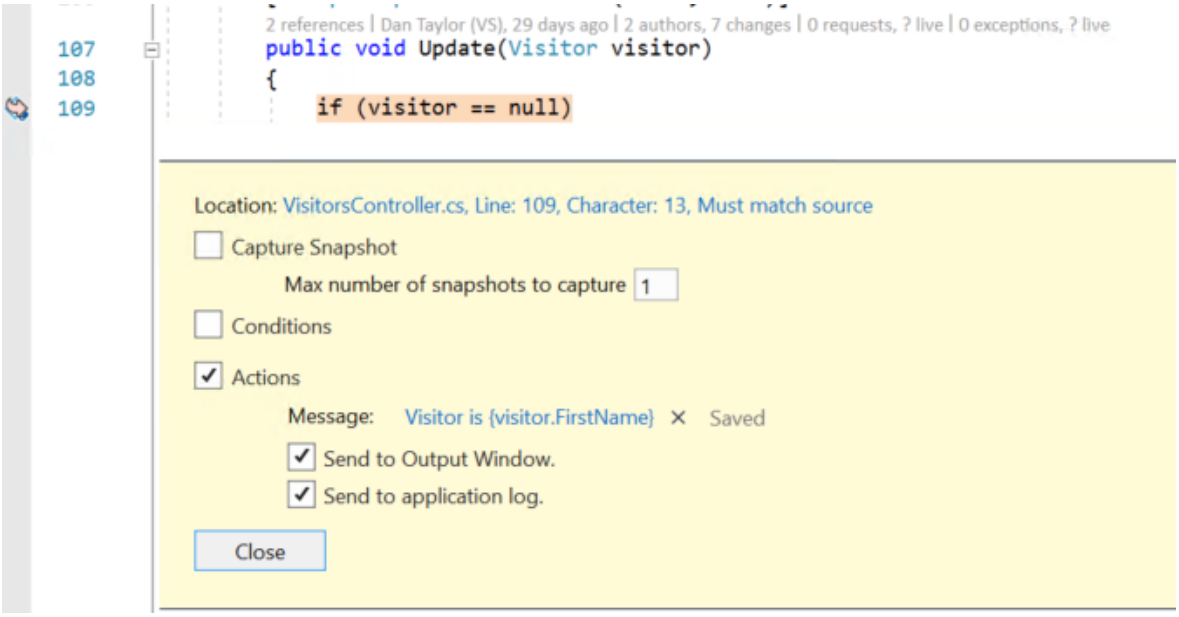
Set a logpoint

In addition to taking a snapshot when a snappoint is hit, you can also configure a snappoint to log a message (that is, create a logpoint). You can set logpoints without having to redeploy your app. Logpoints are executed virtually and cause no impact or side effects to your running application.

To create a logpoint

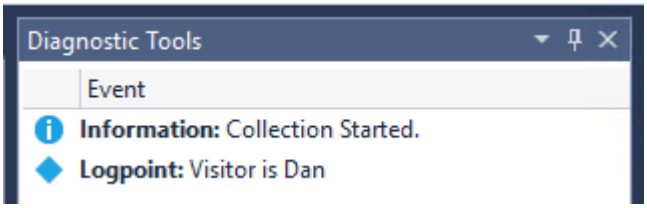
1. Right-click a snappoint icon (the blue hexagon) and choose **Settings**.

2. In the snappoint settings window, select **Actions**.



3. In the **Message** field, you can enter the new log message you want to log. You can also evaluate variables in your log message by placing them inside curly braces.

   If you choose **Send to Output Window**, when the logpoint is hit, the message appears in the Diagnostic Tools window.



   If you choose **Send to application log**, when the logpoint is hit, the message appears anywhere that you can see messages from `System.Diagnostics.Trace` (or `ILogger` in .NET Core), such as App Insights.

Related content

In this tutorial, you've learned how to use the Snapshot Debugger for App Services. You may want to read more details about this feature.

FAQ for snapshot debugging

Feedback

Was this page helpful? Yes No

Use dev tunnels in Visual Studio to debug your web APIs

Article • 11/14/2023

To quickly debug and test your web APIs within Microsoft Power Automate or Power Apps, use *dev tunnels* in Visual Studio. Dev tunnels enables ad-hoc connections between machines that can't directly connect to each other. Once you enable this feature, you'll see that debugging (F5) automatically creates a dev tunnel URL that you can use to connect to Power Apps or Power Automate.

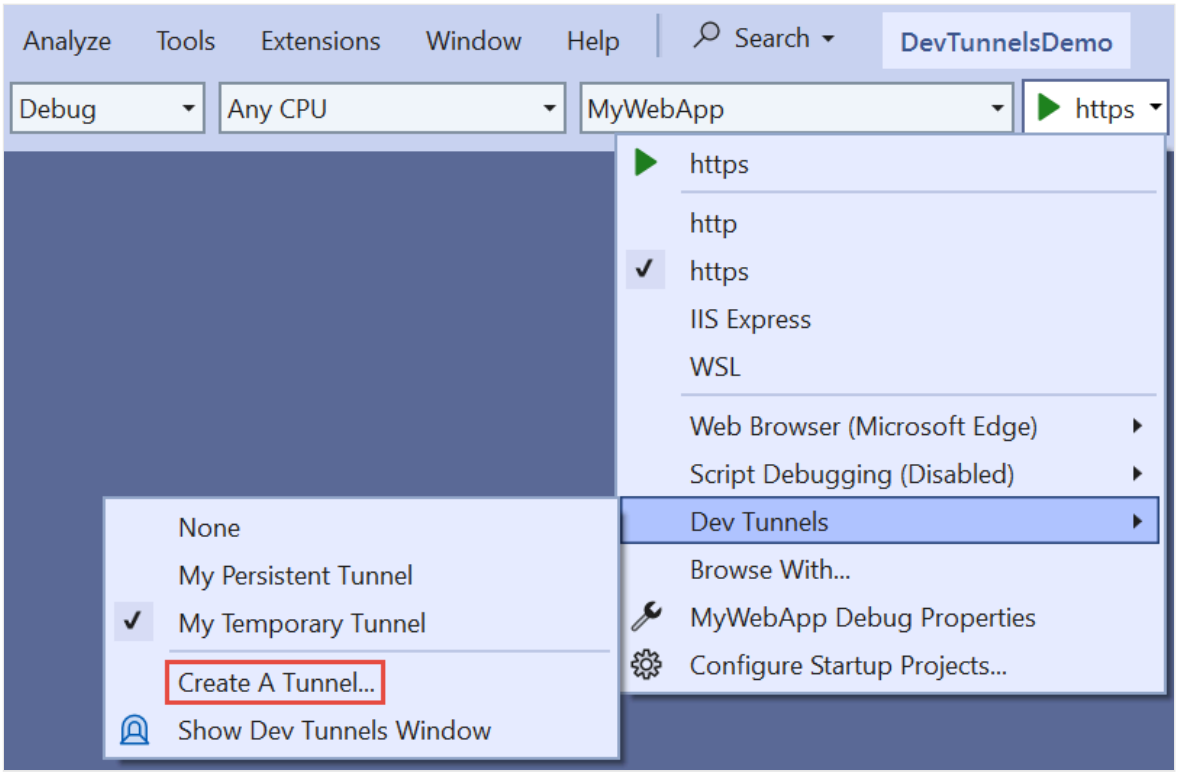
Prerequisites

- Download Visual Studio 2022 Preview version 17.6 or later with the **ASP.NET and web development workload** installed. You need to sign in to Visual Studio to create and use dev tunnels. The feature isn't available in Visual Studio for Mac.
- One of the following Power Platform environments:
  - Power Automate
  - Power Apps

> **Note**
>
> If you need help getting started with Microsoft Power Platform, go to Create a developer environment.

Step 1: Configure your ASP.NET Core project in Visual Studio

1. In the debug dropdown menu, select **Dev Tunnels** > **Create A Tunnel**.



2. The tunnel creation dialog opens and you can configure dev tunnels. Make sure to set authentication type to **Public**.

   To learn more, go to How to use dev tunnels in Visual Studio 2022 with ASP.NET Core apps.

3. Select **OK**. Visual Studio displays confirmation of tunnel creation. The tunnel is now enabled and appears in the debug dropdown **Dev Tunnels** flyout.

4. Select **F5** (**Debug** > **Start Debugging**), or the **Start Debugging** button to see the dev tunnel URL.

URL with and without dev tunnels

To learn more, go to Use a tunnel.

- **Before debugging:** `https://localhost:7223/swagger/index.html`
- **After debugging:** `https://50tt58xr-7223.usw2.devtunnels.ms/swagger/indexf.html`

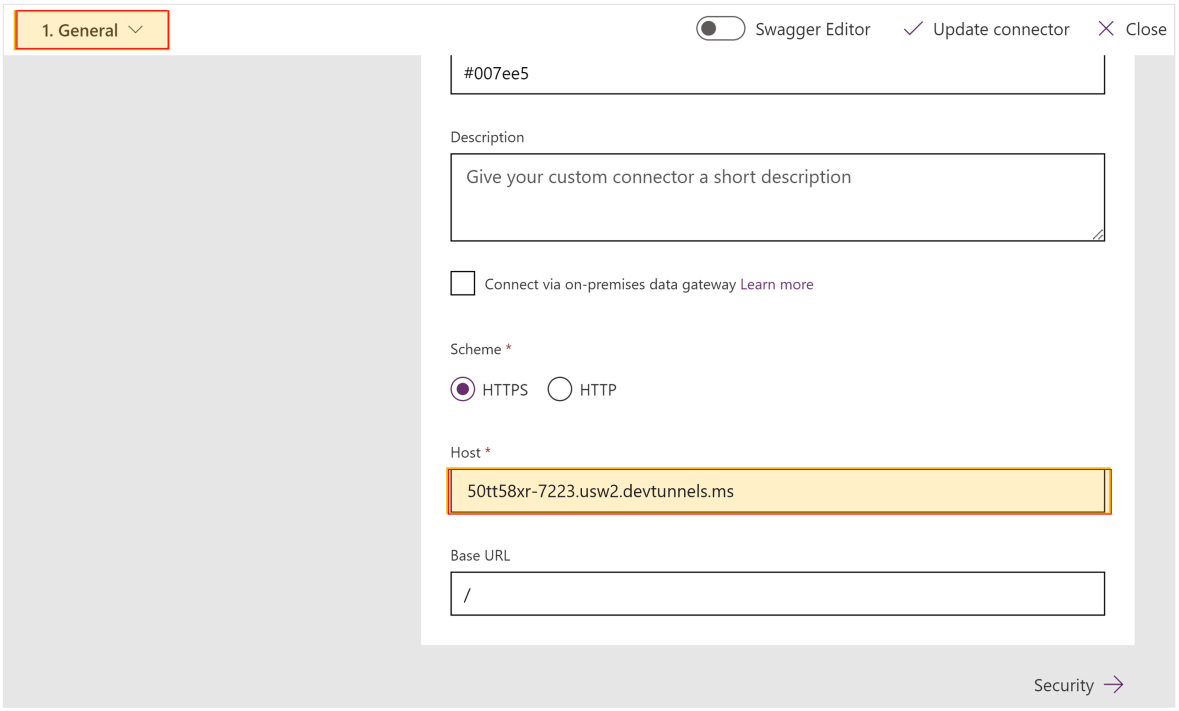
Step 2: Create a custom connector for your web API using the dev tunnel URL

A custom connector is a wrapper around a REST API and allows Power Automate or Power Apps solutions to communicate with your web API. There are many ways to

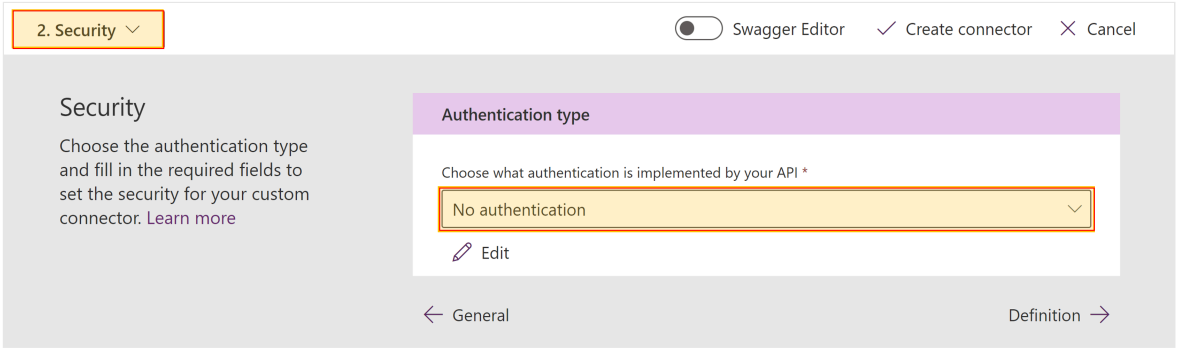
create a custom connector. The following sections explain how to use the dev tunnel URL and create a custom connector from scratch, or with API Management.

Create a custom connector from scratch

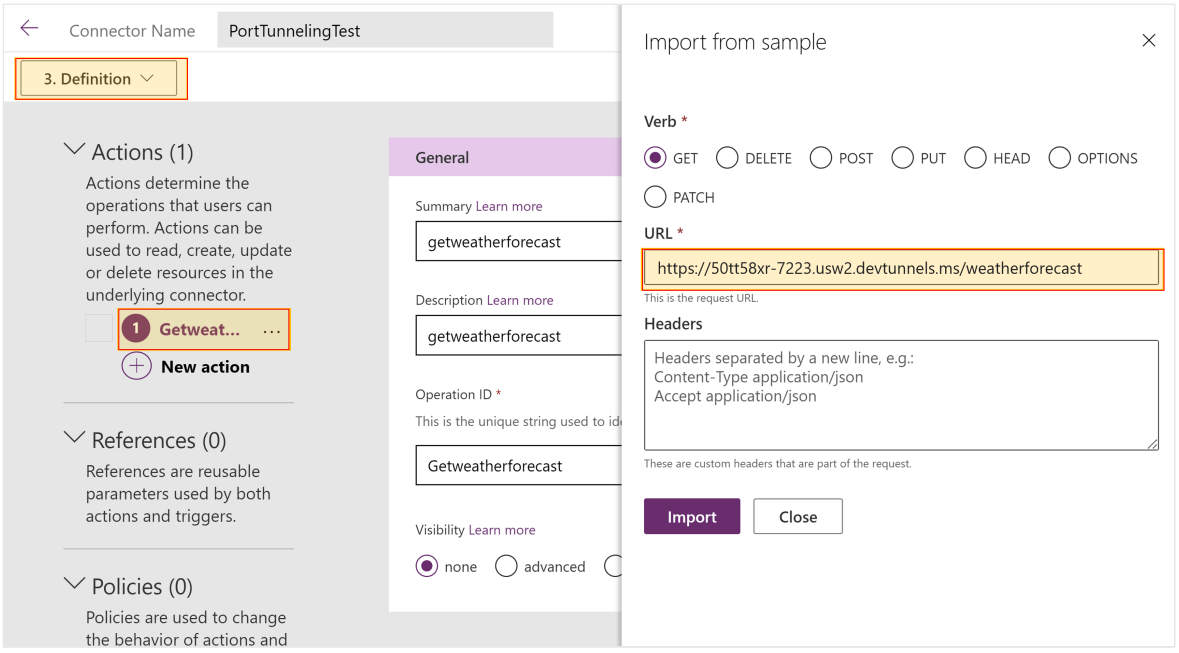
1. On the **General** tab, post the dev tunnel URL into the **Host** field.



2. On the **Security** tab, select **No authentication** from the dropdown menu.



3. On the **Definition** tab, define your HTTP methods by adding actions. For the URL action, use the dev tunnel base URL + /actionName. For an example, go to How to use dev tunnels.

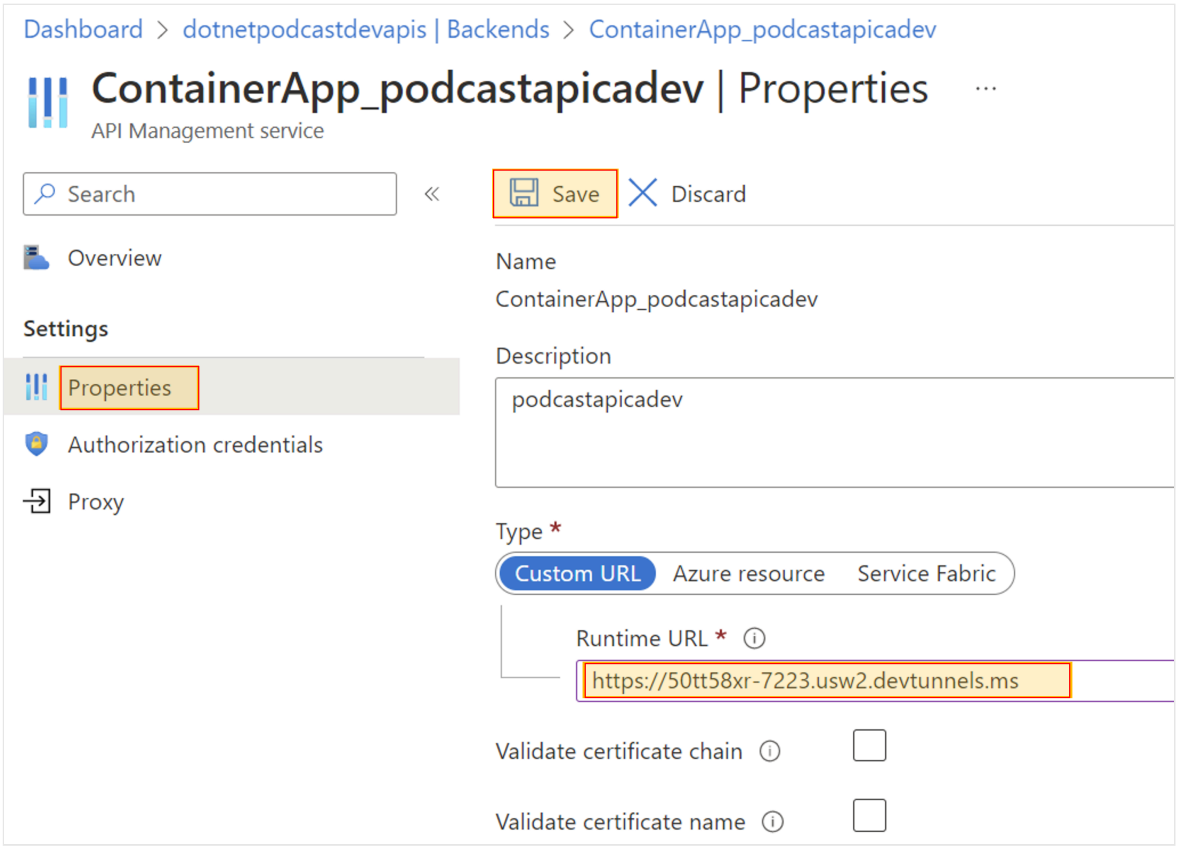


4. You can now test your custom connector. To do this, select the **Test** tab. After adding your connection, you can test your web API.

   For instructions, go to Create a custom connector from scratch.

Create a custom connector with API Management

1. Go to your Azure API Management instance in the Azure portal.
2. Modify the runtime URL of your API in the menu under **Backends** and select your API instance.
3. On the **Properties** tab, replace the **Runtime URL** with the dev tunnel URL and select **Save**.



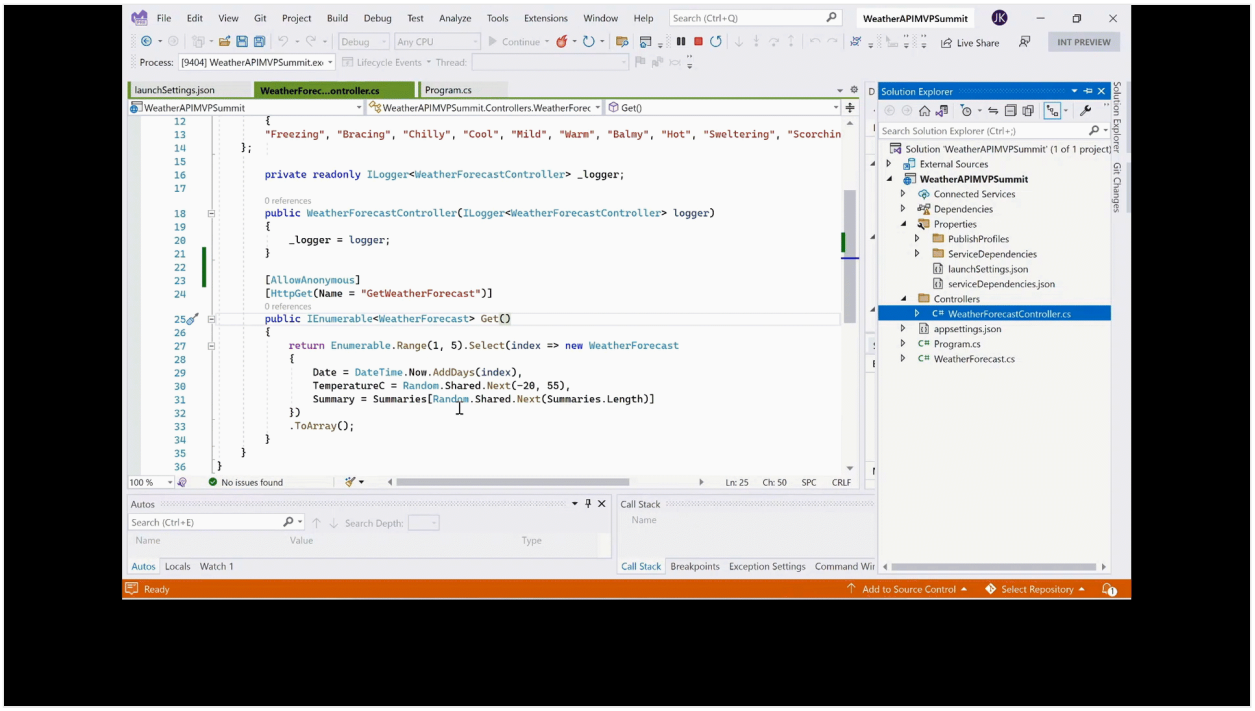
4. On the **Power Platform** tab, you can now create a custom connector. For instructions, go to Export APIs from Azure API Management to the Power Platform.

Step 3: Add the custom connector to Power Apps or Power Automate

To debug your web API, use a custom connector from a Power Apps app or a Power Automate flow.

When your custom connector is integrated in your Power Platform solution, you can set a *breakpoint*, and debug your Power Apps app or Power Automate flow.

> **Note**
>
> Breakpoints are the most basic and essential feature of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code so you review the values of variables, behavior of memory, or if a branch of code is getting run.



Provide feedback

We greatly appreciate feedback on issues with our connector platform, or new feature ideas. To provide feedback, go to Submit issues or get help with connectors and select your feedback type.

Troubleshoot and debug ASP.NET Core projects

Article • 04/11/2023

By Rick Anderson

The following links provide troubleshooting guidance:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- NDC Conference (London, 2018): Diagnosing issues in ASP.NET Core Applications
- ASP.NET Blog: Troubleshooting ASP.NET Core Performance Problems

.NET Core SDK warnings

Both the 32-bit and 64-bit versions of the .NET Core SDK are installed

In the **New Project** dialog for ASP.NET Core, you may see the following warning:

> Both 32-bit and 64-bit versions of the .NET Core SDK are installed. Only templates from the 64-bit versions installed at 'C:\Program Files\dotnet\sdk\' are displayed.

This warning appears when both 32-bit (x86) and 64-bit (x64) versions of the .NET Core SDK are installed. Common reasons both versions may be installed include:

- You originally downloaded the .NET Core SDK installer using a 32-bit machine but then copied it across and installed it on a 64-bit machine.
- The 32-bit .NET Core SDK was installed by another application.
- The wrong version was downloaded and installed.

Uninstall the 32-bit .NET Core SDK to prevent this warning. Uninstall from **Control Panel** > **Programs and Features** > **Uninstall or change a program**. If you understand why the warning occurs and its implications, you can ignore the warning.

The .NET Core SDK is installed in multiple locations

In the **New Project** dialog for ASP.NET Core, you may see the following warning:

> The .NET Core SDK is installed in multiple locations. Only templates from the SDKs installed at 'C:\Program Files\dotnet\sdk\' are displayed.

You see this message when you have at least one installation of the .NET Core SDK in a directory outside of *C:\Program Files\dotnet\sdk*. Usually this happens when the .NET Core SDK has been deployed on a machine using copy/paste instead of the MSI installer.

Uninstall all 32-bit .NET Core SDKs and runtimes to prevent this warning. Uninstall from **Control Panel** > **Programs and Features** > **Uninstall or change a program**. If you understand why the warning occurs and its implications, you can ignore the warning.

No .NET Core SDKs were detected

- In the Visual Studio **New Project** dialog for ASP.NET Core, you may see the following warning:

  > No .NET Core SDKs were detected, ensure they are included in the environment variable `PATH`.

- When executing a `dotnet` command, the warning appears as:

  > It was not possible to find any installed dotnet SDKs.

These warnings appear when the environment variable `PATH` doesn't point to any .NET Core SDKs on the machine. To resolve this problem:

- Install the .NET Core SDK. Obtain the latest installer from .NET Downloads.
- Verify that the `PATH` environment variable points to the location where the SDK is installed (`C:\Program Files\dotnet\` for 64-bit/x64 or `C:\Program Files (x86)\dotnet\` for 32-bit/x86). The SDK installer normally sets the `PATH`. Always install the same bitness SDKs and runtimes on the same machine.

Missing SDK after installing the .NET Core Hosting Bundle

Installing the .NET Core Hosting Bundle modifies the `PATH` when it installs the .NET Core runtime to point to the 32-bit (x86) version of .NET Core (`C:\Program Files (x86)\dotnet\`). This can result in missing SDKs when the 32-bit (x86) .NET Core `dotnet` command is used (No .NET Core SDKs were detected). To resolve this problem, move

`C:\Program Files\dotnet\` to a position before `C:\Program Files (x86)\dotnet\` on the `PATH`.

Obtain data from an app

If an app is capable of responding to requests, you can obtain the following data from the app using middleware:

- Request: Method, scheme, host, pathbase, path, query string, headers
- Connection: Remote IP address, remote port, local IP address, local port, client certificate
- Identity: Name, display name
- Configuration settings
- Environment variables

Place the following middleware code at the beginning of the `Startup.Configure` method's request processing pipeline. The environment is checked before the middleware is run to ensure that the code is only executed in the Development environment.

To obtain the environment, use either of the following approaches:

- Inject the `IHostingEnvironment` into the `Startup.Configure` method and check the environment with the local variable. The following sample code demonstrates this approach.

- Assign the environment to a property in the `Startup` class. Check the environment using the property (for example, `if (Environment.IsDevelopment())`).

C#

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IConfiguration config)
{
    if (env.IsDevelopment())
    {
        app.Run(async (context) =>
        {
            var sb = new StringBuilder();
            var nl = System.Environment.NewLine;
            var rule = string.Concat(nl, new string('-', 40), nl);
            var authSchemeProvider = app.ApplicationServices
                .GetRequiredService<IAuthenticationSchemeProvider>();

            sb.Append($"Request{rule}");
            sb.Append($"{DateTimeOffset.Now}{nl}");
```

```
            sb.Append($"{context.Request.Method} {context.Request.Path}
{nl}");
            sb.Append($"Scheme: {context.Request.Scheme}{nl}");
            sb.Append($"Host: {context.Request.Headers["Host"]}{nl}");
            sb.Append($"PathBase: {context.Request.PathBase.Value}{nl}");
            sb.Append($"Path: {context.Request.Path.Value}{nl}");
            sb.Append($"Query: {context.Request.QueryString.Value}{nl}
{nl}");

            sb.Append($"Connection{rule}");
            sb.Append($"RemoteIp: {context.Connection.RemoteIpAddress}
{nl}");
            sb.Append($"RemotePort: {context.Connection.RemotePort}{nl}");
            sb.Append($"LocalIp: {context.Connection.LocalIpAddress}{nl}");
            sb.Append($"LocalPort: {context.Connection.LocalPort}{nl}");
            sb.Append($"ClientCert: {context.Connection.ClientCertificate}
{nl}{nl}");

            sb.Append($"Identity{rule}");
            sb.Append($"User: {context.User.Identity.Name}{nl}");
            var scheme = await authSchemeProvider
                .GetSchemeAsync(IISDefaults.AuthenticationScheme);
            sb.Append($"DisplayName: {scheme?.DisplayName}{nl}{nl}");

            sb.Append($"Headers{rule}");
            foreach (var header in context.Request.Headers)
            {
                sb.Append($"{header.Key}: {header.Value}{nl}");
            }
            sb.Append(nl);

            sb.Append($"WebSockets{rule}");
            if (context.Features.Get<IHttpUpgradeFeature>() != null)
            {
                sb.Append($"Status: Enabled{nl}{nl}");
            }
            else
            {
                sb.Append($"Status: Disabled{nl}{nl}");
            }

            sb.Append($"Configuration{rule}");
            foreach (var pair in config.AsEnumerable())
            {
                sb.Append($"{pair.Path}: {pair.Value}{nl}");
            }
            sb.Append(nl);

            sb.Append($"Environment Variables{rule}");
            var vars = System.Environment.GetEnvironmentVariables();
            foreach (var key in vars.Keys.Cast<string>().OrderBy(key => key,
                StringComparer.OrdinalIgnoreCase))
            {
                var value = vars[key];
                sb.Append($"{key}: {value}{nl}");
```

```
            }

            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync(sb.ToString());
        });
    }
}
```

Debug ASP.NET Core apps

The following links provide information on debugging ASP.NET Core apps.

- Debugging ASP Core on Linux
- Debugging .NET Core on Unix over SSH
- Quickstart: Debug ASP.NET with the Visual Studio debugger
- See this GitHub issue for more debugging information.

Troubleshoot ASP.NET Core on Azure App Service and IIS

Article • 07/23/2024

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, the ASP.NET Core project default server is Kestrel. Visual studio can be configured to use IIS Express. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally with IIS Express can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:

    - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
    - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
    - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module (ANCM) for IIS.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

This error also may occur when the .NET Core Hosting Bundle isn't installed or is corrupted. Installing or repairing the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express) may fix the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
```

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not
found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error, either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file

deployment model. Use **one** of the following approaches in the app's project file to fix this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.

2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:

    - If deploying a 32-bit (x86) app, set the value to `True`.
    - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Failed to start application (ErrorCode '0x800701b1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x800701b1'.
```

The app failed to start because a Windows Service failed to load.

One common service that needs to be enabled is the "null" service. The following command enables the `null` Windows Service:

Windows Command Prompt

```
sc.exe start null
```

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

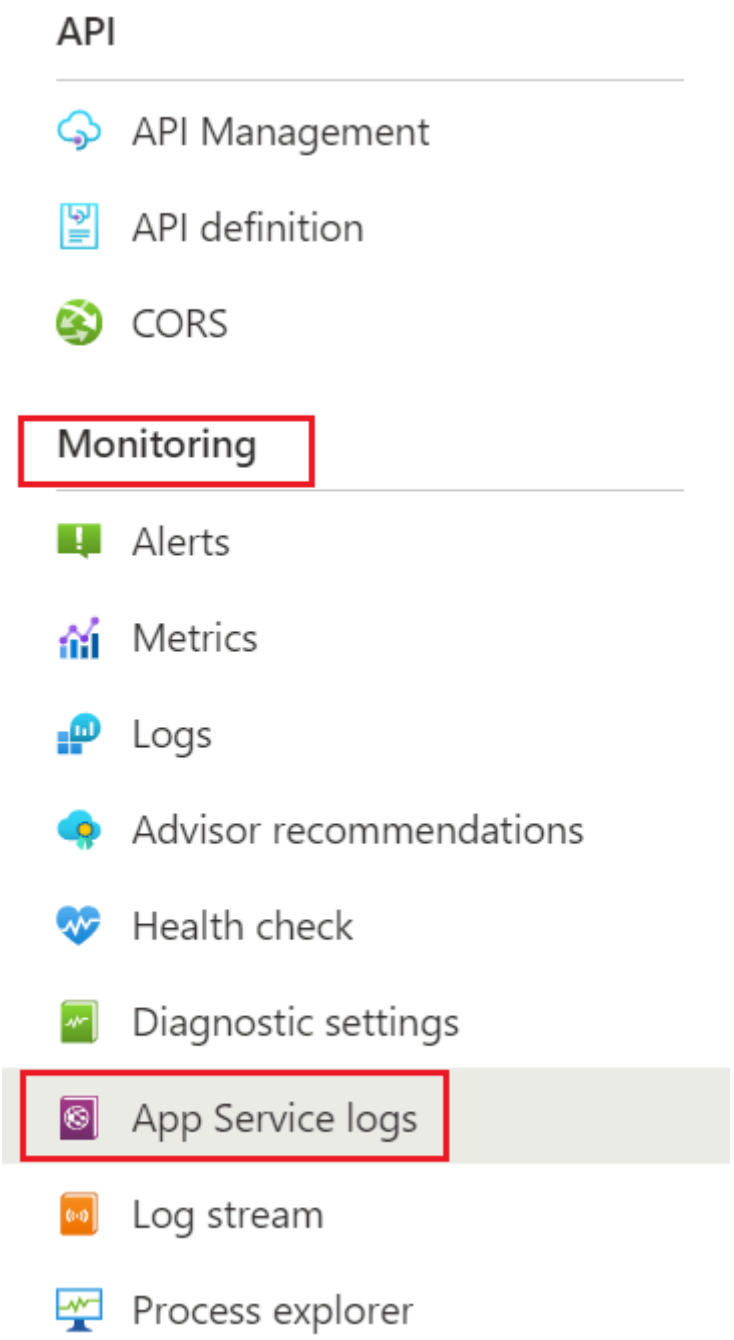
> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Azure App Services Log stream

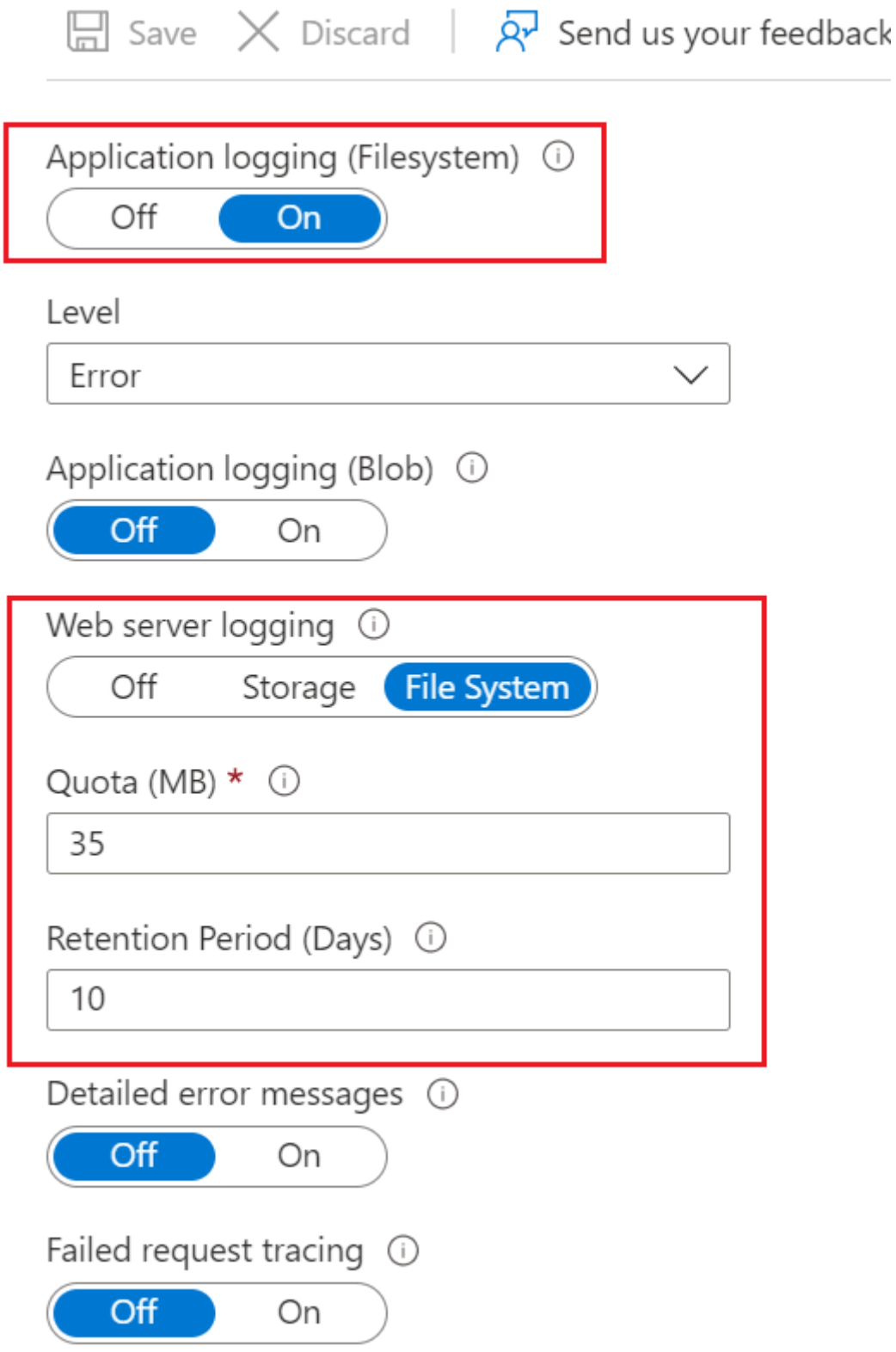
The Azure App Services Log streams logging information as it occurs. To view streaming logs:

1. In the Azure portal, open the app in **App Services**.

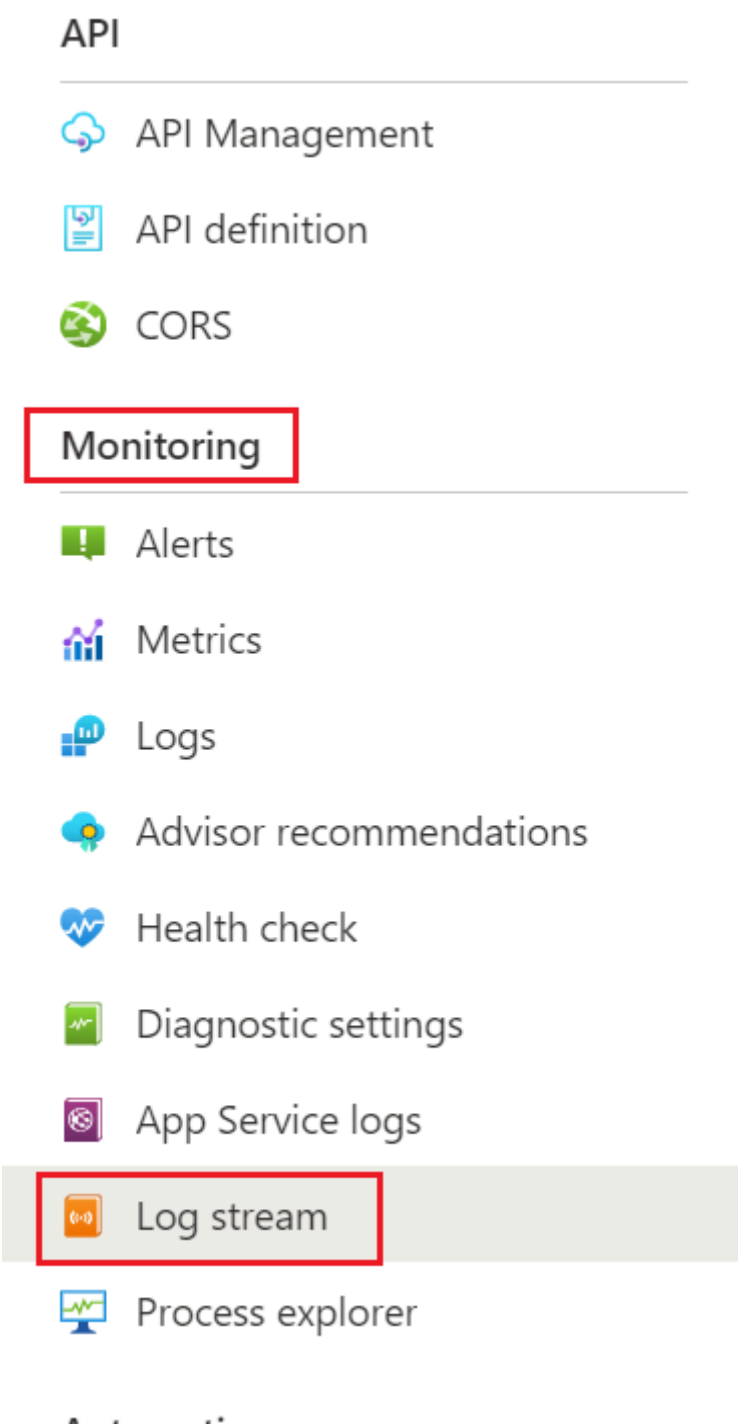
2. In the left pane, navigate to **Monitoring** > **App Service Logs**.



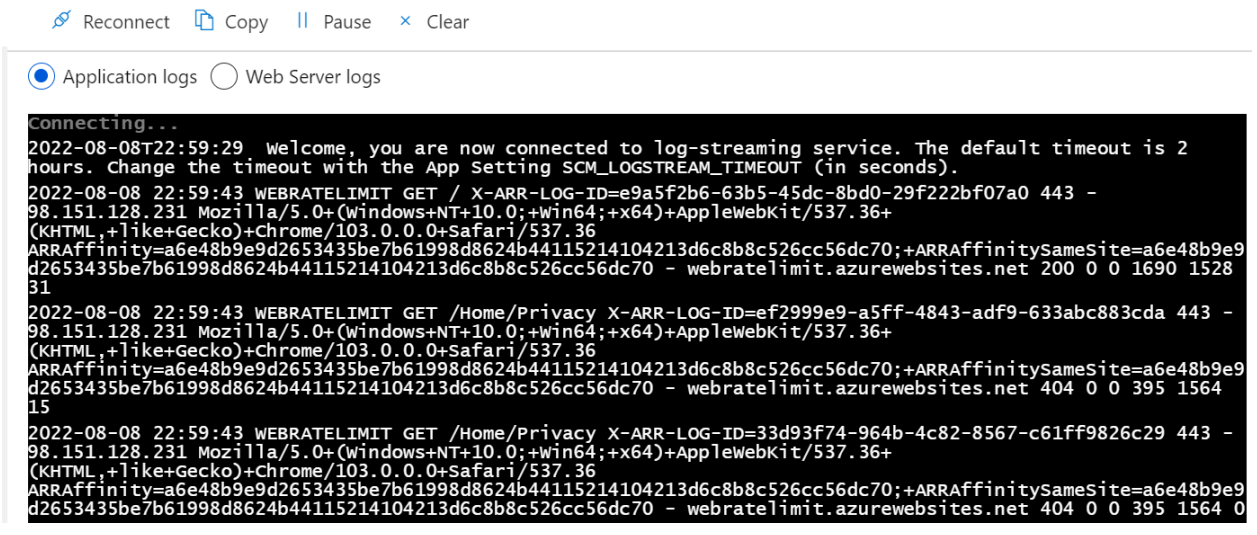
3. Select **File System** for **Web Server Logging**. Optionally enable **Application logging**.



4. In the left pane, navigate to **Monitoring** > **Log stream**, and then select **Application logs** or **Web Server Logs**.



The following images shows the application logs output:



Streaming logs have some latency and might not display immediately.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the `eventlog.xml` file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:

- If the app is a framework-dependent deployment:

  .NET CLI

  ```
  dotnet .\{ASSEMBLY NAME}.dll
  ```

- If the app is a self-contained deployment:

  Console

  ```
  {ASSEMBLY NAME}.exe
  ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:

  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

- If the app is a self-contained deployment:

  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.
4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module (ANCM) for IIS.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:

- Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
- Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
  a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
  b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
  c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see Log creation and redirection with the ASP.NET Core Module.

> ⚠ **Warning**
>
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or nonresponsive app (Azure App Service)

When an app responds slowly or doesn't respond to a request, see Troubleshoot slow web app performance issues in Azure App Service.

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.

5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`).

`stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.

4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module (ANCM) for IIS.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

XML

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see Log creation and redirection with the ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

XML

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
/>
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or non-responding app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.

2. Run the EnableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> ⚠ **Warning**
>
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App doesn't respond, fails during startup, or runs normally

When an app stops responding but doesn't crash, fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Debug .NET and ASP.NET Core source code with Visual Studio
- Troubleshoot and debug ASP.NET Core projects
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module (ANCM) for IIS

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common error troubleshooting for Azure App Service and IIS with ASP.NET Core

Article • 07/26/2024

This topic describes the most common errors and provides troubleshooting advice when hosting ASP.NET Core apps on Azure Apps Service and IIS.

See Troubleshoot ASP.NET Core on Azure App Service and IIS information on common app startup errors and instructions on how to diagnose errors.

Collect the following information:

- Browser behavior such as status code and error message.
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS

    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.

- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**

ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and

Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.

- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure

- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely

- **ASP.NET Core Module stdout Log:** The log file is created but empty.

- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED **--OR--** Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied **--OR--** 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. **--OR--** This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  XML

  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not

found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.

- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Downloads page and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure

- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module (ANCM) for IIS.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure **--OR--** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Code analysis in ASP.NET Core apps

Article • 07/08/2024

.NET compiler platform analyzers inspect application code for code quality and style issues. This document lists diagnostics for ASP.NET Core. For information on .NET diagnostics, see Overview of .NET source code analysis.

The following list contains the diagnostics for ASP.NET Core. Not all of the diagnostics shown are available in older versions of ASP.NET Core.

Diagnostic ID:

- ASP0000
- ASP0001
- ASP0003
- ASP0004
- ASP0005
- ASP0006
- ASP0007
- ASP0008
- ASP0009
- ASP0010
- ASP0011
- ASP0012
- ASP0013
- ASP0014
- ASP0015
- ASP0016
- ASP0017
- ASP0018
- ASP0019
- ASP0020
- ASP0021
- ASP0022
- ASP0023
- ASP0024
- ASP0025
- ASP0026
- BL0001
- BL0002
- BL0003

- BL0004
- BL0005
- BL0006
- MVC1000
- MVC1001
- MVC1002
- MVC1003
- MVC1004
- MVC1005
- MVC1006

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP0000: Do not call 'IServiceCollection.BuildServiceProvider' in 'ConfigureServices'

Article • 06/03/2022

Expand table

	Value
Rule ID	ASP0000
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A call to BuildServiceProvider was detected in the application start up code.

Rule description

Calling 'BuildServiceProvider' from application code results in more than one copy of singleton services being created which might result in incorrect application behavior. Consider alternatives such as dependency injecting services as parameters to 'Configure'.

How to fix violations

Remove the call to `BuildServiceProvider` from the application startup code.

When to suppress warnings

It is safe to suppress this rule if updating the application to remove the call to `BuildServiceProvider` is non-trivial and you have thoroughly tested the application to ensure multiple singleton services are not added.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

ASP0001: Authorization middleware is incorrectly configured

Article • 06/03/2022

Expand table

	Value
Rule ID	ASP0001
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

An out of order call to UseAuthorization was detected in the application start up code.

Rule description

For authorization to be effective for endpoint routes, the call to `UseAuthorization` should appear between the calls to `UseRouting` and `UseEndpoints`. In the absence of this, the fallback policy, if configured, will be used to authorize all requests.

Consider the following code:

C#

```
app.UseStaticFiles();
app.UseAuthorization();

app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapDefaultControllerRoute();
});
```

The call to `UseAuthorization` appears before `UseRouting` and consequently is not endpoint aware.

How to fix violations

Change the order in which the call to `UseAuthorization` and `UseRouting` are performed.

C#

```
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapDefaultControllerRoute();
});
```

When to suppress warnings

It is safe to suppress this rule if the call to `UseAuthorization` is intended to authorize the fallback policy on all outgoing requests, or is meant to authorize resources not routed using endpoint routing.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP0003: Do not use model binding attributes with route handlers

Article • 09/28/2022

	Value
Rule ID	ASP0003
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A route handler delegate includes a parameter that has a [Bind] attribute.

Rule description

Route handler endpoints only support a subset of MVC attributes such as [FromRoute], [FromBody], etc. Unsupported attributes are ignored and result in unexpected binding behavior. For example, the following code results in an analyzer warning:

C#

```
app.MapGet("/todos/{id}", ([Bind] int id) => new Todo { Id = id });
```

How to fix violations

To fix a violation of this rule, make sure that the endpoint uses one of the allowed model binding attributes.

C#

```
app.MapGet("/todos/{id}", ([FromRoute] int id) => new Todo { Id = id });
```

When to suppress warnings

Do ***not*** suppress a warning from this rule. An incorrect model binding setup can result in unexpected behavior when resolving parameters at runtime.

ASP0004: Do not use action results with route handlers

Article • 09/28/2022

	Value
Rule ID	ASP0004
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A route handler delegate returns a value that implements IActionResult.

Rule description

Route handler endpoints do not support executing MVC's `IActionResult` instances. Returning an `IActionResult` that doesn't implement `IResult` results in serializing the result instance rather than executing the result.

C#

```
app.MapGet("/todos/{id}", (int id) => new JsonResult(new Todo { .. }));
```

How to fix violations

To fix a violation of this rule, make sure that endpoint's route handler returns an IResult type by using the Results extension methods.

C#

```
app.MapGet("/todos/{id}", (int id) => Results.Json(new Todo { .. }));
```

When to suppress warnings

Do ***not*** suppress a warning from this rule. Returning an `IActionResult` that doesn't implement `IResult` results in serializing the result instance rather than executing the result.

ASP0005: Do not place attribute on method called by route handler lambda

Article • 09/28/2022

	Value
Rule ID	ASP0005
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

An attribute was applied to a method definition instead of the route handler in a route handler endpoint.

Rule description

When an endpoint is declared, attributes should be applied to the delegate parameter in order to be effective. For example, the Authorize attribute in the following code sample isn't set on the registered endpoint:

C#

```
app.MapGet("/todos/{id}", GetTodoById);

[Authorize]
Todo GetTodoById(int id)
{
  ...
}
```

The attribute must be placed on the route handler parameter as shown in the following code:

C#

```
app.MapGet("/todos/{id}", [Authorize] GetTodoById);

Todo GetTodoById(int id)
{
```

```
  ...
}
```

How to fix violations

To fix a violation of this rule, make sure that endpoint attributes are applied to the route handler parameter:

C#

```
app.MapGet("/todos/{id}", [Authorize] (int id) => {});
app.MapGet("/users/{id}", [Authorize] GetUserById);
```

When to suppress warnings

Do not suppress a warning from this rule. Misplaced attributes can result in unexpected behavior at runtime.

ASP0006: Do not use non-literal sequence numbers

Article • 10/30/2022

	Value
Rule ID	ASP0006
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

An invocation on a method in RenderTreeBuilder containing a sequence number that isn't a literal as a parameter.

Rule description

Blazor's UI diffing algorithm relies on sequence numbers to determine which elements have changed. Computing the sequence number dynamically or using a counter can result in poor diffing performance. Instead, use a literal sequence number that maps to the source code line for the element. For example, the following code produces an error:

C#

```
using Microsoft.AspNetCore.Components.Rendering;
var builder = new RenderTreeBuilder();
var seqNum = 1;
builder.OpenElement(seqNum, "div");
builder.CloseElement();
```

How to fix violations

To fix a violation of this rule, make sure that calls to methods on the RenderTreeBuilder class that take a sequence number as a parameter use a literal sequence number.

C#

```
using Microsoft.AspNetCore.Components.Rendering;
var builder = new RenderTreeBuilder();
```

```
builder.OpenElement(0, "div");
builder.CloseElement();
```

When to suppress warnings

Do ***not*** suppress a warning from this rule. Using a non-literal sequence number can result in performance degradation.

ASP0007: Route parameter and argument optionality is mismatched

Article • 09/28/2022

	Value
Rule ID	ASP0007
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A route parameter is declared as required in the Delegate definition but is marked as optional in the endpoint route.

Rule description

When an endpoint is declared, optionality of parameters can be declared in both the route template and in the route handler arguments. When a parameter is declared as optional in the handler, it must also be declared as optional in the route template. For example, GET `/todos` fails to resolve a match for the following code:

```
C#

app.MapGet("/todos/{id}", (int? id) => {});
```

The preceding code fails to match GET `/todos` because the `id` parameter was not provided, even though it is treated as optional in the route handler.

How to fix violations

To fix a violation of this rule, make sure that the optionality in the route template and the delegate match. For example, for the following code sample:

```
C#

app.MapGet("/todos/{id}", (int? id) => {});
```

If the parameter is intended to be required, make the type non-nullable by removing the `?` from `int?`:

```
C#

app.MapGet("/todos/{id}", (int id) => {});
```

If the parameter is intended to be optional, then the nullable value type operator `?` should be applied:

```
C#

app.MapGet("/todos/{id?}", (int? id) => {});
```

When to suppress warnings

Do ***not*** suppress a warning from this rule. Mismatched parameter optionality can result in unexpected behavior with routing at runtime.

ASP0008: Do not use ConfigureWebHost with WebApplicationBuilder.Host

Article • 09/28/2022

	Value
Rule ID	ASP0008
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

`ConfigureWebHost` can't be used with the `Host` property on `WebApplicationBuilder`.

Rule description

The `WebApplicationBuilder` doesn't support configuring the `WebHost` before build using the `ConfigureWebHost` extension method.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureWebHost(webHostBuilder => {

webHostBuilder.UseContentRoot(Path.Combine(Directory.GetCurrentDirectory(),
"myContentRoot"));
});

var app = builder.Build();

app.Run();
```

How to fix violations

To fix a violation of this rule, configure the `WebHost` directly on the `WebApplicationBuilder`. For example, instead of setting the content root path via

`ConfigureWebHost`.

```C#
var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureWebHost(webHostBuilder =>
{

webHostBuilder.UseContentRoot(Path.Combine(Directory.GetCurrentDirectory(),
"myContentRoot"));
});

var app = builder.Build();

app.Run();
```

Configure the content root path directly on the `WebApplicationBuilder.WebHost`.

```C#
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseContentRoot(Path.Combine(Directory.GetCurrentDirectory(),
"foobar"));

var app = builder.Build();

app.Run();
```

When to suppress warnings

Do ***not*** suppress a warning from this rule. A misconfigured application can result in unexpected behavior at runtime.

ASP0009: Do not use Configure with WebApplicationBuilder.WebHost

Article • 09/28/2022

	Value
Rule ID	ASP0009
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

`Configure` can't be used with the `WebHost` property on `WebApplicationBuilder`.

Rule description

The `WebApplicationBuilder` doesn't support configuring the `WebHost` before build using the `Configure` extension method.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.Configure(webHostBuilder => {

webHostBuilder.UseContentRootPath(Path.Combine(Directory.GetCurrentDirectory
(), "myContentRoot"));
});

var app = builder.Build();

app.Run();
```

How to fix violations

To fix a violation of this rule, configure the `WebHost` directly on the `WebApplicationBuilder`. For example, instead of setting the content root path via `Configure`.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.Configure(webHostBuilder =>
{

webHostBuilder.UseContentRoot(Path.Combine(Directory.GetCurrentDirectory(),
"myContentRoot"));
});

var app = builder.Build();

app.Run();
```

Configure the content root path directly on the `WebApplicationBuilder`.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseContentRoot(Path.Combine(Directory.GetCurrentDirectory(),
"myContentRoot"));

var app = builder.Build();

app.Run();
```

When to suppress warnings

Do ***not*** suppress a warning from this rule. A misconfigured application can result in unexpected behavior at runtime.

ASP0010: Do not use UseStartup with WebApplicationBuilder.WebHost

Article • 09/28/2022

Expand table

	Value
Rule ID	ASP0010
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

`UseStartup` can't be used with `WebApplicationBuilder.WebHost`.

Rule description

The `WebApplicationBuilder` doesn't support configuration via a `Startup` class.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.WebHost.UseStartup<Startup>();

var app = builder.Build();

app.Run();
```

How to fix violations

To fix a violation of this rule, leverage the `Configuration` and `Services` properties on the `WebApplicationBuilder` to modify configuration and DI directly, without the need for a startup class.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication();

var app = builder.Build();

app.Run();
```

When to suppress warnings

Do ***not*** suppress a warning from this rule. A misconfigured application can result in unexpected behavior at runtime.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP0011: Suggest using builder.Logging over Host.ConfigureLogging or WebHost.ConfigureLogging

Article • 09/28/2022

	Value
Rule ID	ASP0011
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

`ConfigureLogging` isn't the recommended strategy for configuring logging in a minimal API application.

Rule description

`ConfigureLogging` isn't the recommended strategy for configuring logging in a minimal API application.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureLogging(logging =>
{
    logging.AddJsonConsole();
})

var app = builder.Build();

app.Run();
```

How to fix violations

To fix a violation of this rule, use the `Logging` property on the `WebApplicationBuilder` to modify the logging configuration directly without the need for an additional

`ConfigureLogging` call.

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Logging.AddJsonConsole();

var app = builder.Build();

app.Run();
```

When to suppress warnings

Do ***not*** suppress a warning from this rule.

ASP0012: Suggest using builder.Services over Host.ConfigureServices or WebHost.ConfigureServices

Article • 09/28/2022

	Value
Rule ID	ASP0012
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

`ConfigureServices` isn't the recommended strategy for registering services in DI in a minimal API application.

Rule description

`ConfigureServices` isn't the recommended strategy for configuring logging in a minimal API application.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureServices(services =>
{
    services.AddAntiforgery();
})

var app = builder.Build();

app.Run();
```

How to fix violations

To fix a violation of this rule, use the `Services` property on the `WebApplicationBuilder` to modify the DI container directly without the need for an additional

`ConfigureServices` call.

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAntiforgery();

var app = builder.Build();

app.Run();
```

When to suppress warnings

Do ***not*** suppress a warning from this rule.

ASP0013: Suggest switching from using Configure methods to WebApplicationBuilder.Configuration

Article • 02/13/2024

Expand table

	Value
Rule ID	ASP0013
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

Configure isn't the recommended strategy for reading and writing to configuration in a minimal API app. `Configure` was designed to be used with Web Host or .NET Generic Host. In a minimal API app, WebApplicationBuilder.Configuration should be used to modify configuration directly.

Rule description

`Configure` isn't the recommended strategy for configuring logging in a minimal API app.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureAppConfiguration(builder =>
{
    builder.AddJsonFile("customAppSettings.json");
})

var app = builder.Build();

app.Run();
```

How to fix violations

To fix a violation of this rule, use WebApplicationBuilder.Configuration to modify application configuration directly without the need for an additional ConfigureAppConfiguration call.

```C#
var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddJsonFile("customAppSettings.json");

var app = builder.Build();

app.Run();
```

When to suppress warnings

Do ***not*** suppress a warning from this rule.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP0014: Suggest using top level route registrations

Article • 02/23/2023

	Value
Rule ID	ASP0014
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

Routes can be registered directly at the top-level of a minimal API application.

Rule description

Routes can be registered directly at the top-level of a minimal API application and don't need to be nested within a `UseEndpoints` call.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", () => "Hello World!");
});

app.Run();
```

How to fix violations

To fix a violation of this rule, register the endpoints directly on the `WebApplication`.

C#

```
var builder = WebApplication.CreateBuilder(args);
```

```
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

When to suppress warnings

Warnings from this rule can be suppressed if the target `UseEndpoints` invocation is invoked without any mappings as a strategy to organize middleware ordering.

C#

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

#pragma warning disable ASP0014
app.UseEndpoints(e => {});
#pragma warning restore ASP0014

app.Run();
```

ASP0015: Suggest using IHeaderDictionary properties

Article • 11/28/2022

	Value
Rule ID	ASP0015
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

IHeaderDictionary properties are the recommended strategy for accessing headers.

Rule description

`IHeaderDictionary` properties are recommended for accessing headers. Accessing headers using an indexer as in the example below is not recommended.

C#

```
var app = WebApplication.Create();

app.MapGet("/", (HttpContext context) => context.Request.Headers[""content-type""]);

app.Run();
```

How to fix violations

To fix a violation of this rule, use the property specified in the analyzer message to access the header specified in the message or apply the associated codefix.

C#

```
var app = WebApplication.Create();
app.MapGet("/", (HttpContext context) =>
context.Request.Headers.ContentType);
app.Run();
```

When to suppress warnings

Do not suppress a warning from this rule.

ASP0016: Do not return a value from RequestDelegate

Article • 11/28/2022

	Value
Rule ID	ASP0016
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A method used to create a RequestDelegate returns `Task<T>`. `RequestDelegate` discards this value.

Rule description

Do not return a value `Delegate`s provided to APIs that expect `RequestDelegate`. For example, the following sample returns a `Task<string>` where the `string` value of the `Task` will be discarded.

C#

```
var app = WebApplication.Create();
app.Use(next =>
{
    return new RequestDelegate((HttpContext context) =>
    {
        return Task.FromResult(""hello world"");
    });
});
```

How to fix violations

To fix a violation of this rule, change the return type to non-generic `Task` or, if the delegate is a route handler, cast it to `Delegate` so the return value is written to the response.

When to suppress warnings

Do not suppress a warning from this rule.

ASP0017: Invalid route pattern

Article • 11/28/2022

	Value
Rule ID	ASP0017
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A route pattern is invalid.

Rule description

This diagnostic is emitted when a route pattern is invalid. In the example below, the route pattern contains an invalid token.

C#

```
var app = WebApplication.Create();

app.MapGet("/{id", (int id) => ...);
```

How to fix violations

To fix a violation of this rule, correct the error identified in the analyzer message.

When to suppress warnings

Do not suppress a warning from this rule.

ASP0018: Unused route parameter

Article • 04/03/2024

Expand table

	Value
Rule ID	ASP0018
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A route parameter is specified but not used.

Rule description

A route parameter is specified but not used. In the example below, the `id` parameter is defined in the route but not in the route handler.

C#

```
var app = WebApplication.Create();

app.MapGet("/{id}", () => ...);
```

How to fix violations

To fix a violation of this rule, remove the route parameter or add code that uses the parameter.

C#

```
var app = WebApplication.Create();

app.MapGet("/{id}", (string id) => ...);
```

When to suppress warnings

In general, do ***not*** suppress a warning from this rule without validating the route parameter is used. Currently properties within the bound model for FromRoute attributes may not be analyzed. For more information, see GitHub issue #54212.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP0019: Suggest using IHeaderDictionary.Append or the indexer

Article • 11/28/2022

	Value
Rule ID	ASP0019
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

`IDictionary.Add` isn't recommended for setting or appending headers. IDictionary.Add throws an `ArgumentException` when attempting to add a duplicate key.

Rule description

`IDictionary.Add` isn't recommended for setting or appending headers.

How to fix violations

To fix a violation of this rule, use `IHeaderDictionary.Append` or the indexer to append or set headers.

When to suppress warnings

Do not suppress a warning from this rule.

ASP0020: Complex types referenced by route parameters must be parsable

Article • 03/27/2023

	Value
Rule ID	ASP0020
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A route parameter is a complex type that isn't parsable.

Rule description

This diagnostic is emitted when a route parameter is a complex type that isn't parsable.

How to fix violations

To fix a violation of this rule, define a `bool TryParse(string, IFormatProvider, out T)` method, where `T` is the complex type identified in the error message. As an alternative, implement IParsable<TSelf>.

When to suppress warnings

Do not suppress a warning from this rule.

ASP0021: The return type of the BindAsync method must be `ValueTask<T>`.

Article • 03/27/2023

	Value
Rule ID	ASP0021
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

An implementation of the BindAsync method has a return type that isn't ValueTask<TResult>.

Rule description

This diagnostic is emitted when an implementation of the `BindAsync` method has a return type that isn't `ValueTask<T>`.

How to fix violations

To fix a violation of this rule, define a `ValueTask<T>` return type for `BindAsync` and consider implementing IBindableFromHttpContext<TSelf> to enforce implementation.

When to suppress warnings

Do not suppress a warning from this rule.

ASP0022: Route conflict detected between route handlers

Article • 11/08/2023

	Value
Rule ID	ASP0022
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

An HTTP request matches multiple routes, resulting in an ambiguous match error.

Rule description

This diagnostic is emitted when a route conflict is detected.

How to fix violations

To fix a violation of this rule, change the route's pattern, HTTP method, or route constraints.

When to suppress warnings

Do not suppress a warning from this rule.

Notes

ASP0023 warns on route conflicts in ASP.NET Core MVC apps, this analyzer warns on route conflicts in minimal API apps.

This analyzer is intentionally conservative about duplicate routes it reports to avoid false positives:

- The analyzer only reports duplicate routes declared in the same code block in a method. Duplicate routes in different branches of an `if` statement aren't reported

because the analyzer can't statically determine which will be used at runtime.

- The analyzer only reports duplicate routes with known metadata. Unknown methods called on a route handler might add new metadata that is then used to customize how the route is matched.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

The ASP.NET Core documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

ASP0023: Route conflict detected between route handlers

Article • 03/29/2023

	Value
Rule ID	ASP0023
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

An HTTP request matches multiple routes, resulting in an ambiguous match error.

Rule description

This diagnostic is emitted when a route conflict is detected.

How to fix violations

To fix a violation of this rule, change the route's pattern, HTTP method, or route constraints.

When to suppress warnings

Do not suppress a warning from this rule.

Notes

ASP0022 warns on route conflicts in minimal API apps, this analyzer warns on route conflicts in ASP.NET Core MVC apps.

ASP0024: Route handler has multiple parameters with the `[FromBody]` attribute

Article • 03/29/2023

	Value
Rule ID	ASP0024
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A route handler has multiple parameters with the [FromBody] attribute or a parameter with an [AsParameters] attribute where the parameter type contains multiple members with `[FromBody]` attributes.

Rule description

Route handler has multiple parameters with the `[FromBody]` attribute.

How to fix violations

To fix a violation of this rule, remove `[FromBody]` attributes from all but one parameter.

When to suppress warnings

Do not suppress a warning from this rule.

ASP0025: Use AddAuthorizationBuilder to register authorization services and construct policies.

Article • 05/15/2023

	Value
Rule ID	ASP0025
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

The use of AddAuthorization can be converted to the new AddAuthorizationBuilder.

Rule description

Use `AddAuthorizationBuilder` to register authorization services and construct policies.

How to fix violations

To fix a violation of this rule, replace the usage of `AddAuthorization` with `AddAuthorizationBuilder`.

The code fix converts any usage of the setters for the following properties of AuthorizationOptions:

- DefaultPolicy
- FallbackPolicy
- InvokeHandlersAfterFailure

These setter usages are converted to equivalent method calls on AuthorizationBuilder:

- SetDefaultPolicy
- SetFallbackPolicy
- SetInvokeHandlersAfterFailure

No diagnostic is reported when the configure action passed to `AddAuthorization` uses any of the following members of `AuthorizationOptions`:

- The GetPolicy(String) method
- The DefaultPolicy getter
- The FallbackPolicy getter
- The InvokeHandlersAfterFailure getter

`AuthorizationBuilder` doesn't have equivalents for these members of `AuthorizationOptions`, so they can't be converted.

No diagnostic is reported if the configure action passed to `AddAuthorization` contains operations unrelated to `AuthorizationOptions`. The code fix would not be able to automatically map unrelated operations to the fluent API of `AddAuthorizationBuilder`.

The following example shows code that triggers this diagnostic:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AtLeast21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});

var app = builder.Build();

app.UseAuthorization();

app.Run();
```

The following example shows the result of applying the code fix:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorizationBuilder()
  .AddPolicy("AtLeast21", policy =>
  {
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
  });

var app = builder.Build();

app.UseAuthorization();
```

```
app.Run();
```

When to suppress warnings

The severity level of this diagnostic is Information. Suppress warnings if you don't want to use the new syntax.

BL0001: Component parameter should have public setters

Article • 06/03/2022

	Value
Rule ID	BL0001
Category	Usage
Fix is breaking or non-breaking	Breaking

Cause

A property on a type deriving from ComponentBase annotated with [Parameter] has a missing or non-public setters.

Rule description

Component parameters are required to have publicly accessible setters to allow the framework to assign values. All of the parameter declarations in the following example result in this diagnostic.

razor

```
@code
{
    [Parameter] int Parameter1 { get; set; }

    [Parameter] public int Parameter2 { get; }

    [Parameter] public int Parameter3 { get; private set; }
}
```

How to fix violations

- Make the property and its setter public.

razor

```
@code
{
```

```
    [Parameter] public int Parameter1 { get; set; }

    [Parameter] public int Parameter2 { get; set; }

    [Parameter] public int Parameter3 { get; set; }
}
```

- If making the property non-public is not possible, consider implementing SetParametersAsync manually.

When to suppress warnings

Do not suppress a warning from this rule.

BL0002: Component has multiple CaptureUnmatchedValues parameters

Article • 06/03/2022

	Value
Rule ID	BL0002
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

More than one parameter on a type deriving from ComponentBase is annotated with `CaptureUnmatchedValues = true`.

Rule description

For a component, exactly one parameter is expected to have the CaptureUnmatchedValues set to `true`.

razor

```
@code
{
    [Parameter(CaptureUnmatchedValues = true)] public Dictionary<string,
object> Parameter1 { get; set; }

    [Parameter(CaptureUnmatchedValues = true)] public Dictionary<string,
object> Parameter2 { get; set; }
}
```

How to fix violations

Limit a single parameter to have `CaptureUnmatchedValues` set.

razor

```
@code
{
    [Parameter(CaptureUnmatchedValues = true)] public Dictionary<string,
```

```
object> Parameter1 { get; set; }

    [Parameter] public Dictionary<string, object> Parameter2 { get; set; }
}
```

When to suppress warnings

Do not suppress a warning from this rule.

BL0003: Component parameter with CaptureUnmatchedValues has the wrong type

Article • 06/03/2022

	Value
Rule ID	BL0003
Category	Usage
Fix is breaking or non-breaking	Breaking

Cause

A parameter on a type deriving from ComponentBase annotated with CaptureUnmatchedValues`= true` is not assignable from `Dictionary<string, object>`

Rule description

Parameters annotated with `CaptureUnmatchedValues = true` must be able to receive a `Dictionary<string, object>` value.

razor

```
@code
{
    [Parameter(CaptureUnmatchedValues = true)] public IDictionary<string,
string> Attributes { get; set; }
}
```

How to fix violations

Change the type of the parameter to either `IDictionary<string, object>` or `Dictionary<string, object>`

razor

```
@code
{
```

```
    [Parameter(CaptureUnmatchedValues = true)] public IDictionary<string,
object> Attributes { get; set; }
}
```

When to suppress warnings

Do not suppress a warning from this rule.

BL0004: Component parameter should be public

Article • 06/03/2022

	Value
Rule ID	BL0004
Category	Usage
Fix is breaking or non-breaking	Breaking

Cause

A property on a type deriving from ComponentBase annotated with [Parameter] is not public.

Rule description

Component parameters are required to be public and must have a public setter.

razor

```
@code
{
    [Parameter] int Parameter1 { get; set; }
}
```

How to fix violations

- Make the property and its setter public.

razor

```
@code
{
    [Parameter] public int Parameter1 { get; set; }
}
```

- If making the property non-public is not possible, consider implementing SetParametersAsync manually.

When to suppress warnings

Do not suppress a warning from this rule.

BL0005: Component parameter should not be set outside of its component

Article • 06/03/2022

	Value
Rule ID	BL0005
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A property on a type deriving from ComponentBase annotated with [Parameter] is being assigned to from outside the component.

Rule description

Component parameters should be assigned to as part of the component initialization or as part of `SetParametersAsync`. Assigning a value to a parameter from an external source results in the value being overwritten the next time the component renders.

How to fix violations

Consider using a distinct property to receive values from other components. Additional code can then be written to decide which of the two values to use in the component.

When to suppress warnings

Do not suppress a warning from this rule.

BL0006: Do not use RenderTree types

Article • 06/03/2022

	Value
Rule ID	BL0006
Category	Usage
Fix is breaking or non-breaking	Non-breaking

Cause

A reference to a type in the Microsoft.AspNetCore.Components.RenderTree was found.

Rule description

Types in the `Microsoft.AspNetCore.Components.RenderTree` namespace are part of Blazor's implementation detail and are subject to breaking changes. Using these types is not recommended in user code.

How to fix violations

Remove the reference to the type.

When to suppress warnings

It is safe to suppress this rule if using this type is essential. Note that this these are subject to breaking changes between major releases.

MVC1000: Use of IHtmlHelper.Partial should be avoided

Article • 06/03/2022

	Value
Rule ID	MVC1000
Fix is breaking or non-breaking	Non-breaking

Cause

The Partial or RenderPartial method was called or referenced.

Rule description

Rendering a partial using `IHtmlHelper.Partial` or `IHtmlHelper.RenderPartial` extension methods results in blocking calls. This may result in performance degradation and application dead locks issues due to thread pool starvation.

How to fix violations

- Use the PartialTagHelper
- Use the PartialAsync or RenderPartialAsync

When to suppress warnings

It's safe to suppress this rule if updating the application to use the suggested fixes is non-trivial. Before the validation is disabled, be sure to consider the risks of thread pool starvation to the application.

MVC1001: Filters cannot be applied to page handler methods

Article • 09/02/2022

Expand table

	Value
Rule ID	MVC1001
Fix is breaking or non-breaking	Non-breaking

Cause

An attribute implementing IFilterMetadata was applied to a Razor Page handler method.

Rule description

Razor Page handler methods are selected after MVC filter execution has started, and consequently cannot contribute filters to execute. Applying a filter to a Razor Page handler is unsupported and always incorrect.

C#

```
public class IndexModel : PageModel
{
    [MyFilter]
    public IActionResult OnGet() => Page();
}
```

How to fix violations

Remove the filter from the handler and apply it to the page model. If a filter has to be applied to a specific handler, consider using multiple Razor Pages.

C#

```
[MyFilter]
public class IndexModel : PageModel
{
```

```
    public IActionResult OnGet() => Page();
}
```

When to suppress warnings

Don't suppress warnings from this rule.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

MVC1002: Route attribute cannot be applied to page handler methods

Article • 06/03/2022

Expand table

	Value
Rule ID	MVC1002
Fix is breaking or non-breaking	Non-breaking

Cause

An attribute implementing IRouteTemplateProvider was applied to a Razor Page handler method.

Rule description

Razor Page handler methods are selected after routing is completed, and consequently cannot contribute a route. Applying a route attribute such as `HttpGet` or `HttpPost` to a Razor Page handler is not supported.

C#

```
public class IndexModel : PageModel
{
    [HttpGet("/my-url")]
    public IActionResult OnGet() => Page();
}
```

How to fix violations

Remove the route attribute from the handler. Routes can be specified for a Razor Page using an `@page` directive or by using conventions. For more information, see custom routes in Razor Pages.

When to suppress warnings

Don't suppress warnings from this rule.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

MVC1003: Route attributes cannot be applied to page models

Article • 06/03/2022

Expand table

	Value
Rule ID	MVC1003
Fix is breaking or non-breaking	Non-breaking

Cause

An attribute implementing IRouteTemplateProvider was applied to a Razor Page model.

Rule description

Razor Page models are identified after routing is completed, and consequently cannot contribute a route. Applying a route attribute such as `Route` to a Razor Page model is not supported.

C#

```
[Route("/my-page-route")]
public class IndexModel : PageModel
{
    public IActionResult OnGet() => Page();
}
```

How to fix violations

Remove the route attribute from the page model. Routes can be specified for a Razor Page using an `@page` directive or by using conventions. For more information, see custom routes in Razor Pages.

When to suppress warnings

Don't suppress warnings from this rule.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

MVC1004: Rename model bound parameter

Article • 06/03/2022

	Value
Rule ID	MVC1004
Fix is breaking or non-breaking	Breaking

Cause

A model bound parameter has the same name as one of its properties.

Rule description

Model binding a complex parameter with a property that has the same name may result in unexpected binding behavior. Consider renaming the parameter, or using a binding attribute to specify a different name.

Consider the following code:

C#

```
public class HomeController : Controller
{
    public IActionResult Get(SearchModel search)
    {
        ...
    }
}

public class SearcModel
{
    public string Search { get; set; }
}
```

In this model, the parameter and its property are both named `Search`, which results in model binding attempting to bind the property as `search.Search`. Naming a parameter and its property the same prevents binding to a value without a prefix such as a query that looks like `?search=MySearchTerm`.

How to fix violations

- Rename the parameter if its prefix is not used during binding:

```
C#

public IActionResult Get(SearchModel model)
{
    ...
}
```

Renaming a parameter on a public type could be considered a breaking change since it changes a library's public API surface.

- If this is problematic, consider using a model binding attribute such as `Bind` to specify the model binding prefix:

```
C#

public IActionResult Get([Bind(Prefix = "")] SearchModel search)
{
    ...
}
```

When to suppress warnings

Warnings can be suppressed if you intend to use the parameter name as a prefix during model binding.

MVC1005: Cannot use UseMvc with Endpoint Routing

Article • 06/03/2022

	Value
Rule ID	MVC1005
Fix is breaking or non-breaking	Non-breaking

Cause

UseMvc was invoked as part of startup.

Rule description

Using MVC via UseMvc or UseMvcWithDefaultRoute requires an explicit opt-in inside `Startup.ConfigureServices`. This is required because MVC must know whether it can rely on the authorization and CORS Middleware during initialization.

How to fix violations

If the app requires legacy IRouter support, disable EnableEndpointRoutingusing any of the following approaches in `Startup.ConfigureServices`:

```
C#

services.AddMvc(options => options.EnableEndpointRouting = false);
```

If legacy `IRouter` support is not required, replace the call to `UseMvc` with `UseEndpoints`. For more details, see the migration guide.

When to suppress warnings

Do not suppress a warning from this rule.

MVC1006: Methods containing TagHelpers must be async and return Task

Article • 06/03/2022

Expand table

	Value
Rule ID	MVC1006
Fix is breaking or non-breaking	Breaking

Cause

A tag helper was defined inside a Razor function that executes synchronously

Rule description

Tag Helper execution is asynchronous. When used inside a method or a lambda within a Razor Page, the containing function must also be declared to be async.

Consider the following cshtml file:

razor

```
void Helper(string controller)
{
    <a asp-controller="@controller">Home</a>
}
```

`asp-controller` is a tag helper and will trigger this rule.

How to fix violations

Declare the function to be async and Task returning:

razor

```
async Task Helper(string controller)
{
    <a asp-controller="@controller">Home</a>
}
```

When to suppress warnings

Do not suppress a warning from this rule.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core Request Delegate Generator (RDG) diagnostics

Article • 01/16/2025

The ASP.NET Core Request Delegate Generator (RDG) is a tool that generates request delegates for ASP.NET Core apps. The RDG is used by the native ahead-of-time (AOT) compiler to generate request delegates for the app's `Map` methods.

> **Note**
>
> - The Native AOT feature is currently in preview.
> - In .NET 8, not all ASP.NET Core features are compatible with Native AOT.

The following list contains the RDG diagnostics for ASP.NET Core:

- RDG002: Unable to resolve endpoint handler
- RDG004: Unable to resolve anonymous type
- RDG005: Invalid abstract type
- RDG006: Invalid constructor parameters
- RDG007: No valid constructor found
- RDG008: Multiple public constructors
- RDG009: Invalid nested AsParameters
- RDG010: InvalidAsParameters Nullable
- RDG011: Type parameters not supported
- RDG012: Unable to resolve inaccessible type
- RDG013: Invalid source attributes

ⓘ **Note:** The author created this article with assistance from AI. Learn more

RDG002: Unable to resolve endpoint handler

Article • 09/27/2023

Expand table

	Value
Rule ID	RDG002
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler that can't be statically analyzed.

Rule description

The Request Delegate Generator runs at compile-time and needs to be able to statically analyze route handlers in an app. The current implementation only supports route handlers that are provided as a lambda expression, method group references, or references to read-only fields or variables.

The following code generates the RDG002 warning because the route handler is provided as a reference to a method:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                        AppJsonSerializerContext.Default);
});

var app = builder.Build();

var del = Wrapper.GetTodos;
app.MapGet("/v1/todos", del);
```

```
app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}

class Wrapper
{
    public static Func<IResult> GetTodos = () =>
        Results.Ok(new Todo(1, "Write test fix"));
}
```

How to fix violations

Declare the route handler using supported syntax, such as an inline lambda:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                   AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos", () => Results.Ok(new Todo(1, "Write tests")));

app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework falls back to generating the request delegate at runtime.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

RDG004: Unable to resolve anonymous type

Article • 09/28/2023

Expand table

	Value
Rule ID	RDG004
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with an anonymous return type.

Rule description

The Request Delegate Generator runs at compile-time and needs to be able to statically analyze route handlers in an app. Anonymous types are generated with a type name only known to the complier and aren't statically analyzable. The following endpoint produces the diagnostic.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                        AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos", () => new { Id = 1, Task = "Write tests" });

app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
```

```
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

How to fix violations

Declare the route handler with a concrete type as the return type.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos", () => new Todo(1, "Write tests fix"));

app.Run();

record Todo(int Id, string Task);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework falls back to generating the request delegate at runtime.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

RDG005: Invalid abstract type

Article • 09/29/2023

Expand table

	Value
Rule ID	RDG005
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter annotated with the [AsParameters] attribute that is an abstract type.

Rule description

The implementation of surrogate binding via the [AsParameters] attribute in minimal APIs only supports types with concrete implementations. Using a parameter with an abstract type as in the following sample produces the diagnostic.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                        AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapPut("/v1/todos/{id}",
    ([AsParameters] TodoRequest todoRequest) =>
Results.Ok(todoRequest.Todo));

app.Run();
```

```
abstract class TodoRequest
{
    public int Id { get; set; }
    public Todo? Todo { get; set; }
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

How to fix violations

Use a concrete type for the surrogate:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                  AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}",
           ([AsParameters] TodoRequest todoRequest) =>
Results.Ok(todoRequest.Todo));

app.Run();

class TodoRequest
{
    public int Id { get; set; }
    public Todo? Todo { get; set; }
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning should not be suppressed. Suppressing the warning will lead to a runtime exception assocaited with the same warning.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

RDG006: Invalid constructor parameters

Article • 09/29/2023

Expand table

	Value
Rule ID	RDG006
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter annotated with the [AsParameters] attribute that contains an invalid constructor.

Rule description

Types that are used for surrogate binding via the [AsParameters] attribute must contain a public parameterized constructor where all parameters to the constructor match the public properties declared on the type. The `TodoRequest` type produces this diagnostic because there is no matching constructor parameter for the `Todo` property.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                       AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}",
           ([AsParameters] TodoRequest todoRequest) =>
Results.Ok(todoRequest.Todo));

app.Run();
```

```
class TodoRequest(int id, string name)
{
    public int Id { get; set; } = id;
    public Todo? Todo { get; set; }
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

How to fix violations

Ensure that all properties on the type have a matching parameter on the constructor.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                    AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}",
           ([AsParameters] TodoRequest todoRequest) =>
Results.Ok(todoRequest.Todo));

app.Run();

class TodoRequest(int id, Todo? todo)
{
    public int Id { get; set; } = id;
    public Todo? Todo { get; set; } = todo;
}

record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning should ***not*** be suppressed. Suppressing the warning leads to a runtime exception associated with the same warning.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

RDG007: No valid constructor found

Article • 10/10/2023

Expand table

	Value
Rule ID	RDG007
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter annotated with the [AsParameters] attribute with no valid constructor.

Rule description

Types that are used for surrogate binding via the `AsParameters` attribute must contain a public constructor. The `TodoRequest` type produces this diagnostic because there is no public constructor.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}", ([AsParameters] TodoRequest request)
    => Results.Ok(request.Todo));

app.Run();

public class TodoRequest
{
    public int Id { get; set; }
    public Todo Todo { get; set; }
```

```
    private TodoRequest()
    {
    }
}

public record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

How to fix violations

Remove the non-public constructor, or add a new public constructor.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);

builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapPut("/v1/todos/{id}", ([AsParameters] TodoRequest request)
    => Results.Ok(request.Todo));
app.Run();

public class TodoRequest(int Id, Todo todo)
{
    public int Id { get; set; } = Id;
    public Todo Todo { get; set; } = todo;
}

public record Todo(int Id, string Task);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning can't be safely suppressed. When suppressed, results in the `InvalidOperationException` runtime exception.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

RDG008: Multiple public constructors

Article • 10/10/2023

Expand table

	Value
Rule ID	RDG008
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter annotated with the [AsParameters] attribute with multiple public constructors.

Rule description

Types that are used for surrogate binding via the `AsParameters` attribute must contain a single public constructor. The `TodoRequest` type produces this diagnostic because there are multiple public constructors.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();
var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests",DateTime.UtcNow.AddDays(1))
};
builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
    return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id)
```

```
is Todo todo
    ? Results.Ok(todo)
    : Results.NotFound();
});

app.Run();

public class TodoItemRequest
{
    public int Id { get; set; }
    public Todo[] Todos { get; set; }

    public TodoItemRequest(int id, Todo[] todos)
    {
        Id = id;
        Todos = todos;
    }

    // Additional Constructor
    public TodoItemRequest()
    {
        Id = 1;
        Todos = [new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2))];
    }

}

public class Todo
{
    public DateTime DueDate { get; }
    public int Id { get; private set; }
    public string Task { get; private set; }

    public Todo(int id, string task, DateTime dueDate)

    {
        Id = id;
        Task = task;
        DueDate = dueDate;
    }
}

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}
```

How to fix violations

Provide a single public constructor.

```
C#
```

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();
var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests",DateTime.UtcNow.AddDays(1))
};
builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
    return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id)
is Todo todo
    ? Results.Ok(todo)
    : Results.NotFound();
});

app.Run();

public class TodoItemRequest
{
    public int Id { get; set; }
    public Todo[] Todos { get; set; }

    public TodoItemRequest(int id, Todo[] todos)
    {
        Id = id;
        Todos = todos;
    }
}

public class Todo
{
    public DateTime DueDate { get; }
    public int Id { get; private set; }
    public string Task { get; private set; }

    public Todo(int id, string task, DateTime dueDate)
    {
        Id = id;
        Task = task;
        DueDate = dueDate;
    }
}
```

```
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}
```

When to suppress warnings

This warning can be safely suppressed.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

RDG009: Invalid nested AsParameters

Article • 11/02/2023

Expand table

	Value
Rule ID	RDG009
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains invalid nested [AsParameters].

Rule description

Types that are used for surrogate binding via the [AsParameters] attribute must not contain nested types that are also annotated with the [AsParameters] attribute:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();

var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests",DateTime.UtcNow.AddDays(1))
};

builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
     return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id) is Todo todo
```

```
    ? Results.Ok(todo)
    : Results.NotFound();
});

app.Run();

struct TodoItemRequest
{
    public int Id { get; set; }
    [AsParameters]
    public Todo[] Todos { get; set; }
}

internal record Todo(int Id, string Task, DateTime DueDate);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}
```

How to fix violations

Remove the nested `AsParameters` attribute:

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder();
var todos = new[]
{
    new Todo(1, "Write tests", DateTime.UtcNow.AddDays(2)),
    new Todo(2, "Fix tests",DateTime.UtcNow.AddDays(1))
};
builder.Services.AddSingleton(todos);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/v1/todos/{id}", ([AsParameters] TodoItemRequest request) =>
{
    return request.Todos.ToList().Find(todoItem => todoItem.Id ==
request.Id) is Todo todo
    ? Results.Ok(todo)
    : Results.NotFound();
});
```

```
app.Run();

struct TodoItemRequest
{
    public int Id { get; set; }
    //[AsParameters]
    public Todo[] Todos { get; set; }
}

internal record Todo(int Id, string Task, DateTime DueDate);

[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
}
```

When to suppress warnings

This warning can ***not*** be suppressed.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

RDG010: InvalidAsParameters Nullable

Article • 10/10/2023

Expand table

	Value
Rule ID	RDG010
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter annotated with the [AsParameters] attribute that is marked as nullable.

Rule description

The implementation of surrogate binding via the [AsParameters] attribute in minimal APIs only supports types that are not nullable.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/todos/{id}", ([AsParameters] TodoRequest? request)
    => Results.Ok(new Todo(request!.Id)));

app.Run();

public record TodoRequest(HttpContext HttpContext, [FromRoute] int Id);
public record Todo(int Id);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{
```

```
}
```

How to fix violations

Declare the parameter as non-nullable.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

var app = builder.Build();

app.MapGet("/todos/{id}", ([AsParameters] TodoRequest request)
    => Results.Ok(new Todo(request.Id)));

app.Run();

public record TodoRequest(HttpContext HttpContext, [FromRoute] int Id);
public record Todo(int Id);
[JsonSerializable(typeof(Todo[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning should ***not*** be suppressed. Suppressing the warning leads to a runtime exception associated with the same warning.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

RDG011: Type parameters not supported

Article • 10/03/2023

Expand table

	Value
Rule ID	RDG011
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler that captures a generic type.

Rule description

Endpoints that use generic type parameters are not supported. The endpoints within `MapEndpoints` produce this diagnostic because of the generic `<T>` parameter.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapEndpoints<Todo>();
app.Run();

public static class RouteBuilderExtensions
{
    public static IEndpointRouteBuilder MapEndpoints<T>(this
IEndpointRouteBuilder app) where T : class, new()
    {
        app.MapPost("/input", (T value) => value);
        app.MapGet("/result", () => new T());
        app.MapPost("/input-with-wrapper", (Wrapper<T> value) => value);
        app.MapGet("/async", async () =>
        {
```

```
            await Task.CompletedTask;
            return new T();
        });
        return app;
    }
}

record Wrapper<T>(T Value);
class Todo
{
    public int Id { get; set; }
    public string Task { get; set; }
}
[JsonSerializable(typeof(Wrapper<Todo>[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

How to fix violations

Remove the generic type from endpoints.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                    AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapTodoEndpoints();
app.Run();

public static class TodoRouteBuilderExtensions
{
    public static IEndpointRouteBuilder MapTodoEndpoints(this
IEndpointRouteBuilder app)
    {
        app.MapPost("/input", (Todo value) => value);
        app.MapGet("/result", () => new Todo(1, "Walk the dog"));
        app.MapPost("/input-with-wrapper", (Wrapper<Todo> value) => value);
        app.MapGet("/async", async () =>
        {
            await Task.CompletedTask;
            return new Todo(1, "Walk the dog");
        });
        return app;
```

```
    }
}

record Wrapper<T>(T Value);
record Todo(int Id, string Task);
[JsonSerializable(typeof(Wrapper<Todo>[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework will fallback to generating the request delegate at runtime.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

RDG012: Unable to resolve inaccessible type

Article • 10/05/2023

Expand table

	Value
Rule ID	RDG012
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter without the appropriate accessibility modifiers.

Rule description

Endpoints that use an inaccessible type (`private` or `protected`) are not supported. The endpoints within `MapEndpoints` produce this diagnostic because of the `Todo` type has the `private` accessibility modifiers.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

var app = builder.Build();
app.MapEndpoints();
app.Run();

public static class TodoRouteBuilderExtensions
{
```

```
    public static IEndpointRouteBuilder MapEndpoints(this
IEndpointRouteBuilder app)
    {
        app.MapPost("/input", (Todo value) => value);
        app.MapGet("/result", () => new Todo(1, "Walk the dog"));
        app.MapPost("/input-with-wrapper", (Wrapper<Todo> value) => value);
        app.MapGet("/async", async () =>
        {
            await Task.CompletedTask;
            return new Todo(1, "Walk the dog");
        });
        return app;
    }

    private record Todo(int Id, string Task);
}

record Wrapper<T>(T Value);
[JsonSerializable(typeof(Wrapper<TodoRouteBuilderExtensions.Todo>[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

How to fix violations

When applicable, set the target parameter type with a friendly accessibility.

C#

```
using System.Text.Json.Serialization;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});


var app = builder.Build();
app.MapEndpoints();
app.Run();

public static class TodoRouteBuilderExtensions
{
```

```
    public static IEndpointRouteBuilder MapEndpoints(this
IEndpointRouteBuilder app)
    {
        app.MapPost("/input", (Todo value) => value);
        app.MapGet("/result", () => new Todo(1, "Walk the dog"));
        app.MapPost("/input-with-wrapper", (Wrapper<Todo> value) => value);
        app.MapGet("/async", async () =>
        {
            await Task.CompletedTask;
            return new Todo(1, "Walk the dog");
        });
        return app;
    }

    public record Todo(int Id, string Task);
}

record Wrapper<T>(T Value);
[JsonSerializable(typeof(Wrapper<TodoRouteBuilderExtensions.Todo>[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning can be safely suppressed. When suppressed, the framework will fallback to generating the request delegate at runtime.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

RDG013: Invalid source attributes

Article • 10/05/2023

Expand table

	Value
Rule ID	RDG013
Fix is breaking or non-breaking	Non-breaking

Cause

This diagnostic is emitted by the Request Delegate Generator when an endpoint contains a route handler with a parameter that contains an invalid combination of service source attributes.

Rule description

ASP.NET Core supports resolving keyed and non-keyed services via dependency injection. It's ***not*** feasible to resolve a service as both keyed and non-keyed. The following code produces the diagnostic and throws a run time error with the same message:

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                   AppJsonSerializerContext.Default);
});
builder.Services.AddKeyedSingleton<IService, FizzService>("fizz");

var app = builder.Build();

app.MapGet("/fizz", ([FromKeyedServices("fizz")][FromServices] IService
service) =>
{
    return Results.Ok(service.Echo());
});
```

```
app.Run();
```

How to fix violations

Resolve the target parameter as either a keyed or non-keyed service.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateSlimBuilder(args);
builder.Services.ConfigureHttpJsonOptions(options =>
{
    options.SerializerOptions.TypeInfoResolverChain.Insert(0,
                                AppJsonSerializerContext.Default);
});

builder.Services.AddKeyedSingleton<IService, FizzService>("fizz");
builder.Services.AddKeyedSingleton<IService, BuzzService>("buzz");
builder.Services.AddSingleton<IService, FizzBuzzService>();
var app = builder.Build();

app.MapGet("/fizz", ([FromKeyedServices("fizz")] IService service) =>
{
    return Results.Ok(service.Echo());
});

app.MapGet("/buzz", ([FromKeyedServices("buzz")] IService service) =>
{
    return Results.Ok(service.Echo());
});

app.MapGet("/fizzbuzz", ([FromServices] IService service) =>
{
    return Results.Ok(service.Echo());
});

app.Run();

public interface IService
{
    string Echo();
}

public class FizzService : IService
{
    public string Echo() => "Fizz";
}

public class BuzzService : IService
```

```
{
    public string Echo() => "Buzz";
}

public class FizzBuzzService : IService
{
    public string Echo()
    {
        return "FizzBuzz";
    }
}
[JsonSerializable(typeof(string[]))]
internal partial class AppJsonSerializerContext : JsonSerializerContext
{

}
```

When to suppress warnings

This warning should ***not*** be suppressed. Suppressing the warning leads to a NotSupportedException runtime exception `The FromKeyedServicesAttribute is not supported on parameters that are also annotated with IFromServiceMetadata.`

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8

Article • 09/27/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Tom Dykstra, Jeremy Likness, and Jon P Smith

This is the first in a series of tutorials that show how to use Entity Framework (EF) Core in an ASP.NET Core Razor Pages app. The tutorials build a web site for a fictional Contoso University. The site includes functionality such as student admission, course creation, and instructor assignments. The tutorial uses the code first approach. For information on following this tutorial using the database first approach, see this Github issue.

Download or view the completed app. Download instructions.

Prerequisites

- If you're new to Razor Pages, go through the Get started with Razor Pages tutorial series before starting this one.

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

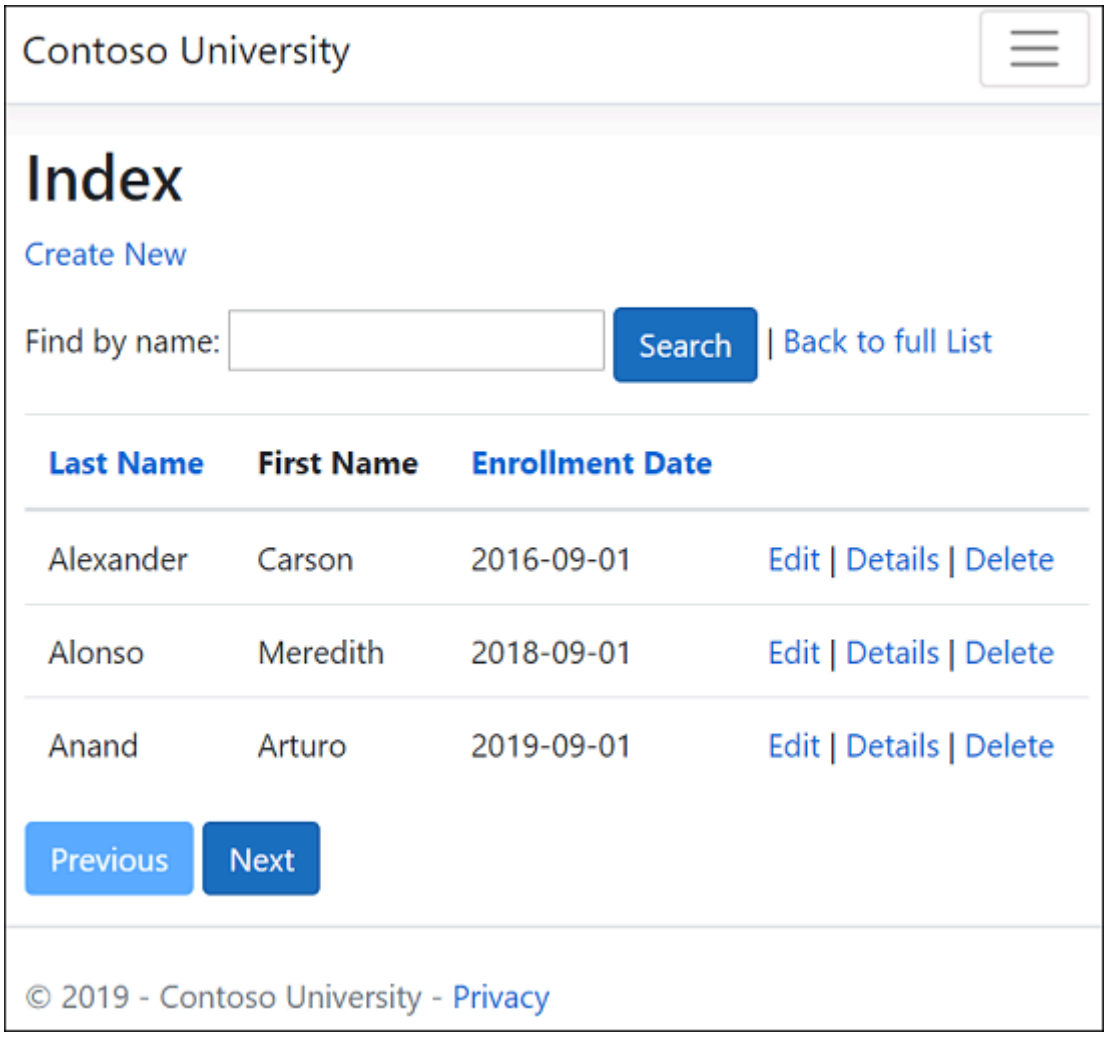
Troubleshooting

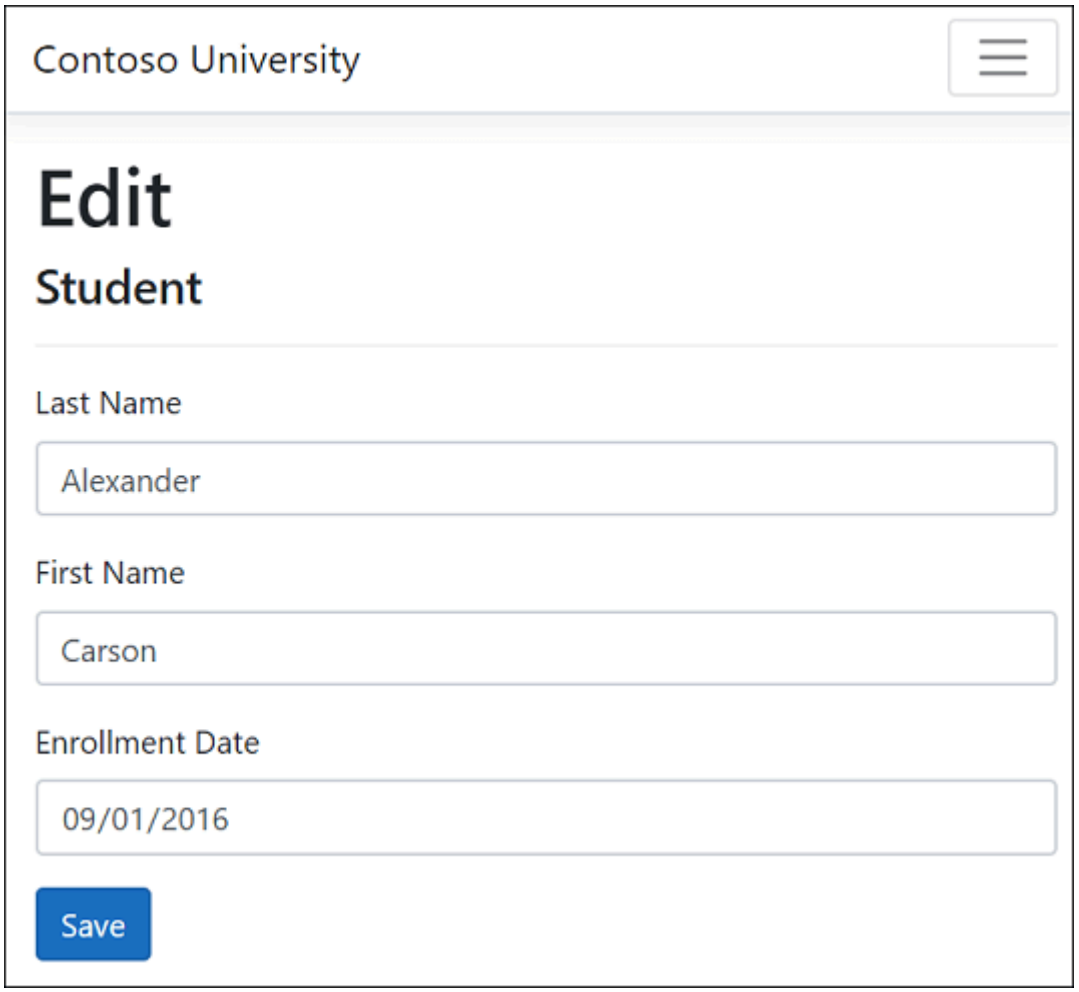
If you run into a problem you can't resolve, compare your code to the completed project. A good way to get help is by posting a question to StackOverflow.com, using

the ASP.NET Core tag or the EF Core tag.

The sample app

The app built in these tutorials is a basic university web site. Users can view and update student, course, and instructor information. Here are a few of the screens created in the tutorial.





The UI style of this site is based on the built-in project templates. The tutorial's focus is on how to use EF Core with ASP.NET Core, not how to customize the UI.

Optional: Build the sample download

This step is optional. Building the completed app is recommended when you have problems you can't solve. If you run into a problem you can't resolve, compare your code to the completed project. Download instructions.

Visual Studio

Select `ContosoUniversity.csproj` to open the project.

- Build the project.
- In Package Manager Console (PMC) run the following command:

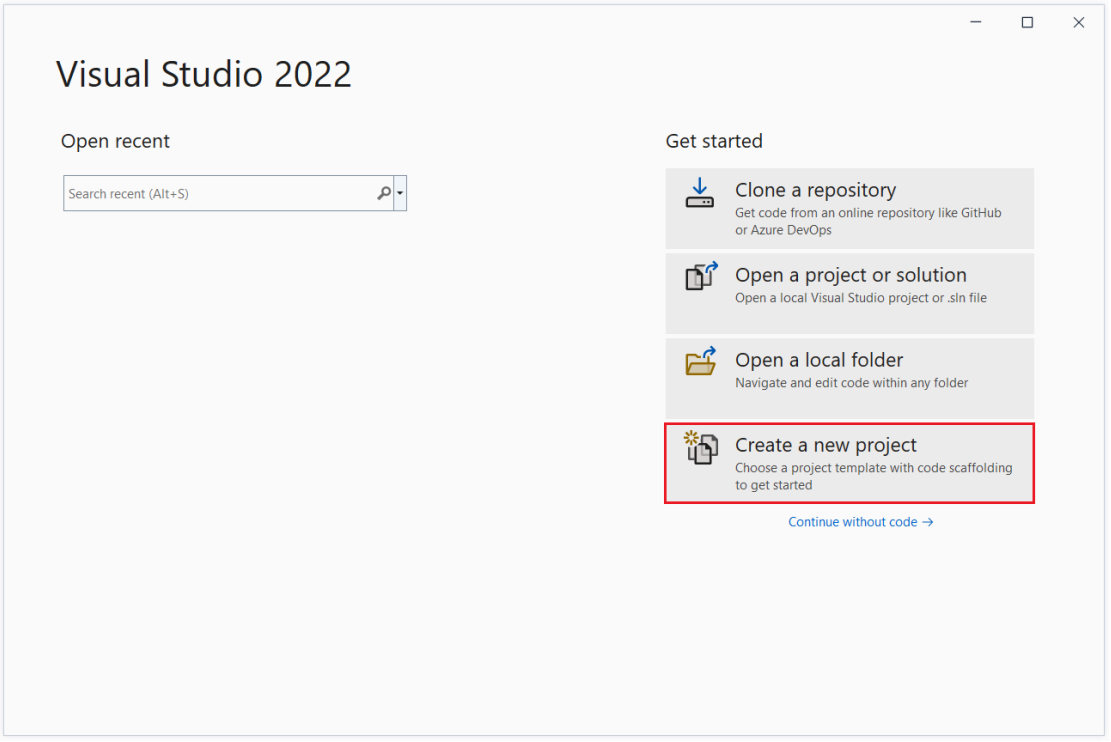
```
PowerShell
Update-Database
```

Run the project to seed the database.

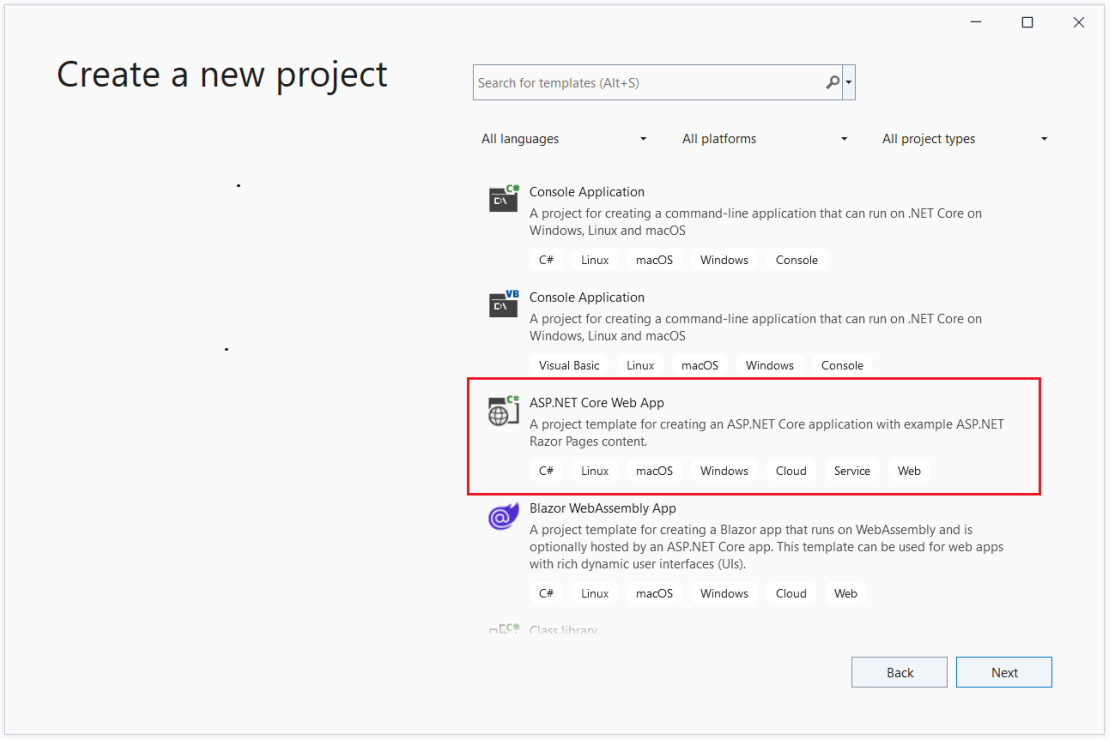
Create the web app project

Visual Studio

1. Start Visual Studio 2022 and select **Create a new project**.



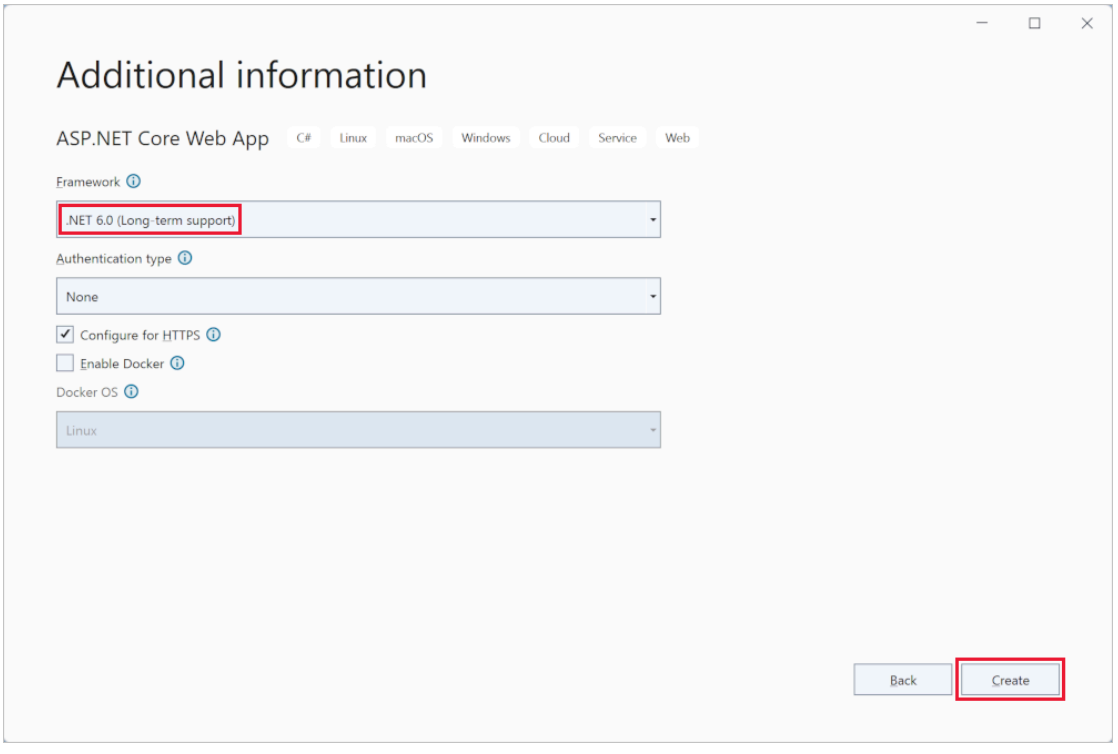
2. In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.



3. In the **Configure your new project** dialog, enter `ContosoUniversity` for **Project name**. It's important to name the project *ContosoUniversity*, including matching the capitalization, so the namespaces will match when you copy and paste example code.

4. Select **Next**.

5. In the **Additional information** dialog, select **.NET 6.0 (Long-term support)** and then select **Create**.



Set up the site style

Copy and paste the following code into the `Pages/Shared/_Layout.cshtml` file:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/ContosoUniversity.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
```

```
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Students/Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Courses/Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Instructors/Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Departments/Index">Departments</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - Contoso University - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

The layout file sets the site header, footer, and menu. The preceding code makes the following changes:

- Each occurrence of "ContosoUniversity" to "Contoso University". There are three occurrences.
- The **Home** and **Privacy** menu entries are deleted.

- Entries are added for **About**, **Students**, **Courses**, **Instructors**, and **Departments**.

In `Pages/Index.cshtml`, replace the contents of the file with the following code:

CSHTML

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Home page";
}

<div class="row mb-auto">
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 mb-4 ">
                <p class="card-text">
                    Contoso University is a sample application that
                    demonstrates how to use Entity Framework Core in an
                    ASP.NET Core Razor Pages web app.
                </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column position-static">
                <p class="card-text mb-auto">
                    You can build the application by following the steps in
a series of tutorials.
                </p>
                <p>
@*                      <a
href="https://docs.microsoft.com/aspnet/core/data/ef-rp/intro"
class="stretched-link">See the tutorial</a>
*@                  </p>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="row no-gutters border mb-4">
            <div class="col p-4 d-flex flex-column">
                <p class="card-text mb-auto">
                    You can download the completed project from GitHub.
                </p>
                <p>
@*                      <a
href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef
-rp/intro/samples" class="stretched-link">See project source code</a>
*@                  </p>
            </div>
        </div>
```

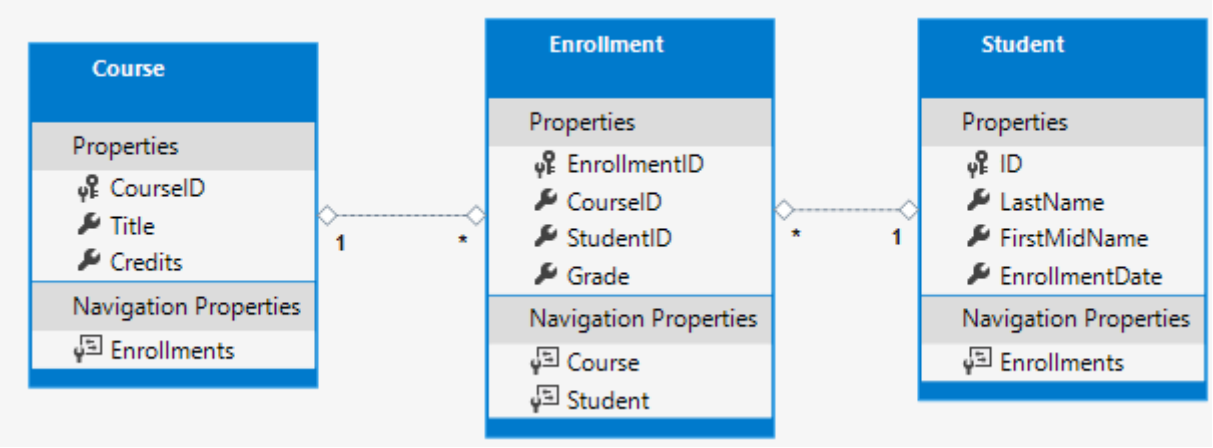
```
    </div>
</div>
```

The preceding code replaces the text about ASP.NET Core with text about this app.

Run the app to verify that the home page appears.

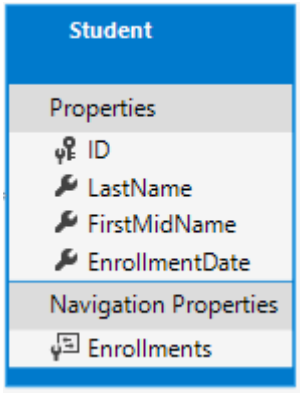
The data model

The following sections create a data model:



A student can enroll in any number of courses, and a course can have any number of students enrolled in it.

The Student entity



- Create a *Models* folder in the project folder.
- Create `Models/Student.cs` with the following code:

```
C#

namespace ContosoUniversity.Models
{
    public class Student
```

```
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

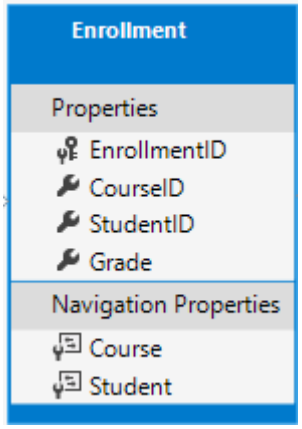
The `ID` property becomes the primary key column of the database table that corresponds to this class. By default, EF Core interprets a property that's named `ID` or `classnameID` as the primary key. So the alternative automatically recognized name for the `Student` class primary key is `StudentID`. For more information, see EF Core - Keys.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. In this case, the `Enrollments` property of a `Student` entity holds all of the `Enrollment` entities that are related to that Student. For example, if a Student row in the database has two related Enrollment rows, the `Enrollments` navigation property contains those two Enrollment entities.

In the database, an Enrollment row is related to a Student row if its `StudentID` column contains the student's ID value. For example, suppose a Student row has ID=1. Related Enrollment rows will have `StudentID` = 1. `StudentID` is a *foreign key* in the Enrollment table.

The `Enrollments` property is defined as `ICollection<Enrollment>` because there may be multiple related Enrollment entities. Other collection types can be used, such as `List<Enrollment>` or `HashSet<Enrollment>`. When `ICollection<Enrollment>` is used, EF Core creates a `HashSet<Enrollment>` collection by default.

The Enrollment entity



Create `Models/Enrollment.cs` with the following code:

```
C#

using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the primary key; this entity uses the `classnameID` pattern instead of `ID` by itself. For a production data model, many developers choose one pattern and use it consistently. This tutorial uses both just to illustrate that both work. Using `ID` without `classname` makes it easier to implement some kinds of data model changes.

The `Grade` property is an `enum`. The question mark after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's null is different from a zero grade—null means a grade isn't known or hasn't been assigned yet.

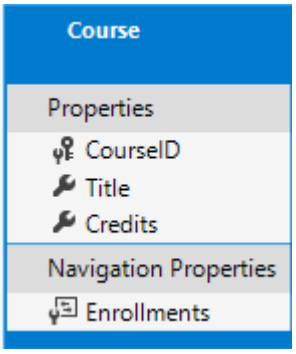
The `StudentID` property is a foreign key, and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property contains a single `Student` entity.

The `CourseID` property is a foreign key, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

EF Core interprets a property as a foreign key if it's named `<navigation property name><primary key property name>`. For example, `StudentID` is the foreign key for the `Student` navigation property, since the `Student` entity's primary key is `ID`. Foreign key properties

can also be named `<primary key property name>`. For example, `CourseID` since the `Course` entity's primary key is `CourseID`.

The Course entity



Create `Models/Course.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The `DatabaseGenerated` attribute allows the app to specify the primary key rather than having the database generate it.

Build the app. The compiler generates several warnings about how `null` values are handled. See this GitHub issue, Nullable reference types, and Tutorial: Express your design intent more clearly with nullable and non-nullable reference types for more information.

To eliminate the warnings from nullable reference types, remove the following line from the `ContosoUniversity.csproj` file:

XML

```
<Nullable>enable</Nullable>
```

The scaffolding engine currently does not support nullable reference types, therefore the models used in scaffold can't either.

Remove the `?` nullable reference type annotation from `public string? RequestId { get; set; }` in `Pages/Error.cshtml.cs` so the project builds without compiler warnings.

Scaffold Student pages

In this section, the ASP.NET Core scaffolding tool is used to generate:

- An EF Core `DbContext` class. The context is the main class that coordinates Entity Framework functionality for a given data model. It derives from the Microsoft.EntityFrameworkCore.DbContext class.
- Razor pages that handle Create, Read, Update, and Delete (CRUD) operations for the `Student` entity.

Visual Studio

- Create a *Pages/Students* folder.
- In **Solution Explorer**, right-click the *Pages/Students* folder and select **Add** > **New Scaffolded Item**.
- In the **Add New Scaffold Item** dialog:
  - In the left tab, select **Installed** > **Common** > **Razor Pages**
  - Select **Razor Pages using Entity Framework (CRUD)** > **ADD**.
- In the **Add Razor Pages using Entity Framework (CRUD)** dialog:
  - In the **Model class** drop-down, select **Student (ContosoUniversity.Models)**.
  - In the **Data context class** row, select the + (plus) sign.
    - Change the data context name to end in `SchoolContext` rather than `ContosoUniversityContext`. The updated context name: `ContosoUniversity.Data.SchoolContext`
    - Select **Add** to finish adding the data context class.
    - Select **Add** to finish the **Add Razor Pages** dialog.

The following packages are automatically installed:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`

- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

If the preceding step fails, build the project and retry the scaffold step.

The scaffolding process:

- Creates Razor pages in the *Pages/Students* folder:
  - `Create.cshtml` and `Create.cshtml.cs`
  - `Delete.cshtml` and `Delete.cshtml.cs`
  - `Details.cshtml` and `Details.cshtml.cs`
  - `Edit.cshtml` and `Edit.cshtml.cs`
  - `Index.cshtml` and `Index.cshtml.cs`
- Creates `Data/SchoolContext.cs`.
- Adds the context to dependency injection in `Program.cs`.
- Adds a database connection string to `appsettings.json`.

Database connection string

The scaffolding tool generates a connection string in the `appsettings.json` file.

Visual Studio

The connection string specifies SQL Server LocalDB:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=
(localdb)\\mssqllocaldb;Database=SchoolContext-
0e9;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. By default, LocalDB creates *.mdf*

files in the `C:/Users/<user>` directory.

Update the database context class

The main class that coordinates EF Core functionality for a given data model is the database context class. The context is derived from Microsoft.EntityFrameworkCore.DbContext. The context specifies which entities are included in the data model. In this project, the class is named `SchoolContext`.

Update `Data/SchoolContext.cs` with the following code:

C#

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext (DbContextOptions<SchoolContext> options)
            : base(options)
        {
        }

        public DbSet<Student> Students { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Course> Courses { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

The preceding code changes from the singular `DbSet<Student> Student` to the plural `DbSet<Student> Students`. To make the Razor Pages code match the new `DBSet` name, make a global change from: `_context.Student.` to: `_context.Students.`

There are 8 occurrences.

Because an entity set contains multiple entities, many developers prefer the `DBSet` property names should be plural.

The highlighted code:

- Creates a DbSet<TEntity> property for each entity set. In EF Core terminology:
  - An entity set typically corresponds to a database table.
  - An entity corresponds to a row in the table.
- Calls OnModelCreating. `OnModelCreating`:
  - Is called when `SchoolContext` has been initialized but before the model has been secured and used to initialize the context.
  - Is required because later in the tutorial the `Student` entity will have references to the other entities.

We hope to fix this issue in a future release.

Program.cs

ASP.NET Core is built with dependency injection. Services such as the `SchoolContext` are registered with dependency injection during app startup. Components that require these services, such as Razor Pages, are provided these services via constructor parameters. The constructor code that gets a database context instance is shown later in the tutorial.

The scaffolding tool automatically registered the context class with the dependency injection container.

Visual Studio

The following highlighted lines were added by the scaffolder:

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));
```

The name of the connection string is passed in to the context by calling a method on a DbContextOptions object. For local development, the ASP.NET Core configuration

system reads the connection string from the `appsettings.json` or the `appsettings.Development.json` file.

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter and UseMigrationsEndPoint as shown in the following code:

Visual Studio

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}
```

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package.

In the Package Manager Console, enter the following to add the NuGet package:

PowerShell

```
Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
```

The `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` NuGet package provides ASP.NET Core middleware for Entity Framework Core error pages. This middleware helps to detect and diagnose errors with Entity Framework Core migrations.

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment for EF migrations errors.

Create the database

Update `Program.cs` to create the database if it doesn't exist:

Visual Studio

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("SchoolCo
ntext")));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseMigrationsEndPoint();
}

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    // DbInitializer.Initialize(context);
}
```

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The EnsureCreated method takes no action if a database for the context exists. If no database exists, it creates the database and schema. `EnsureCreated` enables the following workflow for handling data model changes:

- Delete the database. Any existing data is lost.
- Change the data model. For example, add an `EmailAddress` field.
- Run the app.
- `EnsureCreated` creates a database with the new schema.

This workflow works early in development when the schema is rapidly evolving, as long as data doesn't need to be preserved. The situation is different when data that has been entered into the database needs to be preserved. When that is the case, use migrations.

Later in the tutorial series, the database is deleted that was created by `EnsureCreated` and migrations is used. A database that is created by `EnsureCreated` can't be updated by using migrations.

Test the app

- Run the app.
- Select the **Students** link and then **Create New**.
- Test the Edit, Details, and Delete links.

Seed the database

The `EnsureCreated` method creates an empty database. This section adds code that populates the database with test data.

Create `Data/DbInitializer.cs` with the following code:

```
C#
```

```
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.P
arse("2019-09-01")},
                new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Pa
rse("2017-09-01")},
                new
Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse
("2018-09-01")},
                new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Pa
rse("2017-09-01")},
                new
Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2017
-09-01")},
                new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Pars
e("2016-09-01")},
                new
Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse
("2018-09-01")},
                new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Pars
e("2019-09-01")}
            };

            context.Students.AddRange(students);
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course{CourseID=1050,Title="Chemistry",Credits=3},
                new Course{CourseID=4022,Title="Microeconomics",Credits=3},
                new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
                new Course{CourseID=1045,Title="Calculus",Credits=4},
                new Course{CourseID=3141,Title="Trigonometry",Credits=4},
                new Course{CourseID=2021,Title="Composition",Credits=3},
                new Course{CourseID=2042,Title="Literature",Credits=4}
```

```
            };

            context.Courses.AddRange(courses);
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
                new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
                new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
                new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
                new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
                new Enrollment{StudentID=3,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=1050},
                new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
                new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
                new Enrollment{StudentID=6,CourseID=1045},
                new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };

            context.Enrollments.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The code checks if there are any students in the database. If there are no students, it adds test data to the database. It creates the test data in arrays rather than `List<T>` collections to optimize performance.

- In `Program.cs`, remove `//` from the `DbInitializer.Initialize` line:

C#

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    var context = services.GetRequiredService<SchoolContext>();
    context.Database.EnsureCreated();
    DbInitializer.Initialize(context);
}
```

Visual Studio

- Stop the app if it's running, and run the following command in the **Package Manager Console** (PMC):

PowerShell

```
Drop-Database -Confirm
```

- Respond with `Y` to delete the database.

- Restart the app.
- Select the Students page to see the seeded data.

View the database

Visual Studio

- Open **SQL Server Object Explorer** (SSOX) from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB** > **Databases** > **SchoolContext-{GUID}**. The database name is generated from the context name provided earlier plus a dash and a GUID.
- Expand the **Tables** node.
- Right-click the **Student** table and click **View Data** to see the columns created and the rows inserted into the table.
- Right-click the **Student** table and click **View Code** to see how the `Student` model maps to the `Student` table schema.

Asynchronous EF methods in ASP.NET Core web apps

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't doing work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server can handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time. For low traffic situations, the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, the async keyword, `Task` return value, `await` keyword, and `ToListAsync` method make the code execute asynchronously.

```
C#

public async Task OnGetAsync()
{
    Students = await _context.Students.ToListAsync();
}
```

- The `async` keyword tells the compiler to:
  - Generate callbacks for parts of the method body.
  - Create the Task object that's returned.
- The `Task` return type represents ongoing work.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF Core:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes `ToListAsync`, `SingleOrDefaultAsync`, `FirstOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF Core context isn't thread safe: don't try to do multiple operations in parallel.
- To take advantage of the performance benefits of async code, verify that library packages (such as for paging) use async if they call EF Core methods that send queries to the database.

For more information about asynchronous programming in .NET, see Async Overview and Asynchronous programming with async and await.

> ⚠ **Warning**
>
> The async implementation of Microsoft.Data.SqlClient has some known issues (#593, #601, and others). If you're seeing unexpected performance problems,

try using sync command execution instead, especially when dealing with large text or binary values.

Performance considerations

In general, a web page shouldn't be loading an arbitrary number of rows. A query should use paging or a limiting approach. For example, the preceding query could use `Take` to limit the rows returned:

C#

```
public async Task OnGetAsync()
{
    Student = await _context.Students.Take(10).ToListAsync();
}
```

Enumerating a large table in a view could return a partially constructed HTTP 200 response if a database exception occurs part way through the enumeration.

Paging is covered later in the tutorial.

For more information, see Performance considerations (EF).

Next steps

Use SQLite for development, SQL Server for production

Next tutorial

Part 2, Razor Pages with EF Core in ASP.NET Core - CRUD

Article • 04/11/2023

By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

In this tutorial, the scaffolded CRUD (create, read, update, delete) code is reviewed and customized.

No repository

Some developers use a service layer or repository pattern to create an abstraction layer between the UI (Razor Pages) and the data access layer. This tutorial doesn't do that. To minimize complexity and keep the tutorial focused on EF Core, EF Core code is added directly to the page model classes.

Update the Details page

The scaffolded code for the Students pages doesn't include enrollment data. In this section, enrollments are added to the `Details` page.

Read enrollments

To display a student's enrollment data on the page, the enrollment data must be read. The scaffolded code in `Pages/Students/Details.cshtml.cs` reads only the `Student` data, without the `Enrollment` data:

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
```

```
    }

    Student = await _context.Students.FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

Replace the `OnGetAsync` method with the following code to read enrollment data for the selected student. The changes are highlighted.

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    Student = await _context.Students
        .Include(s => s.Enrollments)
        .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}
```

The Include and ThenInclude methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. These methods are examined in detail in the Read related data tutorial.

The AsNoTracking method improves performance in scenarios where the entities returned are not updated in the current context. `AsNoTracking` is discussed later in this tutorial.

Display enrollments

Replace the code in `Pages/Students/Details.cshtml` with the following code to display a list of enrollments. The changes are highlighted.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h1>Details</h1>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
```

```
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.Enrollments)
        </dt>
        <dd class="col-sm-10">
            <table class="table">
                <tr>
                    <th>Course Title</th>
                    <th>Grade</th>
                </tr>
                @foreach (var item in Model.Student.Enrollments)
                {
                    <tr>
                        <td>
                            @Html.DisplayFor(modelItem => item.Course.Title)
                        </td>
                        <td>
                            @Html.DisplayFor(modelItem => item.Grade)
                        </td>
                    </tr>
                }
            </table>
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Student.ID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

The preceding code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the `Course` entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. The list of courses and grades for the selected student is displayed.

Ways to read one entity

The generated code uses FirstOrDefaultAsync to read one entity. This method returns null if nothing is found; otherwise, it returns the first row found that satisfies the query filter criteria. `FirstOrDefaultAsync` is generally a better choice than the following alternatives:

- SingleOrDefaultAsync - Throws an exception if there's more than one entity that satisfies the query filter. To determine if more than one row could be returned by the query, `SingleOrDefaultAsync` tries to fetch multiple rows. This extra work is

unnecessary if the query can only return one entity, as when it searches on a unique key.

- FindAsync - Finds an entity with the primary key (PK). If an entity with the PK is being tracked by the context, it's returned without a request to the database. This method is optimized to look up a single entity, but you can't call `Include` with `FindAsync`. So if related data is needed, `FirstOrDefaultAsync` is the better choice.

Route data vs. query string

The URL for the Details page is `https://localhost:<port>/Students/Details?id=1`. The entity's primary key value is in the query string. Some developers prefer to pass the key value in route data: `https://localhost:<port>/Students/Details/1`. For more information, see Update the generated code.

Update the Create page

The scaffolded `OnPostAsync` code for the Create page is vulnerable to overposting. Replace the `OnPostAsync` method in `Pages/Students/Create.cshtml.cs` with the following code.

C#

```
public async Task<IActionResult> OnPostAsync()
{
    var emptyStudent = new Student();

    if (await TryUpdateModelAsync<Student>(
        emptyStudent,
        "student",   // Prefix for form value.
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        _context.Students.Add(emptyStudent);
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

TryUpdateModelAsync

The preceding code creates a Student object and then uses posted form fields to update the Student object's properties. The TryUpdateModelAsync method:

- Uses the posted form values from the PageContext property in the PageModel.
- Updates only the properties listed (`s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate`).
- Looks for form fields with a "student" prefix. For example, `Student.FirstMidName`. It's not case sensitive.
- Uses the model binding system to convert form values from strings to the types in the `Student` model. For example, `EnrollmentDate` is converted to `DateTime`.

Run the app, and create a student entity to test the Create page.

Overposting

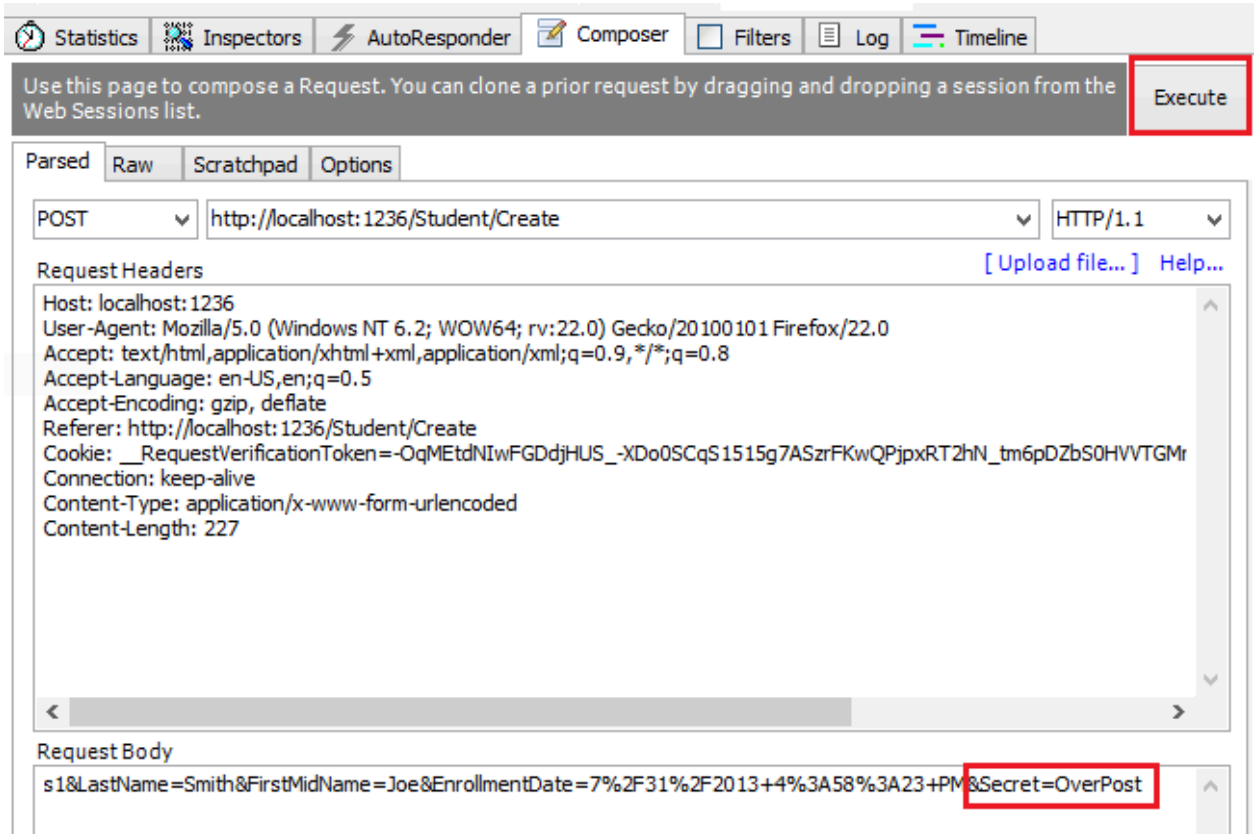
Using `TryUpdateModel` to update fields with posted values is a security best practice because it prevents overposting. For example, suppose the Student entity includes a `Secret` property that this web page shouldn't update or add:

C#

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if the app doesn't have a `Secret` field on the create or update Razor Page, a hacker could set the `Secret` value by overposting. A hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. The original code doesn't limit the fields that the model binder uses when it creates a Student instance.

Whatever value the hacker specified for the `Secret` form field is updated in the database. The following image shows the Fiddler tool adding the `Secret` field, with the value "OverPost", to the posted form values.



The value "OverPost" is successfully added to the `Secret` property of the inserted row. That happens even though the app designer never intended the `Secret` property to be set with the Create page.

View model

View models provide an alternative way to prevent overposting.

The application model is often called the domain model. The domain model typically contains all the properties required by the corresponding entity in the database. The view model contains only the properties needed for the UI page, for example, the Create page.

In addition to the view model, some apps use a binding model or input model to pass data between the Razor Pages page model class and the browser.

Consider the following `StudentVM` view model:

C#

```
public class StudentVM
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
```

```
    public DateTime EnrollmentDate { get; set; }
}
```

The following code uses the `StudentVM` view model to create a new student:

C#

```
[BindProperty]
public StudentVM StudentVM { get; set; }

public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    var entry = _context.Add(new Student());
    entry.CurrentValues.SetValues(StudentVM);
    await _context.SaveChangesAsync();
    return RedirectToPage("./Index");
}
```

The SetValues method sets the values of this object by reading values from another PropertyValues object. `SetValues` uses property name matching. The view model type:

- Doesn't need to be related to the model type.
- Needs to have properties that match.

Using `StudentVM` requires the Create page use `StudentVM` rather than `Student`:

CSHTML

```
@page
@model CreateVMModel

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Student</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
```

```
                <label asp-for="StudentVM.LastName" class="control-label">
</label>
                <input asp-for="StudentVM.LastName" class="form-control" />
                <span asp-validation-for="StudentVM.LastName" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.FirstMidName" class="control-
label"></label>
                <input asp-for="StudentVM.FirstMidName" class="form-control"
/>
                <span asp-validation-for="StudentVM.FirstMidName"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="StudentVM.EnrollmentDate" class="control-
label"></label>
                <input asp-for="StudentVM.EnrollmentDate" class="form-
control" />
                <span asp-validation-for="StudentVM.EnrollmentDate"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Edit page

In `Pages/Students/Edit.cshtml.cs`, replace the `OnGetAsync` and `OnPostAsync` methods with the following code.

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
```

```
    Student = await _context.Students.FindAsync(id);

    if (Student == null)
    {
        return NotFound();
    }
    return Page();
}

public async Task<IActionResult> OnPostAsync(int id)
{
    var studentToUpdate = await _context.Students.FindAsync(id);

    if (studentToUpdate == null)
    {
        return NotFound();
    }

    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "student",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }

    return Page();
}
```

The code changes are similar to the Create page with a few exceptions:

- `FirstOrDefaultAsync` has been replaced with FindAsync. When you don't have to include related data, `FindAsync` is more efficient.
- `OnPostAsync` has an `id` parameter.
- The current student is fetched from the database, rather than creating an empty student.

Run the app, and test it by creating and editing a student.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database. This tracking information determines what happens when SaveChangesAsync is called. For example, when a new entity is passed to the AddAsync method, that entity's state is set to Added. When `SaveChangesAsync` is called, the database context issues a SQL `INSERT` command.

An entity may be in one of the following states:

- `Added`: The entity doesn't yet exist in the database. The `SaveChanges` method issues an `INSERT` statement.
- `Unchanged`: No changes need to be saved with this entity. An entity has this status when it's read from the database.
- `Modified`: Some or all of the entity's property values have been modified. The `SaveChanges` method issues an `UPDATE` statement.
- `Deleted`: The entity has been marked for deletion. The `SaveChanges` method issues a `DELETE` statement.
- `Detached`: The entity isn't being tracked by the database context.

In a desktop app, state changes are typically set automatically. An entity is read, changes are made, and the entity state is automatically changed to `Modified`. Calling `SaveChanges` generates a SQL `UPDATE` statement that updates only the changed properties.

In a web app, the `DbContext` that reads an entity and displays the data is disposed after a page is rendered. When a page's `OnPostAsync` method is called, a new web request is made and with a new instance of the `DbContext`. Rereading the entity in that new context simulates desktop processing.

Update the Delete page

In this section, a custom error message is implemented when the call to `SaveChanges` fails.

Replace the code in `Pages/Students/Delete.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
```

```
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<DeleteModel> _logger;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context,
                           ILogger<DeleteModel> logger)
        {
            _context = context;
            _logger = logger;
        }

        [BindProperty]
        public Student Student { get; set; }
        public string ErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id, bool?
saveChangesError = false)
        {
            if (id == null)
            {
                return NotFound();
            }

            Student = await _context.Students
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Student == null)
            {
                return NotFound();
            }

            if (saveChangesError.GetValueOrDefault())
            {
                ErrorMessage = String.Format("Delete {ID} failed. Try
again", id);
            }

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var student = await _context.Students.FindAsync(id);

            if (student == null)
            {
                return NotFound();
            }
```

```
            try
            {
                _context.Students.Remove(student);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            catch (DbUpdateException ex)
            {
                _logger.LogError(ex, ErrorMessage);

                return RedirectToAction("./Delete",
                                        new { id, saveChangesError = true });
            }
        }
    }
}
```

The preceding code:

- Adds Logging.
- Adds the optional parameter `saveChangesError` to the `OnGetAsync` method signature. `saveChangesError` indicates whether the method was called after a failure to delete the student object.

The delete operation might fail because of transient network problems. Transient network errors are more likely when the database is in the cloud. The `saveChangesError` parameter is `false` when the Delete page `OnGetAsync` is called from the UI. When `OnGetAsync` is called by `OnPostAsync` because the delete operation failed, the `saveChangesError` parameter is `true`.

The `OnPostAsync` method retrieves the selected entity, then calls the Remove method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL `DELETE` command is generated. If `Remove` fails:

- The database exception is caught.
- The Delete pages `OnGetAsync` method is called with `saveChangesError=true`.

Add an error message to `Pages/Students/Delete.cshtml`:

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.DeleteModel

@{
    ViewData["Title"] = "Delete";
}
```

```
<h1>Delete</h1>

<p class="text-danger">@Model.ErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Student</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.LastName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.LastName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.FirstMidName)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.FirstMidName)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Student.EnrollmentDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Student.EnrollmentDate)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Student.ID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Run the app and delete a student to test the Delete page.

Next steps

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Part 3, Razor Pages with EF Core in ASP.NET Core - Sort, Filter, Paging

Article • 04/11/2023

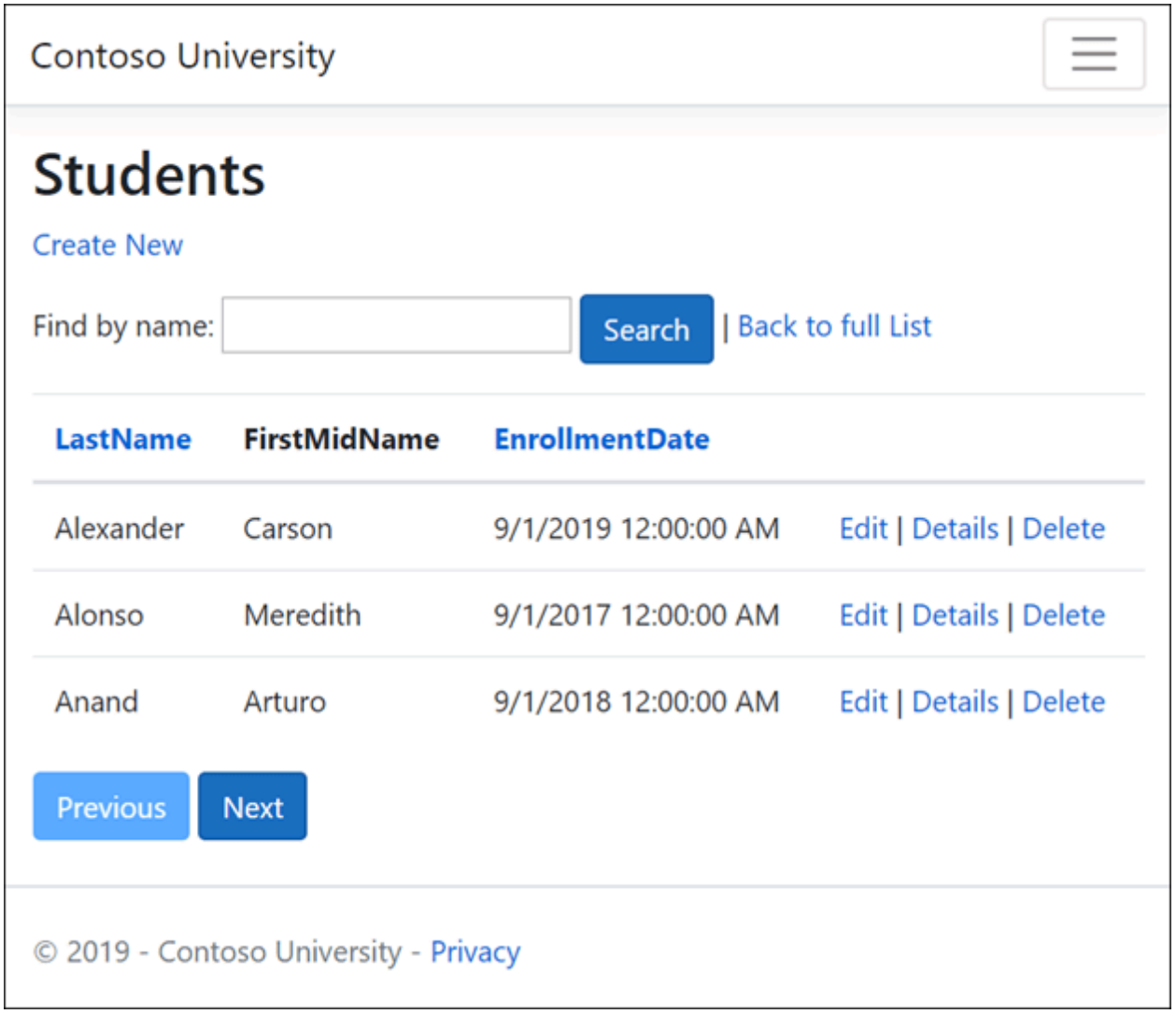
By Tom Dykstra, Jeremy Likness, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial adds sorting, filtering, and paging functionality to the Students pages.

The following illustration shows a completed page. The column headings are clickable links to sort the column. Click a column heading repeatedly to switch between ascending and descending sort order.



Add sorting

Replace the code in `Pages/Students/Index.cshtml.cs` with the following code to add sorting.

C#

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;
    public IndexModel(SchoolContext context)
    {
        _context = context;
    }

    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder)
    {
        // using System;
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;

        switch (sortOrder)
        {
            case "name_desc":
                studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                break;
            case "Date":
                studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                break;
            case "date_desc":
                studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                break;
            default:
                studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                break;
        }

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Requires adding `using System;`.
- Adds properties to contain the sorting parameters.
- Changes the name of the `Student` property to `Students`.
- Replaces the code in the `OnGetAsync` method.

The `OnGetAsync` method receives a `sortOrder` parameter from the query string in the URL. The URL and query string is generated by the Anchor Tag Helper.

The `sortOrder` parameter is either `Name` or `Date`. The `sortOrder` parameter is optionally followed by `_desc` to specify descending order. The default sort order is ascending.

When the Index page is requested from the **Students** link, there's no query string. The students are displayed in ascending order by last name. Ascending order by last name is the `default` in the `switch` statement. When the user clicks a column heading link, the appropriate `sortOrder` value is provided in the query string value.

`NameSort` and `DateSort` are used by the Razor Page to configure the column heading hyperlinks with the appropriate query string values:

```
C#
```

```
NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
DateSort = sortOrder == "Date" ? "date_desc" : "Date";
```

The code uses the C# conditional operator ?:. The `?:` operator is a ternary operator, it takes three operands. The first line specifies that when `sortOrder` is null or empty, `NameSort` is set to `name_desc`. If `sortOrder` is ***not*** null or empty, `NameSort` is set to an empty string.

These two statements enable the page to set the column heading hyperlinks as follows:

Expand table

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code initializes an `IQueryable<Student>` before the switch statement, and modifies it in the switch statement:

C#

```
IQueryable<Student> studentsIQ = from s in _context.Students
                                 select s;

switch (sortOrder)
{
    case "name_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
        break;
    case "Date":
        studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
        break;
    case "date_desc":
        studentsIQ = studentsIQ.OrderByDescending(s => s.EnrollmentDate);
        break;
    default:
        studentsIQ = studentsIQ.OrderBy(s => s.LastName);
        break;
}

Students = await studentsIQ.AsNoTracking().ToListAsync();
```

When an `IQueryable` is created or modified, no query is sent to the database. The query isn't executed until the `IQueryable` object is converted into a collection. `IQueryable` are converted to a collection by calling a method such as `ToListAsync`. Therefore, the `IQueryable` code results in a single query that's not executed until the following statement:

C#

```
Students = await studentsIQ.AsNoTracking().ToListAsync();
```

`OnGetAsync` could get verbose with a large number of sortable columns. For information about an alternative way to code this functionality, see Use dynamic LINQ to simplify code in the MVC version of this tutorial series.

Add column heading hyperlinks to the Student Index page

Replace the code in `Students/Index.cshtml`, with the following code. The changes are highlighted.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>
<p>
    <a asp-page="Create">Create New</a>
</p>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
```

```
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code:

- Adds hyperlinks to the `LastName` and `EnrollmentDate` column headings.
- Uses the information in `NameSort` and `DateSort` to set up hyperlinks with the current sort order values.
- Changes the page heading from Index to Students.
- Changes `Model.Student` to `Model.Students`.

To verify that sorting works:

- Run the app and select the **Students** tab.
- Click the column headings.

Add filtering

To add filtering to the Students Index page:

- A text box and a submit button is added to the Razor Page. The text box supplies a search string on the first or last name.
- The page model is updated to use the text box value.

Update the OnGetAsync method

Replace the code in `Students/Index.cshtml.cs` with the following code to add filtering:

C#

```
public class IndexModel : PageModel
{
    private readonly SchoolContext _context;

    public IndexModel(SchoolContext context)
    {
        _context = context;
    }
```

```
    public string NameSort { get; set; }
    public string DateSort { get; set; }
    public string CurrentFilter { get; set; }
    public string CurrentSort { get; set; }

    public IList<Student> Students { get; set; }

    public async Task OnGetAsync(string sortOrder, string searchString)
    {
        NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
        DateSort = sortOrder == "Date" ? "date_desc" : "Date";

        CurrentFilter = searchString;

        IQueryable<Student> studentsIQ = from s in _context.Students
                                         select s;
        if (!String.IsNullOrEmpty(searchString))
        {
            studentsIQ = studentsIQ.Where(s =>
s.LastName.Contains(searchString)
                                   ||
s.FirstMidName.Contains(searchString));
        }

        switch (sortOrder)
        {
            case "name_desc":
                studentsIQ = studentsIQ.OrderByDescending(s => s.LastName);
                break;
            case "Date":
                studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                break;
            case "date_desc":
                studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                break;
            default:
                studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                break;
        }

        Students = await studentsIQ.AsNoTracking().ToListAsync();
    }
}
```

The preceding code:

- Adds the `searchString` parameter to the `OnGetAsync` method, and saves the parameter value in the `CurrentFilter` property. The search string value is received from a text box that's added in the next section.
- Adds to the LINQ statement a `Where` clause. The `Where` clause selects only students whose first name or last name contains the search string. The LINQ statement is

executed only if there's a value to search for.

IQueryable vs. IEnumerable

The code calls the Where method on an `IQueryable` object, and the filter is processed on the server. In some scenarios, the app might be calling the `Where` method as an extension method on an in-memory collection. For example, suppose `_context.Students` changes from EF Core `DbSet` to a repository method that returns an `IEnumerable` collection. The result would normally be the same but in some cases may be different.

For example, the .NET Framework implementation of `Contains` performs a case-sensitive comparison by default. In SQL Server, `Contains` case-sensitivity is determined by the collation setting of the SQL Server instance. SQL Server defaults to case-insensitive. SQLite defaults to case-sensitive. `ToUpper` could be called to make the test explicitly case-insensitive:

```
C#

Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())`
```

The preceding code would ensure that the filter is case-insensitive even if the `Where` method is called on an `IEnumerable` or runs on SQLite.

When `Contains` is called on an `IEnumerable` collection, the .NET Core implementation is used. When `Contains` is called on an `IQueryable` object, the database implementation is used.

Calling `Contains` on an `IQueryable` is usually preferable for performance reasons. With `IQueryable`, the filtering is done by the database server. If an `IEnumerable` is created first, all the rows have to be returned from the database server.

There's a performance penalty for calling `ToUpper`. The `ToUpper` code adds a function in the WHERE clause of the TSQL SELECT statement. The added function prevents the optimizer from using an index. Given that SQL is installed as case-insensitive, it's best to avoid the `ToUpper` call when it's not needed.

For more information, see How to use case-insensitive query with Sqlite provider.

Update the Razor page

Replace the code in `Pages/Students/Index.cshtml` to add a **Search** button.

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString"
value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
```

```
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

The preceding code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST. With POST, the parameters are passed in the HTTP message body and not in the URL. When HTTP GET is used, the form data is passed in the URL as query strings. Passing the data with query strings enables users to bookmark the URL. The W3C guidelines recommend that GET should be used when the action doesn't result in an update.

Test the app:

- Select the **Students** tab and enter a search string. If you're using SQLite, the filter is case-insensitive only if you implemented the optional `ToUpper` code shown earlier.
- Select **Search**.

Notice that the URL contains the search string. For example:

```
browser-address-bar
```

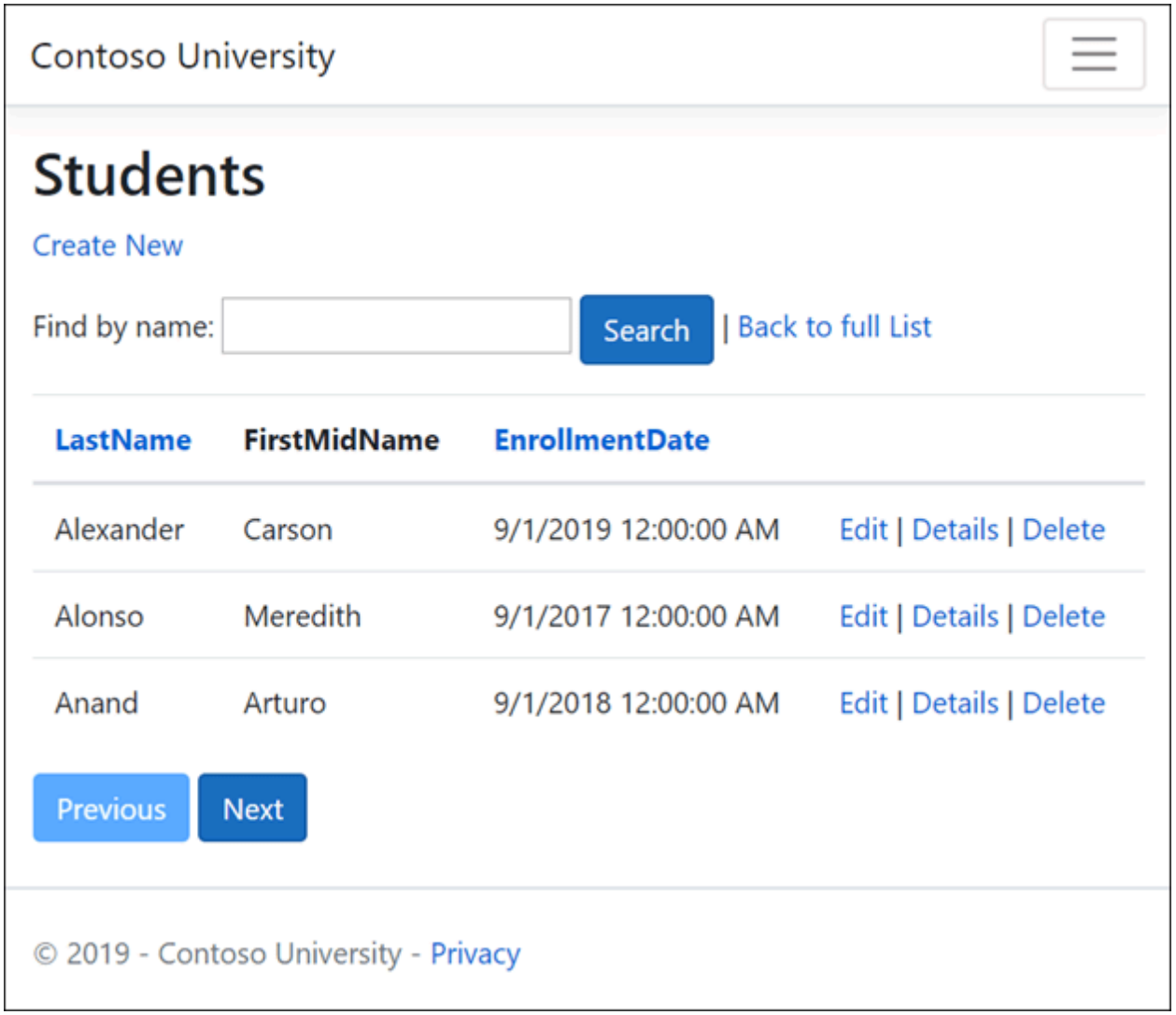
```
https://localhost:5001/Students?SearchString=an
```

If the page is bookmarked, the bookmark contains the URL to the page and the `SearchString` query string. The `method="get"` in the `form` tag is what caused the query string to be generated.

Currently, when a column heading sort link is selected, the filter value from the **Search** box is lost. The lost filter value is fixed in the next section.

Add paging

In this section, a `PaginatedList` class is created to support paging. The `PaginatedList` class uses `Skip` and `Take` statements to filter data on the server instead of retrieving all rows of the table. The following illustration shows the paging buttons.



Create the PaginatedList class

In the project folder, create `PaginatedList.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
```

```
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int
pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }

        public bool HasPreviousPage => PageIndex > 1;

        public bool HasNextPage => PageIndex < TotalPages;

        public static async Task<PaginatedList<T>> CreateAsync(
            IQueryable<T> source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip(
                (pageIndex - 1) * pageSize)
                .Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in the preceding code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it returns a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` are used to enable or disable **Previous** and **Next** paging buttons.

The `CreateAsync` method is used to create the `PaginatedList<T>`. A constructor can't create the `PaginatedList<T>` object; constructors can't run asynchronous code.

Add page size to configuration

Add `PageSize` to the `appsettings.json` Configuration file:

JSON

```
{
  "PageSize": 3,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
```

```
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SchoolContext": "Server=(localdb)\\mssqllocaldb;Database=CU-
1;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

Add paging to IndexModel

Replace the code in `Students/Index.cshtml.cs` to add paging.

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Students
{
    public class IndexModel : PageModel
    {
        private readonly SchoolContext _context;
        private readonly IConfiguration Configuration;

        public IndexModel(SchoolContext context, IConfiguration
configuration)
        {
            _context = context;
            Configuration = configuration;
        }

        public string NameSort { get; set; }
        public string DateSort { get; set; }
        public string CurrentFilter { get; set; }
        public string CurrentSort { get; set; }
```

```
        public PaginatedList<Student> Students { get; set; }

        public async Task OnGetAsync(string sortOrder,
            string currentFilter, string searchString, int? pageIndex)
        {
            CurrentSort = sortOrder;
            NameSort = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
            DateSort = sortOrder == "Date" ? "date_desc" : "Date";
            if (searchString != null)
            {
                pageIndex = 1;
            }
            else
            {
                searchString = currentFilter;
            }

            CurrentFilter = searchString;

            IQueryable<Student> studentsIQ = from s in _context.Students
                                             select s;
            if (!String.IsNullOrEmpty(searchString))
            {
                studentsIQ = studentsIQ.Where(s =>
s.LastName.Contains(searchString)
                                       ||
s.FirstMidName.Contains(searchString));
            }
            switch (sortOrder)
            {
                case "name_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s =>
s.LastName);
                    break;
                case "Date":
                    studentsIQ = studentsIQ.OrderBy(s => s.EnrollmentDate);
                    break;
                case "date_desc":
                    studentsIQ = studentsIQ.OrderByDescending(s =>
s.EnrollmentDate);
                    break;
                default:
                    studentsIQ = studentsIQ.OrderBy(s => s.LastName);
                    break;
            }

            var pageSize = Configuration.GetValue("PageSize", 4);
            Students = await PaginatedList<Student>.CreateAsync(
                studentsIQ.AsNoTracking(), pageIndex ?? 1, pageSize);
        }
    }
}
```

The preceding code:

- Changes the type of the `Students` property from `IList<Student>` to `PaginatedList<Student>`.
- Adds the page index, the current `sortOrder`, and the `currentFilter` to the `OnGetAsync` method signature.
- Saves the sort order in the `CurrentSort` property.
- Resets page index to 1 when there's a new search string.
- Uses the `PaginatedList` class to get Student entities.
- Sets `pageSize` to 3 from Configuration, 4 if configuration fails.

All the parameters that `OnGetAsync` receives are null when:

- The page is called from the **Students** link.
- The user hasn't clicked a paging or sorting link.

When a paging link is clicked, the page index variable contains the page number to display.

The `CurrentSort` property provides the Razor Page with the current sort order. The current sort order must be included in the paging links to keep the sort order while paging.

The `CurrentFilter` property provides the Razor Page with the current filter string. The `CurrentFilter` value:

- Must be included in the paging links in order to maintain the filter settings during paging.
- Must be restored to the text box when the page is redisplayed.

If the search string is changed while paging, the page is reset to 1. The page has to be reset to 1 because the new filter can result in different data to display. When a search value is entered and **Submit** is selected:

- The search string is changed.
- The `searchString` parameter isn't null.

The `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is passed to the Razor Page.

The two question marks after `pageIndex` in the `PaginatedList.CreateAsync` call represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type. The expression `pageIndex ?? 1` returns the value of `pageIndex` if it has a value, otherwise, it returns 1.

Add paging links

Replace the code in `Students/Index.cshtml` with the following code. The changes are highlighted:

CSHTML

```
@page
@model ContosoUniversity.Pages.Students.IndexModel

@{
    ViewData["Title"] = "Students";
}

<h2>Students</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>

<form asp-page="./Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name:
            <input type="text" name="SearchString"
value="@Model.CurrentFilter" />
            <input type="submit" value="Search" class="btn btn-primary" /> |
            <a asp-page="./Index">Back to full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model =>
model.Students[0].LastName)
                </a>
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Students[0].FirstMidName)
            </th>
            <th>
                <a asp-page="./Index" asp-route-sortOrder="@Model.DateSort"
                   asp-route-currentFilter="@Model.CurrentFilter">
                    @Html.DisplayNameFor(model =>
model.Students[0].EnrollmentDate)
                </a>
            </th>
```

```
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Students)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

```
@{
    var prevDisabled = !Model.Students.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.Students.HasNextPage ? "disabled" : "";
}

<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

The column header links use the query string to pass the current search string to the `OnGetAsync` method:

CSHTML

```
<a asp-page="./Index" asp-route-sortOrder="@Model.NameSort"
   asp-route-currentFilter="@Model.CurrentFilter">
    @Html.DisplayNameFor(model => model.Students[0].LastName)
</a>
```

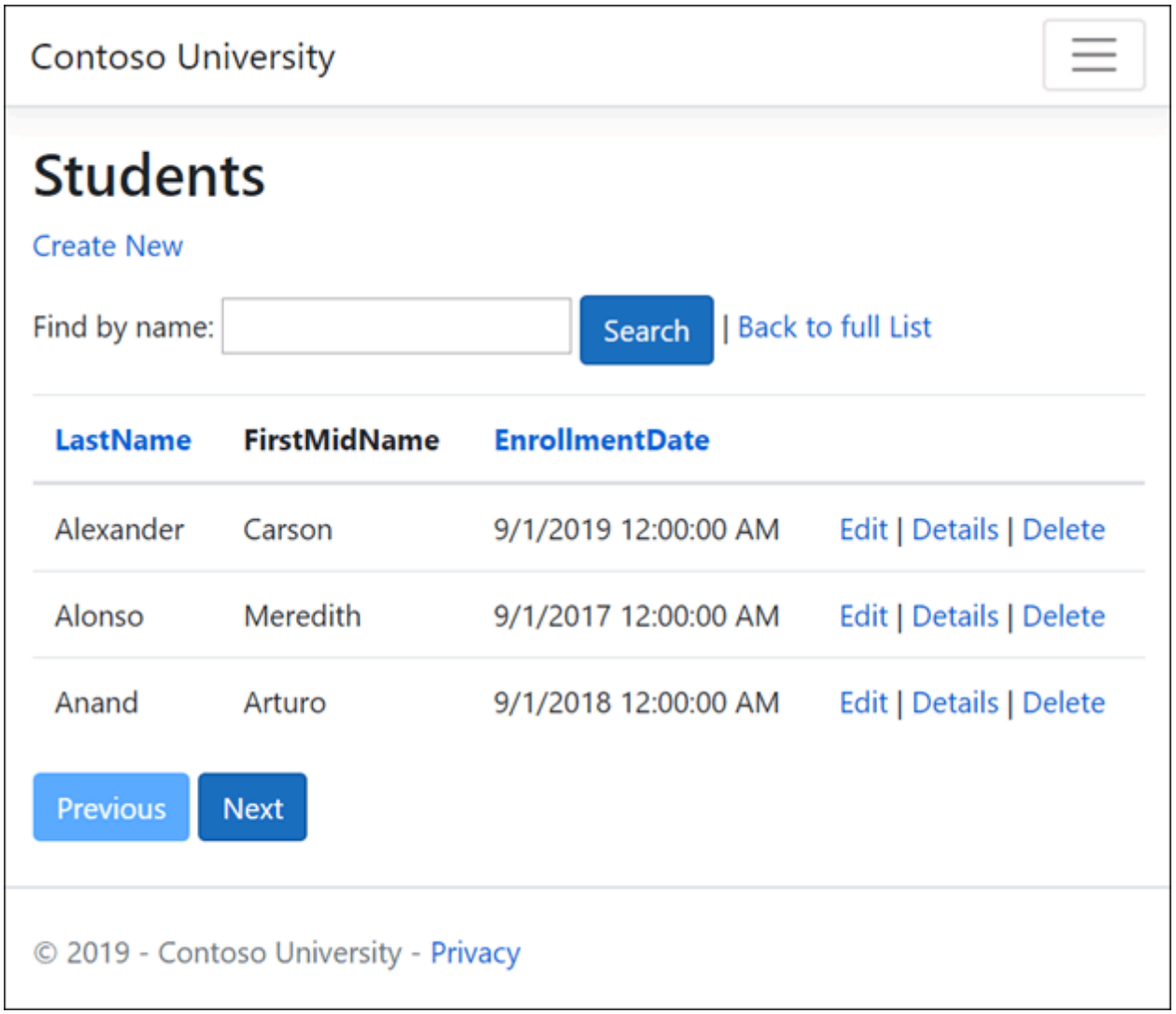
The paging buttons are displayed by tag helpers:

CSHTML

```
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex - 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @prevDisabled">
    Previous
</a>
<a asp-page="./Index"
   asp-route-sortOrder="@Model.CurrentSort"
   asp-route-pageIndex="@(Model.Students.PageIndex + 1)"
   asp-route-currentFilter="@Model.CurrentFilter"
   class="btn btn-primary @nextDisabled">
    Next
</a>
```

Run the app and navigate to the students page.

- To make sure paging works, click the paging links in different sort orders.
- To verify that paging works correctly with sorting and filtering, enter a search string and try paging.



Grouping

This section creates an `About` page that displays how many students have enrolled for each enrollment date. The update uses grouping and includes the following steps:

- Create a view model for the data used by the `About` page.
- Update the `About` page to use the view model.

Create the view model

Create a *Models/SchoolViewModels* folder.

Create `SchoolViewModels/EnrollmentDateGroup.cs` with the following code:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
```

```
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Create the Razor Page

Create a `Pages/About.cshtml` file with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.AboutModel

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>

<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model.Students)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Create the page model

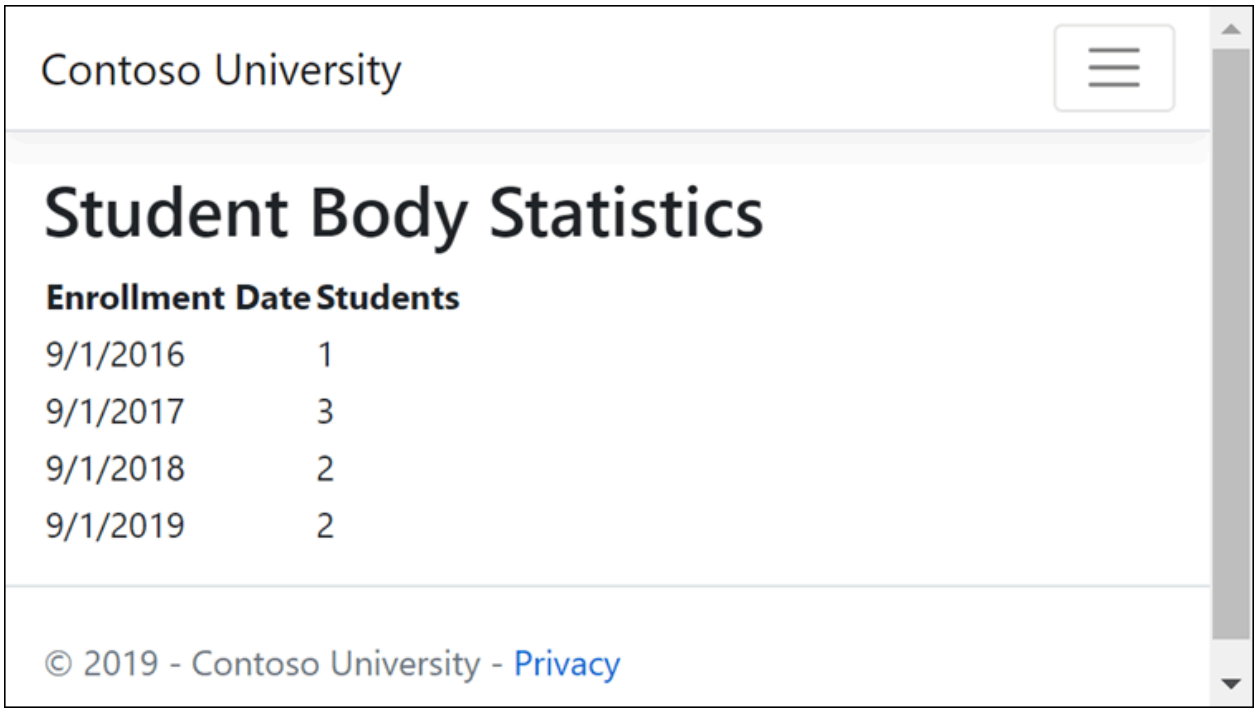
Update the `Pages/About.cshtml.cs` file with the following code:

C#

```
using ContosoUniversity.Models.SchoolViewModels;
using ContosoUniversity.Data;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using ContosoUniversity.Models;

namespace ContosoUniversity.Pages
{
    public class AboutModel : PageModel
    {
        private readonly SchoolContext _context;

        public AboutModel(SchoolContext context)
        {
            _context = context;
        }

        public IList<EnrollmentDateGroup> Students { get; set; }

        public async Task OnGetAsync()
        {
            IQueryable<EnrollmentDateGroup> data =
                from student in _context.Students
                group student by student.EnrollmentDate into dateGroup
                select new EnrollmentDateGroup()
                {
                    EnrollmentDate = dateGroup.Key,
                    StudentCount = dateGroup.Count()
                };

            Students = await data.AsNoTracking().ToListAsync();
        }
    }
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Run the app and navigate to the About page. The count of students for each enrollment date is displayed in a table.



Next steps

In the next tutorial, the app uses migrations to update the data model.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 4, Razor Pages with EF Core migrations in ASP.NET Core

Article • 07/28/2023

By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial introduces the EF Core migrations feature for managing data model changes.

When a new app is developed, the data model changes frequently. Each time the model changes, the model gets out of sync with the database. This tutorial series started by configuring the Entity Framework to create the database if it doesn't exist. Each time the data model changes, the database needs to be dropped. The next time the app runs, the call to `EnsureCreated` re-creates the database to match the new data model. The `DbInitializer` class then runs to seed the new database.

This approach to keeping the DB in sync with the data model works well until the app needs to be deployed to production. When the app is running in production, it's usually storing data that needs to be maintained. The app can't start with a test DB each time a change is made (such as adding a new column). The EF Core Migrations feature solves this problem by enabling EF Core to update the DB schema instead of creating a new database.

Rather than dropping and recreating the database when the data model changes, migrations updates the schema and retains existing data.

> **Note**
>
> **SQLite limitations**
>
> This tutorial uses the Entity Framework Core migrations feature where possible. Migrations updates the database schema to match changes in the data model. However, migrations only does the kinds of changes that the database engine supports, and SQLite's schema change capabilities are limited. For example, adding

a column is supported, but removing a column is not supported. If a migration is created to remove a column, the `ef migrations add` command succeeds but the `ef database update` command fails.

The workaround for the SQLite limitations is to manually write migrations code to perform a table rebuild when something in the table changes. The code goes in the `Up` and `Down` methods for a migration and involves:

- Creating a new table.
- Copying data from the old table to the new table.
- Dropping the old table.
- Renaming the new table.

Writing database-specific code of this type is outside the scope of this tutorial. Instead, this tutorial drops and re-creates the database whenever an attempt to apply a migration would fail. For more information, see the following resources:

- SQLite EF Core Database Provider Limitations
- Customize migration code
- Data seeding
- SQLite ALTER TABLE statement

Drop the database

Visual Studio

Use **SQL Server Object Explorer** (SSOX) to delete the database, or run the following command in the **Package Manager Console** (PMC):

PowerShell

```
Drop-Database
```

Create an initial migration

Visual Studio

Run the following commands in the PMC:

PowerShell

```
Add-Migration InitialCreate
Update-Database
```

Remove EnsureCreated

This tutorial series started by using EnsureCreated. `EnsureCreated` doesn't create a migrations history table and so can't be used with migrations. It's designed for testing or rapid prototyping where the database is dropped and re-created frequently.

From this point forward, the tutorials will use migrations.

In `Program.cs`, delete the following line:

C#

```
context.Database.EnsureCreated();
```

Run the app and verify that the database is seeded.

Up and Down methods

The EF Core `migrations add` command generated code to create the database. This migrations code is in the `Migrations\<timestamp>_InitialCreate.cs` file. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets. The `Down` method deletes them, as shown in the following example:

C#

```
using System;
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace ContosoUniversity.Migrations
{
    public partial class InitialCreate : Migration
    {
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Course",
                columns: table => new
                {
```

```
                    CourseID = table.Column<int>(nullable: false),
                    Title = table.Column<string>(nullable: true),
                    Credits = table.Column<int>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Course", x => x.CourseID);
                });

            migrationBuilder.CreateTable(
                name: "Student",
                columns: table => new
                {
                    ID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    LastName = table.Column<string>(nullable: true),
                    FirstMidName = table.Column<string>(nullable: true),
                    EnrollmentDate = table.Column<DateTime>(nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Student", x => x.ID);
                });

            migrationBuilder.CreateTable(
                name: "Enrollment",
                columns: table => new
                {
                    EnrollmentID = table.Column<int>(nullable: false)
                        .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
                    CourseID = table.Column<int>(nullable: false),
                    StudentID = table.Column<int>(nullable: false),
                    Grade = table.Column<int>(nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Enrollment", x => x.EnrollmentID);
                    table.ForeignKey(
                        name: "FK_Enrollment_Course_CourseID",
                        column: x => x.CourseID,
                        principalTable: "Course",
                        principalColumn: "CourseID",
                        onDelete: ReferentialAction.Cascade);
                    table.ForeignKey(
                        name: "FK_Enrollment_Student_StudentID",
                        column: x => x.StudentID,
                        principalTable: "Student",
                        principalColumn: "ID",
                        onDelete: ReferentialAction.Cascade);
                });

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_CourseID",
```

```
                table: "Enrollment",
                column: "CourseID");

            migrationBuilder.CreateIndex(
                name: "IX_Enrollment_StudentID",
                table: "Enrollment",
                column: "StudentID");
        }

        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Enrollment");

            migrationBuilder.DropTable(
                name: "Course");

            migrationBuilder.DropTable(
                name: "Student");
        }
    }
}
```

The preceding code is for the initial migration. The code:

- Was generated by the `migrations add InitialCreate` command.
- Is executed by the `database update` command.
- Creates a database for the data model specified by the database context class.

The migration name parameter (`InitialCreate` in the example) is used for the file name. The migration name can be any valid file name. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, a migration that added a department table might be called "AddDepartmentTable."

The migrations history table

- Use SSOX or SQLite tool to inspect the database.
- Notice the addition of an `__EFMigrationsHistory` table. The `__EFMigrationsHistory` table keeps track of which migrations have been applied to the database.
- View the data in the `__EFMigrationsHistory` table. It shows one row for the first migration.

The data model snapshot

Migrations creates a *snapshot* of the current data model in `Migrations/SchoolContextModelSnapshot.cs`. When add a migration is added, EF determines what changed by comparing the current data model to the snapshot file.

Because the snapshot file tracks the state of the data model, a migration cannot be deleted by deleting the `<timestamp>_<migrationname>.cs` file. To back out the most recent migration, use the migrations remove command. `migrations remove` deletes the migration and ensures the snapshot is correctly reset. For more information, see dotnet ef migrations remove.

See Resetting all migrations to remove all migrations.

Applying migrations in production

We recommend that production apps **not** call Database.Migrate at application startup. `Migrate` shouldn't be called from an app that is deployed to a server farm. If the app is scaled out to multiple server instances, it's hard to ensure database schema updates don't happen from multiple servers or conflict with read/write access.

Database migration should be done as part of deployment, and in a controlled way. Production database migration approaches include:

- Using migrations to create SQL scripts and using the SQL scripts in deployment.
- Running `dotnet ef database update` from a controlled environment.

Troubleshooting

If the app uses SQL Server LocalDB and displays the following exception:

```text
SqlException: Cannot open database "ContosoUniversity" requested by the
login.
The login failed.
Login failed for user 'user name'.
```

The solution may be to run `dotnet ef database update` at a command prompt.

Additional resources

- EF Core CLI.
- dotnet ef migrations CLI commands

- Package Manager Console (Visual Studio)

Next steps

The next tutorial builds out the data model, adding entity properties and new entities.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Part 5, Razor Pages with EF Core in ASP.NET Core - Data Model

Article • 04/11/2023

By Tom Dykstra, Jeremy Likness, and Jon P Smith

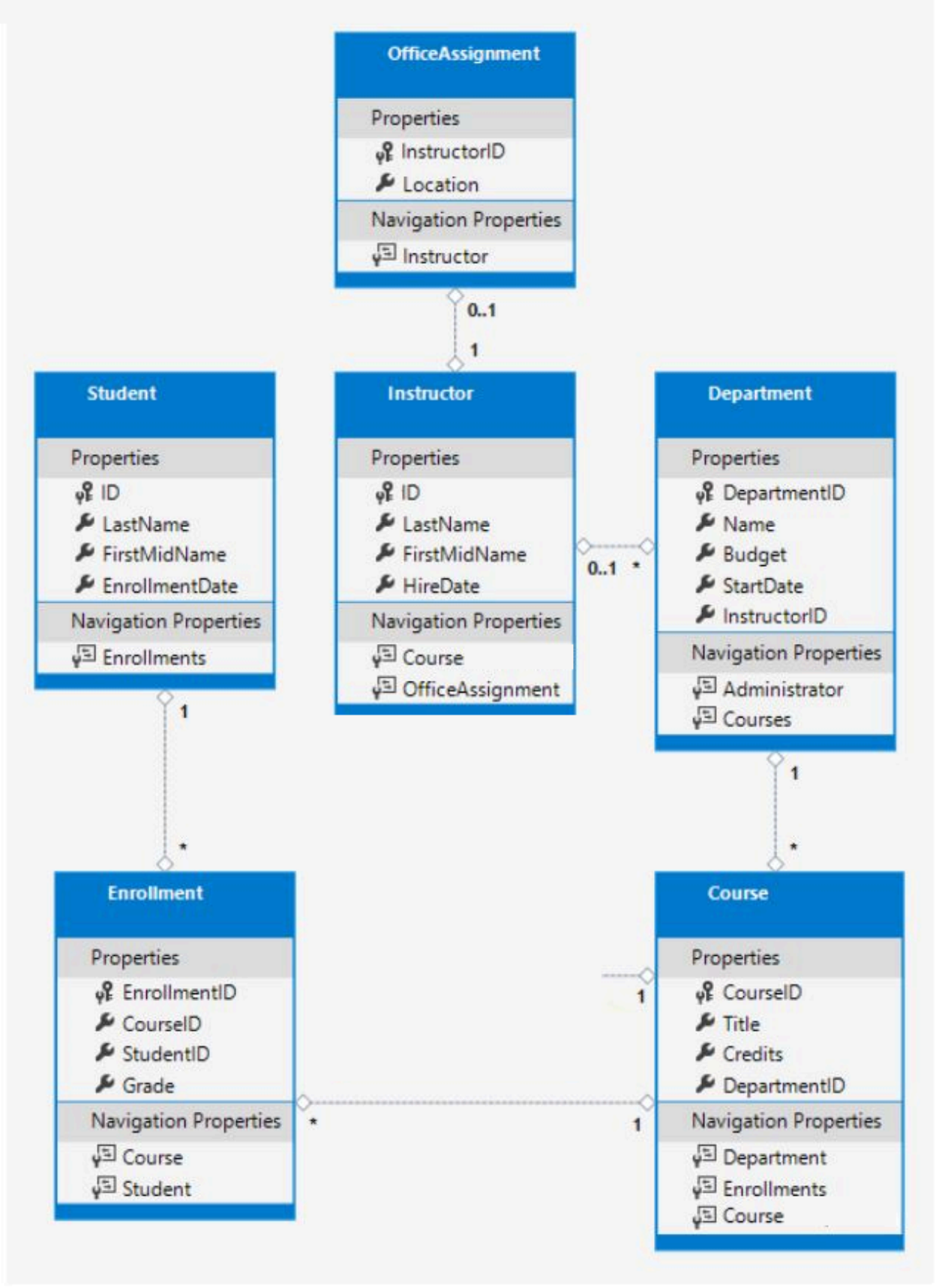
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

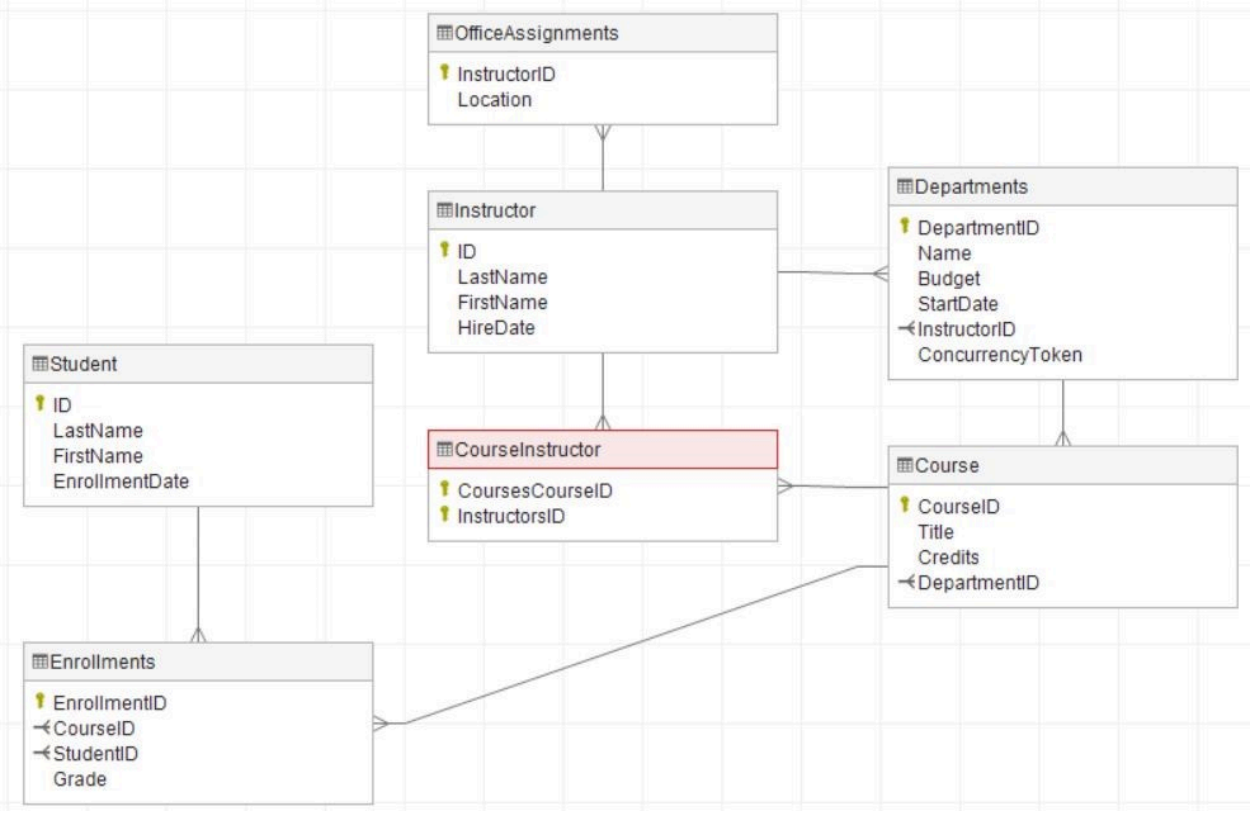
The previous tutorials worked with a basic data model that was composed of three entities. In this tutorial:

- More entities and relationships are added.
- The data model is customized by specifying formatting, validation, and database mapping rules.

The completed data model is shown in the following illustration:



The following database diagram was made with Dataedo:



To create a database diagram with Dataedo:

- Deploy the app to Azure
- Download and install Dataedo on your computer.
- Follow the instructions Generate documentation for Azure SQL Database in 5 minutes

In the preceding Dataedo diagram, the `CourseInstructor` is a join table created by Entity Framework. For more information, see Many-to-many

The Student entity

Replace the code in `Models/Student.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
```

```
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than
50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The preceding code adds a `FullName` property and adds the following attributes to existing properties:

- [DataType]
- [DisplayFormat]
- [StringLength]
- [Column]
- [Required]
- [Display]

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. `FullName` can't be set, so it has only a get accessor. No `FullName` column is created in the database.

The DataType attribute

```
C#
```

```
[DataType(DataType.Date)]
```

For student enrollment dates, all of the pages currently display the time of day along with the date, although only the date is relevant. By using data annotation attributes, you can make one code change that will fix the display format in every page that shows the data.

The DataType attribute specifies a data type that's more specific than the database intrinsic type. In this case only the date should be displayed, not the date and time. The DataType Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, etc. The `DataType` attribute can also enable the app to automatically provide type-specific features. For example:

- The `mailto:` link is automatically created for `DataType.EmailAddress`.
- The date selector is provided for `DataType.Date` in most browsers.

The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes. The `DataType` attributes don't provide validation.

The DisplayFormat attribute

C#

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
```

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the date field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format. The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied to the edit UI. Some fields shouldn't use `ApplyFormatInEditMode`. For example, the currency symbol should generally not be displayed in an edit text box.

The `DisplayFormat` attribute can be used by itself. It's generally a good idea to use the `DataType` attribute with the `DisplayFormat` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen. The `DataType` attribute provides the following benefits that are not available in `DisplayFormat`:

- The browser can enable HTML5 features. For example, show a calendar control, the locale-appropriate currency symbol, email links, and client-side input validation.
- By default, the browser renders data using the correct format based on the locale.

For more information, see the <input> Tag Helper documentation.

The StringLength attribute

C#

```
[StringLength(50, ErrorMessage = "First name cannot be longer than 50
characters.")]
```

Data validation rules and validation error messages can be specified with attributes. The StringLength attribute specifies the minimum and maximum length of characters that are allowed in a data field. The code shown limits names to no more than 50 characters. An example that sets the minimum string length is shown later.

The `StringLength` attribute also provides client-side and server-side validation. The minimum value has no impact on the database schema.

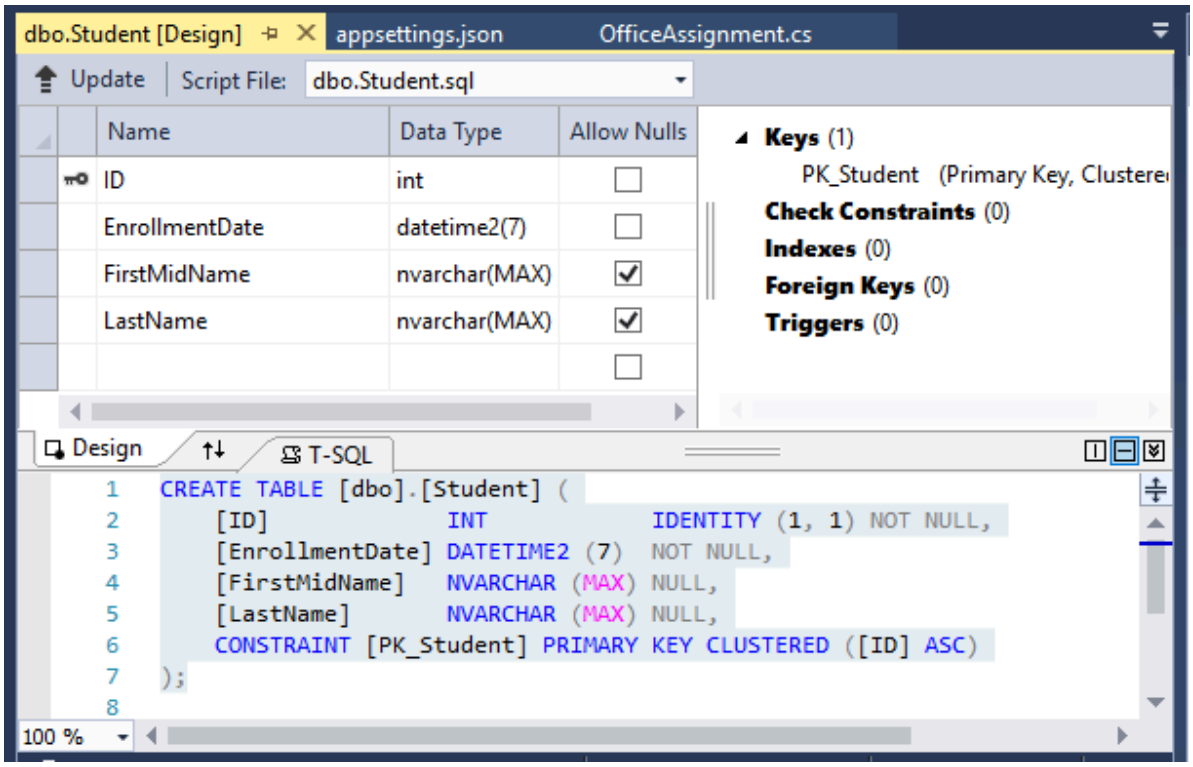
The `StringLength` attribute doesn't prevent a user from entering white space for a name. The RegularExpression attribute can be used to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

C#

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

Visual Studio

In **SQL Server Object Explorer** (SSOX), open the Student table designer by double-clicking the **Student** table.



The preceding image shows the schema for the `Student` table. The name fields have type `nvarchar(MAX)`. When a migration is created and applied later in this tutorial, the name fields become `nvarchar(50)` as a result of the string length attributes.

The Column attribute

C#

```
[Column("FirstName")]
public string FirstMidName { get; set; }
```

Attributes can control how classes and properties are mapped to the database. In the `Student` model, the `Column` attribute is used to map the name of the `FirstMidName` property to "FirstName" in the database.

When the database is created, property names on the model are used for column names (except when the `Column` attribute is used). The `Student` model uses `FirstMidName` for the first-name field because the field might also contain a middle name.

With the `[Column]` attribute, `Student.FirstMidName` in the data model maps to the `FirstName` column of the `Student` table. The addition of the `Column` attribute changes the model backing the `SchoolContext`. The model backing the `SchoolContext` no longer matches the database. That discrepancy will be resolved by adding a migration later in this tutorial.

The Required attribute

```
C#

[Required]
```

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (for example, `DateTime`, `int`, and `double`). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

```
C#

[Display(Name = "Last Name")]
[Required]
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

`MinimumLength` and `Required` allow whitespace to satisfy the validation. Use the `RegularExpression` attribute for full control over the string.

The Display attribute

```
C#

[Display(Name = "Last Name")]
```

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date." The default captions had no space dividing the words, for example "Lastname."

Create a migration

Run the app and go to the Students page. An exception is thrown. The `[Column]` attribute causes EF to expect to find a column named `FirstName`, but the column name in the database is still `FirstMidName`.

Visual Studio

The error message is similar to the following example:

```
SqlException: Invalid column name 'FirstName'.
There are pending model changes
Pending model changes are detected in the following:

SchoolContext
```

- In the PMC, enter the following commands to create a new migration and update the database:

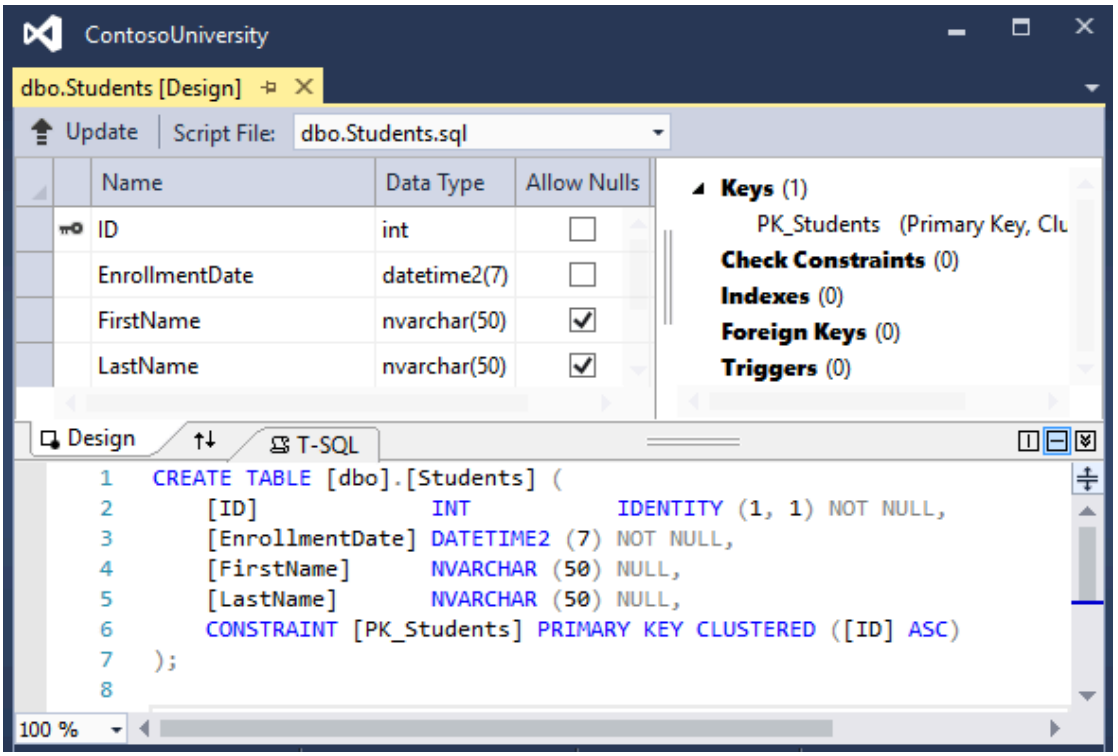
  ```PowerShell
  Add-Migration ColumnFirstName
  Update-Database
  ```

  The first of these commands generates the following warning message:

  ```text
  An operation was scaffolded that may result in the loss of data.
  Please review the migration for accuracy.
  ```

  The warning is generated because the name fields are now limited to 50 characters. If a name in the database had more than 50 characters, the 51 to last character would be lost.

- Open the Student table in SSOX:



Before the migration was applied, the name columns were of type nvarchar(MAX). The name columns are now `nvarchar(50)`. The column name has changed from `FirstMidName` to `FirstName`.

- Run the app and go to the Students page.
- Notice that times are not input or displayed along with dates.
- Select **Create New**, and try to enter a name longer than 50 characters.

> ⓘ Note
>
> In the following sections, building the app at some stages generates compiler errors. The instructions specify when to build the app.

The Instructor Entity

Create `Models/Instructor.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
```

```
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<Course> Courses { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Multiple attributes can be on one line. The `HireDate` attributes could be written as follows:

C#

```
[DataType(DataType.Date),Display(Name = "Hire
Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

Navigation properties

The `Courses` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `Courses` is defined as a collection.

C#

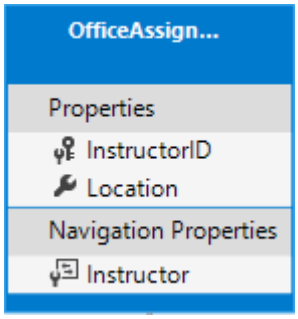
```
public ICollection<Course> Courses { get; set; }
```

An instructor can have at most one office, so the `OfficeAssignment` property holds a single `OfficeAssignment` entity. `OfficeAssignment` is null if no office is assigned.

C#

```
public OfficeAssignment OfficeAssignment { get; set; }
```

The OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }

        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

The [Key] attribute is used to identify a property as the primary key (PK) when the property name is something other than `classnameID` or `ID`.

There's a one-to-zero-or-one relationship between the `Instructor` and `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to. The `OfficeAssignment` PK is also its foreign key (FK) to the `Instructor` entity. A one-to-zero-or-one relationship occurs when a PK in one table is both a PK and a FK in another table.

EF Core can't automatically recognize `InstructorID` as the PK of `OfficeAssignment` because `InstructorID` doesn't follow the ID or classnameID naming convention. Therefore, the `Key` attribute is used to identify `InstructorID` as the PK:

C#

```
[Key]
public int InstructorID { get; set; }
```

By default, EF Core treats the key as non-database-generated because the column is for an identifying relationship. For more information, see EF Keys.

The Instructor navigation property

The `Instructor.OfficeAssignment` navigation property can be null because there might not be an `OfficeAssignment` row for a given instructor. An instructor might not have an office assignment.

The `OfficeAssignment.Instructor` navigation property will always have an instructor entity because the foreign key `InstructorID` type is `int`, a non-nullable value type. An office assignment can't exist without an instructor.

When an `Instructor` entity has a related `OfficeAssignment` entity, each entity has a reference to the other one in its navigation property.

The Course Entity

Update `Models/Course.cs` with the following code:

C#

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
```

```
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<Instructor> Instructors { get; set; }
    }
}
```

The `Course` entity has a foreign key (FK) property `DepartmentID`. `DepartmentID` points to the related `Department` entity. The `Course` entity has a `Department` navigation property.

EF Core doesn't require a foreign key property for a data model when the model has a navigation property for a related entity. EF Core automatically creates FKs in the database wherever they're needed. EF Core creates shadow properties for automatically created FKs. However, explicitly including the FK in the data model can make updates simpler and more efficient. For example, consider a model where the FK property `DepartmentID` is ***not*** included. When a course entity is fetched to edit:

- The `Department` property is `null` if it's not explicitly loaded.
- To update the course entity, the `Department` entity must first be fetched.

When the FK property `DepartmentID` is included in the data model, there's no need to fetch the `Department` entity before an update.

The DatabaseGenerated attribute

The `[DatabaseGenerated(DatabaseGeneratedOption.None)]` attribute specifies that the PK is provided by the application rather than generated by the database.

C#

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, EF Core assumes that PK values are generated by the database. Database-generated is generally the best approach. For `Course` entities, the user specifies the PK. For example, a course number such as a 1000 series for the math department, a 2000 series for the English department.

The `DatabaseGenerated` attribute can also be used to generate default values. For example, the database can automatically generate a date field to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key (FK) properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` FK and a `Department` navigation property.

C#

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

C#

```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `Instructors` navigation property is a collection:

C#

```
public ICollection<Instructor> Instructors { get; set; }
```

The Department entity

Create `Models/Department.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
                       ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Previously the `Column` attribute was used to change column name mapping. In the code for the `Department` entity, the `Column` attribute is used to change SQL data type mapping. The `Budget` column is defined using the SQL Server money type in the database:

C#

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required. EF Core chooses the appropriate SQL Server data type based on the CLR type for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. `Budget` is for currency, and the money data type is more appropriate for currency.

Foreign key and navigation properties

The FK and navigation properties reflect the following relationships:

- A department may or may not have an administrator.
- An administrator is always an instructor. Therefore the `InstructorID` property is included as the FK to the `Instructor` entity.

The navigation property is named `Administrator` but holds an `Instructor` entity:

C#

```
public int? InstructorID { get; set; }
public Instructor Administrator { get; set; }
```

The `?` in the preceding code specifies the property is nullable.

A department may have many courses, so there's a Courses navigation property:

C#

```
public ICollection<Course> Courses { get; set; }
```

By convention, EF Core enables cascade delete for non-nullable FKs and for many-to-many relationships. This default behavior can result in circular cascade delete rules. Circular cascade delete rules cause an exception when a migration is added.

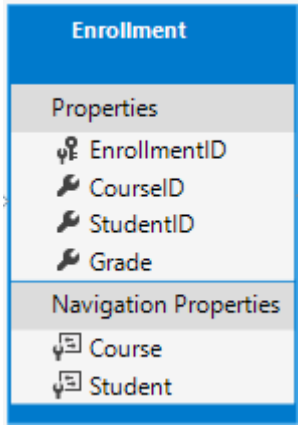
For example, if the `Department.InstructorID` property was defined as non-nullable, EF Core would configure a cascade delete rule. In that case, the department would be deleted when the instructor assigned as its administrator is deleted. In this scenario, a restrict rule would make more sense. The following fluent API would set a restrict rule and disable cascade delete.

C#

```
modelBuilder.Entity<Department>()
    .HasOne(d => d.Administrator)
    .WithMany()
    .OnDelete(DeleteBehavior.Restrict)
```

The Enrollment foreign key and navigation properties

An enrollment record is for one course taken by one student.



Update `Models/Enrollment.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The FK properties and navigation properties reflect the following relationships:

An enrollment record is for one course, so there's a `CourseID` FK property and a `Course` navigation property:

C#

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

An enrollment record is for one student, so there's a `StudentID` FK property and a `Student` navigation property:

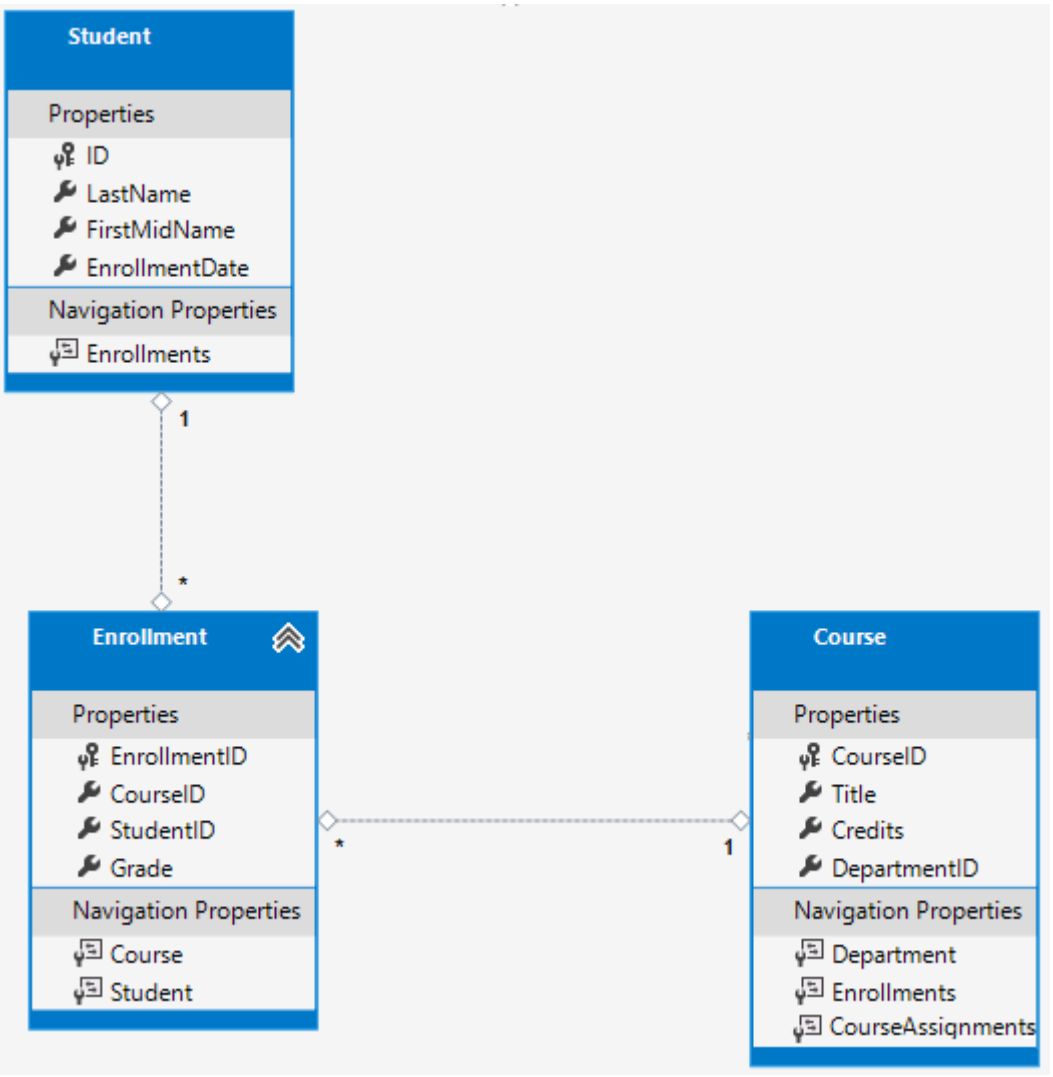
C#

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many Relationships

There's a many-to-many relationship between the `Student` and `Course` entities. The `Enrollment` entity functions as a many-to-many join table ***with payload*** in the database. ***With payload*** means that the `Enrollment` table contains additional data besides FKs for the joined tables. In the `Enrollment` entity, the additional data besides FKs are the PK and `Grade`.

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using EF Power Tools for EF 6.x. Creating the diagram isn't part of the tutorial.)



Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two FKs, `CourseID` and `StudentID`. A many-to-many join table without payload is sometimes called a pure join table (PJT).

The `Instructor` and `Course` entities have a many-to-many relationship using a PJT.

Update the database context

Update `Data/SchoolContext.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable(nameof(Course))
                .HasMany(c => c.Instructors)
                .WithMany(i => i.Courses);
            modelBuilder.Entity<Student>().ToTable(nameof(Student));
            modelBuilder.Entity<Instructor>().ToTable(nameof(Instructor));
        }
    }
}
```

The preceding code adds the new entities and configures the many-to-many relationship between the `Instructor` and `Course` entities.

Fluent API alternative to attributes

The `OnModelCreating` method in the preceding code uses the *fluent API* to configure EF Core behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement. The following code is an example of the fluent API:

```
C#
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

In this tutorial, the fluent API is used only for database mapping that can't be done with attributes. However, the fluent API can specify most of the formatting, validation, and mapping rules that can be done with attributes.

Some attributes such as `MinimumLength` can't be applied with the fluent API. `MinimumLength` doesn't change the schema, it only applies a minimum length validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes *clean*. Attributes and the fluent API can be mixed. There are some configurations that can only be done with the fluent API, for example, specifying a composite PK. There are some configurations that can only be done with attributes (`MinimumLength`). The recommended practice for using fluent API or attributes:

- Choose one of these two approaches.
- Use the chosen approach consistently as much as possible.

Some of the attributes used in this tutorial are used for:

- Validation only (for example, `MinimumLength`).
- EF Core configuration only (for example, `HasKey`).
- Validation and EF Core configuration (for example, `[StringLength(50)]`).

For more information about attributes vs. fluent API, see Methods of configuration.

Seed the database

Update the code in `Data/DbInitializer.cs`:

```
C#
```

```
using ContosoUniversity.Models;
using System;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var alexander = new Student
            {
                FirstMidName = "Carson",
                LastName = "Alexander",
                EnrollmentDate = DateTime.Parse("2016-09-01")
            };

            var alonso = new Student
            {
                FirstMidName = "Meredith",
                LastName = "Alonso",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var anand = new Student
            {
                FirstMidName = "Arturo",
                LastName = "Anand",
                EnrollmentDate = DateTime.Parse("2019-09-01")
            };

            var barzdukas = new Student
            {
                FirstMidName = "Gytis",
                LastName = "Barzdukas",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var li = new Student
            {
                FirstMidName = "Yan",
                LastName = "Li",
                EnrollmentDate = DateTime.Parse("2018-09-01")
            };

            var justice = new Student
            {
```

```
            FirstMidName = "Peggy",
            LastName = "Justice",
            EnrollmentDate = DateTime.Parse("2017-09-01")
        };

        var norman = new Student
        {
            FirstMidName = "Laura",
            LastName = "Norman",
            EnrollmentDate = DateTime.Parse("2019-09-01")
        };

        var olivetto = new Student
        {
            FirstMidName = "Nino",
            LastName = "Olivetto",
            EnrollmentDate = DateTime.Parse("2011-09-01")
        };

        var students = new Student[]
        {
            alexander,
            alonso,
            anand,
            barzdukas,
            li,
            justice,
            norman,
            olivetto
        };

        context.AddRange(students);

        var abercrombie = new Instructor
        {
            FirstMidName = "Kim",
            LastName = "Abercrombie",
            HireDate = DateTime.Parse("1995-03-11")
        };

        var fakhouri = new Instructor
        {
            FirstMidName = "Fadi",
            LastName = "Fakhouri",
            HireDate = DateTime.Parse("2002-07-06")
        };

        var harui = new Instructor
        {
            FirstMidName = "Roger",
            LastName = "Harui",
            HireDate = DateTime.Parse("1998-07-01")
        };

        var kapoor = new Instructor
```

```
            {
                FirstMidName = "Candace",
                LastName = "Kapoor",
                HireDate = DateTime.Parse("2001-01-15")
            };

            var zheng = new Instructor
            {
                FirstMidName = "Roger",
                LastName = "Zheng",
                HireDate = DateTime.Parse("2004-02-12")
            };

            var instructors = new Instructor[]
            {
                abercrombie,
                fakhouri,
                harui,
                kapoor,
                zheng
            };

            context.AddRange(instructors);

            var officeAssignments = new OfficeAssignment[]
            {
                new OfficeAssignment {
                    Instructor = fakhouri,
                    Location = "Smith 17" },
                new OfficeAssignment {
                    Instructor = harui,
                    Location = "Gowan 27" },
                new OfficeAssignment {
                    Instructor = kapoor,
                    Location = "Thompson 304" }
            };

            context.AddRange(officeAssignments);

            var english = new Department
            {
                Name = "English",
                Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = abercrombie
            };

            var mathematics = new Department
            {
                Name = "Mathematics",
                Budget = 100000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = fakhouri
            };
```

```
            var engineering = new Department
            {
                Name = "Engineering",
                Budget = 350000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = harui
            };

            var economics = new Department
            {
                Name = "Economics",
                Budget = 100000,
                StartDate = DateTime.Parse("2007-09-01"),
                Administrator = kapoor
            };

            var departments = new Department[]
            {
                english,
                mathematics,
                engineering,
                economics
            };

            context.AddRange(departments);

            var chemistry = new Course
            {
                CourseID = 1050,
                Title = "Chemistry",
                Credits = 3,
                Department = engineering,
                Instructors = new List<Instructor> { kapoor, harui }
            };

            var microeconomics = new Course
            {
                CourseID = 4022,
                Title = "Microeconomics",
                Credits = 3,
                Department = economics,
                Instructors = new List<Instructor> { zheng }
            };

            var macroeconmics = new Course
            {
                CourseID = 4041,
                Title = "Macroeconomics",
                Credits = 3,
                Department = economics,
                Instructors = new List<Instructor> { zheng }
            };

            var calculus = new Course
            {
```

```
            CourseID = 1045,
            Title = "Calculus",
            Credits = 4,
            Department = mathematics,
            Instructors = new List<Instructor> { fakhouri }
        };

        var trigonometry = new Course
        {
            CourseID = 3141,
            Title = "Trigonometry",
            Credits = 4,
            Department = mathematics,
            Instructors = new List<Instructor> { harui }
        };

        var composition = new Course
        {
            CourseID = 2021,
            Title = "Composition",
            Credits = 3,
            Department = english,
            Instructors = new List<Instructor> { abercrombie }
        };

        var literature = new Course
        {
            CourseID = 2042,
            Title = "Literature",
            Credits = 4,
            Department = english,
            Instructors = new List<Instructor> { abercrombie }
        };

        var courses = new Course[]
        {
            chemistry,
            microeconomics,
            macroeconmics,
            calculus,
            trigonometry,
            composition,
            literature
        };

        context.AddRange(courses);

        var enrollments = new Enrollment[]
        {
            new Enrollment {
                Student = alexander,
                Course = chemistry,
                Grade = Grade.A
            },
            new Enrollment {
```

```
                    Student = alexander,
                    Course = microeconomics,
                    Grade = Grade.C
                },
                new Enrollment {
                    Student = alexander,
                    Course = macroeconmics,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = alonso,
                    Course = calculus,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = alonso,
                    Course = trigonometry,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = alonso,
                    Course = composition,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = anand,
                    Course = chemistry
                },
                new Enrollment {
                    Student = anand,
                    Course = microeconomics,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = barzdukas,
                    Course = chemistry,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = li,
                    Course = composition,
                    Grade = Grade.B
                },
                new Enrollment {
                    Student = justice,
                    Course = literature,
                    Grade = Grade.B
                }
            };

            context.AddRange(enrollments);
            context.SaveChanges();
        }
    }
}
```

The preceding code provides seed data for the new entities. Most of this code creates new entity objects and loads sample data. The sample data is used for testing.

Apply the migration or drop and re-create

With the existing database, there are two approaches to changing the database:

- Drop and re-create the database. Choose this section when using SQLite.
- Apply the migration to the existing database. The instructions in this section work for SQL Server only, ***not for SQLite***.

Either choice works for SQL Server. While the apply-migration method is more complex and time-consuming, it's the preferred approach for real-world, production environments.

Drop and re-create the database

To force EF Core to create a new database, drop and update the database:

Visual Studio

- Delete the *Migrations* folder.
- In the **Package Manager Console** (PMC), run the following commands:

PowerShell

```
Drop-Database
Add-Migration InitialCreate
Update-Database
```

Run the app. Running the app runs the `DbInitializer.Initialize` method. The `DbInitializer.Initialize` populates the new database.

Visual Studio

Open the database in SSOX:

- If SSOX was opened previously, click the **Refresh** button.
- Expand the **Tables** node. The created tables are displayed.

Next steps

The next two tutorials show how to read and update related data.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 6, Razor Pages with EF Core in ASP.NET Core - Read Related Data

Article • 04/11/2023

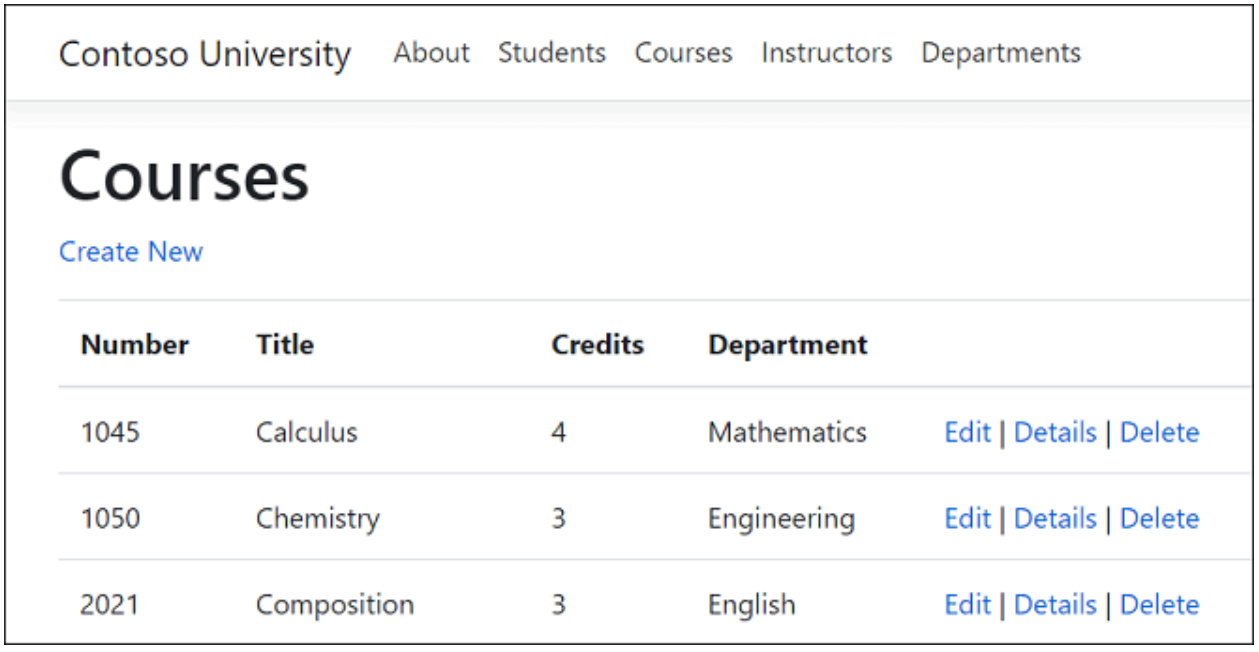
By Tom Dykstra, Jon P Smith, and Rick Anderson

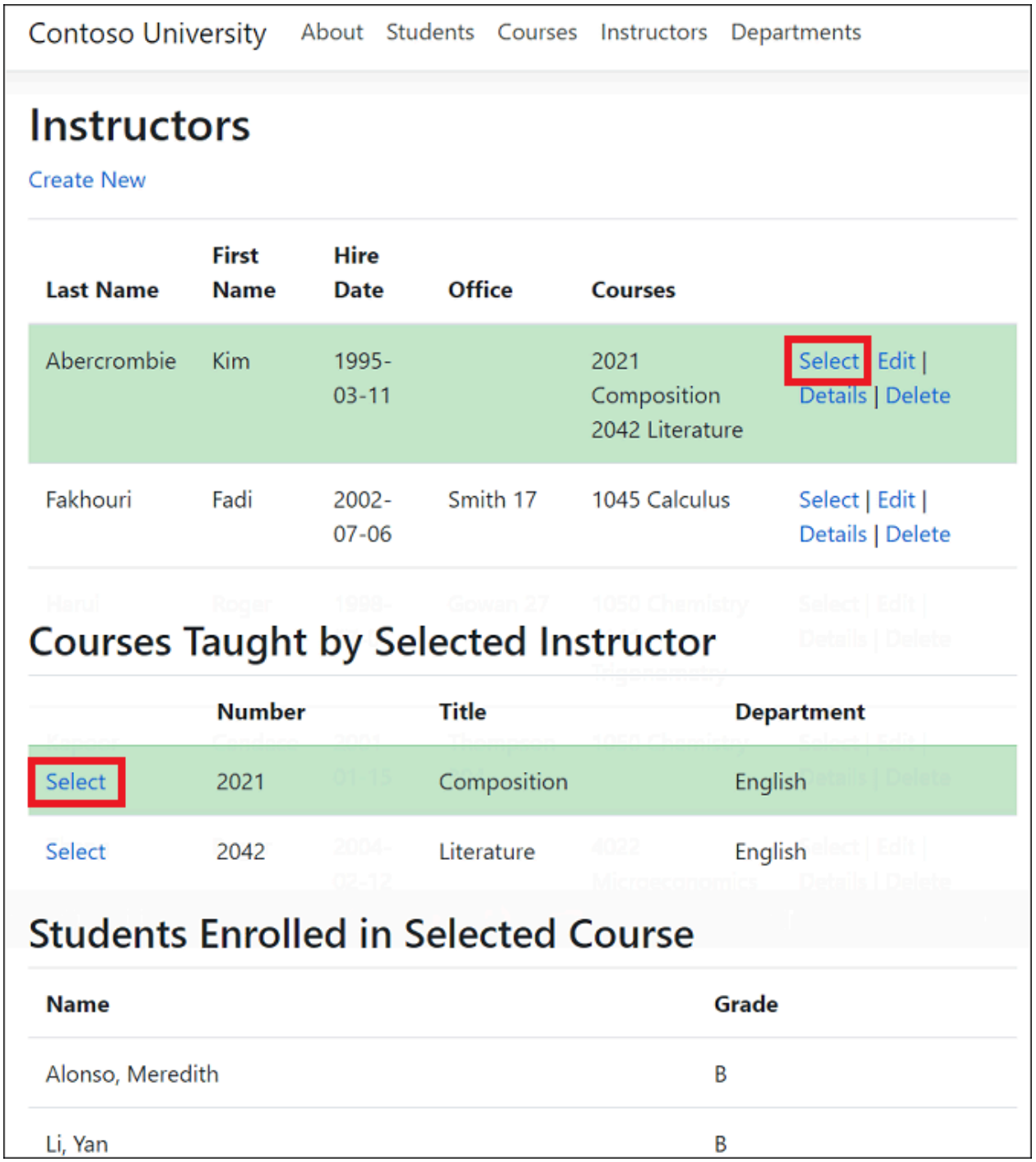
The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to read and display related data. Related data is data that EF Core loads into navigation properties.

The following illustrations show the completed pages for this tutorial:





Eager, explicit, and lazy loading

There are several ways that EF Core can load related data into the navigation properties of an entity:

- Eager loading. Eager loading is when a query for one type of entity also loads related entities. When an entity is read, its related data is retrieved. This typically results in a single join query that retrieves all of the data that's needed. EF Core will issue multiple queries for some types of eager loading. Issuing multiple queries can be more efficient than a large single query. Eager loading is specified with the Include and ThenInclude methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department entities and related Course entities

Eager loading sends multiple queries when a collection navigation is included:

- One query for the main query
- One query for each collection "edge" in the load tree.

- Separate queries with `Load`: The data can be retrieved in separate queries, and EF Core "fixes up" the navigation properties. "Fixes up" means that EF Core automatically populates the navigation properties. Separate queries with `Load` is more like explicit loading than eager loading.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

Query: Course rows related to Department d

Note: EF Core automatically fixes up navigation properties to any other entities that were previously loaded into the context instance. Even if the data for a navigation property is *not* explicitly included, the property may still be populated if some or all of the related entities were previously loaded.

- Explicit loading. When the entity is first read, related data isn't retrieved. Code must be written to retrieve the related data when it's needed. Explicit loading with separate queries results in multiple queries sent to the database. With explicit loading, the code specifies the navigation properties to be loaded. Use the `Load` method to do explicit loading. For example:

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Entry(d).Collection(p => p.Courses).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

Query: all Department rows

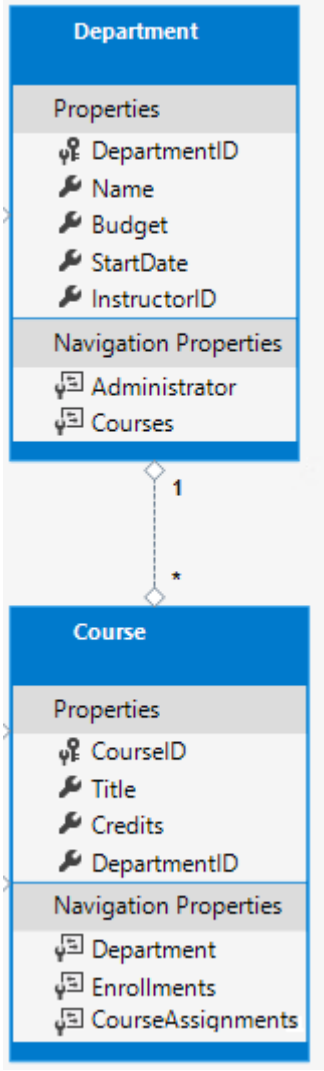
Query: Course rows related to Department d

- Lazy loading. When the entity is first read, related data isn't retrieved. The first time a navigation property is accessed, the data required for that navigation property is automatically retrieved. A query is sent to the database each time a navigation property is accessed for the first time. Lazy loading can hurt performance, for

example when developers use N+1 queries. N+1 queries load a parent and enumerate through children.

Create Course pages

The `Course` entity includes a navigation property that contains the related `Department` entity.



To display the name of the assigned department for a course:

- Load the related `Department` entity into the `Course.Department` navigation property.
- Get the name from the `Department` entity's `Name` property.

Scaffold Course pages

Visual Studio

- Follow the instructions in Scaffold Student pages with the following exceptions:
  - Create a *Pages/Courses* folder.
  - Use `Course` for the model class.
  - Use the existing context class instead of creating a new one.

- Open `Pages/Courses/Index.cshtml.cs` and examine the `OnGetAsync` method. The scaffolding engine specified eager loading for the `Department` navigation property. The `Include` method specifies eager loading.

- Run the app and select the **Courses** link. The department column displays the `DepartmentID`, which isn't useful.

Display the department name

Update Pages/Courses/Index.cshtml.cs with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IList<Course> Courses { get; set; }

        public async Task OnGetAsync()
        {
            Courses = await _context.Courses
                .Include(c => c.Department)
                .AsNoTracking()
                .ToListAsync();
        }
```

```
    }
}
```

The preceding code changes the `Course` property to `Courses` and adds `AsNoTracking`.

No-tracking queries are useful when the results are used in a read-only scenario. They're generally quicker to execute because there's no need to set up the change tracking information. If the entities retrieved from the database don't need to be updated, then a no-tracking query is likely to perform better than a tracking query.

In some cases a tracking query is more efficient than a no-tracking query. For more information, see Tracking vs. No-Tracking Queries. In the preceding code, `AsNoTracking` is called because the entities aren't updated in the current context.

Update `Pages/Courses/Index.cshtml` with the following code.

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.IndexModel

@{
    ViewData["Title"] = "Courses";
}

<h1>Courses</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Courses[0].Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Courses)
{
```

```
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.CourseID)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Title)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Credits)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Department.Name)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.CourseID">Edit</a>
|
                <a asp-page="./Details" asp-route-
id="@item.CourseID">Details</a> |
                <a asp-page="./Delete" asp-route-
id="@item.CourseID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

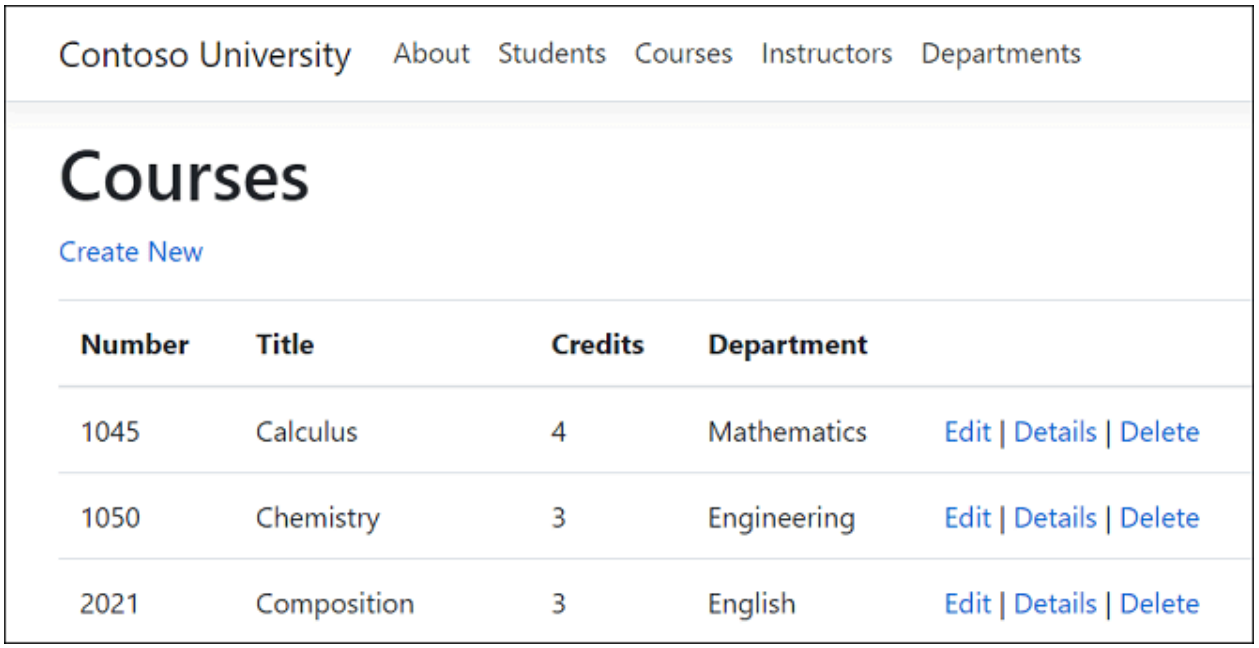
The following changes have been made to the scaffolded code:

- Changed the `Course` property name to `Courses`.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

  HTML

  ```
  @Html.DisplayFor(modelItem => item.Department.Name)
  ```

Run the app and select the **Courses** tab to see the list with department names.



Loading related data with Select

The `OnGetAsync` method loads related data with the `Include` method. The `Select` method is an alternative that loads only the related data needed. For single items, like the `Department.Name` it uses a `SQL INNER JOIN`. For collections, it uses another database access, but so does the `Include` operator on collections.

The following code loads related data with the `Select` method:

C#

```
public IList<CourseViewModel> CourseVM { get; set; }

public async Task OnGetAsync()
{
    CourseVM = await _context.Courses
    .Select(p => new CourseViewModel
    {
        CourseID = p.CourseID,
        Title = p.Title,
        Credits = p.Credits,
        DepartmentName = p.Department.Name
    }).ToListAsync();
}
```

The preceding code doesn't return any entity types, therefore no tracking is done. For more information about the EF tracking, see Tracking vs. No-Tracking Queries.

The `CourseViewModel`:

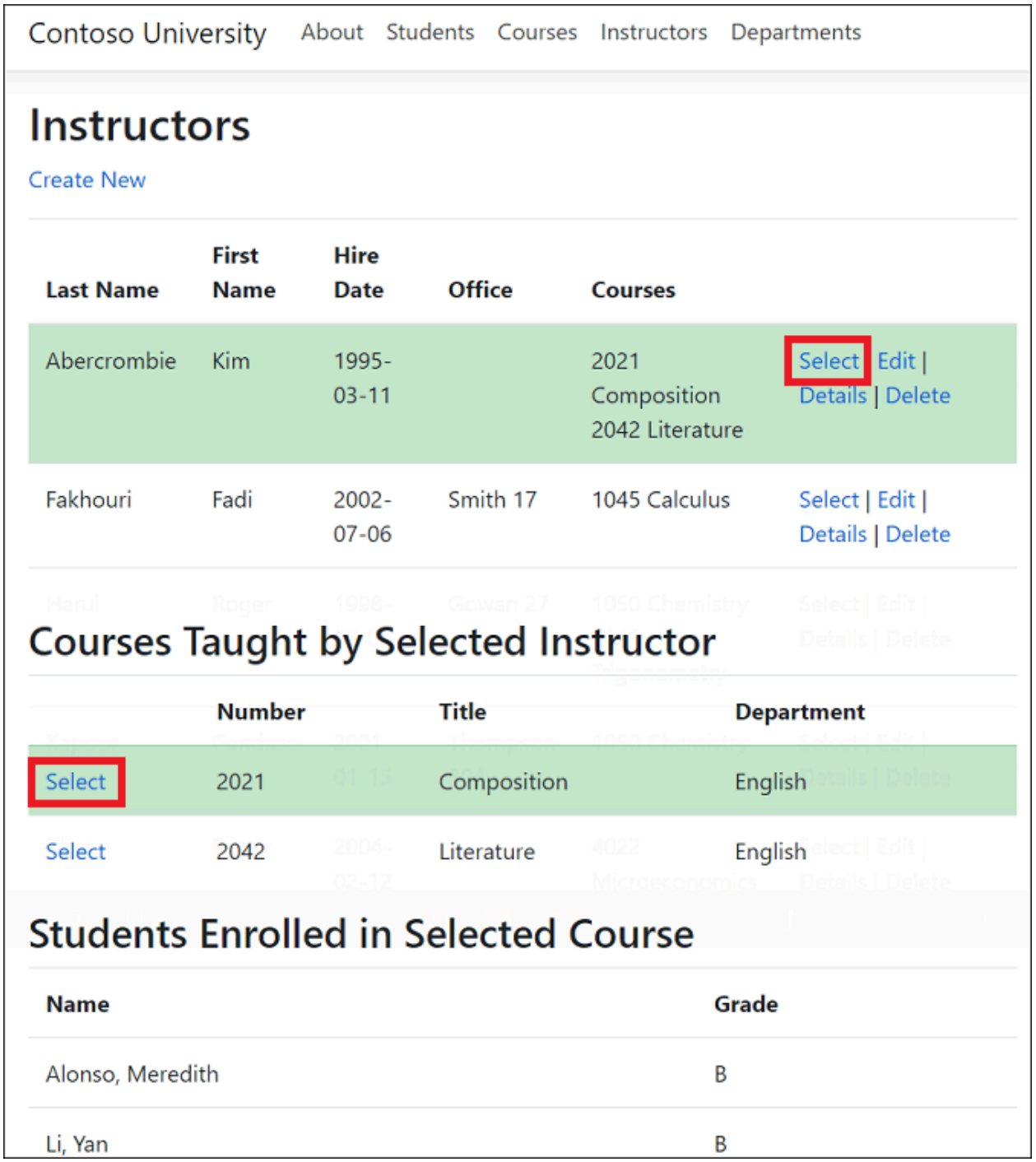
C#

```
public class CourseViewModel
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public string DepartmentName { get; set; }
}
```

See IndexSelectModel for the complete Razor Pages.

Create Instructor pages

This section scaffolds Instructor pages and adds related Courses and Enrollments to the Instructors Index page.



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity (Office in the preceding image). The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. Eager loading is used for the `OfficeAssignment` entities. Eager loading is typically more efficient when the related data needs to be displayed. In this case, office assignments for the instructors are displayed.
- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. Eager loading is used for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because only courses for the selected

instructor are needed. This example shows how to use eager loading for navigation properties in entities that are in navigation properties.

- When the user selects a course, related data from the `Enrollments` entity is displayed. In the preceding image, student name and grade are displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship.

Create a view model

The instructors page shows data from three different tables. A view model is needed that includes three properties representing the three tables.

Create `Models/SchoolViewModels/InstructorIndexData.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Scaffold Instructor pages

Visual Studio

- Follow the instructions in Scaffold the student pages with the following exceptions:
  - Create a *Pages/Instructors* folder.
  - Use `Instructor` for the model class.
  - Use the existing context class instead of creating a new one.

Run the app and navigate to the Instructors page.

Update `Pages/Instructors/Index.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;  // Add VM
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class IndexModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public IndexModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public InstructorIndexData InstructorData { get; set; }
        public int InstructorID { get; set; }
        public int CourseID { get; set; }

        public async Task OnGetAsync(int? id, int? courseID)
        {
            InstructorData = new InstructorIndexData();
            InstructorData.Instructors = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.Courses)
                    .ThenInclude(c => c.Department)
                .OrderBy(i => i.LastName)
                .ToListAsync();

            if (id != null)
            {
                InstructorID = id.Value;
                Instructor instructor = InstructorData.Instructors
                    .Where(i => i.ID == id.Value).Single();
                InstructorData.Courses = instructor.Courses;
            }

            if (courseID != null)
            {
                CourseID = courseID.Value;
                IEnumerable<Enrollment> Enrollments = await
_context.Enrollments
                    .Where(x => x.CourseID == CourseID)
                    .Include(i=>i.Student)
                    .ToListAsync();
                InstructorData.Enrollments = Enrollments;
```

```
            }
        }
    }
}
```

The `OnGetAsync` method accepts optional route data for the ID of the selected instructor.

Examine the query in the `Pages/Instructors/Index.cshtml.cs` file:

C#

```
InstructorData = new InstructorIndexData();
InstructorData.Instructors = await _context.Instructors
    .Include(i => i.OfficeAssignment)
    .Include(i => i.Courses)
        .ThenInclude(c => c.Department)
    .OrderBy(i => i.LastName)
    .ToListAsync();
```

The code specifies eager loading for the following navigation properties:

- `Instructor.OfficeAssignment`
- `Instructor.Courses`
  - `Course.Department`

The following code executes when an instructor is selected, that is, `id != null`.

C#

```
if (id != null)
{
    InstructorID = id.Value;
    Instructor instructor = InstructorData.Instructors
        .Where(i => i.ID == id.Value).Single();
    InstructorData.Courses = instructor.Courses;
}
```

The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is loaded with the `Course` entities from the selected instructor's `Courses` navigation property.

The `Where` method returns a collection. In this case, the filter select a single entity, so the `Single` method is called to convert the collection into a single `Instructor` entity. The `Instructor` entity provides access to the `Course` navigation property.

The Single method is used on a collection when the collection has only one item. The `Single` method throws an exception if the collection is empty or if there's more than one item. An alternative is SingleOrDefault, which returns a default value if the collection is empty. For this query, `null` in the default returned.

The following code populates the view model's `Enrollments` property when a course is selected:

C#

```
if (courseID != null)
{
    CourseID = courseID.Value;
    IEnumerable<Enrollment> Enrollments = await _context.Enrollments
        .Where(x => x.CourseID == CourseID)
        .Include(i=>i.Student)
        .ToListAsync();
    InstructorData.Enrollments = Enrollments;
}
```

Update the instructors Index page

Update `Pages/Instructors/Index.cshtml` with the following code.

CSHTML

```
@page "{id:int?}"
@model ContosoUniversity.Pages.Instructors.IndexModel

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
```

```
    <tbody>
        @foreach (var item in Model.InstructorData.Instructors)
        {
            string selectedRow = "";
            if (item.ID == Model.InstructorID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @{
                        foreach (var course in item.Courses)
                        {
                            @course.CourseID @:  @course.Title <br />
                        }
                    }
                </td>
                <td>
                    <a asp-page="./Index" asp-route-id="@item.ID">Select</a>
|
                    <a asp-page="./Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>

@if (Model.InstructorData.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
```

```
        </tr>

        @foreach (var item in Model.InstructorData.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == Model.CourseID)
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    <a asp-page="./Index" asp-route-
courseID="@item.CourseID">Select</a>
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}

@if (Model.InstructorData.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.InstructorData.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

The preceding code makes the following changes:

- Updates the `page` directive to `@page "{id:int?}"`. `"{id:int?}"` is a route template. The route template changes integer query strings in the URL to route data. For example, clicking on the **Select** link for an instructor with only the `@page` directive produces a URL like the following:

  `https://localhost:5001/Instructors?id=2`

  When the page directive is `@page "{id:int?}"`, the URL is: `https://localhost:5001/Instructors/2`

- Adds an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.

  HTML

  ```
  @if (item.OfficeAssignment != null)
  {
      @item.OfficeAssignment.Location
  }
  ```

- Adds a **Courses** column that displays courses taught by each instructor. See Explicit line transition for more about this razor syntax.

- Adds code that dynamically adds `class="table-success"` to the `tr` element of the selected instructor and course. This sets a background color for the selected row using a Bootstrap class.

  HTML

  ```
  string selectedRow = "";
  if (item.CourseID == Model.CourseID)
  {
      selectedRow = "table-success";
  }
  <tr class="@selectedRow">
  ```

- Adds a new hyperlink labeled **Select**. This link sends the selected instructor's ID to the `Index` method and sets a background color.

  HTML

  ```
  <a asp-action="Index" asp-route-id="@item.ID">Select</a> |
  ```

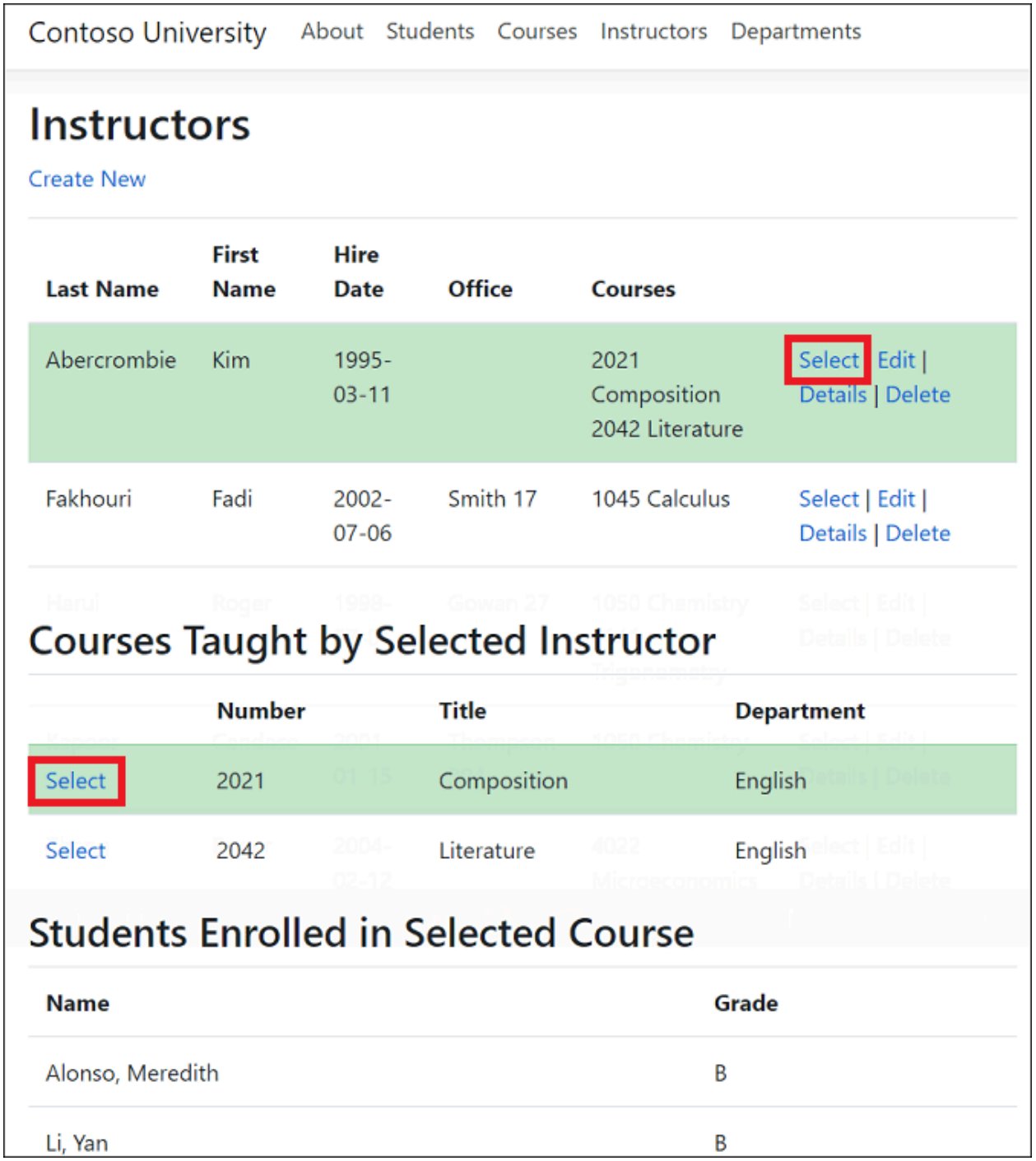
- Adds a table of courses for the selected Instructor.

- Adds a table of student enrollments for the selected course.

Run the app and select the **Instructors** tab. The page displays the `Location` (office) from the related `OfficeAssignment` entity. If `OfficeAssignment` is null, an empty table cell is displayed.

Click on the **Select** link for an instructor. The row style changes and courses assigned to that instructor are displayed.

Select a course to see the list of enrolled students and their grades.



Next steps

The next tutorial shows how to update related data.

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 7, Razor Pages with EF Core in ASP.NET Core - Update Related Data

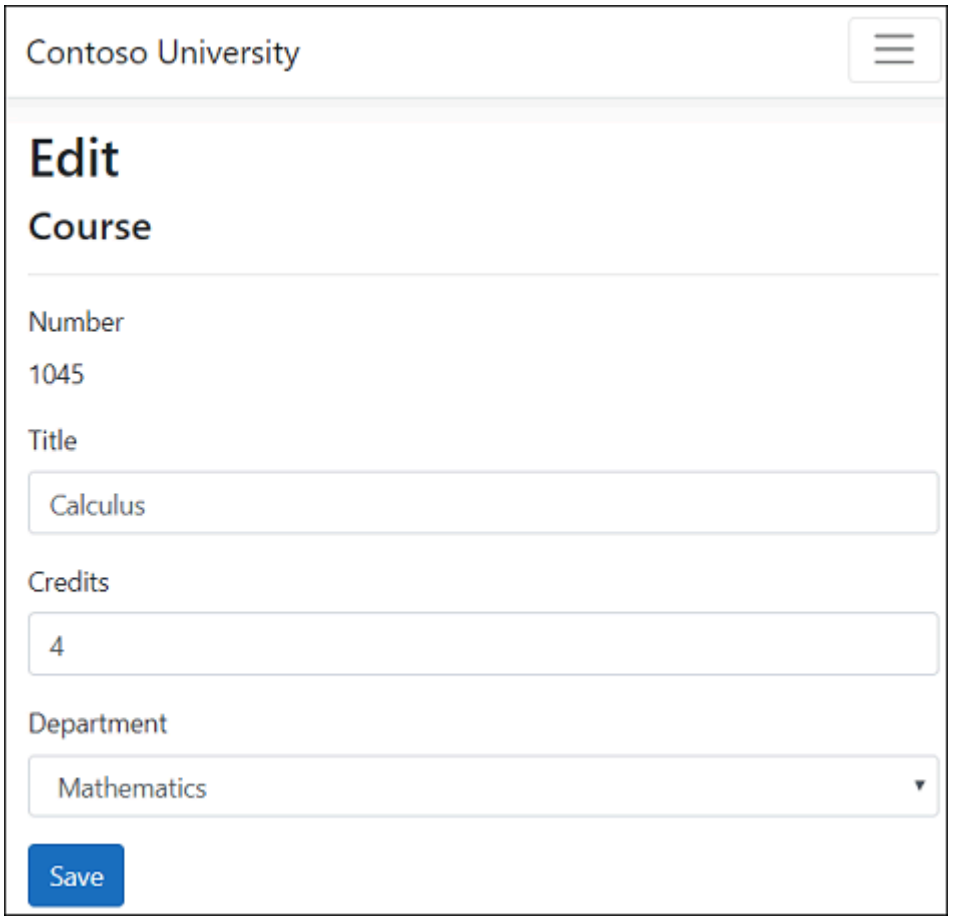
Article • 04/11/2023

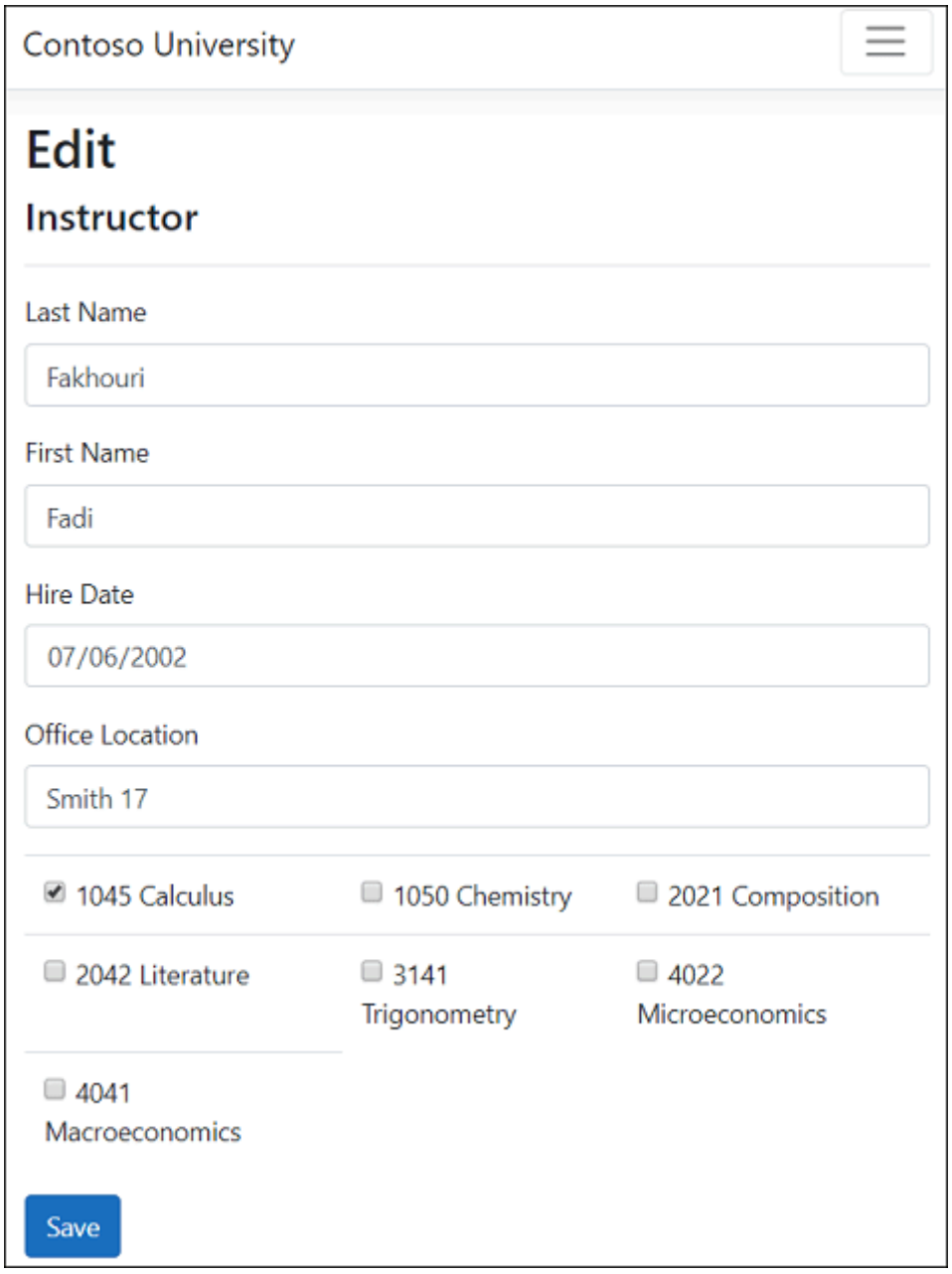
By Tom Dykstra, Jon P Smith, and Rick Anderson

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to update related data. The following illustrations show some of the completed pages.


Contoso University
Edit
Course
Number
1045
Title
Calculus
Credits
4
Department
Mathematics
Save



Update the Course Create and Edit pages

The scaffolded code for the Course Create and Edit pages has a Department drop-down list that shows `DepartmentID`, an `int`. The drop-down should show the Department name, so both of these pages need a list of department names. To provide that list, use a base class for the Create and Edit pages.

Create a base class for Course Create and Edit

Create a `Pages/Courses/DepartmentNamePageModel.cs` file with the following code:

```
C#

using ContosoUniversity.Data;
using ContosoUniversity.Models;
```

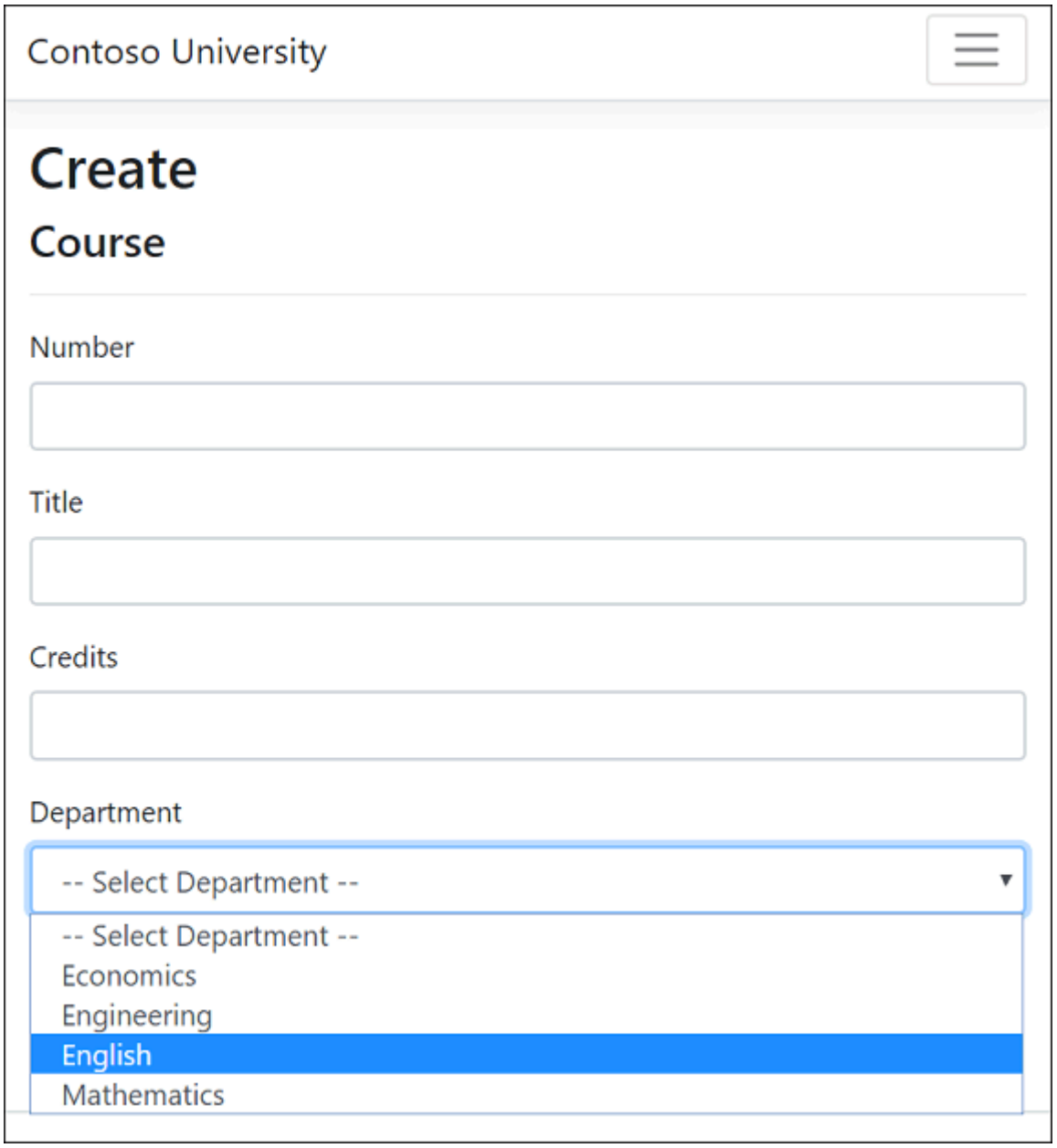
```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace ContosoUniversity.Pages.Courses
{
    public class DepartmentNamePageModel : PageModel
    {
        public SelectList DepartmentNameSL { get; set; }

        public void PopulateDepartmentsDropDownList(SchoolContext _context,
            object selectedDepartment = null)
        {
            var departmentsQuery = from d in _context.Departments
                                   orderby d.Name // Sort by name.
                                   select d;

            DepartmentNameSL = new
SelectList(departmentsQuery.AsNoTracking(),
                nameof(Department.DepartmentID),
                nameof(Department.Name),
                selectedDepartment);
        }
    }
}
```

The preceding code creates a SelectList to contain the list of department names. If `selectedDepartment` is specified, that department is selected in the `SelectList`.

The Create and Edit page model classes will derive from `DepartmentNamePageModel`.

Update the Course Create page model

A Course is assigned to a Department. The base class for the Create and Edit pages provides a `SelectList` for selecting the department. The drop-down list that uses the `SelectList` sets the `Course.DepartmentID` foreign key (FK) property. EF Core uses the `Course.DepartmentID` FK to load the `Department` navigation property.



Update `Pages/Courses/Create.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class CreateModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public CreateModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        public IActionResult OnGet()
```

```
        {
            PopulateDepartmentsDropDownList(_context);
            return Page();
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            var emptyCourse = new Course();

            if (await TryUpdateModelAsync<Course>(
                 emptyCourse,
                 "course",   // Prefix for form value.
                 s => s.CourseID, s => s.DepartmentID, s => s.Title, s =>
s.Credits))
            {
                _context.Courses.Add(emptyCourse);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context,
emptyCourse.DepartmentID);
            return Page();
        }
    }
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The preceding code:

- Derives from `DepartmentNamePageModel`.
- Uses TryUpdateModelAsync to prevent overposting.
- Removes `ViewData["DepartmentID"]`. The `DepartmentNameSL` `SelectList` is a strongly typed model and will be used by the Razor page. Strongly typed models are preferred over weakly typed. For more information, see Weakly typed data (ViewData and ViewBag).

Update the Course Create Razor page

Update `Pages/Courses/Create.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.CreateModel
@{
    ViewData["Title"] = "Create Course";
}
<h2>Create</h2>
<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label">
</label>
                <input asp-for="Course.CourseID" class="form-control" />
                <span asp-validation-for="Course.CourseID" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label">
</label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Department" class="control-label">
</label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL">
                    <option value="">-- Select Department --</option>
                </select>
                <span asp-validation-for="Course.DepartmentID" class="text-
danger" />
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
/>
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="Index">Back to List</a>
</div>
@section Scripts {
```

```
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Changes the caption from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL` (from the base class).
- Adds the "Select Department" option. This change renders "Select Department" in the drop-down when no department has been selected yet, rather than the first department.
- Adds a validation message when the department isn't selected.

The Razor Page uses the Select Tag Helper:

CSHTML

```
<div class="form-group">
    <label asp-for="Course.Department" class="control-label"></label>
    <select asp-for="Course.DepartmentID" class="form-control"
            asp-items="@Model.DepartmentNameSL">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="Course.DepartmentID" class="text-danger" />
</div>
```

Test the Create page. The Create page displays the department name rather than the department ID.

Update the Course Edit page model

Update `Pages/Courses/Edit.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Courses
{
    public class EditModel : DepartmentNamePageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
```

```
            _context = context;
        }

        [BindProperty]
        public Course Course { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Course = await _context.Courses
                .Include(c => c.Department).FirstOrDefaultAsync(m =>
m.CourseID == id);

            if (Course == null)
            {
                return NotFound();
            }

            // Select current DepartmentID.
            PopulateDepartmentsDropDownList(_context, Course.DepartmentID);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            var courseToUpdate = await _context.Courses.FindAsync(id);

            if (courseToUpdate == null)
            {
                return NotFound();
            }

            if (await TryUpdateModelAsync<Course>(
                 courseToUpdate,
                 "course",   // Prefix for form value.
                   c => c.Credits, c => c.DepartmentID, c => c.Title))
            {
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }

            // Select DepartmentID if TryUpdateModelAsync fails.
            PopulateDepartmentsDropDownList(_context,
courseToUpdate.DepartmentID);
            return Page();
        }
```

```
    }
}
```

The changes are similar to those made in the Create page model. In the preceding code, `PopulateDepartmentsDropDownList` passes in the department ID, which selects that department in the drop-down list.

Update the Course Edit Razor page

Update `Pages/Courses/Edit.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.EditModel

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Course</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Course.CourseID" />
            <div class="form-group">
                <label asp-for="Course.CourseID" class="control-label">
</label>
                <div>@Html.DisplayFor(model => model.Course.CourseID)</div>
            </div>
            <div class="form-group">
                <label asp-for="Course.Title" class="control-label"></label>
                <input asp-for="Course.Title" class="form-control" />
                <span asp-validation-for="Course.Title" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="Course.Credits" class="control-label">
</label>
                <input asp-for="Course.Credits" class="form-control" />
                <span asp-validation-for="Course.Credits" class="text-
danger"></span>
            </div>
            <div class="form-group">
```

```
                <label asp-for="Course.Department" class="control-label">
</label>
                <select asp-for="Course.DepartmentID" class="form-control"
                        asp-items="@Model.DepartmentNameSL"></select>
                <span asp-validation-for="Course.DepartmentID" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code makes the following changes:

- Displays the course ID. Generally the Primary Key (PK) of an entity isn't displayed. PKs are usually meaningless to users. In this case, the PK is the course number.
- Changes the caption for the Department drop-down from **DepartmentID** to **Department**.
- Replaces `"ViewBag.DepartmentID"` with `DepartmentNameSL`, which is in the base class.

The page contains a hidden field (`<input type="hidden">`) for the course number. Adding a `<label>` tag helper with `asp-for="Course.CourseID"` doesn't eliminate the need for the hidden field. `<input type="hidden">` is required for the course number to be included in the posted data when the user selects **Save**.

Update the Course page models

AsNoTracking can improve performance when tracking isn't required.

Update `Pages/Courses/Delete.cshtml.cs` and `Pages/Courses/Details.cshtml.cs` by adding `AsNoTracking` to the `OnGetAsync` methods:

C#

```
public async Task<IActionResult> OnGetAsync(int? id)
{
    if (id == null)
```

```
    {
        return NotFound();
    }

    Course = await _context.Courses
        .AsNoTracking()
        .Include(c => c.Department)
        .FirstOrDefaultAsync(m => m.CourseID == id);

    if (Course == null)
    {
        return NotFound();
    }
    return Page();
}
```

Update the Course Razor pages

Update `Pages/Courses/Delete.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
```

```
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Course.CourseID" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

Make the same changes to the Details page.

CSHTML

```
@page
@model ContosoUniversity.Pages.Courses.DetailsModel

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Course.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Credits)
        </dd>
        <dt class="col-sm-2">
```

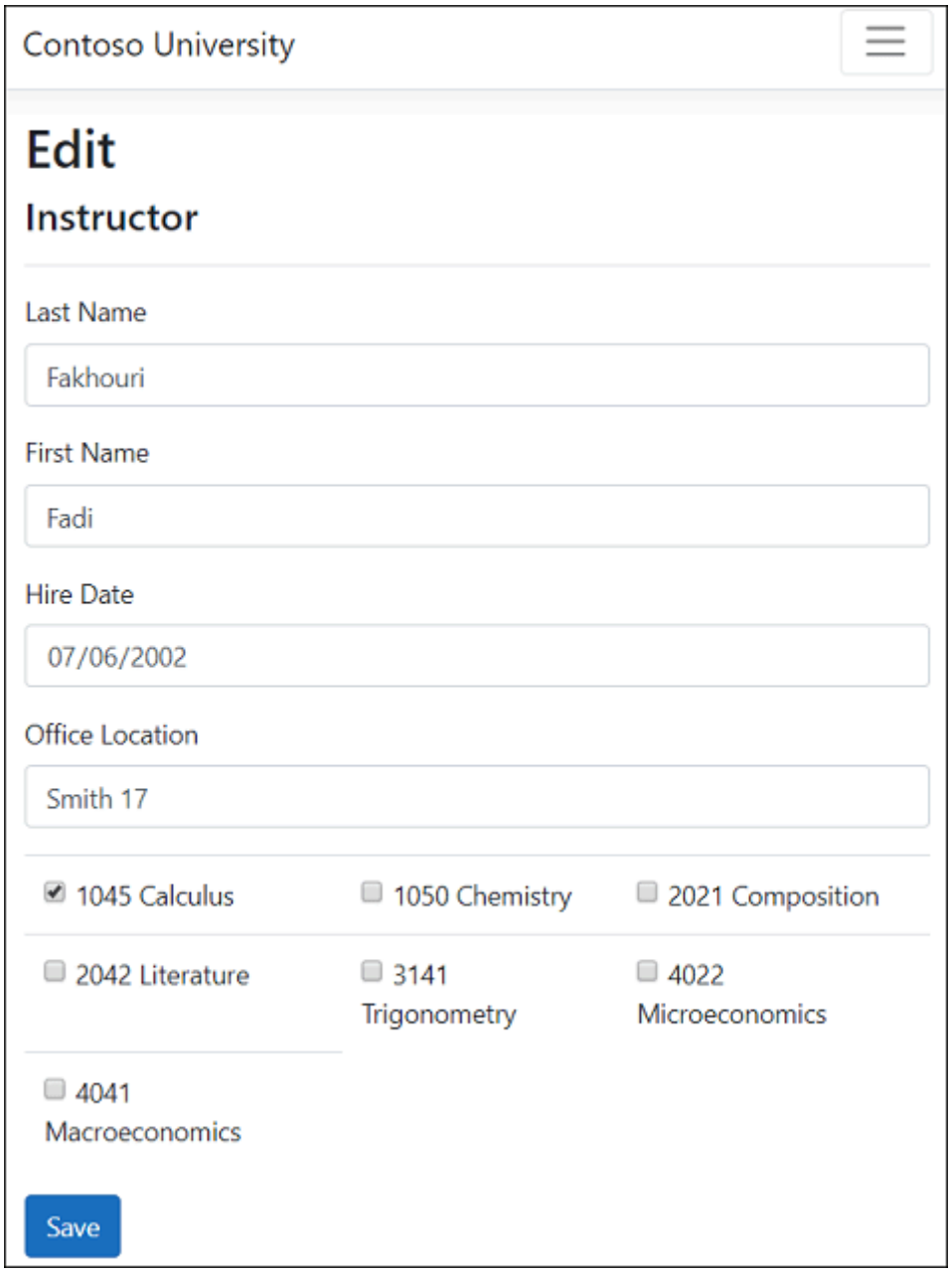
```
            @Html.DisplayNameFor(model => model.Course.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Course.Department.Name)
        </dd>
    </dl>
</div>
<div>
    <a asp-page="./Edit" asp-route-id="@Model.Course.CourseID">Edit</a> |
    <a asp-page="./Index">Back to List</a>
</div>
```

Test the Course pages

Test the create, edit, details, and delete pages.

Update the instructor Create and Edit pages

Instructors may teach any number of courses. The following image shows the instructor Edit page with an array of course checkboxes.



The checkboxes enable changes to courses an instructor is assigned to. A checkbox is displayed for every course in the database. Courses that the instructor is assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, a different UI might work better. But the method of managing a many-to-many relationship shown here wouldn't change. To create or delete relationships, you manipulate a join entity.

Create a class for assigned courses data

Create `Models/SchoolViewModels/AssignedCourseData.cs` with the following code:

C#

```
namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
```

```
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

The `AssignedCourseData` class contains data to create the checkboxes for courses assigned to an instructor.

Create an Instructor page model base class

Create the `Pages/Instructors/InstructorCoursesPageModel.cs` base class:

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Collections.Generic;
using System.Linq;

namespace ContosoUniversity.Pages.Instructors
{
    public class InstructorCoursesPageModel : PageModel
    {
        public List<AssignedCourseData> AssignedCourseDataList;

        public void PopulateAssignedCourseData(SchoolContext context,
                                               Instructor instructor)
        {
            var allCourses = context.Courses;
            var instructorCourses = new HashSet<int>(
                instructor.Courses.Select(c => c.CourseID));
            AssignedCourseDataList = new List<AssignedCourseData>();
            foreach (var course in allCourses)
            {
                AssignedCourseDataList.Add(new AssignedCourseData
                {
                    CourseID = course.CourseID,
                    Title = course.Title,
                    Assigned = instructorCourses.Contains(course.CourseID)
                });
            }
        }
    }
}
```

The `InstructorCoursesPageModel` is the base class for the Edit and Create page models. `PopulateAssignedCourseData` reads all `Course` entities to populate `AssignedCourseDataList`. For each course, the code sets the `CourseID`, title, and whether or not the instructor is assigned to the course. A HashSet is used for efficient lookups.

Handle office location

Another relationship the edit page has to handle is the one-to-zero-or-one relationship that the Instructor entity has with the `OfficeAssignment` entity. The instructor edit code must handle the following scenarios:

- If the user clears the office assignment, delete the `OfficeAssignment` entity.
- If the user enters an office assignment and it was empty, create a new `OfficeAssignment` entity.
- If the user changes the office assignment, update the `OfficeAssignment` entity.

Update the Instructor Edit page model

Update `Pages/Instructors/Edit.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class EditModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
```

```
            {
                return NotFound();
            }

            Instructor = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.Courses)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.ID == id);

            if (Instructor == null)
            {
                return NotFound();
            }
            PopulateAssignedCourseData(_context, Instructor);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id, string[]
selectedCourses)
        {
            if (id == null)
            {
                return NotFound();
            }

            var instructorToUpdate = await _context.Instructors
                .Include(i => i.OfficeAssignment)
                .Include(i => i.Courses)
                .FirstOrDefaultAsync(s => s.ID == id);

            if (instructorToUpdate == null)
            {
                return NotFound();
            }

            if (await TryUpdateModelAsync<Instructor>(
                instructorToUpdate,
                "Instructor",
                i => i.FirstMidName, i => i.LastName,
                i => i.HireDate, i => i.OfficeAssignment))
            {
                if (String.IsNullOrWhiteSpace(
                    instructorToUpdate.OfficeAssignment?.Location))
                {
                    instructorToUpdate.OfficeAssignment = null;
                }
                UpdateInstructorCourses(selectedCourses,
instructorToUpdate);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            UpdateInstructorCourses(selectedCourses, instructorToUpdate);
            PopulateAssignedCourseData(_context, instructorToUpdate);
            return Page();
```

```
        }

        public void UpdateInstructorCourses(string[] selectedCourses,
                                            Instructor instructorToUpdate)
        {
            if (selectedCourses == null)
            {
                instructorToUpdate.Courses = new List<Course>();
                return;
            }

            var selectedCoursesHS = new HashSet<string>(selectedCourses);
            var instructorCourses = new HashSet<int>
                (instructorToUpdate.Courses.Select(c => c.CourseID));
            foreach (var course in _context.Courses)
            {
                if (selectedCoursesHS.Contains(course.CourseID.ToString()))
                {
                    if (!instructorCourses.Contains(course.CourseID))
                    {
                        instructorToUpdate.Courses.Add(course);
                    }
                }
                else
                {
                    if (instructorCourses.Contains(course.CourseID))
                    {
                        var courseToRemove =
instructorToUpdate.Courses.Single(
                                                              c => c.CourseID ==
course.CourseID);
                        instructorToUpdate.Courses.Remove(courseToRemove);
                    }
                }
            }
        }
    }
}
```

The preceding code:

- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` and `Courses` navigation properties.
- Updates the retrieved `Instructor` entity with values from the model binder. TryUpdateModelAsync prevents overposting.
- If the office location is blank, sets `Instructor.OfficeAssignment` to null. When `Instructor.OfficeAssignment` is null, the related row in the `OfficeAssignment` table is deleted.
- Calls `PopulateAssignedCourseData` in `OnGetAsync` to provide information for the checkboxes using the `AssignedCourseData` view model class.

- Calls `UpdateInstructorCourses` in `OnPostAsync` to apply information from the checkboxes to the Instructor entity being edited.
- Calls `PopulateAssignedCourseData` and `UpdateInstructorCourses` in `OnPostAsync` if TryUpdateModelAsync fails. These method calls restore the assigned course data entered on the page when it is redisplayed with an error message.

Since the Razor page doesn't have a collection of Course entities, the model binder can't automatically update the `Courses` navigation property. Instead of using the model binder to update the `Courses` navigation property, that's done in the new `UpdateInstructorCourses` method. Therefore you need to exclude the `Courses` property from model binding. This doesn't require any change to the code that calls TryUpdateModelAsync because you're using the overload with declared properties and `Courses` isn't in the include list.

If no checkboxes were selected, the code in `UpdateInstructorCourses` initializes the `instructorToUpdate.Courses` with an empty collection and returns:

C#

```
if (selectedCourses == null)
{
    instructorToUpdate.Courses = new List<Course>();
    return;
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the page. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the checkbox for a course is selected but the course is ***not*** in the `Instructor.Courses` navigation property, the course is added to the collection in the navigation property.

C#

```
if (selectedCoursesHS.Contains(course.CourseID.ToString()))
{
    if (!instructorCourses.Contains(course.CourseID))
    {
        instructorToUpdate.Courses.Add(course);
    }
}
```

If the checkbox for a course is ***not*** selected, but the course is in the `Instructor.Courses` navigation property, the course is removed from the navigation property.

C#

```
else
{
    if (instructorCourses.Contains(course.CourseID))
    {
        var courseToRemove = instructorToUpdate.Courses.Single(
                                        c => c.CourseID == course.CourseID);
        instructorToUpdate.Courses.Remove(courseToRemove);
    }
}
```

Update the Instructor Edit Razor page

Update `Pages/Instructors/Edit.cshtml` with the following code:

CSHTML

```
@page
@model ContosoUniversity.Pages.Instructors.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Instructor.ID" />
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label">
</label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-
label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-
control" />
                <span asp-validation-for="Instructor.FirstMidName"
class="text-danger"></span>
            </div>
            <div class="form-group">
```

```
                <label asp-for="Instructor.HireDate" class="control-label">
</label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location"
class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location"
class="form-control" />
                <span asp-validation-
for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in
Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                               @course.CourseID @:
@course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="./Index">Back to List</a>
</div>

@section Scripts {
```

```
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code creates an HTML table that has three columns. Each column has a checkbox and a caption containing the course number and title. The checkboxes all have the same name ("selectedCourses"). Using the same name informs the model binder to treat them as a group. The value attribute of each checkbox is set to `CourseID`. When the page is posted, the model binder passes an array that consists of the `CourseID` values for only the checkboxes that are selected.

When the checkboxes are initially rendered, courses assigned to the instructor are selected.

Note: The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be more useable and efficient.

Run the app and test the updated Instructors Edit page. Change some course assignments. The changes are reflected on the Index page.

Update the Instructor Create page

Update the Instructor Create page model and with code similar to the Edit page:

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class CreateModel : InstructorCoursesPageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;
        private readonly ILogger<InstructorCoursesPageModel> _logger;

        public CreateModel(SchoolContext context,
                           ILogger<InstructorCoursesPageModel> logger)
        {
            _context = context;
            _logger = logger;
        }
```

```
        public IActionResult OnGet()
        {
            var instructor = new Instructor();
            instructor.Courses = new List<Course>();

            // Provides an empty collection for the foreach loop
            // foreach (var course in Model.AssignedCourseDataList)
            // in the Create Razor page.
            PopulateAssignedCourseData(_context, instructor);
            return Page();
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnPostAsync(string[]
selectedCourses)
        {
            var newInstructor = new Instructor();

            if (selectedCourses.Length > 0)
            {
                newInstructor.Courses = new List<Course>();
                // Load collection with one DB call.
                _context.Courses.Load();
            }

            // Add selected Courses courses to the new instructor.
            foreach (var course in selectedCourses)
            {
                var foundCourse = await
_context.Courses.FindAsync(int.Parse(course));
                if (foundCourse != null)
                {
                    newInstructor.Courses.Add(foundCourse);
                }
                else
                {
                    _logger.LogWarning("Course {course} not found", course);
                }
            }

            try
            {
                if (await TryUpdateModelAsync<Instructor>(
                                newInstructor,
                                "Instructor",
                                i => i.FirstMidName, i => i.LastName,
                                i => i.HireDate, i => i.OfficeAssignment))
                {
                    _context.Instructors.Add(newInstructor);
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
```

```
                return RedirectToPage("./Index");
            }
            catch (Exception ex)
            {
                _logger.LogError(ex.Message);
            }

            PopulateAssignedCourseData(_context, newInstructor);
            return Page();
        }
    }
}
```

The preceding code:

- Adds logging for warning and error messages.
- Calls Load, which fetches all the Courses in one database call. For small collections this is an optimization when using FindAsync. `FindAsync` returns the tracked entity without a request to the database.

```
C#

public async Task<IActionResult> OnPostAsync(string[] selectedCourses)
{
    var newInstructor = new Instructor();

    if (selectedCourses.Length > 0)
    {
        newInstructor.Courses = new List<Course>();
        // Load collection with one DB call.
        _context.Courses.Load();
    }

    // Add selected Courses courses to the new instructor.
    foreach (var course in selectedCourses)
    {
        var foundCourse = await
_context.Courses.FindAsync(int.Parse(course));
        if (foundCourse != null)
        {
            newInstructor.Courses.Add(foundCourse);
        }
        else
        {
            _logger.LogWarning("Course {course} not found", course);
        }
    }

    try
    {
        if (await TryUpdateModelAsync<Instructor>(
```

```
                        newInstructor,
                        "Instructor",
                        i => i.FirstMidName, i => i.LastName,
                        i => i.HireDate, i => i.OfficeAssignment))
            {
                _context.Instructors.Add(newInstructor);
                await _context.SaveChangesAsync();
                return RedirectToPage("./Index");
            }
            return RedirectToPage("./Index");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex.Message);
        }

        PopulateAssignedCourseData(_context, newInstructor);
        return Page();
    }
```

- `_context.Instructors.Add(newInstructor)` creates a new `Instructor` using many-to-many relationships without explicitly mapping the join table. Many-to-many was added in EF 5.0.

Test the instructor Create page.

Update the Instructor Create Razor page with code similar to the Edit page:

CSHTML

```
@page
@model ContosoUniversity.Pages.Instructors.CreateModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Instructor</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Instructor.LastName" class="control-label">
</label>
                <input asp-for="Instructor.LastName" class="form-control" />
                <span asp-validation-for="Instructor.LastName" class="text-
danger"></span>
```

```
            </div>
            <div class="form-group">
                <label asp-for="Instructor.FirstMidName" class="control-
label"></label>
                <input asp-for="Instructor.FirstMidName" class="form-
control" />
                <span asp-validation-for="Instructor.FirstMidName"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.HireDate" class="control-label">
</label>
                <input asp-for="Instructor.HireDate" class="form-control" />
                <span asp-validation-for="Instructor.HireDate" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Instructor.OfficeAssignment.Location"
class="control-label"></label>
                <input asp-for="Instructor.OfficeAssignment.Location"
class="form-control" />
                <span asp-validation-
for="Instructor.OfficeAssignment.Location" class="text-danger" />
            </div>
            <div class="form-group">
                <div class="table">
                    <table>
                        <tr>
                            @{
                                int cnt = 0;

                                foreach (var course in
Model.AssignedCourseDataList)
                                {
                                    if (cnt++ % 3 == 0)
                                    {
                                        @:</tr><tr>
                                    }
                                    @:<td>
                                        <input type="checkbox"
                                               name="selectedCourses"
                                               value="@course.CourseID"
                                               @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />

                                               @course.CourseID @:
@course.Title
                                    @:</td>
                                }
                                @:</tr>
                            }
                    </table>
                </div>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary"
```

```
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-page="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Update the Instructor Delete page

Update `Pages/Instructors/Delete.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Instructors
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Instructor Instructor { get; set; }

        public async Task<IActionResult> OnGetAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor = await _context.Instructors.FirstOrDefaultAsync(m =>
m.ID == id);
```

```
            if (Instructor == null)
            {
                return NotFound();
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int? id)
        {
            if (id == null)
            {
                return NotFound();
            }

            Instructor instructor = await _context.Instructors
                .Include(i => i.Courses)
                .SingleAsync(i => i.ID == id);

            if (instructor == null)
            {
                return RedirectToPage("./Index");
            }

            var departments = await _context.Departments
                .Where(d => d.InstructorID == id)
                .ToListAsync();
            departments.ForEach(d => d.InstructorID = null);

            _context.Instructors.Remove(instructor);

            await _context.SaveChangesAsync();
            return RedirectToPage("./Index");
        }
    }
}
```

The preceding code makes the following changes:

- Uses eager loading for the `Courses` navigation property. `Courses` must be included or they aren't deleted when the instructor is deleted. To avoid needing to read them, configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Run the app and test the Delete page.

Next steps

Previous tutorial | Next tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Part 8, Razor Pages with EF Core in ASP.NET Core - Concurrency

Article • 04/11/2023

Tom Dykstra, and Jon P Smith

The Contoso University web app demonstrates how to create Razor Pages web apps using EF Core and Visual Studio. For information about the tutorial series, see the first tutorial.

If you run into problems you can't solve, download the completed app and compare that code to what you created by following the tutorial.

This tutorial shows how to handle conflicts when multiple users update an entity concurrently.

Concurrency conflicts

A concurrency conflict occurs when:

- A user navigates to the edit page for an entity.
- Another user updates the same entity before the first user's change is written to the database.

If concurrency detection isn't enabled, whoever updates the database last overwrites the other user's changes. If this risk is acceptable, the cost of programming for concurrency might outweigh the benefit.

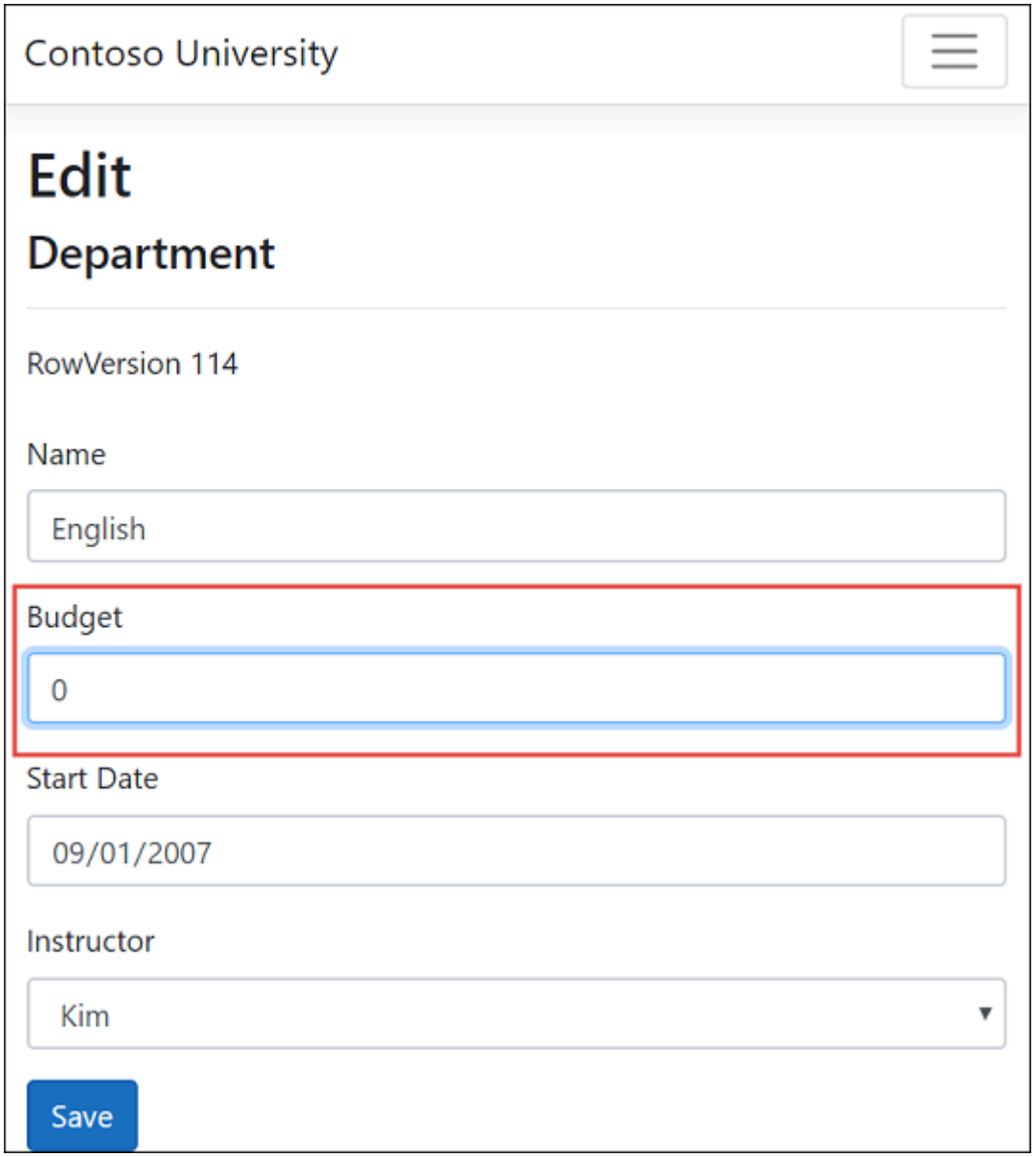
Pessimistic concurrency

One way to prevent concurrency conflicts is to use database locks. This is called pessimistic concurrency. Before the app reads a database row that it intends to update, it requests a lock. Once a row is locked for update access, no other users are allowed to lock the row until the first lock is released.

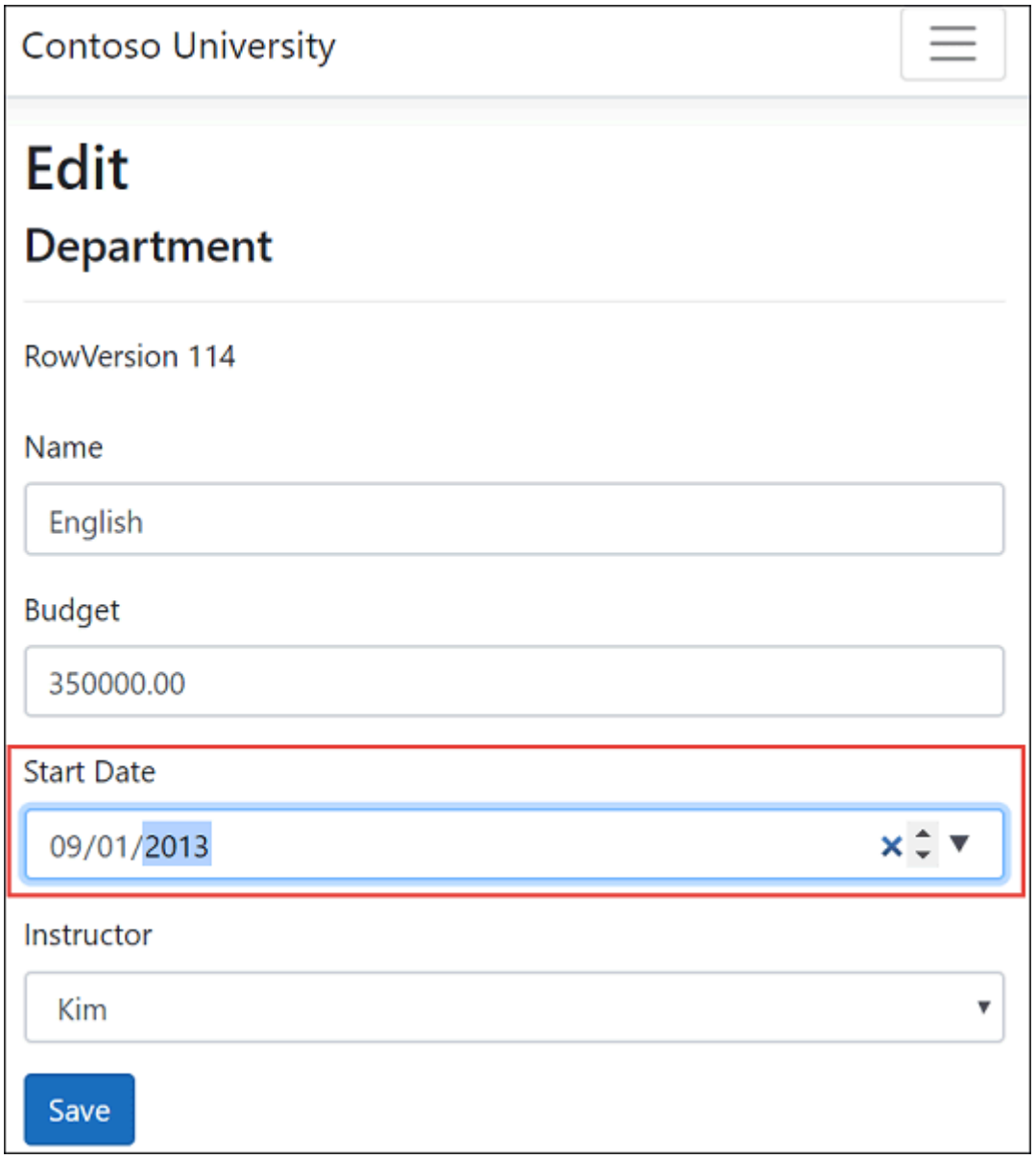
Managing locks has disadvantages. It can be complex to program and can cause performance problems as the number of users increases. Entity Framework Core provides no built-in support for pessimistic concurrency.

Optimistic concurrency

Optimistic concurrency allows concurrency conflicts to happen, and then reacts appropriately when they do. For example, Jane visits the Department edit page and changes the budget for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change take effect, since the browser displays the Index page with zero as the Budget amount.

John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts:

- Keep track of which property a user has modified and update only the corresponding columns in the database.

  In the scenario, no data would be lost. Different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes. This method of updating can reduce the number of conflicts that could result in data loss. This approach has some disadvantages:
  - Can't avoid data loss if competing changes are made to the same property.
  - Is generally not practical in a web app. It requires maintaining significant state in order to keep track of all fetched values and new values. Maintaining large amounts of state can affect app performance.

    - Can increase app complexity compared to concurrency detection on an entity.

- Let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the fetched $350,000.00 value. This approach is called a *Client Wins* or *Last in Wins* scenario. All values from the client take precedence over what's in the data store. The scaffolded code does no concurrency handling, Client Wins happens automatically.

- Prevent John's change from being updated in the database. Typically, the app would:
    - Display an error message.
    - Show the current state of the data.
    - Allow the user to reapply the changes.

  This is called a *Store Wins* scenario. The data-store values take precedence over the values submitted by the client. The Store Wins scenario is used in this tutorial. This method ensures that no changes are overwritten without a user being alerted.

Conflict detection in EF Core

Properties configured as concurrency tokens are used to implement optimistic concurrency control. When an update or delete operation is triggered by SaveChanges or SaveChangesAsync, the value of the concurrency token in the database is compared against the original value read by EF Core:

- If the values match, the operation can complete.
- If the values do not match, EF Core assumes that another user has performed a conflicting operation, aborts the current transaction, and throws a DbUpdateConcurrencyException.

Another user or process performing an operation that conflicts with the current operation is known as *concurrency conflict*.

On relational databases EF Core checks for the value of the concurrency token in the `WHERE` clause of `UPDATE` and `DELETE` statements to detect a concurrency conflict.

The data model must be configured to enable conflict detection by including a tracking column that can be used to determine when a row has been changed. EF provides two approaches for concurrency tokens:

- Applying [ConcurrencyCheck] or IsConcurrencyToken to a property on the model. This approach is not recommended. For more information, see Concurrency Tokens in EF Core.
- Applying TimestampAttribute or IsRowVersion to a concurrency token in the model. This is the approach used in this tutorial.

The SQL Server approach and SQLite implementation details are slightly different. A difference file is shown later in the tutorial listing the differences. The Visual Studio tab shows the SQL Server approach. The Visual Studio Code tab shows the approach for non-SQL Server databases, such as SQLite.

Visual Studio

- In the model, include a tracking column that is used to determine when a row has been changed.
- Apply the TimestampAttribute to the concurrency property.

Update the `Models/Department.cs` file with the following highlighted code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
                       ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
```

```
        public byte[] ConcurrencyToken { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The TimestampAttribute is what identifies the column as a concurrency tracking column. The fluent API is an alternative way to specify the tracking property:

C#

```
modelBuilder.Entity<Department>()
  .Property<byte[]>("ConcurrencyToken")
  .IsRowVersion();
```

The [Timestamp] attribute on an entity property generates the following code in the ModelBuilder method:

C#

```
 b.Property<byte[]>("ConcurrencyToken")
     .IsConcurrencyToken()
     .ValueGeneratedOnAddOrUpdate()
     .HasColumnType("rowversion");
```

The preceding code:

- Sets the property type `ConcurrencyToken` to byte array. `byte[]` is the required type for SQL Server.
- Calls IsConcurrencyToken. `IsConcurrencyToken` configures the property as a concurrency token. On updates, the concurrency token value in the database is compared to the original value to ensure it has not changed since the instance was retrieved from the database. If it has changed, a DbUpdateConcurrencyException is thrown and changes are not applied.
- Calls ValueGeneratedOnAddOrUpdate, which configures the `ConcurrencyToken` property to have a value automatically generated when adding or updating an entity.
- `HasColumnType("rowversion")` sets the column type in the SQL Server database to rowversion.

The following code shows a portion of the T-SQL generated by EF Core when the `Department` name is updated:

```
```

SQL

```
SET NOCOUNT ON;
UPDATE [Departments] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [ConcurrencyToken] = @p2;
SELECT [ConcurrencyToken]
FROM [Departments]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

The preceding highlighted code shows the `WHERE` clause containing `ConcurrencyToken`. If the database `ConcurrencyToken` doesn't equal the `ConcurrencyToken` parameter `@p2`, no rows are updated.

The following highlighted code shows the T-SQL that verifies exactly one row was updated:

SQL

```
SET NOCOUNT ON;
UPDATE [Departments] SET [Name] = @p0
WHERE [DepartmentID] = @p1 AND [ConcurrencyToken] = @p2;
SELECT [ConcurrencyToken]
FROM [Departments]
WHERE @@ROWCOUNT = 1 AND [DepartmentID] = @p1;
```

@@ROWCOUNT returns the number of rows affected by the last statement. If no rows are updated, EF Core throws a `DbUpdateConcurrencyException`.

Add a migration

Adding the `ConcurrencyToken` property changes the data model, which requires a migration.

Build the project.

Visual Studio

Run the following commands in the PMC:

PowerShell

```
Add-Migration RowVersion
Update-Database
```

The preceding commands:

- Creates the `Migrations/{time stamp}_RowVersion.cs` migration file.
- Updates the `Migrations/SchoolContextModelSnapshot.cs` file. The update adds the following code to the `BuildModel` method:

C#

```
b.Property<byte[]>("ConcurrencyToken")
    .IsConcurrencyToken()
    .ValueGeneratedOnAddOrUpdate()
    .HasColumnType("rowversion");
```

Scaffold Department pages

Visual Studio

Follow the instructions in Scaffold Student pages with the following exceptions:

- Create a *Pages/Departments* folder.
- Use `Department` for the model class.
- Use the existing context class instead of creating a new one.

Add a utility class

In the project folder, create the `Utility` class with the following code:

Visual Studio

C#

```
namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(byte[] token)
        {
            return token[7].ToString();
        }
    }
}
```

The `Utility` class provides the `GetLastChars` method used to display the last few characters of the concurrency token. The following code shows the code that works with both SQLite ad SQL Server:

C#

```
#if SQLiteVersion
using System;

namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(Guid token)
        {
            return token.ToString().Substring(
                                    token.ToString().Length - 3);
        }
    }
}
#else
namespace ContosoUniversity
{
    public static class Utility
    {
        public static string GetLastChars(byte[] token)
        {
            return token[7].ToString();
        }
    }
}
#endif
```

The `#if SQLiteVersion` preprocessor directive isolates the differences in the SQLite and SQL Server versions and helps:

- The author maintain one code base for both versions.
- SQLite developers deploy the app to Azure and use SQL Azure.

Build the project.

Update the Index page

The scaffolding tool created a `ConcurrencyToken` column for the Index page, but that field wouldn't be displayed in a production app. In this tutorial, the last portion of the `ConcurrencyToken` is displayed to help show how concurrency handling works. The last portion isn't guaranteed to be unique by itself.

Update *Pages\Departments\Index.cshtml* page:

- Replace Index with Departments.
- Change the code containing `ConcurrencyToken` to show just the last few characters.
- Replace `FirstMidName` with `FullName`.

The following code shows the updated page:

CSHTML

```
@page
@model ContosoUniversity.Pages.Departments.IndexModel

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department[0].StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model =>
model.Department[0].Administrator)
            </th>
            <th>
                Token
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Department)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
```

```
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem =>
item.Administrator.FullName)
                </td>
                <td>
                    @Utility.GetLastChars(item.ConcurrencyToken)
                </td>
                <td>
                    <a asp-page="./Edit" asp-route-
id="@item.DepartmentID">Edit</a> |
                    <a asp-page="./Details" asp-route-
id="@item.DepartmentID">Details</a> |
                    <a asp-page="./Delete" asp-route-
id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Update the Edit page model

Update `Pages/Departments/Edit.cshtml.cs` with the following code:

Visual Studio

C#

```
using ContosoUniversity.Data;
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class EditModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public EditModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
```

```
        }

        [BindProperty]
        public Department Department { get; set; }
        // Replace ViewData["InstructorID"]
        public SelectList InstructorNameSL { get; set; }

        public async Task<IActionResult> OnGetAsync(int id)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)  // eager loading
                .AsNoTracking()                 // tracking not required
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                return NotFound();
            }

            // Use strongly typed data rather than ViewData.
            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FirstMidName");

            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            // Fetch current department from DB.
            // ConcurrencyToken may have changed.
            var departmentToUpdate = await _context.Departments
                .Include(i => i.Administrator)
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (departmentToUpdate == null)
            {
                return HandleDeletedDepartment();
            }

            // Set ConcurrencyToken to value read in OnGetAsync
            _context.Entry(departmentToUpdate).Property(
                 d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;

            if (await TryUpdateModelAsync<Department>(
                departmentToUpdate,
                "Department",
                s => s.Name, s => s.StartDate, s => s.Budget, s =>
s.InstructorID))
            {
```

```
                try
                {
                    await _context.SaveChangesAsync();
                    return RedirectToPage("./Index");
                }
                catch (DbUpdateConcurrencyException ex)
                {
                    var exceptionEntry = ex.Entries.Single();
                    var clientValues =
(Department)exceptionEntry.Entity;
                    var databaseEntry =
exceptionEntry.GetDatabaseValues();
                    if (databaseEntry == null)
                    {
                        ModelState.AddModelError(string.Empty, "Unable
to save. " +
                            "The department was deleted by another
user.");
                        return Page();
                    }

                    var dbValues = (Department)databaseEntry.ToObject();
                    await SetDbErrorMessage(dbValues, clientValues,
_context);

                    // Save the current ConcurrencyToken so next
postback
                    // matches unless an new concurrency issue happens.
                    Department.ConcurrencyToken =
(byte[])dbValues.ConcurrencyToken;
                    // Clear the model error for the next postback.
                    ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
                }
            }

            InstructorNameSL = new SelectList(_context.Instructors,
                "ID", "FullName", departmentToUpdate.InstructorID);

            return Page();
        }

        private IActionResult HandleDeletedDepartment()
        {
            // ModelState contains the posted data because of the
deletion error
            // and overides the Department instance values when
displaying Page().
            ModelState.AddModelError(string.Empty,
                "Unable to save. The department was deleted by another
user.");
            InstructorNameSL = new SelectList(_context.Instructors,
"ID", "FullName", Department.InstructorID);
            return Page();
        }
```

```
        private async Task SetDbErrorMessage(Department dbValues,
                Department clientValues, SchoolContext context)
        {

            if (dbValues.Name != clientValues.Name)
            {
                ModelState.AddModelError("Department.Name",
                    $"Current value: {dbValues.Name}");
            }
            if (dbValues.Budget != clientValues.Budget)
            {
                ModelState.AddModelError("Department.Budget",
                    $"Current value: {dbValues.Budget:c}");
            }
            if (dbValues.StartDate != clientValues.StartDate)
            {
                ModelState.AddModelError("Department.StartDate",
                    $"Current value: {dbValues.StartDate:d}");
            }
            if (dbValues.InstructorID != clientValues.InstructorID)
            {
                Instructor dbInstructor = await _context.Instructors
                   .FindAsync(dbValues.InstructorID);
                ModelState.AddModelError("Department.InstructorID",
                    $"Current value: {dbInstructor?.FullName}");
            }

            ModelState.AddModelError(string.Empty,
                "The record you attempted to edit "
              + "was modified by another user after you. The "
              + "edit operation was canceled and the current values in
the database "
              + "have been displayed. If you still want to edit this
record, click "
              + "the Save button again.");
        }
    }
}
```

The concurrency updates

OriginalValue is updated with the `ConcurrencyToken` value from the entity when it was fetched in the `OnGetAsync` method. EF Core generates a `SQL UPDATE` command with a `WHERE` clause containing the original `ConcurrencyToken` value. If no rows are affected by the `UPDATE` command, a `DbUpdateConcurrencyException` exception is thrown. No rows are affected by the `UPDATE` command when no rows have the original `ConcurrencyToken` value.

Visual Studio

C#

```
public async Task<IActionResult> OnPostAsync(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // Fetch current department from DB.
    // ConcurrencyToken may have changed.
    var departmentToUpdate = await _context.Departments
        .Include(i => i.Administrator)
        .FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
        return HandleDeletedDepartment();
    }

    // Set ConcurrencyToken to value read in OnGetAsync
    _context.Entry(departmentToUpdate).Property(
         d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;
```

In the preceding highlighted code:

- The value in `Department.ConcurrencyToken` is the value when the entity was fetched in the `Get` request for the `Edit` page. The value is provided to the `OnPost` method by a hidden field in the Razor page that displays the entity to be edited. The hidden field value is copied to `Department.ConcurrencyToken` by the model binder.
- `OriginalValue` is what EF Core uses in the `WHERE` clause. Before the highlighted line of code executes:
  - `OriginalValue` has the value that was in the database when `FirstOrDefaultAsync` was called in this method.
  - This value might be different from what was displayed on the Edit page.
- The highlighted code makes sure that EF Core uses the original `ConcurrencyToken` value from the displayed `Department` entity in the SQL `UPDATE` statement's `WHERE` clause.

The following code shows the `Department` model. `Department` is initialized in the:

- `OnGetAsync` method by the EF query.

- `OnPostAsync` method by the hidden field in the Razor page using model binding:

Visual Studio

C#

```
public class EditModel : PageModel
{
    private readonly ContosoUniversity.Data.SchoolContext _context;

    public EditModel(ContosoUniversity.Data.SchoolContext context)
    {
        _context = context;
    }

    [BindProperty]
    public Department Department { get; set; }
    // Replace ViewData["InstructorID"]
    public SelectList InstructorNameSL { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Department = await _context.Departments
            .Include(d => d.Administrator)  // eager loading
            .AsNoTracking()                 // tracking not required
            .FirstOrDefaultAsync(m => m.DepartmentID == id);

        if (Department == null)
        {
            return NotFound();
        }

        // Use strongly typed data rather than ViewData.
        InstructorNameSL = new SelectList(_context.Instructors,
            "ID", "FirstMidName");

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        // Fetch current department from DB.
        // ConcurrencyToken may have changed.
        var departmentToUpdate = await _context.Departments
            .Include(i => i.Administrator)
            .FirstOrDefaultAsync(m => m.DepartmentID == id);

        if (departmentToUpdate == null)
```

```
            {
                return HandleDeletedDepartment();
            }

            // Set ConcurrencyToken to value read in OnGetAsync
            _context.Entry(departmentToUpdate).Property(
                 d => d.ConcurrencyToken).OriginalValue =
    Department.ConcurrencyToken;
```

The preceding code shows the `ConcurrencyToken` value of the `Department` entity from the `HTTP POST` request is set to the `ConcurrencyToken` value from the `HTTP GET` request.

When a concurrency error happens, the following highlighted code gets the client values (the values posted to this method) and the database values.

C#

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await SetDbErrorMessage(dbValues, clientValues, _context);

        // Save the current ConcurrencyToken so next postback
        // matches unless an new concurrency issue happens.
        Department.ConcurrencyToken = dbValues.ConcurrencyToken;
        // Clear the model error for the next postback.
        ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
    }
```

The following code adds a custom error message for each column that has database values different from what was posted to `OnPostAsync`:

C#

```
private async Task SetDbErrorMessage(Department dbValues,
        Department clientValues, SchoolContext context)
{

    if (dbValues.Name != clientValues.Name)
    {
        ModelState.AddModelError("Department.Name",
            $"Current value: {dbValues.Name}");
    }
    if (dbValues.Budget != clientValues.Budget)
    {
        ModelState.AddModelError("Department.Budget",
            $"Current value: {dbValues.Budget:c}");
    }
    if (dbValues.StartDate != clientValues.StartDate)
    {
        ModelState.AddModelError("Department.StartDate",
            $"Current value: {dbValues.StartDate:d}");
    }
    if (dbValues.InstructorID != clientValues.InstructorID)
    {
        Instructor dbInstructor = await _context.Instructors
           .FindAsync(dbValues.InstructorID);
        ModelState.AddModelError("Department.InstructorID",
            $"Current value: {dbInstructor?.FullName}");
    }

    ModelState.AddModelError(string.Empty,
        "The record you attempted to edit "
      + "was modified by another user after you. The "
      + "edit operation was canceled and the current values in the database
"
      + "have been displayed. If you still want to edit this record, click "
      + "the Save button again.");
}
```

The following highlighted code sets the `ConcurrencyToken` value to the new value retrieved from the database. The next time the user clicks **Save**, only concurrency errors that happen since the last display of the Edit page will be caught.

C#

```
if (await TryUpdateModelAsync<Department>(
    departmentToUpdate,
    "Department",
    s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
```

```
{
    try
    {
        await _context.SaveChangesAsync();
        return RedirectToPage("./Index");
    }
    catch (DbUpdateConcurrencyException ex)
    {
        var exceptionEntry = ex.Entries.Single();
        var clientValues = (Department)exceptionEntry.Entity;
        var databaseEntry = exceptionEntry.GetDatabaseValues();
        if (databaseEntry == null)
        {
            ModelState.AddModelError(string.Empty, "Unable to save. " +
                "The department was deleted by another user.");
            return Page();
        }

        var dbValues = (Department)databaseEntry.ToObject();
        await SetDbErrorMessage(dbValues, clientValues, _context);

        // Save the current ConcurrencyToken so next postback
        // matches unless an new concurrency issue happens.
        Department.ConcurrencyToken = dbValues.ConcurrencyToken;
        // Clear the model error for the next postback.
        ModelState.Remove($"{nameof(Department)}.
{nameof(Department.ConcurrencyToken)}");
    }
```

The ModelState.Remove statement is required because `ModelState` has the previous `ConcurrencyToken` value. In the Razor Page, the `ModelState` value for a field takes precedence over the model property values when both are present.

SQL Server vs SQLite code differences

The following shows the differences between the SQL Server and SQLite versions:

diff

```
+ using System;    // For GUID on SQLite

+ departmentToUpdate.ConcurrencyToken = Guid.NewGuid();

 _context.Entry(departmentToUpdate)
    .Property(d => d.ConcurrencyToken).OriginalValue =
Department.ConcurrencyToken;

- Department.ConcurrencyToken = (byte[])dbValues.ConcurrencyToken;
+ Department.ConcurrencyToken = dbValues.ConcurrencyToken;
```

Update the Edit Razor page

Update `Pages/Departments/Edit.cshtml` with the following code:

CSHTML

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.EditModel
@{
    ViewData["Title"] = "Edit";
}
<h2>Edit</h2>
<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="Department.DepartmentID" />
            <input type="hidden" asp-for="Department.ConcurrencyToken" />
            <div class="form-group">
                <label>Version</label>
                @Utility.GetLastChars(Model.Department.ConcurrencyToken)
            </div>
            <div class="form-group">
                <label asp-for="Department.Name" class="control-label">
</label>
                <input asp-for="Department.Name" class="form-control" />
                <span asp-validation-for="Department.Name" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.Budget" class="control-label">
</label>
                <input asp-for="Department.Budget" class="form-control" />
                <span asp-validation-for="Department.Budget" class="text-
danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Department.StartDate" class="control-label">
</label>
                <input asp-for="Department.StartDate" class="form-control"
/>
                <span asp-validation-for="Department.StartDate" class="text-
danger">
                </span>
            </div>
            <div class="form-group">
                <label class="control-label">Instructor</label>
```

```
                <select asp-for="Department.InstructorID" class="form-
control"
                        asp-items="@Model.InstructorNameSL"></select>
                <span asp-validation-for="Department.InstructorID"
class="text-danger">
                </span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
<div>
    <a asp-page="./Index">Back to List</a>
</div>
@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

The preceding code:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds a hidden row version. `ConcurrencyToken` must be added so postback binds the value.
- Displays the last byte of `ConcurrencyToken` for debugging purposes.
- Replaces `ViewData` with the strongly-typed `InstructorNameSL`.

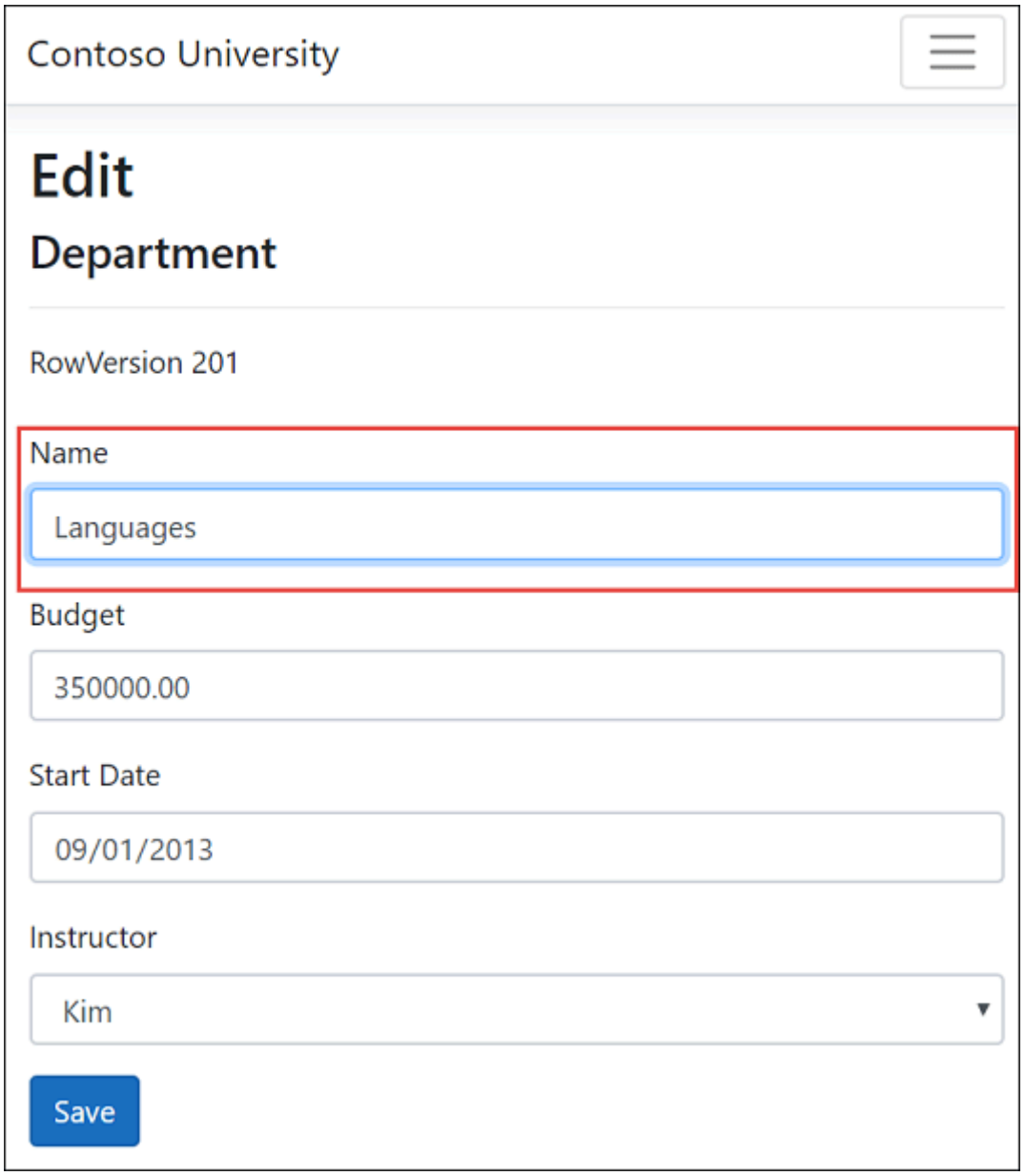
Test concurrency conflicts with the Edit page

Open two browsers instances of Edit on the English department:

- Run the app and select Departments.
- Right-click the **Edit** hyperlink for the English department and select **Open in new tab**.
- In the first tab, click the **Edit** hyperlink for the English department.

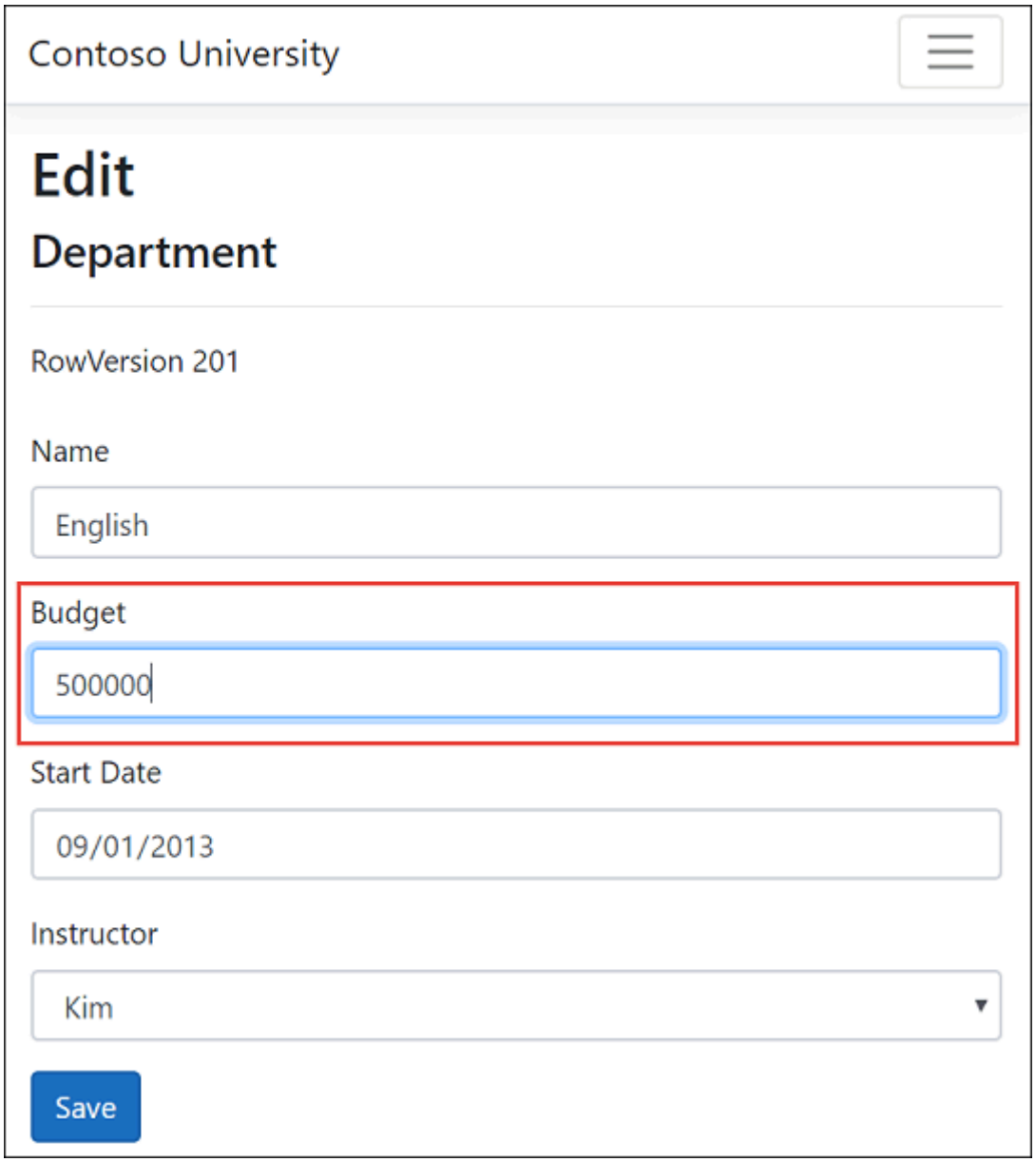
The two browser tabs display the same information.

Change the name in the first browser tab and click **Save**.

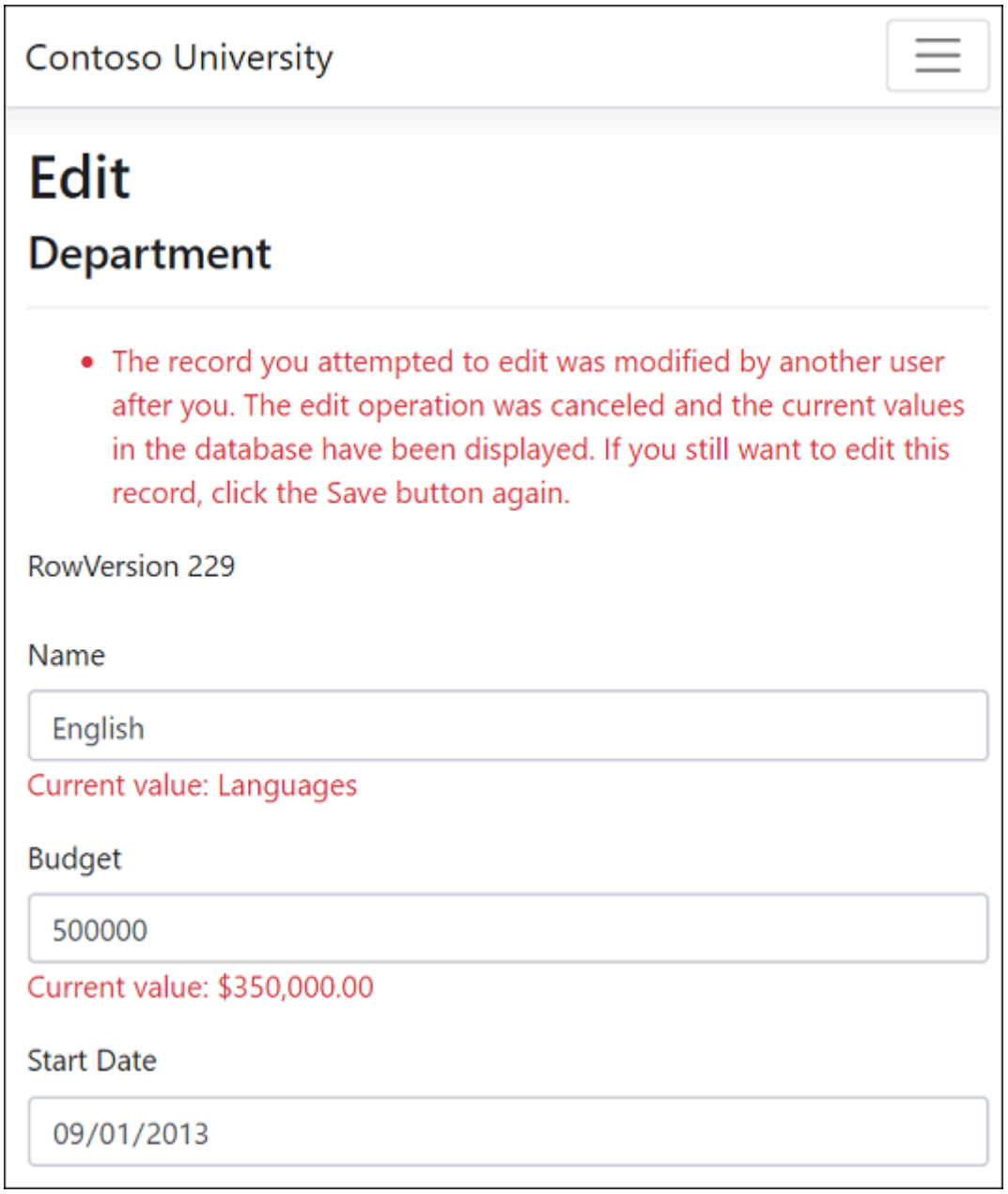


The browser shows the Index page with the changed value and updated `ConcurrencyToken` indicator. Note the updated `ConcurrencyToken` indicator, it's displayed on the second postback in the other tab.

Change a different field in the second browser tab.



Click **Save**. You see error messages for all fields that don't match the database values:



This browser window didn't intend to change the Name field. Copy and paste the current value (Languages) into the Name field. Tab out. Client-side validation removes the error message.

Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values in the Index page.

Update the Delete page model

Update `Pages/Departments/Delete.cshtml.cs` with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
```

```
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;

namespace ContosoUniversity.Pages.Departments
{
    public class DeleteModel : PageModel
    {
        private readonly ContosoUniversity.Data.SchoolContext _context;

        public DeleteModel(ContosoUniversity.Data.SchoolContext context)
        {
            _context = context;
        }

        [BindProperty]
        public Department Department { get; set; }
        public string ConcurrencyErrorMessage { get; set; }

        public async Task<IActionResult> OnGetAsync(int id, bool?
concurrencyError)
        {
            Department = await _context.Departments
                .Include(d => d.Administrator)
                .AsNoTracking()
                .FirstOrDefaultAsync(m => m.DepartmentID == id);

            if (Department == null)
            {
                 return NotFound();
            }

            if (concurrencyError.GetValueOrDefault())
            {
                ConcurrencyErrorMessage = "The record you attempted to
delete "
                  + "was modified by another user after you selected delete.
"
                  + "The delete operation was canceled and the current
values in the "
                  + "database have been displayed. If you still want to
delete this "
                  + "record, click the Delete button again.";
            }
            return Page();
        }

        public async Task<IActionResult> OnPostAsync(int id)
        {
            try
            {
                if (await _context.Departments.AnyAsync(
                    m => m.DepartmentID == id))
                {
                    // Department.ConcurrencyToken value is from when the
entity
```

```
                    // was fetched. If it doesn't match the DB, a
                    // DbUpdateConcurrencyException exception is thrown.
                    _context.Departments.Remove(Department);
                    await _context.SaveChangesAsync();
                }
                return RedirectToPage("./Index");
            }
            catch (DbUpdateConcurrencyException)
            {
                return RedirectToPage("./Delete",
                    new { concurrencyError = true, id = id });
            }
        }
    }
}
```

The Delete page detects concurrency conflicts when the entity has changed after it was fetched. `Department.ConcurrencyToken` is the row version when the entity was fetched. When EF Core creates the `SQL DELETE` command, it includes a WHERE clause with `ConcurrencyToken`. If the `SQL DELETE` command results in zero rows affected:

- The `ConcurrencyToken` in the `SQL DELETE` command doesn't match `ConcurrencyToken` in the database.
- A `DbUpdateConcurrencyException` exception is thrown.
- `OnGetAsync` is called with the `concurrencyError`.

Update the Delete Razor page

Update `Pages/Departments/Delete.cshtml` with the following code:

CSHTML

```
@page "{id:int}"
@model ContosoUniversity.Pages.Departments.DeleteModel

@{
    ViewData["Title"] = "Delete";
}

<h1>Delete</h1>

<p class="text-danger">@Model.ConcurrencyErrorMessage</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.Department.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.ConcurrencyToken)
        </dt>
        <dd class="col-sm-10">
            @Utility.GetLastChars(Model.Department.ConcurrencyToken)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model =>
model.Department.Administrator.FullName)
        </dd>
    </dl>

    <form method="post">
        <input type="hidden" asp-for="Department.DepartmentID" />
        <input type="hidden" asp-for="Department.ConcurrencyToken" />
        <input type="submit" value="Delete" class="btn btn-danger" /> |
        <a asp-page="./Index">Back to List</a>
    </form>
</div>
```

The preceding code makes the following changes:

- Updates the `page` directive from `@page` to `@page "{id:int}"`.
- Adds an error message.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Changes `ConcurrencyToken` to display the last byte.
- Adds a hidden row version. `ConcurrencyToken` must be added so postback binds the value.

Test concurrency conflicts

Create a test department.

Open two browsers instances of Delete on the test department:

- Run the app and select Departments.
- Right-click the **Delete** hyperlink for the test department and select **Open in new tab**.
- Click the **Edit** hyperlink for the test department.

The two browser tabs display the same information.

Change the budget in the first browser tab and click **Save**.

The browser shows the Index page with the changed value and updated `ConcurrencyToken` indicator. Note the updated `ConcurrencyToken` indicator, it's displayed on the second postback in the other tab.

Delete the test department from the second tab. A concurrency error is display with the current values from the database. Clicking **Delete** deletes the entity, unless `ConcurrencyToken` has been updated.

Additional resources

- Concurrency Tokens in EF Core
- Handle concurrency in EF Core
- Debugging ASP.NET Core 2.x source

Next steps

This is the last tutorial in the series. Additional topics are covered in the MVC version of this tutorial series.

Previous tutorial

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

ASP.NET Core MVC with EF Core - tutorial series

Article • 04/11/2023

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model. For new development, we recommend Razor Pages over MVC with controllers and views. See the Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- Best practices for EF.

1. Get started
2. Create, Read, Update, and Delete operations
3. Sorting, filtering, paging, and grouping
4. Migrations
5. Create a complex data model
6. Reading related data
7. Updating related data
8. Handle concurrency conflicts
9. Inheritance
10. Advanced topics

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Tutorial: Get started with EF Core in an ASP.NET MVC web app

Article • 04/11/2023

By Tom Dykstra and Rick Anderson

This tutorial teaches ASP.NET Core MVC and Entity Framework Core with controllers and views. Razor Pages is an alternative programming model. For new development, we recommend Razor Pages over MVC with controllers and views. See the Razor Pages version of this tutorial. Each tutorial covers some material the other doesn't:

Some things this MVC tutorial has that the Razor Pages tutorial doesn't:

- Implement inheritance in the data model
- Perform raw SQL queries
- Use dynamic LINQ to simplify code

Some things the Razor Pages tutorial has that this one doesn't:

- Use Select method to load related data
- Best practices for EF.

The Contoso University sample web app demonstrates how to create an ASP.NET Core MVC web app using Entity Framework (EF) Core and Visual Studio.

The sample app is a web site for a fictional Contoso University. It includes functionality such as student admission, course creation, and instructor assignments. This is the first in a series of tutorials that explain how to build the Contoso University sample app.

Prerequisites

- If you're new to ASP.NET Core MVC, go through the Get started with ASP.NET Core MVC tutorial series before starting this one.

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

This tutorial has not been updated for ASP.NET Core 6 or later. The tutorial's instructions will not work correctly if you create a project that targets ASP.NET Core 6 or later. For example, the ASP.NET Core 6 and later web templates use the minimal hosting model, which unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file.

Another difference introduced in .NET 6 is the NRT (nullable reference types) feature. The project templates enable this feature by default. Problems can happen where EF considers a property to be required in .NET 6 which is nullable in .NET 5. For example, the Create Student page will fail silently unless the `Enrollments` property is made nullable or the `asp-validation-summary` helper tag is changed from `ModelOnly` to `All`.

We recommend that you install and use the .NET 5 SDK for this tutorial. Until this tutorial is updated, see Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8 on how to use Entity Framework with ASP.NET Core 6 or later.

Database engines

The Visual Studio instructions use SQL Server LocalDB, a version of SQL Server Express that runs only on Windows.

Solve problems and troubleshoot

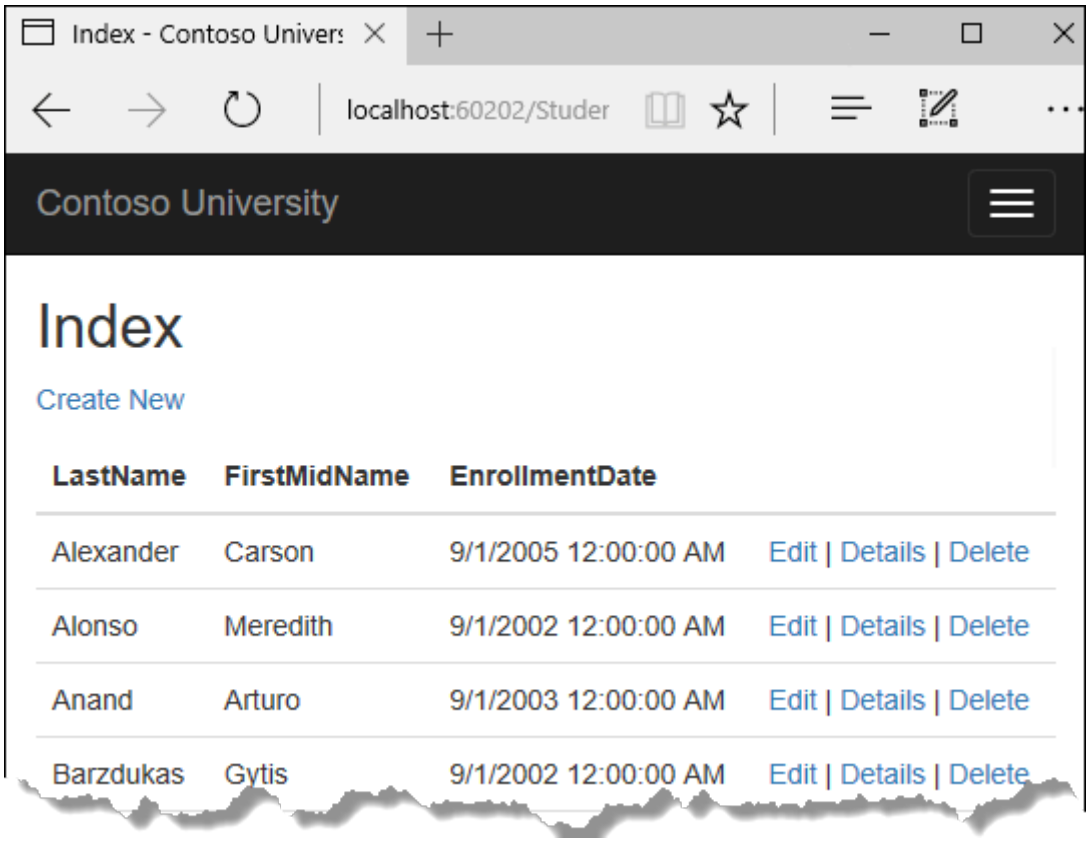
If you run into a problem you can't resolve, you can generally find the solution by comparing your code to the completed project. For a list of common errors and how to solve them, see the Troubleshooting section of the last tutorial in the series. If you don't find what you need there, you can post a question to StackOverflow.com for ASP.NET Core or EF Core.

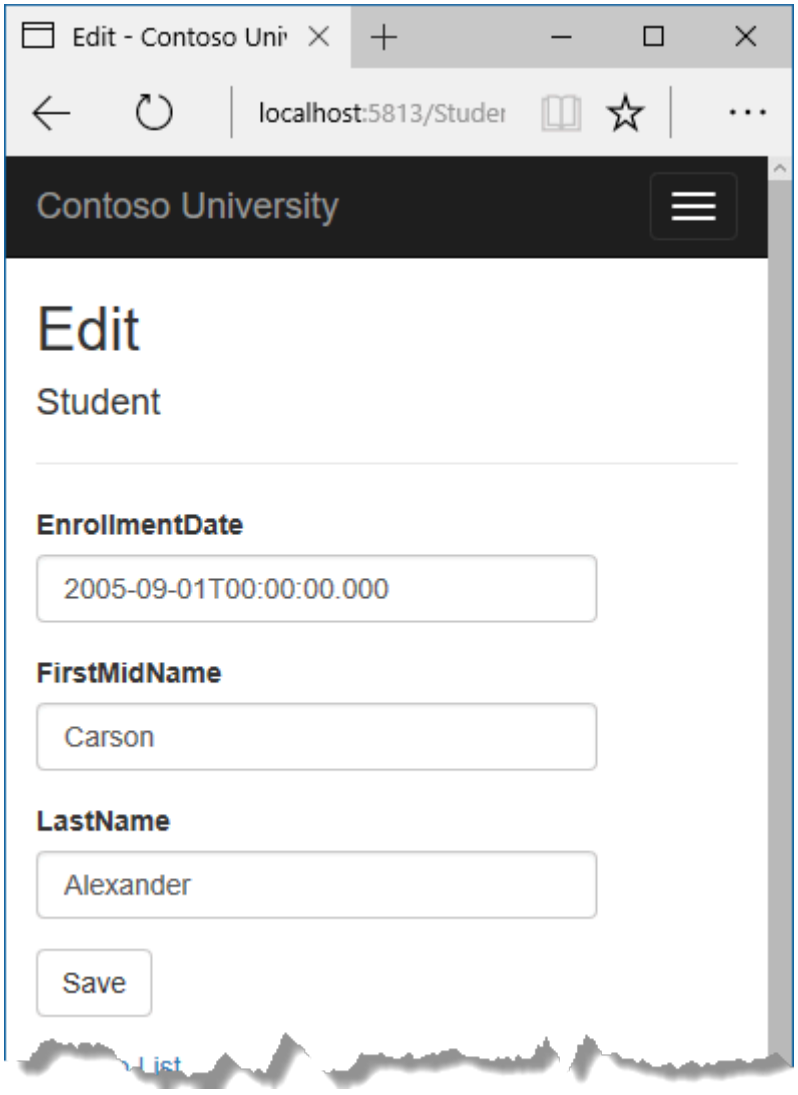
> **Tip**
>
> This is a series of 10 tutorials, each of which builds on what is done in earlier tutorials. Consider saving a copy of the project after each successful tutorial completion. Then if you run into problems, you can start over from the previous tutorial instead of going back to the beginning of the whole series.

Contoso University web app

The app built in these tutorials is a basic university web site.

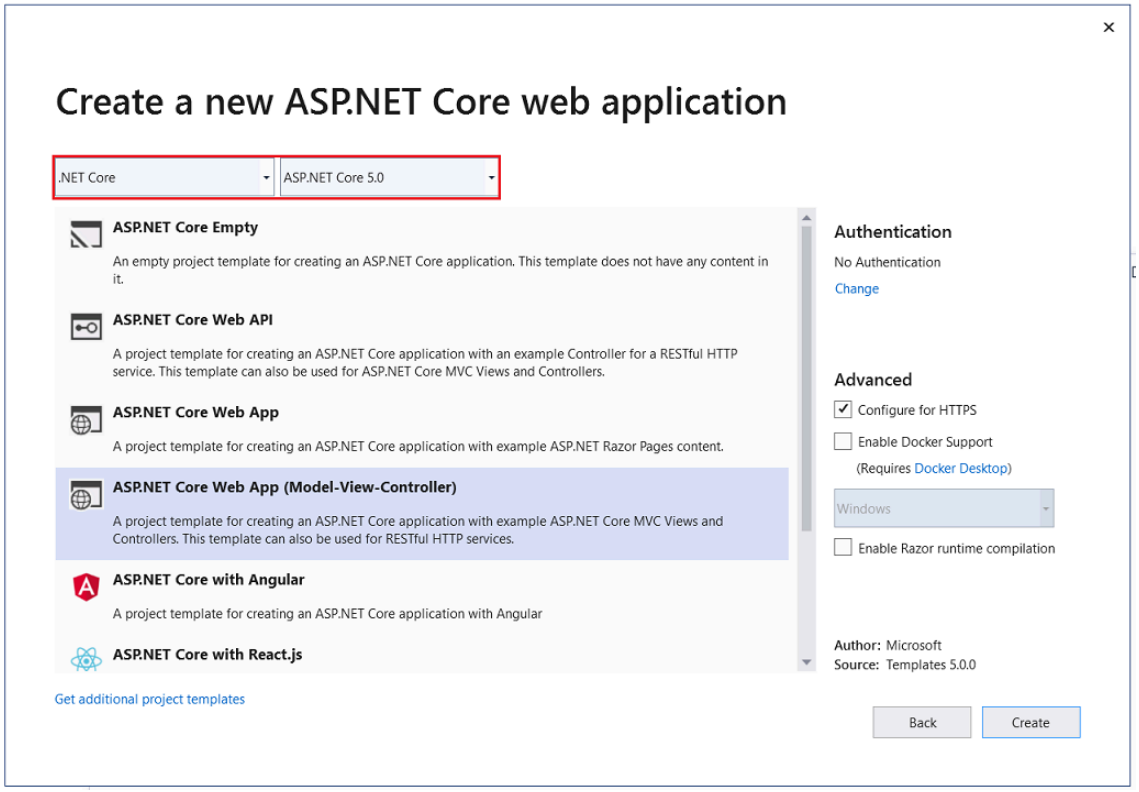
Users can view and update student, course, and instructor information. Here are a few of the screens in the app:


Index - Contoso Univers
localhost:60202/Studer
Contoso University
Index
Create New
LastName
FirstMidName
EnrollmentDate
Alexander
Carson
9/1/2005 12:00:00 AM
Edit | Details | Delete
Alonso
Meredith
9/1/2002 12:00:00 AM
Edit | Details | Delete
Anand
Arturo
9/1/2003 12:00:00 AM
Edit | Details | Delete
Barzdukas
Gytis
9/1/2002 12:00:00 AM
Edit | Details | Delete


Edit - Contoso Uni
localhost:5813/Studer
Contoso University
Edit
Student
EnrollmentDate
2005-09-01T00:00:00.000
FirstMidName
Carson
LastName
Alexander
Save

Create web app

1. Start Visual Studio and select **Create a new project**.
2. In the **Create a new project** dialog, select **ASP.NET Core Web Application** > **Next**.
3. In the **Configure your new project** dialog, enter `ContosoUniversity` for **Project name**. It's important to use this exact name including capitalization, so each `namespace` matches when code is copied.
4. Select **Create**.
5. In the **Create a new ASP.NET Core web application** dialog, select:
   a. **.NET Core** and **ASP.NET Core 5.0** in the dropdowns.
   b. **ASP.NET Core Web App (Model-View-Controller)**.
   c. **Create**



Set up the site style

A few basic changes set up the site menu, layout, and home page.

Open `Views/Shared/_Layout.cshtml` and make the following changes:

- Change each occurrence of `ContosoUniversity` to `Contoso University`. There are three occurrences.
- Add menu entries for **About**, **Students**, **Courses**, **Instructors**, and **Departments**, and delete the **Privacy** menu entry.

The preceding changes are highlighted in the following code:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Contoso University</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-controller="Home"
asp-action="Index">Contoso University</a>
                <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="About">About</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Students" asp-action="Index">Students</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Courses" asp-action="Index">Courses</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Instructors" asp-action="Index">Instructors</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Departments" asp-action="Index">Departments</a>
                        </li>
                    </ul>
                </div>
```

```
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2020 - Contoso University - <a asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

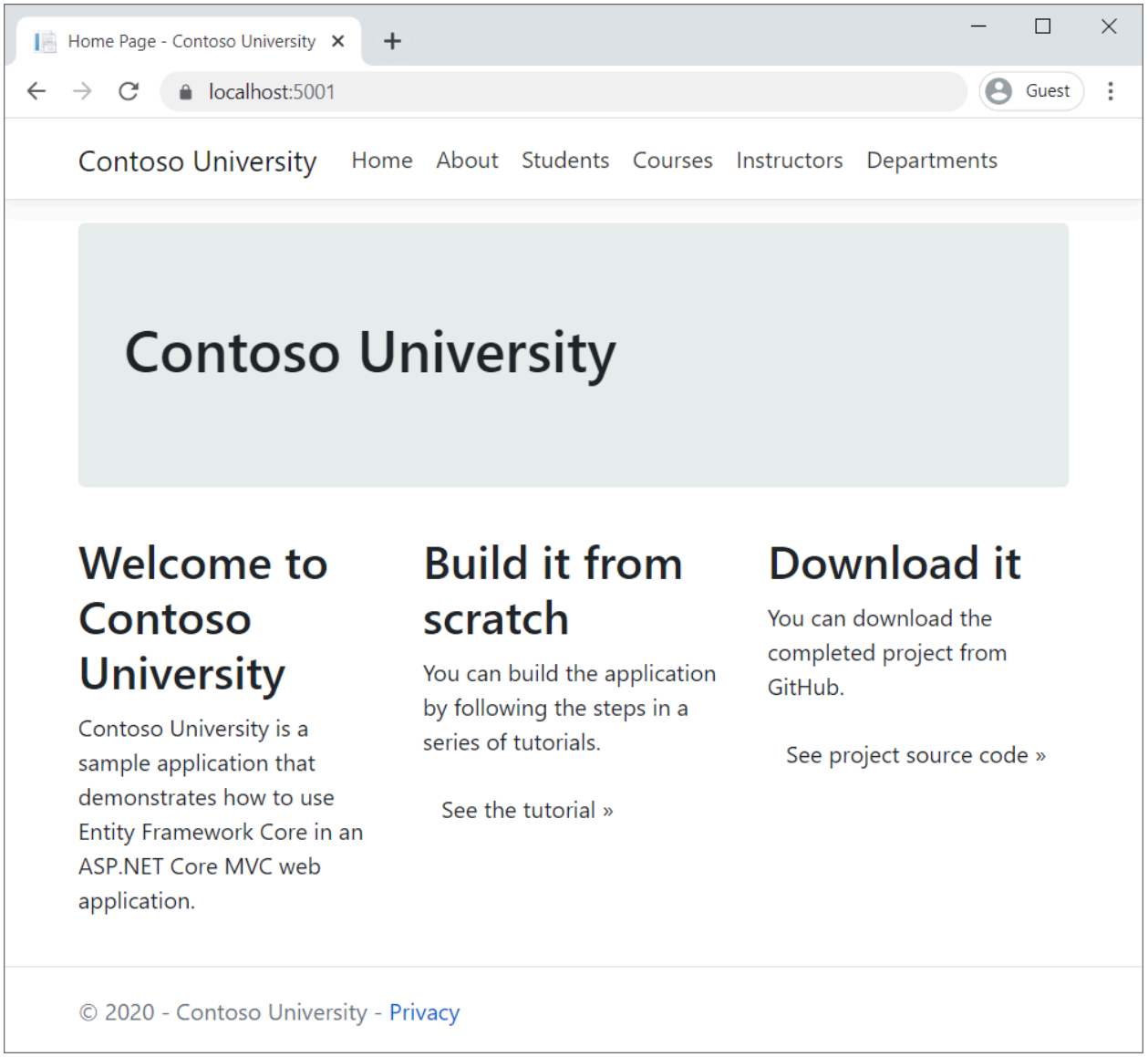
In `Views/Home/Index.cshtml`, replace the contents of the file with the following markup:

CSHTML

```
@{
    ViewData["Title"] = "Home Page";
}

<div class="jumbotron">
    <h1>Contoso University</h1>
</div>
<div class="row">
    <div class="col-md-4">
        <h2>Welcome to Contoso University</h2>
        <p>
            Contoso University is a sample application that
            demonstrates how to use Entity Framework Core in an
            ASP.NET Core MVC web application.
        </p>
    </div>
    <div class="col-md-4">
        <h2>Build it from scratch</h2>
        <p>You can build the application by following the steps in a series
of tutorials.</p>
        <p><a class="btn btn-default"
href="https://docs.asp.net/en/latest/data/ef-mvc/intro.html">See the
tutorial &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Download it</h2>
        <p>You can download the completed project from GitHub.</p>
```

```
        <p><a class="btn btn-default"
href="https://github.com/dotnet/AspNetCore.Docs/tree/main/aspnetcore/data/ef
-mvc/intro/samples/5cu-final">See project source code &raquo;</a></p>
    </div>
</div>
```

Press CTRL+F5 to run the project or choose **Debug** > **Start Without Debugging** from the menu. The home page is displayed with tabs for the pages created in this tutorial.



EF Core NuGet packages

This tutorial uses SQL Server, and the provider package is Microsoft.EntityFrameworkCore.SqlServer.

The EF SQL Server package and its dependencies, `Microsoft.EntityFrameworkCore` and `Microsoft.EntityFrameworkCore.Relational`, provide runtime support for EF.

Add the Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore NuGet package. In the Package Manager Console (PMC), enter the following commands to add the NuGet packages:

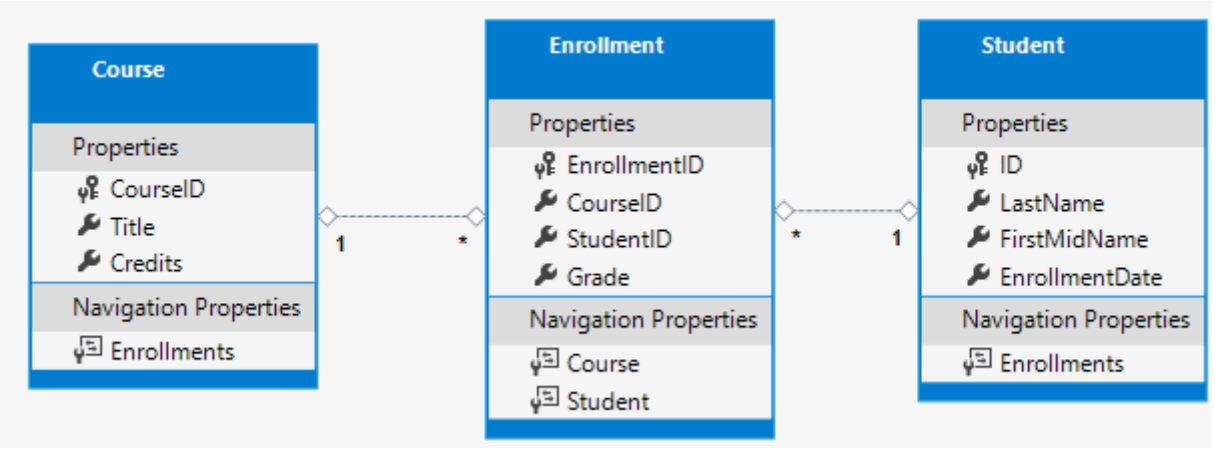
```
PowerShell

Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

The `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore` NuGet package provides ASP.NET Core middleware for EF Core error pages. This middleware helps to detect and diagnose errors with EF Core migrations.

For information about other database providers that are available for EF Core, see Database providers.

Create the data model

The following entity classes are created for this app:

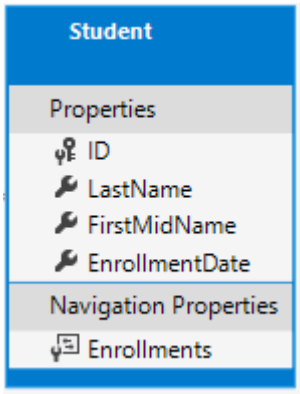


The preceding entities have the following relationships:

- A one-to-many relationship between `Student` and `Enrollment` entities. A student can be enrolled in any number of courses.
- A one-to-many relationship between `Course` and `Enrollment` entities. A course can have any number of students enrolled in it.

In the following sections, a class is created for each of these entities.

The Student entity



In the *Models* folder, create the `Student` class with the following code:

C#

```
using System;
using System.Collections.Generic;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `ID` property is the primary key (**PK**) column of the database table that corresponds to this class. By default, EF interprets a property that's named `ID` or `classnameID` as the primary key. For example, the PK could be named `StudentID` rather than `ID`.

The `Enrollments` property is a navigation property. Navigation properties hold other entities that are related to this entity. The `Enrollments` property of a `Student` entity:

- Contains all of the `Enrollment` entities that are related to that `Student` entity.
- If a specific `Student` row in the database has two related `Enrollment` rows:
  - That `Student` entity's `Enrollments` navigation property contains those two `Enrollment` entities.

`Enrollment` rows contain a student's PK value in the `StudentID` foreign key (**FK**) column.

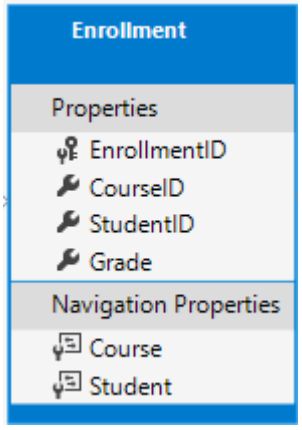
If a navigation property can hold multiple entities:

- The type must be a list, such as `ICollection<T>`, `List<T>`, or `HashSet<T>`.

- Entities can be added, deleted, and updated.

Many-to-many and one-to-many navigation relationships can contain multiple entities. When `ICollection<T>` is used, EF creates a `HashSet<T>` collection by default.

The Enrollment entity



In the *Models* folder, create the `Enrollment` class with the following code:

C#

```
namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

The `EnrollmentID` property is the PK. This entity uses the `classnameID` pattern instead of `ID` by itself. The `Student` entity used the `ID` pattern. Some developers prefer to use one pattern throughout the data model. In this tutorial, the variation illustrates that either pattern can be used. A later tutorial shows how using `ID` without classname makes it easier to implement inheritance in the data model.

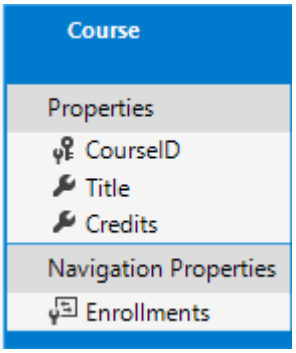
The `Grade` property is an `enum`. The `?` after the `Grade` type declaration indicates that the `Grade` property is nullable. A grade that's `null` is different from a zero grade. `null` means a grade isn't known or hasn't been assigned yet.

The `StudentID` property is a foreign key (FK), and the corresponding navigation property is `Student`. An `Enrollment` entity is associated with one `Student` entity, so the property can only hold a single `Student` entity. This differs from the `Student.Enrollments` navigation property, which can hold multiple `Enrollment` entities.

The `CourseID` property is a FK, and the corresponding navigation property is `Course`. An `Enrollment` entity is associated with one `Course` entity.

Entity Framework interprets a property as a FK property if it's named `<`navigation property name`><`primary key property name`>`. For example, `StudentID` for the `Student` navigation property since the `Student` entity's PK is `ID`. FK properties can also be named `<`primary key property name`>`. For example, `CourseID` because the `Course` entity's PK is `CourseID`.

The Course entity



In the *Models* folder, create the `Course` class with the following code:

C#

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int CourseID { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
```

```
    }
}
```

The `Enrollments` property is a navigation property. A `Course` entity can be related to any number of `Enrollment` entities.

The DatabaseGenerated attribute is explained in a later tutorial. This attribute allows entering the PK for the course rather than having the database generate it.

Create the database context

The main class that coordinates EF functionality for a given data model is the DbContext database context class. This class is created by deriving from the `Microsoft.EntityFrameworkCore.DbContext` class. The `DbContext` derived class specifies which entities are included in the data model. Some EF behaviors can be customized. In this project, the class is named `SchoolContext`.

In the project folder, create a folder named `Data`.

In the *Data* folder create a `SchoolContext` class with the following code:

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

The preceding code creates a `DbSet` property for each entity set. In EF terminology:

- An entity set typically corresponds to a database table.
- An entity corresponds to a row in the table.

The `DbSet<Enrollment>` and `DbSet<Course>` statements could be omitted and it would work the same. EF would include them implicitly because:

- The `Student` entity references the `Enrollment` entity.
- The `Enrollment` entity references the `Course` entity.

When the database is created, EF creates tables that have names the same as the `DbSet` property names. Property names for collections are typically plural. For example, `Students` rather than `Student`. Developers disagree about whether table names should be pluralized or not. For these tutorials, the default behavior is overridden by specifying singular table names in the `DbContext`. To do that, add the following highlighted code after the last DbSet property.

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
        }
    }
}
```

Register the `SchoolContext`

ASP.NET Core includes dependency injection. Services, such as the EF database context, are registered with dependency injection during app startup. Components that require these services, such as MVC controllers, are provided these services via constructor

parameters. The controller constructor code that gets a context instance is shown later in this tutorial.

To register `SchoolContext` as a service, open `Startup.cs`, and add the highlighted lines to the `ConfigureServices` method.

C#

```
using ContosoUniversity.Data;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace ContosoUniversity
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

            services.AddControllersWithViews();
        }
```

The name of the connection string is passed in to the context by calling a method on a `DbContextOptionsBuilder` object. For local development, the ASP.NET Core configuration system reads the connection string from the `appsettings.json` file.

Open the `appsettings.json` file and add a connection string as shown in the following markup:

JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity1;Trusted_Connection=True;
MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Add the database exception filter

Add AddDatabaseDeveloperPageExceptionFilter to `ConfigureServices` as shown in the following code:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<SchoolContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

    services.AddDatabaseDeveloperPageExceptionFilter();

    services.AddControllersWithViews();
}
```

The `AddDatabaseDeveloperPageExceptionFilter` provides helpful error information in the development environment.

SQL Server Express LocalDB

The connection string specifies SQL Server LocalDB. LocalDB is a lightweight version of the SQL Server Express Database Engine and is intended for app development, not production use. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB creates *.mdf* DB files in the `C:/Users/<user>` directory.

Initialize DB with test data

EF creates an empty database. In this section, a method is added that's called after the database is created in order to populate it with test data.

The `EnsureCreated` method is used to automatically create the database. In a later tutorial, you see how to handle model changes by using Code First Migrations to change the database schema instead of dropping and re-creating the database.

In the *Data* folder, create a new class named `DbInitializer` with the following code:

C#

```
using ContosoUniversity.Models;
using System;
using System.Linq;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
            new
Student{FirstMidName="Carson",LastName="Alexander",EnrollmentDate=DateTime.P
arse("2005-09-01")},
            new
Student{FirstMidName="Meredith",LastName="Alonso",EnrollmentDate=DateTime.Pa
rse("2002-09-01")},
            new
Student{FirstMidName="Arturo",LastName="Anand",EnrollmentDate=DateTime.Parse
("2003-09-01")},
            new
Student{FirstMidName="Gytis",LastName="Barzdukas",EnrollmentDate=DateTime.Pa
rse("2002-09-01")},
            new
Student{FirstMidName="Yan",LastName="Li",EnrollmentDate=DateTime.Parse("2002
-09-01")},
            new
Student{FirstMidName="Peggy",LastName="Justice",EnrollmentDate=DateTime.Pars
e("2001-09-01")},
            new
```

```
Student{FirstMidName="Laura",LastName="Norman",EnrollmentDate=DateTime.Parse
("2003-09-01")},
            new
Student{FirstMidName="Nino",LastName="Olivetto",EnrollmentDate=DateTime.Pars
e("2005-09-01")}
            };
            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
            new Course{CourseID=1050,Title="Chemistry",Credits=3},
            new Course{CourseID=4022,Title="Microeconomics",Credits=3},
            new Course{CourseID=4041,Title="Macroeconomics",Credits=3},
            new Course{CourseID=1045,Title="Calculus",Credits=4},
            new Course{CourseID=3141,Title="Trigonometry",Credits=4},
            new Course{CourseID=2021,Title="Composition",Credits=3},
            new Course{CourseID=2042,Title="Literature",Credits=4}
            };
            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
            new Enrollment{StudentID=1,CourseID=1050,Grade=Grade.A},
            new Enrollment{StudentID=1,CourseID=4022,Grade=Grade.C},
            new Enrollment{StudentID=1,CourseID=4041,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=1045,Grade=Grade.B},
            new Enrollment{StudentID=2,CourseID=3141,Grade=Grade.F},
            new Enrollment{StudentID=2,CourseID=2021,Grade=Grade.F},
            new Enrollment{StudentID=3,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=1050},
            new Enrollment{StudentID=4,CourseID=4022,Grade=Grade.F},
            new Enrollment{StudentID=5,CourseID=4041,Grade=Grade.C},
            new Enrollment{StudentID=6,CourseID=1045},
            new Enrollment{StudentID=7,CourseID=3141,Grade=Grade.A},
            };
            foreach (Enrollment e in enrollments)
            {
                context.Enrollments.Add(e);
            }
            context.SaveChanges();
        }
    }
}
```

The preceding code checks if the database exists:

- If the database is not found;
  - It is created and loaded with test data. It loads test data into arrays rather than `List<T>` collections to optimize performance.
- If the database is found, it takes no action.

Update `Program.cs` with the following code:

C#

```
using ContosoUniversity.Data;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

namespace ContosoUniversity
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = CreateHostBuilder(args).Build();

            CreateDbIfNotExists(host);

            host.Run();
        }

        private static void CreateDbIfNotExists(IHost host)
        {
            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;
                try
                {
                    var context = services.GetRequiredService<SchoolContext>
();
                    DbInitializer.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger =
services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred creating the
DB.");
                }
            }
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
```

```
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

`Program.cs` does the following on app startup:

- Get a database context instance from the dependency injection container.
- Call the `DbInitializer.Initialize` method.
- Dispose the context when the `Initialize` method completes as shown in the following code:

C#

```
public static void Main(string[] args)
{
     var host = CreateWebHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        try
        {
            var context = services.GetRequiredService<SchoolContext>();
            DbInitializer.Initialize(context);
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred while seeding the
database.");
        }
    }

    host.Run();
}
```

The first time the app is run, the database is created and loaded with test data. Whenever the data model changes:

- Delete the database.
- Update the seed method, and start afresh with a new database.

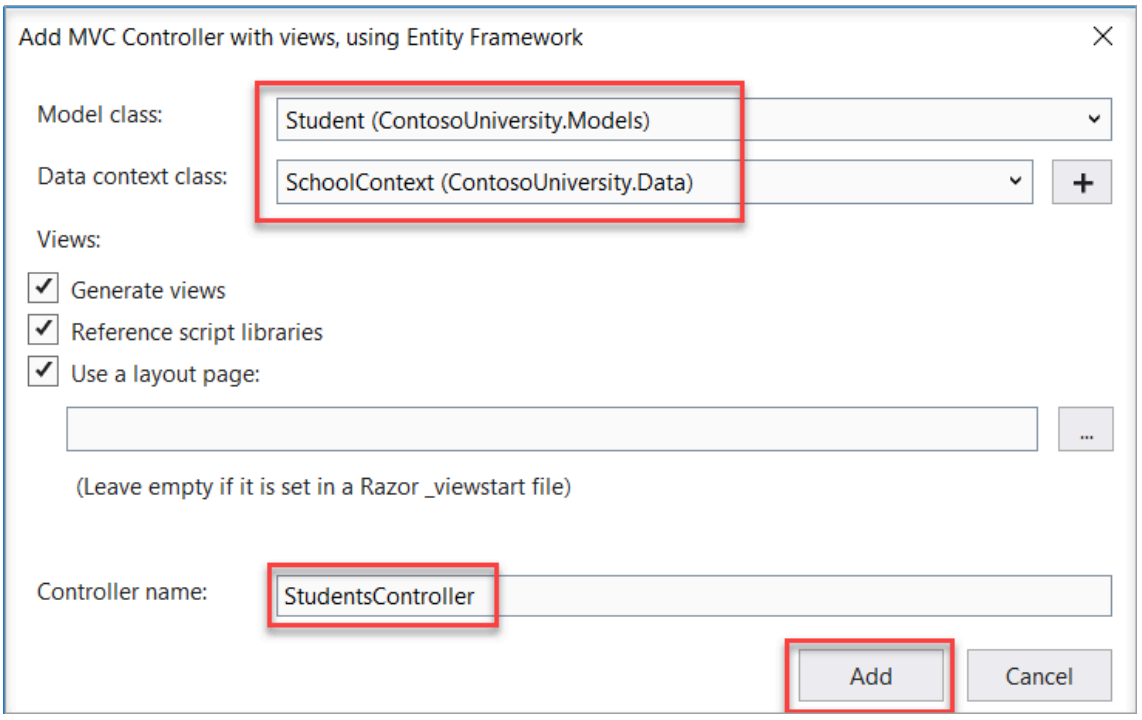
In later tutorials, the database is modified when the data model changes, without deleting and re-creating it. No data is lost when the data model changes.

Create controller and views

Use the scaffolding engine in Visual Studio to add an MVC controller and views that will use EF to query and save data.

The automatic creation of CRUD action methods and views is known as scaffolding.

- In **Solution Explorer**, right-click the `Controllers` folder and select **Add** > **New Scaffolded Item**.
- In the **Add Scaffold** dialog box:
  - Select **MVC controller with views, using Entity Framework**.
  - Click **Add**. The **Add MVC Controller with views, using Entity Framework** dialog box appears:



  - In **Model class**, select **Student**.
  - In **Data context class**, select **SchoolContext**.
  - Accept the default **StudentsController** as the name.
  - Click **Add**.

The Visual Studio scaffolding engine creates a `StudentsController.cs` file and a set of views (`*.cshtml` files) that work with the controller.

Notice the controller takes a `SchoolContext` as a constructor parameter.

C#

```
namespace ContosoUniversity.Controllers
{
    public class StudentsController : Controller
    {
```

```
        private readonly SchoolContext _context;

        public StudentsController(SchoolContext context)
        {
            _context = context;
        }
```

ASP.NET Core dependency injection takes care of passing an instance of `SchoolContext` into the controller. You configured that in the `Startup` class.

The controller contains an `Index` action method, which displays all students in the database. The method gets a list of students from the Students entity set by reading the `Students` property of the database context instance:

C#

```
public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

The asynchronous programming elements in this code are explained later in the tutorial.

The `Views/Students/Index.cshtml` view displays this list in a table:

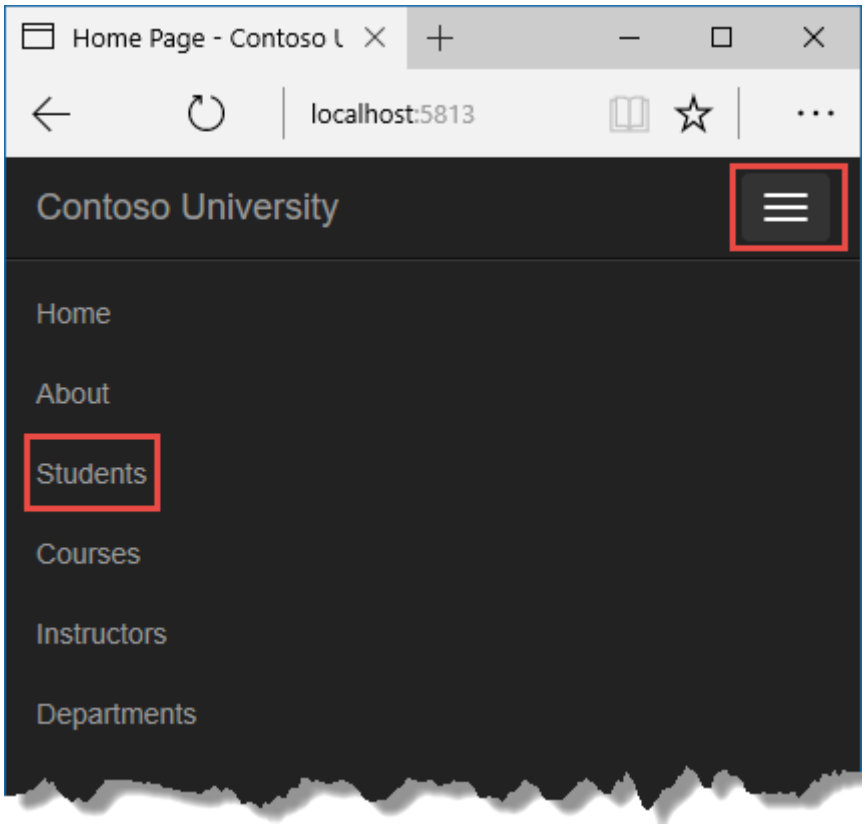
CSHTML

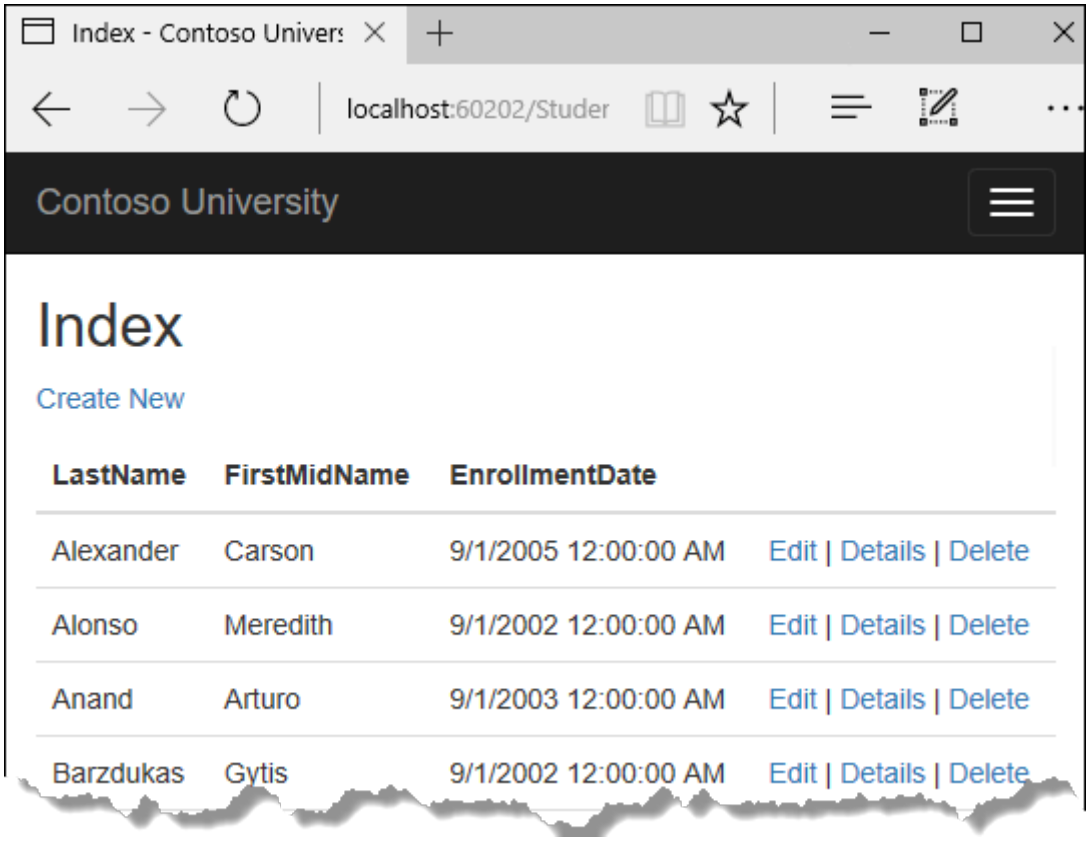
```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    @Html.DisplayNameFor(model => model.LastName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    @Html.DisplayNameFor(model => model.EnrollmentDate)
                </th>
```

```
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a>
|
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Press CTRL+F5 to run the project or choose **Debug** > **Start Without Debugging** from the menu.

Click the Students tab to see the test data that the `DbInitializer.Initialize` method inserted. Depending on how narrow your browser window is, you'll see the `Students` tab link at the top of the page or you'll have to click the navigation icon in the upper right corner to see the link.





View the database

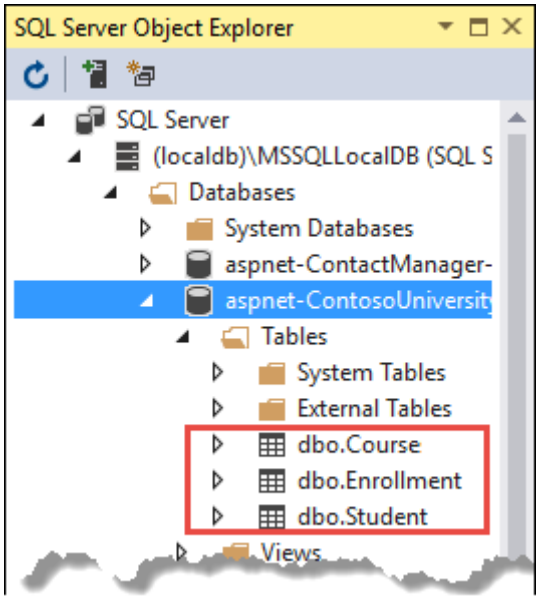
When the app is started, the `DbInitializer.Initialize` method calls `EnsureCreated`. EF saw that there was no database:

- So it created a database.

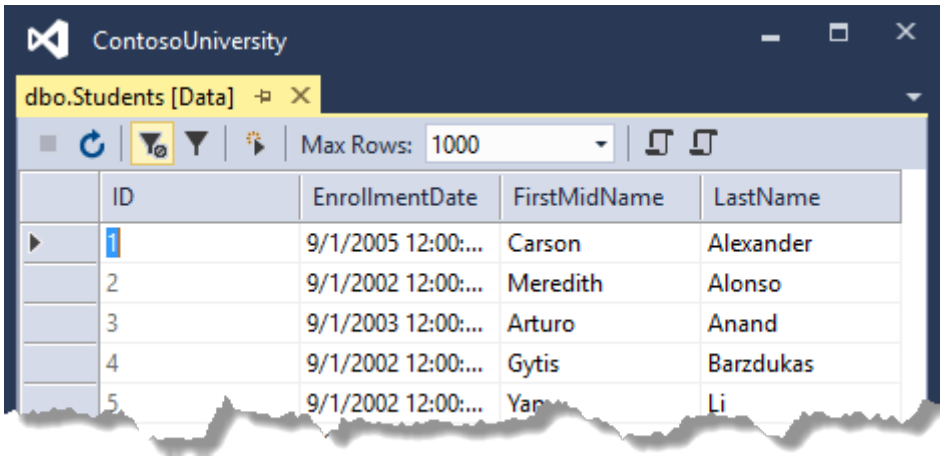
- The `Initialize` method code populated the database with data.

Use **SQL Server Object Explorer** (SSOX) to view the database in Visual Studio:

- Select **SQL Server Object Explorer** from the **View** menu in Visual Studio.
- In SSOX, select **(localdb)\MSSQLLocalDB** > **Databases**.
- Select `ContosoUniversity1`, the entry for the database name that's in the connection string in the `appsettings.json` file.
- Expand the **Tables** node to see the tables in the database.



Right-click the **Student** table and click **View Data** to see the data in the table.



The `*.mdf` and `*.ldf` database files are in the *C:\Users\<username>* folder.

Because `EnsureCreated` is called in the initializer method that runs on app start, you could:

- Make a change to the `Student` class.
- Delete the database.

- Stop, then start the app. The database is automatically re-created to match the change.

For example, if an `EmailAddress` property is added to the `Student` class, a new `EmailAddress` column in the re-created table. The view won't display the new `EmailAddress` property.

Conventions

The amount of code written in order for the EF to create a complete database is minimal because of the use of the conventions EF uses:

- The names of `DbSet` properties are used as table names. For entities not referenced by a `DbSet` property, entity class names are used as table names.
- Entity property names are used for column names.
- Entity properties that are named `ID` or `classnameID` are recognized as PK properties.
- A property is interpreted as a FK property if it's named `<`navigation property name`><`PK property name`>`. For example, `StudentID` for the `Student` navigation property since the `Student` entity's PK is `ID`. FK properties can also be named `<`primary key property name`>`. For example, `EnrollmentID` since the `Enrollment` entity's PK is `EnrollmentID`.

Conventional behavior can be overridden. For example, table names can be explicitly specified, as shown earlier in this tutorial. Column names and any property can be set as a PK or FK.

Asynchronous code

Asynchronous programming is the default mode for ASP.NET Core and EF Core.

A web server has a limited number of threads available, and in high load situations all of the available threads might be in use. When that happens, the server can't process new requests until the threads are freed up. With synchronous code, many threads may be tied up while they aren't actually doing any work because they're waiting for I/O to complete. With asynchronous code, when a process is waiting for I/O to complete, its thread is freed up for the server to use for processing other requests. As a result, asynchronous code enables server resources to be used more efficiently, and the server is enabled to handle more traffic without delays.

Asynchronous code does introduce a small amount of overhead at run time, but for low traffic situations the performance hit is negligible, while for high traffic situations, the potential performance improvement is substantial.

In the following code, `async`, `Task<T>`, `await`, and `ToListAsync` make the code execute asynchronously.

```
C#

public async Task<IActionResult> Index()
{
    return View(await _context.Students.ToListAsync());
}
```

- The `async` keyword tells the compiler to generate callbacks for parts of the method body and to automatically create the `Task<IActionResult>` object that's returned.
- The return type `Task<IActionResult>` represents ongoing work with a result of type `IActionResult`.
- The `await` keyword causes the compiler to split the method into two parts. The first part ends with the operation that's started asynchronously. The second part is put into a callback method that's called when the operation completes.
- `ToListAsync` is the asynchronous version of the `ToList` extension method.

Some things to be aware of when writing asynchronous code that uses EF:

- Only statements that cause queries or commands to be sent to the database are executed asynchronously. That includes, for example, `ToListAsync`, `SingleOrDefaultAsync`, and `SaveChangesAsync`. It doesn't include, for example, statements that just change an `IQueryable`, such as `var students = context.Students.Where(s => s.LastName == "Davolio")`.
- An EF context isn't thread safe: don't try to do multiple operations in parallel. When you call any async EF method, always use the `await` keyword.
- To take advantage of the performance benefits of async code, make sure that any library packages used also use async if they call any EF methods that cause queries to be sent to the database.

For more information about asynchronous programming in .NET, see Async Overview.

Limit entities fetched

See Performance considerations for information on limiting the number of entities returned from a query.

SQL Logging of Entity Framework Core

Logging configuration is commonly provided by the `Logging` section of `appsettings.{Environment}.json` files. To log SQL statements, add `"Microsoft.EntityFrameworkCore.Database.Command": "Information"` to the `appsettings.Development.json` file:

JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=MyDB-
2;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
     ,"Microsoft.EntityFrameworkCore.Database.Command": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

With the preceding JSON, SQL statements are displayed on the command line and in the Visual Studio output window.

For more information, see Logging in .NET Core and ASP.NET Core and this GitHub issue.

Advance to the next tutorial to learn how to perform basic CRUD (create, read, update, delete) operations.

Implement basic CRUD functionality

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Tutorial: Implement CRUD Functionality - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you created an MVC application that stores and displays data using the Entity Framework and SQL Server LocalDB. In this tutorial, you'll review and customize the CRUD (create, read, update, delete) code that the MVC scaffolding automatically creates for you in controllers and views.

> **Note**
>
> It's a common practice to implement the repository pattern in order to create an abstraction layer between your controller and the data access layer. To keep these tutorials simple and focused on teaching how to use the Entity Framework itself, they don't use repositories. For information about repositories with EF, see the last tutorial in this series.

In this tutorial, you:

- ✓ Customize the Details page
- ✓ Update the Create page
- ✓ Update the Edit page
- ✓ Update the Delete page
- ✓ Close database connections

Prerequisites

- Get started with EF Core and ASP.NET Core MVC

Customize the Details page

The scaffolded code for the Students Index page left out the `Enrollments` property, because that property holds a collection. In the **Details** page, you'll display the contents of the collection in an HTML table.

In `Controllers/StudentsController.cs`, the action method for the Details view uses the `FirstOrDefaultAsync` method to retrieve a single `Student` entity. Add code that calls

`Include`. `ThenInclude`, and `AsNoTracking` methods, as shown in the following highlighted code.

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .Include(s => s.Enrollments)
            .ThenInclude(e => e.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);

    if (student == null)
    {
        return NotFound();
    }

    return View(student);
}
```

The `Include` and `ThenInclude` methods cause the context to load the `Student.Enrollments` navigation property, and within each enrollment the `Enrollment.Course` navigation property. You'll learn more about these methods in the read related data tutorial.

The `AsNoTracking` method improves performance in scenarios where the entities returned won't be updated in the current context's lifetime. You'll learn more about `AsNoTracking` at the end of this tutorial.

Route data

The key value that's passed to the `Details` method comes from *route data*. Route data is data that the model binder found in a segment of the URL. For example, the default route specifies controller, action, and id segments:

C#

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
```

```
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

In the following URL, the default route maps Instructor as the controller, Index as the action, and 1 as the id; these are route data values.

```
http://localhost:1230/Instructor/Index/1?courseID=2021
```

The last part of the URL ("?courseID=2021") is a query string value. The model binder will also pass the ID value to the `Index` method `id` parameter if you pass it as a query string value:

```
http://localhost:1230/Instructor/Index?id=1&CourseID=2021
```

In the Index page, hyperlink URLs are created by tag helper statements in the Razor view. In the following Razor code, the `id` parameter matches the default route, so `id` is added to the route data.

HTML

```
<a asp-action="Edit" asp-route-id="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

HTML

```
<a href="/Students/Edit/6">Edit</a>
```

In the following Razor code, `studentID` doesn't match a parameter in the default route, so it's added as a query string.

HTML

```
<a asp-action="Edit" asp-route-studentID="@item.ID">Edit</a>
```

This generates the following HTML when `item.ID` is 6:

HTML

```
<a href="/Students/Edit?studentID=6">Edit</a>
```

For more information about tag helpers, see Tag Helpers in ASP.NET Core.

Add enrollments to the Details view

Open `Views/Students/Details.cshtml`. Each field is displayed using `DisplayNameFor` and `DisplayFor` helpers, as shown in the following example:

CSHTML

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.LastName)
</dt>
<dd class="col-sm-10">
    @Html.DisplayFor(model => model.LastName)
</dd>
```

After the last field and immediately before the closing `</dl>` tag, add the following code to display a list of enrollments:

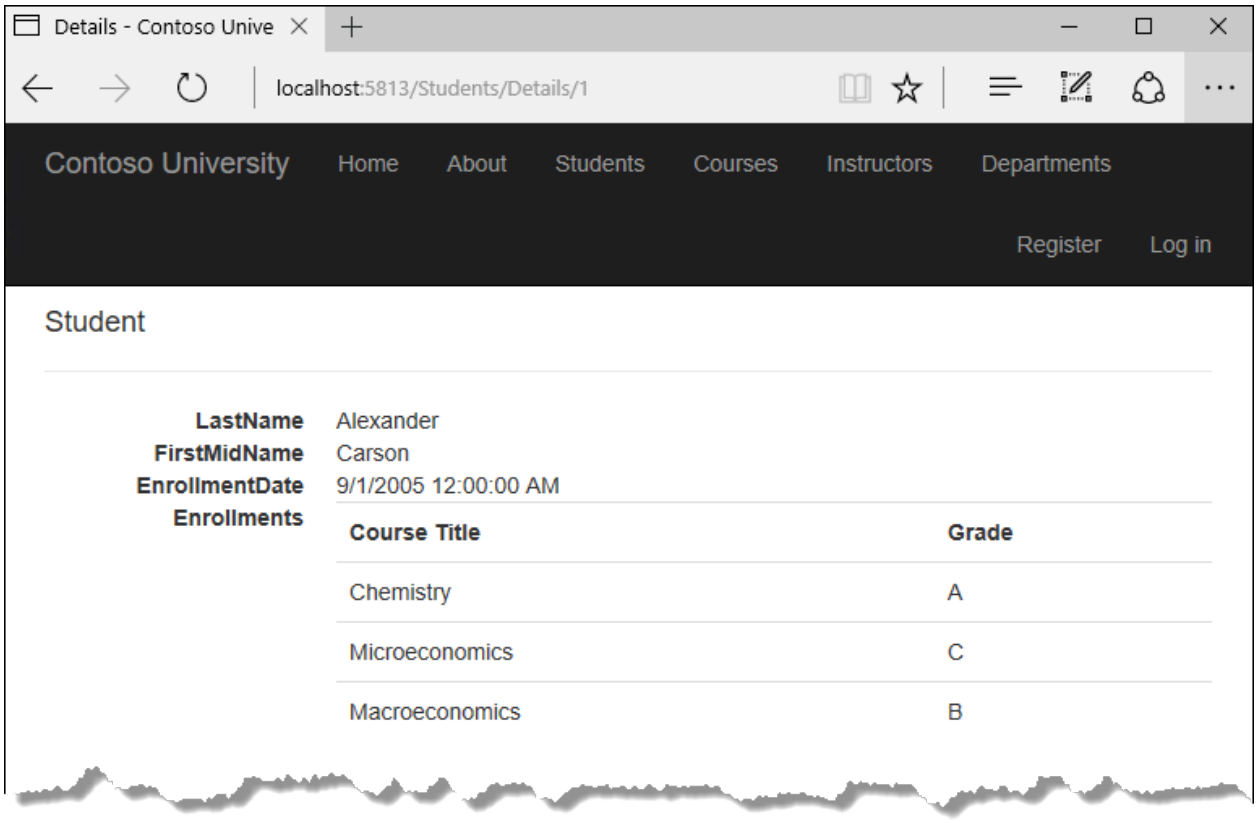
CSHTML

```
<dt class="col-sm-2">
    @Html.DisplayNameFor(model => model.Enrollments)
</dt>
<dd class="col-sm-10">
    <table class="table">
        <tr>
            <th>Course Title</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Course.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
</dd>
```

If code indentation is wrong after you paste the code, press CTRL-K-D to correct it.

This code loops through the entities in the `Enrollments` navigation property. For each enrollment, it displays the course title and the grade. The course title is retrieved from the Course entity that's stored in the `Course` navigation property of the Enrollments entity.

Run the app, select the **Students** tab, and click the **Details** link for a student. You see the list of courses and grades for the selected student:



Update the Create page

In `StudentsController.cs`, modify the HttpPost `Create` method by adding a try-catch block and removing ID from the `Bind` attribute.

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
```

```
            }
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists " +
                "see your system administrator.");
        }
        return View(student);
    }
```

This code adds the Student entity created by the ASP.NET Core MVC model binder to the Students entity set and then saves the changes to the database. (Model binder refers to the ASP.NET Core MVC functionality that makes it easier for you to work with data submitted by a form; a model binder converts posted form values to CLR types and passes them to the action method in parameters. In this case, the model binder instantiates a Student entity for you using property values from the Form collection.)

You removed `ID` from the `Bind` attribute because ID is the primary key value which SQL Server will set automatically when the row is inserted. Input from the user doesn't set the ID value.

Other than the `Bind` attribute, the try-catch block is the only change you've made to the scaffolded code. If an exception that derives from `DbUpdateException` is caught while the changes are being saved, a generic error message is displayed. `DbUpdateException` exceptions are sometimes caused by something external to the application rather than a programming error, so the user is advised to try again. Although not implemented in this sample, a production quality application would log the exception. For more information, see the **Log for insight** section in Monitoring and Telemetry (Building Real-World Cloud Apps with Azure).

The `ValidateAntiForgeryToken` attribute helps prevent cross-site request forgery (CSRF) attacks. The token is automatically injected into the view by the FormTagHelper and is included when the form is submitted by the user. The token is validated by the `ValidateAntiForgeryToken` attribute. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

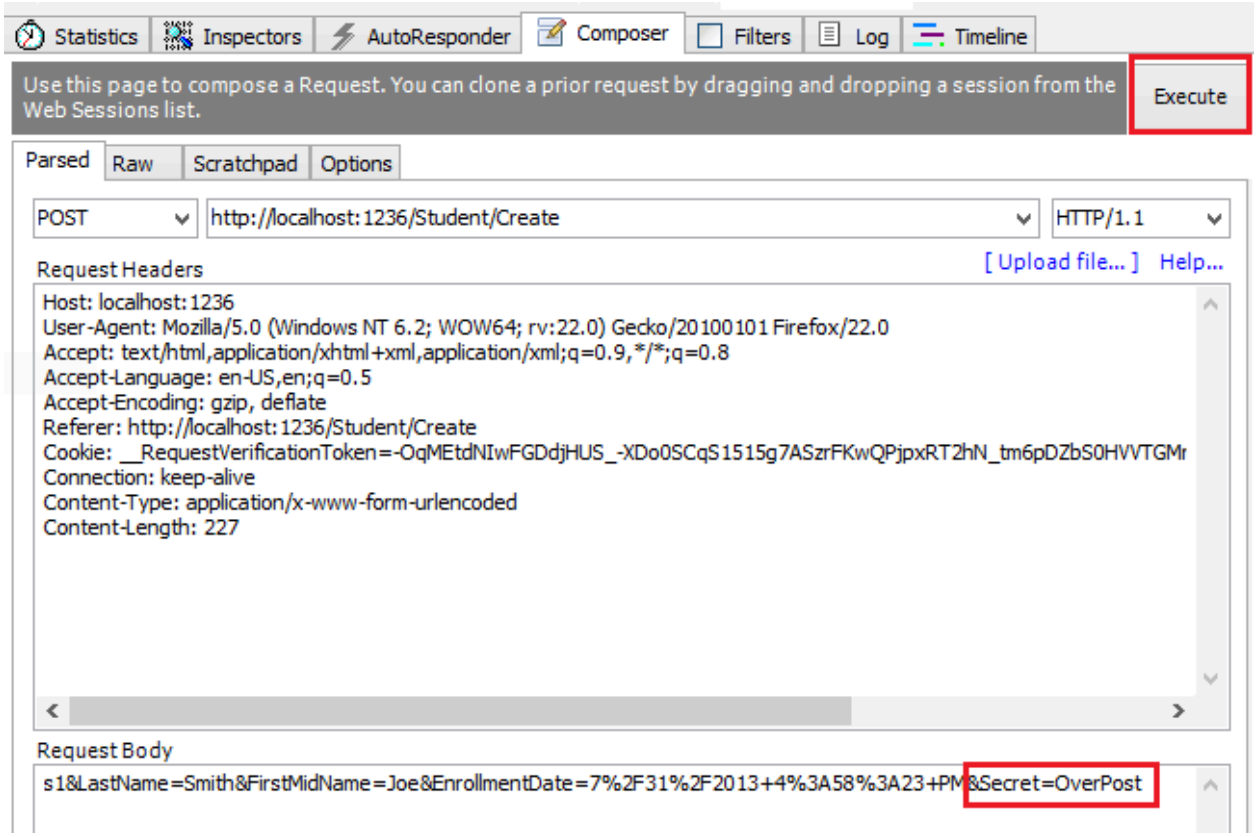
Security note about overposting

The `Bind` attribute that the scaffolded code includes on the `Create` method is one way to protect against overposting in create scenarios. For example, suppose the Student entity includes a `Secret` property that you don't want this web page to set.

C#

```
public class Student
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
    public string Secret { get; set; }
}
```

Even if you don't have a `Secret` field on the web page, a hacker could use a tool such as Fiddler, or write some JavaScript, to post a `Secret` form value. Without the `Bind` attribute limiting the fields that the model binder uses when it creates a Student instance, the model binder would pick up that `Secret` form value and use it to create the Student entity instance. Then whatever value the hacker specified for the `Secret` form field would be updated in your database. The following image shows the Fiddler tool adding the `Secret` field (with the value "OverPost") to the posted form values.



The value "OverPost" would then be successfully added to the `Secret` property of the inserted row, although you never intended that the web page be able to set that property.

You can prevent overposting in edit scenarios by reading the entity from the database first and then calling `TryUpdateModel`, passing in an explicit allowed properties list. That's the method used in these tutorials.

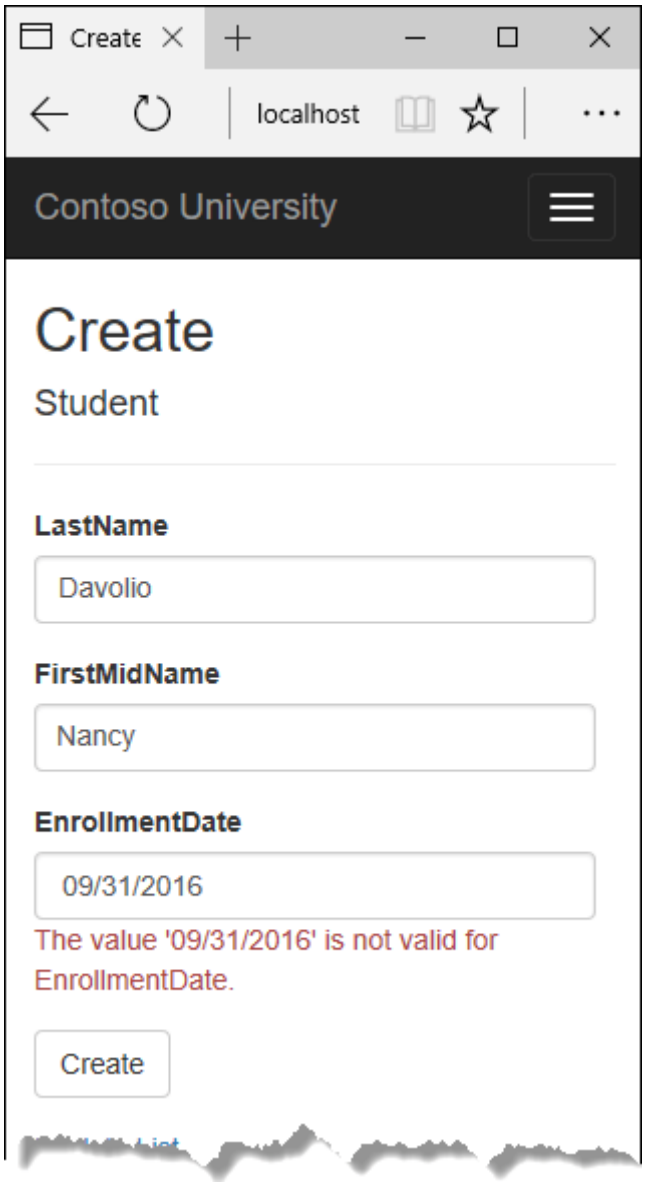
An alternative way to prevent overposting that's preferred by many developers is to use view models rather than entity classes with model binding. Include only the properties you want to update in the view model. Once the MVC model binder has finished, copy the view model properties to the entity instance, optionally using a tool such as AutoMapper. Use `_context.Entry` on the entity instance to set its state to `Unchanged`, and then set `Property("PropertyName").IsModified` to true on each entity property that's included in the view model. This method works in both edit and create scenarios.

Test the Create page

The code in `Views/Students/Create.cshtml` uses `label`, `input`, and `span` (for validation messages) tag helpers for each field.

Run the app, select the **Students** tab, and click **Create New**.

Enter names and a date. Try entering an invalid date if your browser lets you do that. (Some browsers force you to use a date picker.) Then click **Create** to see the error message.



This is server-side validation that you get by default; in a later tutorial you'll see how to add attributes that will generate code for client-side validation also. The following highlighted code shows the model validation check in the `Create` method.

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("EnrollmentDate,FirstMidName,LastName")] Student student)
{
    try
    {
        if (ModelState.IsValid)
        {
            _context.Add(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
    }
    catch (DbUpdateException /* ex */)
```

```
    {
        //Log the error (uncomment ex variable name and write a log.
        ModelState.AddModelError("", "Unable to save changes. " +
            "Try again, and if the problem persists " +
            "see your system administrator.");
    }
    return View(student);
}
```

Change the date to a valid value and click **Create** to see the new student appear in the **Index** page.

Update the Edit page

In `StudentController.cs`, the HttpGet `Edit` method (the one without the `HttpPost` attribute) uses the `FirstOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the `Details` method. You don't need to change this method.

Recommended HttpPost Edit code: Read and update

Replace the HttpPost Edit action method with the following code.

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
    var studentToUpdate = await _context.Students.FirstOrDefaultAsync(s =>
s.ID == id);
    if (await TryUpdateModelAsync<Student>(
        studentToUpdate,
        "",
        s => s.FirstMidName, s => s.LastName, s => s.EnrollmentDate))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
```

```
                "see your system administrator.");
        }
    }
    return View(studentToUpdate);
}
```

These changes implement a security best practice to prevent overposting. The scaffolder generated a `Bind` attribute and added the entity created by the model binder to the entity set with a `Modified` flag. That code isn't recommended for many scenarios because the `Bind` attribute clears out any pre-existing data in fields not listed in the `Include` parameter.

The new code reads the existing entity and calls `TryUpdateModel` to update fields in the retrieved entity based on user input in the posted form data. The Entity Framework's automatic change tracking sets the `Modified` flag on the fields that are changed by form input. When the `SaveChanges` method is called, the Entity Framework creates SQL statements to update the database row. Concurrency conflicts are ignored, and only the table columns that were updated by the user are updated in the database. (A later tutorial shows how to handle concurrency conflicts.)

As a best practice to prevent overposting, the fields that you want to be updateable by the **Edit** page are declared in the `TryUpdateModel` parameters. (The empty string preceding the list of fields in the parameter list is for a prefix to use with the form fields names.) Currently there are no extra fields that you're protecting, but listing the fields that you want the model binder to bind ensures that if you add fields to the data model in the future, they're automatically protected until you explicitly add them here.

As a result of these changes, the method signature of the HttpPost `Edit` method is the same as the HttpGet `Edit` method; therefore you've renamed the method `EditPost`.

Alternative HttpPost Edit code: Create and attach

The recommended HttpPost edit code ensures that only changed columns get updated and preserves data in properties that you don't want included for model binding. However, the read-first approach requires an extra database read, and can result in more complex code for handling concurrency conflicts. An alternative is to attach an entity created by the model binder to the EF context and mark it as modified. (Don't update your project with this code, it's only shown to illustrate an optional approach.)

C#

```
public async Task<IActionResult> Edit(int id,
[Bind("ID,EnrollmentDate,FirstMidName,LastName")] Student student)
```

```
{
    if (id != student.ID)
    {
        return NotFound();
    }
    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(student);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
    }
    return View(student);
}
```

You can use this approach when the web page UI includes all of the fields in the entity and can update any of them.

The scaffolded code uses the create-and-attach approach but only catches `DbUpdateConcurrencyException` exceptions and returns 404 error codes. The example shown catches any database update exception and displays an error message.

Entity States

The database context keeps track of whether entities in memory are in sync with their corresponding rows in the database, and this information determines what happens when you call the `SaveChanges` method. For example, when you pass a new entity to the `Add` method, that entity's state is set to `Added`. Then when you call the `SaveChanges` method, the database context issues a SQL INSERT command.

An entity may be in one of the following states:

- `Added`. The entity doesn't yet exist in the database. The `SaveChanges` method issues an INSERT statement.
- `Unchanged`. Nothing needs to be done with this entity by the `SaveChanges` method. When you read an entity from the database, the entity starts out with this status.

- `Modified`. Some or all of the entity's property values have been modified. The `SaveChanges` method issues an UPDATE statement.
- `Deleted`. The entity has been marked for deletion. The `SaveChanges` method issues a DELETE statement.
- `Detached`. The entity isn't being tracked by the database context.

In a desktop application, state changes are typically set automatically. You read an entity and make changes to some of its property values. This causes its entity state to automatically be changed to `Modified`. Then when you call `SaveChanges`, the Entity Framework generates a SQL UPDATE statement that updates only the actual properties that you changed.

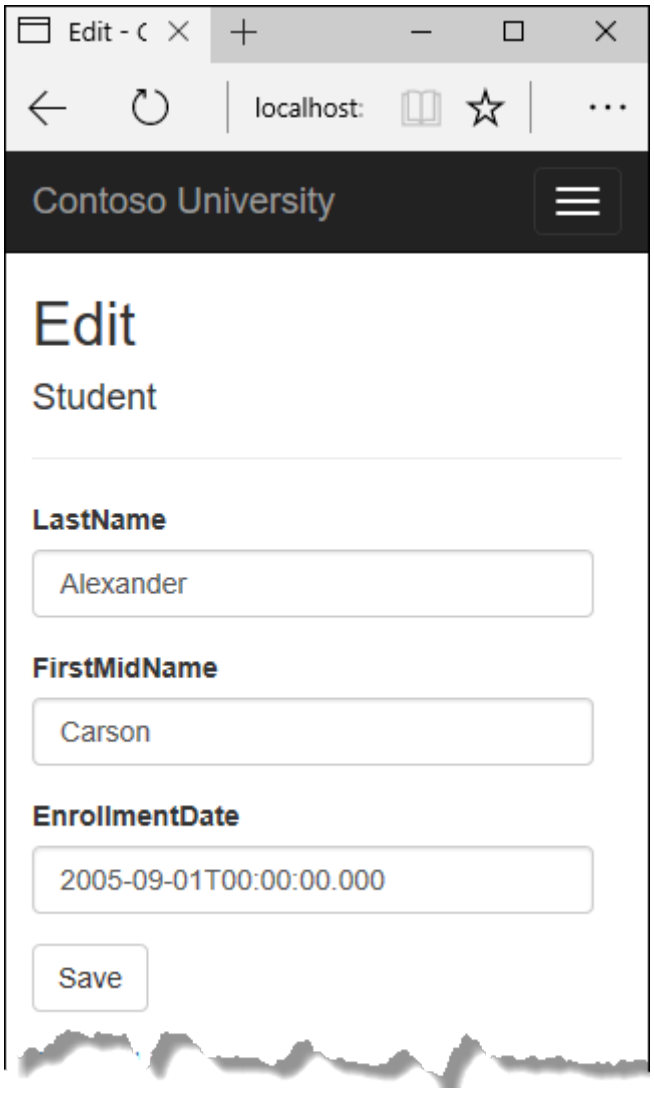
In a web app, the `DbContext` that initially reads an entity and displays its data to be edited is disposed after a page is rendered. When the HttpPost `Edit` action method is called, a new web request is made and you have a new instance of the `DbContext`. If you re-read the entity in that new context, you simulate desktop processing.

But if you don't want to do the extra read operation, you have to use the entity object created by the model binder. The simplest way to do this is to set the entity state to Modified as is done in the alternative HttpPost Edit code shown earlier. Then when you call `SaveChanges`, the Entity Framework updates all columns of the database row, because the context has no way to know which properties you changed.

If you want to avoid the read-first approach, but you also want the SQL UPDATE statement to update only the fields that the user actually changed, the code is more complex. You have to save the original values in some way (such as by using hidden fields) so that they're available when the HttpPost `Edit` method is called. Then you can create a Student entity using the original values, call the `Attach` method with that original version of the entity, update the entity's values to the new values, and then call `SaveChanges`.

Test the Edit page

Run the app, select the **Students** tab, then click an **Edit** hyperlink.



Change some of the data and click **Save**. The **Index** page opens and you see the changed data.

Update the Delete page

In `StudentController.cs`, the template code for the HttpGet `Delete` method uses the `FirstOrDefaultAsync` method to retrieve the selected Student entity, as you saw in the Details and Edit methods. However, to implement a custom error message when the call to `SaveChanges` fails, you'll add some functionality to this method and its corresponding view.

As you saw for update and create operations, delete operations require two action methods. The method that's called in response to a GET request displays a view that gives the user a chance to approve or cancel the delete operation. If the user approves it, a POST request is created. When that happens, the HttpPost `Delete` method is called and then that method actually performs the delete operation.

You'll add a try-catch block to the HttpPost `Delete` method to handle any errors that might occur when the database is updated. If an error occurs, the HttpPost Delete

method calls the HttpGet Delete method, passing it a parameter that indicates that an error has occurred. The HttpGet Delete method then redisplays the confirmation page along with the error message, giving the user an opportunity to cancel or try again.

Replace the HttpGet `Delete` action method with the following code, which manages error reporting.

C#

```
public async Task<IActionResult> Delete(int? id, bool? saveChangesError =
false)
{
    if (id == null)
    {
        return NotFound();
    }

    var student = await _context.Students
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (student == null)
    {
        return NotFound();
    }

    if (saveChangesError.GetValueOrDefault())
    {
        ViewData["ErrorMessage"] =
            "Delete failed. Try again, and if the problem persists " +
            "see your system administrator.";
    }

    return View(student);
}
```

This code accepts an optional parameter that indicates whether the method was called after a failure to save changes. This parameter is false when the HttpGet `Delete` method is called without a previous failure. When it's called by the HttpPost `Delete` method in response to a database update error, the parameter is true and an error message is passed to the view.

The read-first approach to HttpPost Delete

Replace the HttpPost `Delete` action method (named `DeleteConfirmed`) with the following code, which performs the actual delete operation and catches any database update errors.

C#

```
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var student = await _context.Students.FindAsync(id);
    if (student == null)
    {
        return RedirectToAction(nameof(Index));
    }

    try
    {
        _context.Students.Remove(student);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id,
saveChangesError = true });
    }
}
```

This code retrieves the selected entity, then calls the `Remove` method to set the entity's status to `Deleted`. When `SaveChanges` is called, a SQL DELETE command is generated.

The create-and-attach approach to HttpPost Delete

If improving performance in a high-volume application is a priority, you could avoid an unnecessary SQL query by instantiating a Student entity using only the primary key value and then setting the entity state to `Deleted`. That's all that the Entity Framework needs in order to delete the entity. (Don't put this code in your project; it's here just to illustrate an alternative.)

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    try
    {
        Student studentToDelete = new Student() { ID = id };
        _context.Entry(studentToDelete).State = EntityState.Deleted;
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
```

```
    }
    catch (DbUpdateException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { id = id,
saveChangesError = true });
    }
}
```

If the entity has related data that should also be deleted, make sure that cascade delete is configured in the database. With this approach to entity deletion, EF might not realize there are related entities to be deleted.

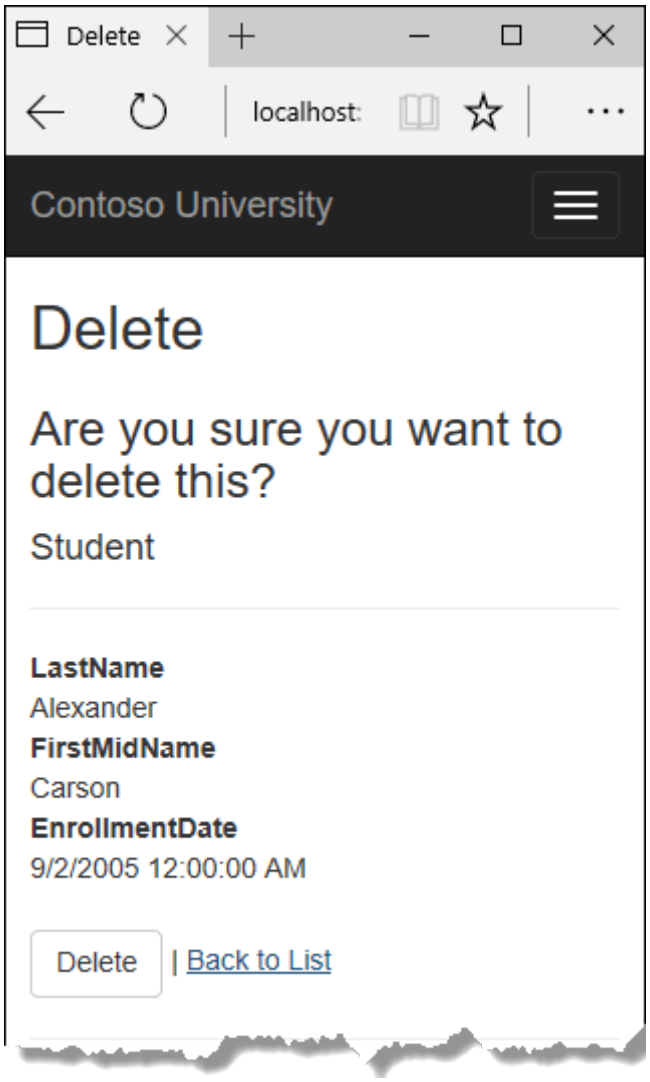
Update the Delete view

In `Views/Student/Delete.cshtml`, add an error message between the h2 heading and the h3 heading, as shown in the following example:

CSHTML

```
<h2>Delete</h2>
<p class="text-danger">@ViewData["ErrorMessage"]</p>
<h3>Are you sure you want to delete this?</h3>
```

Run the app, select the **Students** tab, and click a **Delete** hyperlink:



Click **Delete**. The Index page is displayed without the deleted student. (You'll see an example of the error handling code in action in the concurrency tutorial.)

Close database connections

To free up the resources that a database connection holds, the context instance must be disposed as soon as possible when you are done with it. The ASP.NET Core built-in dependency injection takes care of that task for you.

In `Startup.cs`, you call the AddDbContext extension method to provision the `DbContext` class in the ASP.NET Core DI container. That method sets the service lifetime to `Scoped` by default. `Scoped` means the context object lifetime coincides with the web request life time, and the `Dispose` method will be called automatically at the end of the web request.

Handle transactions

By default the Entity Framework implicitly implements transactions. In scenarios where you make changes to multiple rows or tables and then call `SaveChanges`, the Entity Framework automatically makes sure that either all of your changes succeed or they all fail. If some changes are done first and then an error happens, those changes are automatically rolled back. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

No-tracking queries

When a database context retrieves table rows and creates entity objects that represent them, by default it keeps track of whether the entities in memory are in sync with what's in the database. The data in memory acts as a cache and is used when you update an entity. This caching is often unnecessary in a web application because context instances are typically short-lived (a new one is created and disposed for each request) and the context that reads an entity is typically disposed before that entity is used again.

You can disable tracking of entity objects in memory by calling the `AsNoTracking` method. Typical scenarios in which you might want to do that include the following:

- During the context lifetime you don't need to update any entities, and you don't need EF to automatically load navigation properties with entities retrieved by separate queries. Frequently these conditions are met in a controller's HttpGet action methods.
- You are running a query that retrieves a large volume of data, and only a small portion of the returned data will be updated. It may be more efficient to turn off tracking for the large query, and run a query later for the few entities that need to be updated.
- You want to attach an entity in order to update it, but earlier you retrieved the same entity for a different purpose. Because the entity is already being tracked by the database context, you can't attach the entity that you want to change. One way to handle this situation is to call `AsNoTracking` on the earlier query.

For more information, see Tracking vs. No-Tracking.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized the Details page
- ✓ Updated the Create page
- ✓ Updated the Edit page
- ✓ Updated the Delete page
- ✓ Closed database connections

Advance to the next tutorial to learn how to expand the functionality of the **Index** page by adding sorting, filtering, and paging.

Next: Sorting, filtering, and paging

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

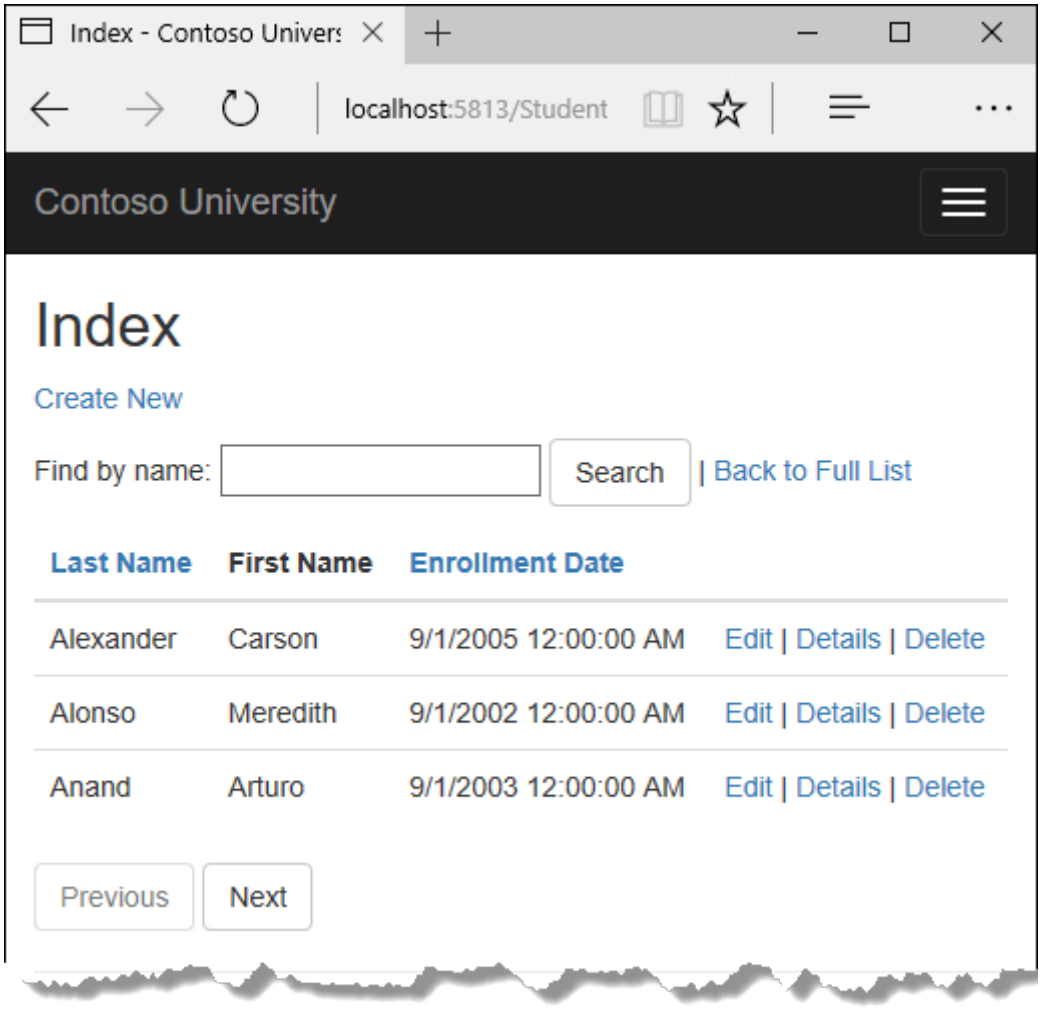
Provide product feedback

Tutorial: Add sorting, filtering, and paging - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you implemented a set of web pages for basic CRUD operations for Student entities. In this tutorial you'll add sorting, filtering, and paging functionality to the Students Index page. You'll also create a page that does simple grouping.

The following illustration shows what the page will look like when you're done. The column headings are links that the user can click to sort by that column. Clicking a column heading repeatedly toggles between ascending and descending sort order.



In this tutorial, you:

- ✓ Add column sort links
- ✓ Add a Search box
- ✓ Add paging to Students Index
- ✓ Add paging to Index method
- ✓ Add paging links

✔ Create an About page

Prerequisites

- Implement CRUD Functionality

Add column sort links

To add sorting to the Student Index page, you'll change the `Index` method of the Students controller and add code to the Student Index view.

Add sorting Functionality to the Index method

In `StudentsController.cs`, replace the `Index` method with the following code:

C#

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

This code receives a `sortOrder` parameter from the query string in the URL. The query string value is provided by ASP.NET Core MVC as a parameter to the action method. The parameter will be a string that's either "Name" or "Date", optionally followed by an

underscore and the string "desc" to specify descending order. The default sort order is ascending.

The first time the Index page is requested, there's no query string. The students are displayed in ascending order by last name, which is the default as established by the fall-through case in the `switch` statement. When the user clicks a column heading hyperlink, the appropriate `sortOrder` value is provided in the query string.

The two `ViewData` elements (NameSortParm and DateSortParm) are used by the view to configure the column heading hyperlinks with the appropriate query string values.

C#

```
public async Task<IActionResult> Index(string sortOrder)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in _context.Students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

These are ternary statements. The first one specifies that if the `sortOrder` parameter is null or empty, NameSortParm should be set to "name_desc"; otherwise, it should be set to an empty string. These two statements enable the view to set the column heading hyperlinks as follows:

Expand table

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name ascending	descending	ascending

Current sort order	Last Name Hyperlink	Date Hyperlink
Last Name descending	ascending	ascending
Date ascending	ascending	descending
Date descending	ascending	ascending

The method uses LINQ to Entities to specify the column to sort by. The code creates an `IQueryable` variable before the switch statement, modifies it in the switch statement, and calls the `ToListAsync` method after the `switch` statement. When you create and modify `IQueryable` variables, no query is sent to the database. The query isn't executed until you convert the `IQueryable` object into a collection by calling a method such as `ToListAsync`. Therefore, this code results in a single query that's not executed until the `return View` statement.

This code could get verbose with a large number of columns. The last tutorial in this series shows how to write code that lets you pass the name of the `OrderBy` column in a string variable.

Add column heading hyperlinks to the Student Index view

Replace the code in `Views/Students/Index.cshtml`, with the following code to add column heading hyperlinks. The changed lines are highlighted.

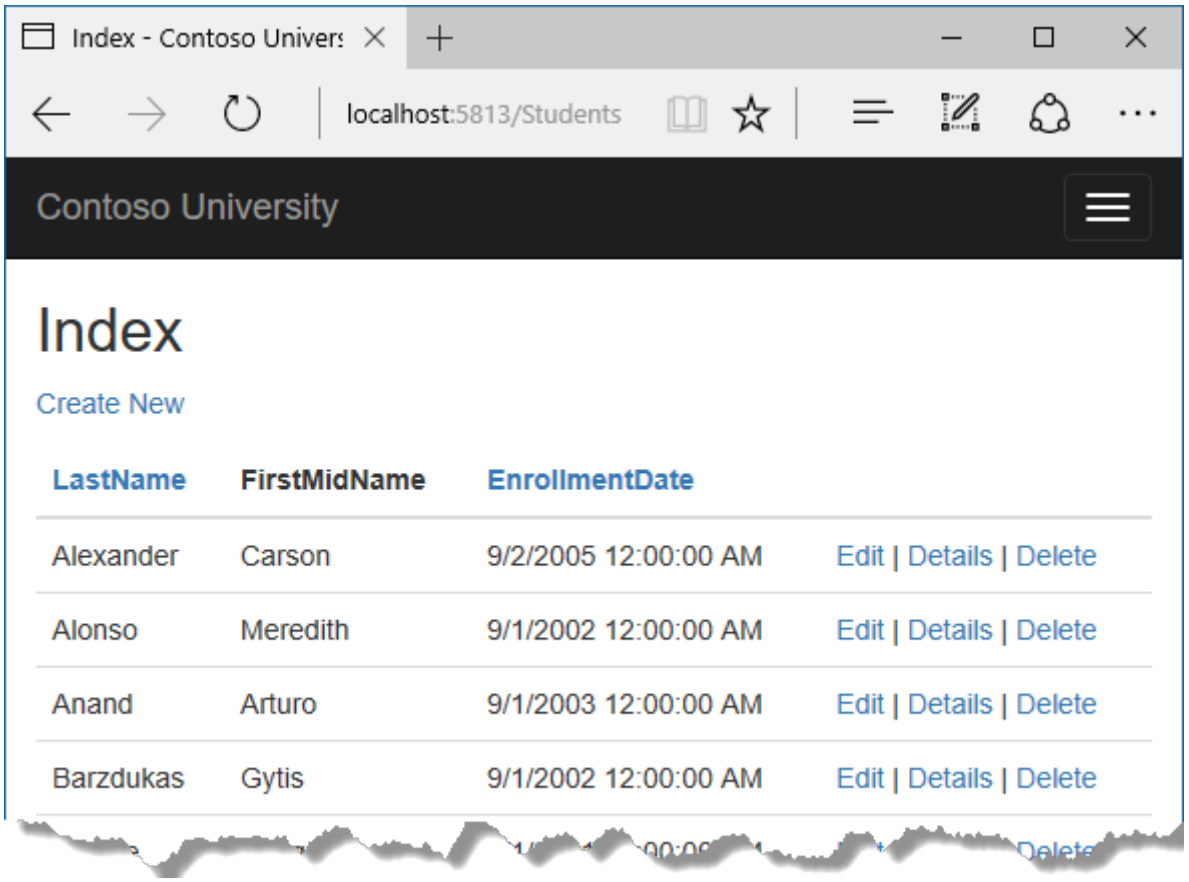
CSHTML

```
@model IEnumerable<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]">@Html.DisplayNameFor(model =>
model.LastName)</a>
                </th>
```

```
                <th>
                    @Html.DisplayNameFor(model => model.FirstMidName)
                </th>
                <th>
                    <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]">@Html.DisplayNameFor(model =>
model.EnrollmentDate)</a>
                </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FirstMidName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.ID">Details</a>
|
                <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

This code uses the information in `ViewData` properties to set up hyperlinks with the appropriate query string values.

Run the app, select the **Students** tab, and click the **Last Name** and **Enrollment Date** column headings to verify that sorting works.



Add a Search box

To add filtering to the Students Index page, you'll add a text box and a submit button to the view and make corresponding changes in the `Index` method. The text box will let you enter a string to search for in the first name and last name fields.

Add filtering functionality to the Index method

In `StudentsController.cs`, replace the `Index` method with the following code (the changes are highlighted).

C#

```
public async Task<IActionResult> Index(string sortOrder, string
searchString)
{
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
```

```
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(await students.AsNoTracking().ToListAsync());
}
```

You've added a `searchString` parameter to the `Index` method. The search string value is received from a text box that you'll add to the Index view. You've also added to the LINQ statement a where clause that selects only students whose first name or last name contains the search string. The statement that adds the where clause is executed only if there's a value to search for.

> ⓘ **Note**
>
> Here you are calling the `Where` method on an `IQueryable` object, and the filter will be processed on the server. In some scenarios you might be calling the `Where` method as an extension method on an in-memory collection. (For example, suppose you change the reference to `_context.Students` so that instead of an EF `DbSet` it references a repository method that returns an `IEnumerable` collection.) The result would normally be the same but in some cases may be different.
>
> For example, the .NET Framework implementation of the `Contains` method performs a case-sensitive comparison by default, but in SQL Server this is determined by the collation setting of the SQL Server instance. That setting defaults to case-insensitive. You could call the `ToUpper` method to make the test explicitly case-insensitive: *Where(s => s.LastName.ToUpper().Contains(searchString.ToUpper())*. That would ensure that results stay the same if you change the code later to use a repository which returns an `IEnumerable` collection instead of an `IQueryable` object. (When you call the `Contains` method on an `IEnumerable` collection, you get the

.NET Framework implementation; when you call it on an `IQueryable` object, you get the database provider implementation.) However, there's a performance penalty for this solution. The `ToUpper` code would put a function in the WHERE clause of the TSQL SELECT statement. That would prevent the optimizer from using an index. Given that SQL is mostly installed as case-insensitive, it's best to avoid the `ToUpper` code until you migrate to a case-sensitive data store.

Add a Search Box to the Student Index View

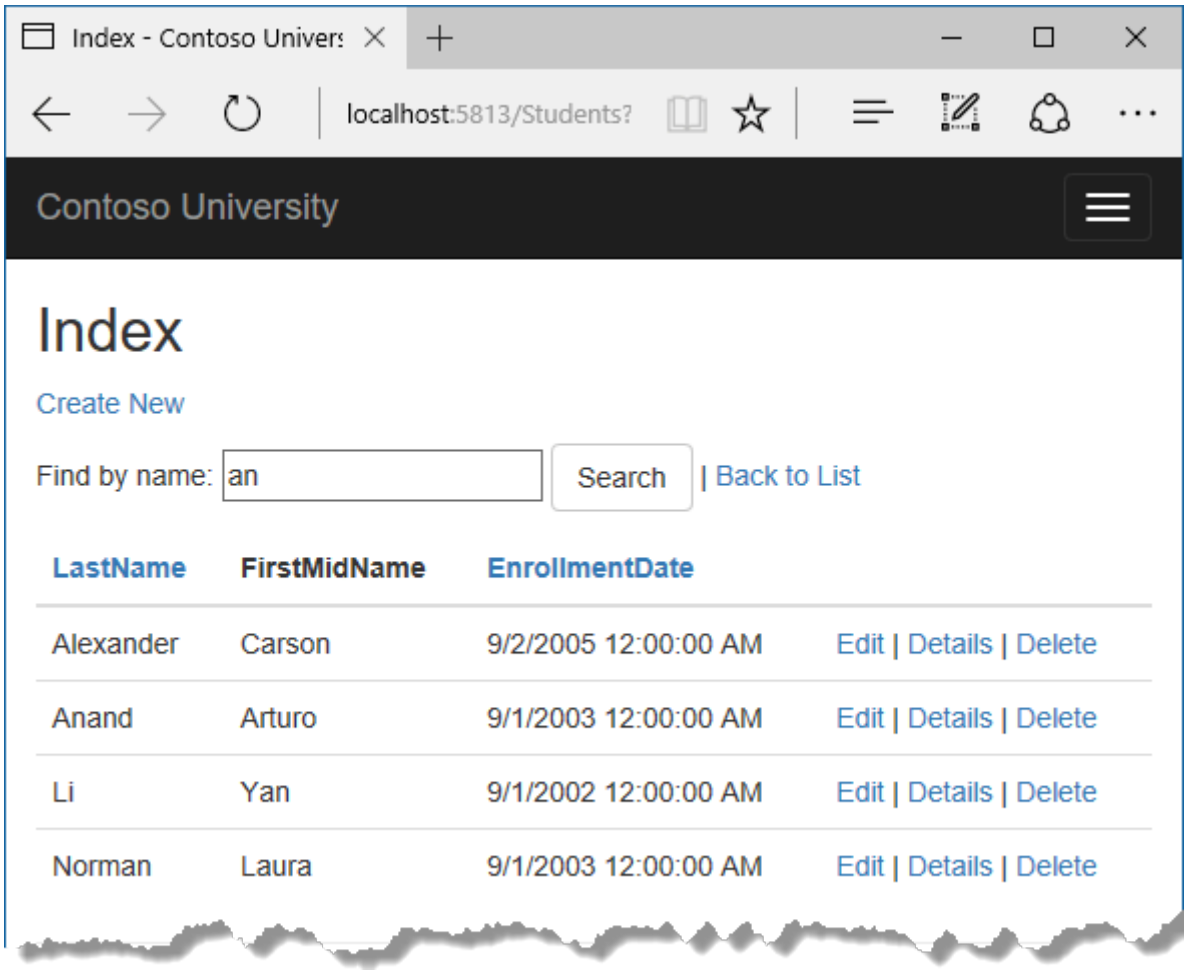
In `Views/Student/Index.cshtml`, add the highlighted code immediately before the opening table tag in order to create a caption, a text box, and a **Search** button.

CSHTML

```
<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString"
value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
```

This code uses the `<form>` tag helper to add the search text box and button. By default, the `<form>` tag helper submits form data with a POST, which means that parameters are passed in the HTTP message body and not in the URL as query strings. When you specify HTTP GET, the form data is passed in the URL as query strings, which enables users to bookmark the URL. The W3C guidelines recommend that you should use GET when the action doesn't result in an update.

Run the app, select the **Students** tab, enter a search string, and click Search to verify that filtering is working.



Notice that the URL contains the search string.

```
HTML
```

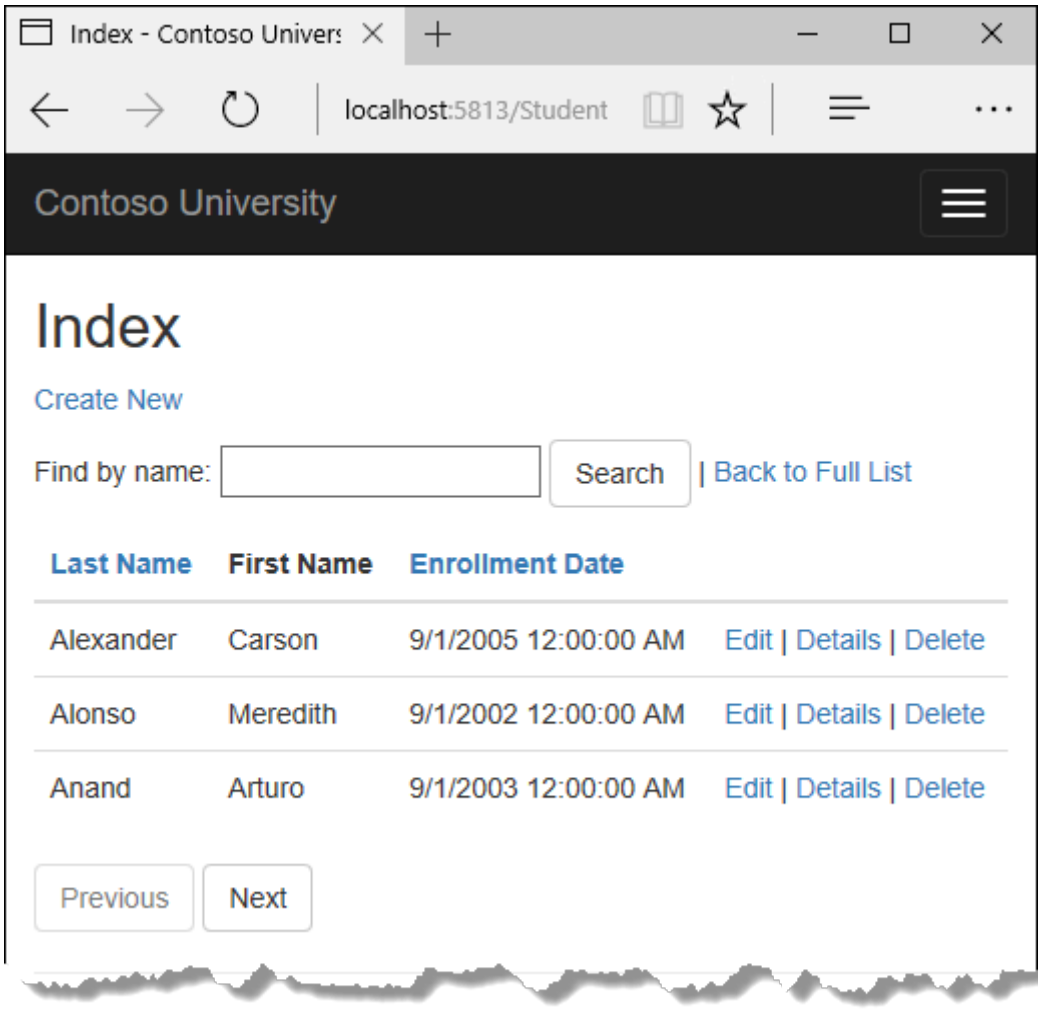
```
http://localhost:5813/Students?SearchString=an
```

If you bookmark this page, you'll get the filtered list when you use the bookmark. Adding `method="get"` to the `form` tag is what caused the query string to be generated.

At this stage, if you click a column heading sort link you'll lose the filter value that you entered in the **Search** box. You'll fix that in the next section.

Add paging to Students Index

To add paging to the Students Index page, you'll create a `PaginatedList` class that uses `Skip` and `Take` statements to filter data on the server instead of always retrieving all rows of the table. Then you'll make additional changes in the `Index` method and add paging buttons to the `Index` view. The following illustration shows the paging buttons.



In the project folder, create `PaginatedList.cs`, and then replace the template code with the following code.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity
{
    public class PaginatedList<T> : List<T>
    {
        public int PageIndex { get; private set; }
        public int TotalPages { get; private set; }

        public PaginatedList(List<T> items, int count, int pageIndex, int
pageSize)
        {
            PageIndex = pageIndex;
            TotalPages = (int)Math.Ceiling(count / (double)pageSize);

            this.AddRange(items);
        }
```

```
        public bool HasPreviousPage => PageIndex > 1;

        public bool HasNextPage => PageIndex < TotalPages;

        public static async Task<PaginatedList<T>> CreateAsync(IQueryable<T>
source, int pageIndex, int pageSize)
        {
            var count = await source.CountAsync();
            var items = await source.Skip((pageIndex - 1) *
pageSize).Take(pageSize).ToListAsync();
            return new PaginatedList<T>(items, count, pageIndex, pageSize);
        }
    }
}
```

The `CreateAsync` method in this code takes page size and page number and applies the appropriate `Skip` and `Take` statements to the `IQueryable`. When `ToListAsync` is called on the `IQueryable`, it will return a List containing only the requested page. The properties `HasPreviousPage` and `HasNextPage` can be used to enable or disable **Previous** and **Next** paging buttons.

A `CreateAsync` method is used instead of a constructor to create the `PaginatedList<T>` object because constructors can't run asynchronous code.

Add paging to Index method

In `StudentsController.cs`, replace the `Index` method with the following code.

C#

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
{
    ViewData["CurrentSort"] = sortOrder;
    ViewData["NameSortParm"] = String.IsNullOrEmpty(sortOrder) ? "name_desc"
: "";
    ViewData["DateSortParm"] = sortOrder == "Date" ? "date_desc" : "Date";
```

```
    if (searchString != null)
    {
        pageNumber = 1;
    }
    else
    {
        searchString = currentFilter;
    }

    ViewData["CurrentFilter"] = searchString;

    var students = from s in _context.Students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.LastName.Contains(searchString)
                               || s.FirstMidName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    int pageSize = 3;
    return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1,
pageSize));
}
```

This code adds a page number parameter, a current sort order parameter, and a current filter parameter to the method signature.

C#

```
public async Task<IActionResult> Index(
    string sortOrder,
    string currentFilter,
    string searchString,
    int? pageNumber)
```

The first time the page is displayed, or if the user hasn't clicked a paging or sorting link, all the parameters will be null. If a paging link is clicked, the page variable will contain the page number to display.

The `ViewData` element named CurrentSort provides the view with the current sort order, because this must be included in the paging links in order to keep the sort order the same while paging.

The `ViewData` element named CurrentFilter provides the view with the current filter string. This value must be included in the paging links in order to maintain the filter settings during paging, and it must be restored to the text box when the page is redisplayed.

If the search string is changed during paging, the page has to be reset to 1, because the new filter can result in different data to display. The search string is changed when a value is entered in the text box and the Submit button is pressed. In that case, the `searchString` parameter isn't null.

C#

```
if (searchString != null)
{
    pageNumber = 1;
}
else
{
    searchString = currentFilter;
}
```

At the end of the `Index` method, the `PaginatedList.CreateAsync` method converts the student query to a single page of students in a collection type that supports paging. That single page of students is then passed to the view.

C#

```
return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(), pageNumber ?? 1,
pageSize));
```

The `PaginatedList.CreateAsync` method takes a page number. The two question marks represent the null-coalescing operator. The null-coalescing operator defines a default value for a nullable type; the expression `(pageNumber ?? 1)` means return the value of `pageNumber` if it has a value, or return 1 if `pageNumber` is null.

Add paging links

In `Views/Students/Index.cshtml`, replace the existing code with the following code. The changes are highlighted.

CSHTML

```
@model PaginatedList<ContosoUniversity.Models.Student>

@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-action="Index" method="get">
    <div class="form-actions no-color">
        <p>
            Find by name: <input type="text" name="SearchString"
value="@ViewData["CurrentFilter"]" />
            <input type="submit" value="Search" class="btn btn-default" /> |
            <a asp-action="Index">Back to Full List</a>
        </p>
    </div>
</form>

<table class="table">
    <thead>
        <tr>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["NameSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Last Name</a>
            </th>
            <th>
                First Name
            </th>
            <th>
                <a asp-action="Index" asp-route-
sortOrder="@ViewData["DateSortParm"]" asp-route-
currentFilter="@ViewData["CurrentFilter"]">Enrollment Date</a>
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
```

```
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.EnrollmentDate)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

```
@{
    var prevDisabled = !Model.HasPreviousPage ? "disabled" : "";
    var nextDisabled = !Model.HasNextPage ? "disabled" : "";
}

<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
    Previous
</a>
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex + 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @nextDisabled">
    Next
</a>
```

The `@model` statement at the top of the page specifies that the view now gets a `PaginatedList<T>` object instead of a `List<T>` object.

The column header links use the query string to pass the current search string to the controller so that the user can sort within filter results:

HTML

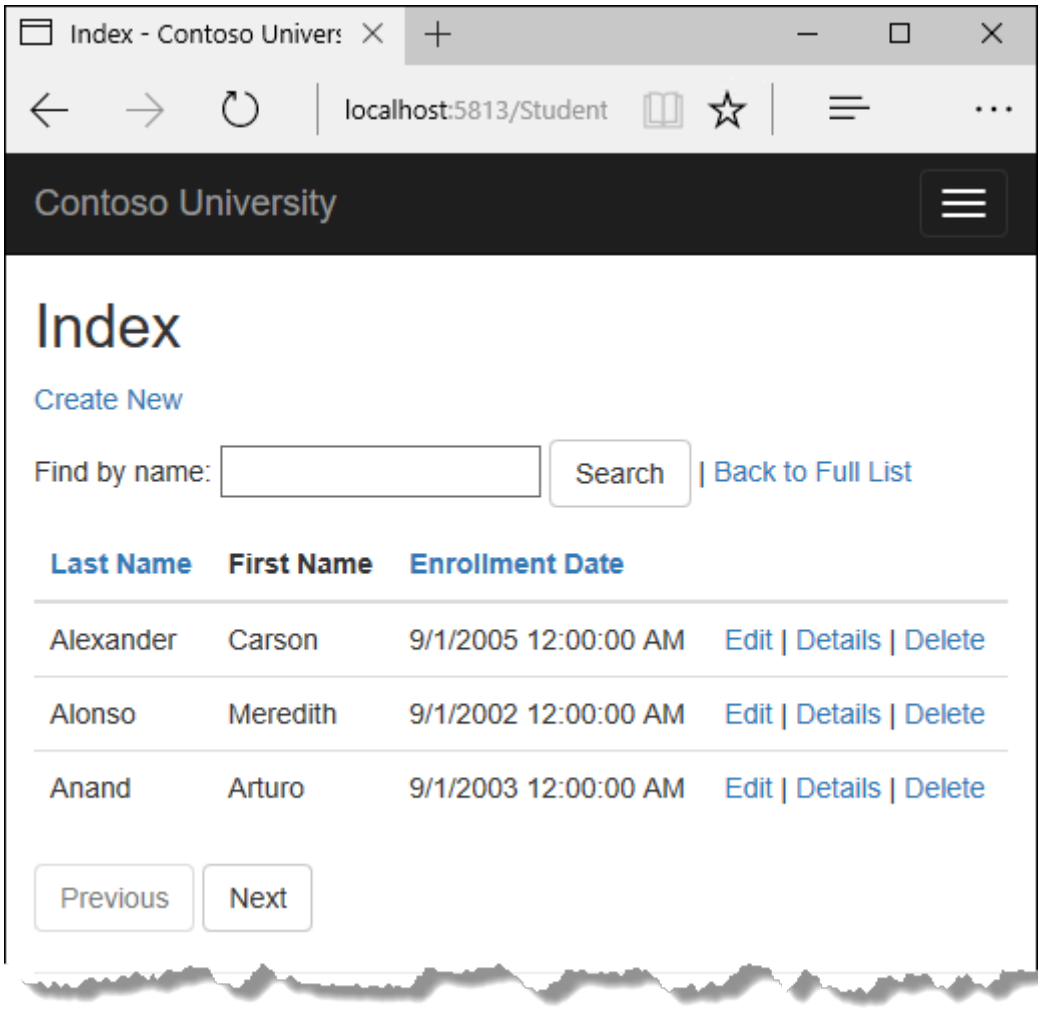
```
<a asp-action="Index" asp-route-sortOrder="@ViewData["DateSortParm"]" asp-
route-currentFilter ="@ViewData["CurrentFilter"]">Enrollment Date</a>
```

The paging buttons are displayed by tag helpers:

HTML

```
<a asp-action="Index"
   asp-route-sortOrder="@ViewData["CurrentSort"]"
   asp-route-pageNumber="@(Model.PageIndex - 1)"
   asp-route-currentFilter="@ViewData["CurrentFilter"]"
   class="btn btn-default @prevDisabled">
   Previous
</a>
```

Run the app and go to the Students page.



Click the paging links in different sort orders to make sure paging works. Then enter a search string and try paging again to verify that paging also works correctly with sorting and filtering.

Create an About page

For the Contoso University website's **About** page, you'll display how many students have enrolled for each enrollment date. This requires grouping and simple calculations

on the groups. To accomplish this, you'll do the following:

- Create a view model class for the data that you need to pass to the view.
- Create the About method in the Home controller.
- Create the About view.

Create the view model

Create a *SchoolViewModels* folder in the *Models* folder.

In the new folder, add a class file `EnrollmentDateGroup.cs` and replace the template code with the following code:

C#

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class EnrollmentDateGroup
    {
        [DataType(DataType.Date)]
        public DateTime? EnrollmentDate { get; set; }

        public int StudentCount { get; set; }
    }
}
```

Modify the Home Controller

In `HomeController.cs`, add the following using statements at the top of the file:

C#

```
using Microsoft.EntityFrameworkCore;
using ContosoUniversity.Data;
using ContosoUniversity.Models.SchoolViewModels;
using Microsoft.Extensions.Logging;
```

Add a class variable for the database context immediately after the opening curly brace for the class, and get an instance of the context from ASP.NET Core DI:

C#

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;
    private readonly SchoolContext _context;

    public HomeController(ILogger<HomeController> logger, SchoolContext
context)
    {
        _logger = logger;
        _context = context;
    }
```

Add an `About` method with the following code:

C#

```
public async Task<ActionResult> About()
{
    IQueryable<EnrollmentDateGroup> data =
        from student in _context.Students
        group student by student.EnrollmentDate into dateGroup
        select new EnrollmentDateGroup()
        {
            EnrollmentDate = dateGroup.Key,
            StudentCount = dateGroup.Count()
        };
    return View(await data.AsNoTracking().ToListAsync());
}
```

The LINQ statement groups the student entities by enrollment date, calculates the number of entities in each group, and stores the results in a collection of `EnrollmentDateGroup` view model objects.

Create the About View

Add a `Views/Home/About.cshtml` file with the following code:

CSHTML

```
@model
IEnumerable<ContosoUniversity.Models.SchoolViewModels.EnrollmentDateGroup>

@{
    ViewData["Title"] = "Student Body Statistics";
}

<h2>Student Body Statistics</h2>
```

```
<table>
    <tr>
        <th>
            Enrollment Date
        </th>
        <th>
            Students
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.EnrollmentDate)
            </td>
            <td>
                @item.StudentCount
            </td>
        </tr>
    }
</table>
```

Run the app and go to the About page. The count of students for each enrollment date is displayed in a table.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Added column sort links
- ✓ Added a Search box
- ✓ Added paging to Students Index
- ✓ Added paging to Index method
- ✓ Added paging links
- ✓ Created an About page

Advance to the next tutorial to learn how to handle data model changes by using migrations.

Next: Handle data model changes

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Tutorial: Part 5, apply migrations to the Contoso University sample

Article • 07/28/2023

In this tutorial, you start using the EF Core migrations feature for managing data model changes. In later tutorials, you'll add more migrations as you change the data model.

In this tutorial, you:

- ✓ Learn about migrations
- ✓ Create an initial migration
- ✓ Examine Up and Down methods
- ✓ Learn about the data model snapshot
- ✓ Apply the migration

Prerequisites

- Sorting, filtering, and paging

About migrations

When you develop a new application, your data model changes frequently, and each time the model changes, it gets out of sync with the database. You started these tutorials by configuring the Entity Framework to create the database if it doesn't exist. Then each time you change the data model -- add, remove, or change entity classes or change your DbContext class -- you can delete the database and EF creates a new one that matches the model, and seeds it with test data.

This method of keeping the database in sync with the data model works well until you deploy the application to production. When the application is running in production it's usually storing data that you want to keep, and you don't want to lose everything each time you make a change such as adding a new column. The EF Core Migrations feature solves this problem by enabling EF to update the database schema instead of creating a new database.

To work with migrations, you can use the **Package Manager Console** (PMC) or the CLI. These tutorials show how to use CLI commands. Information about the PMC is at the end of this tutorial.

Drop the database

Install EF Core tools as a global tool and delete the database:

.NET CLI

```
dotnet tool install --global dotnet-ef
dotnet ef database drop
```

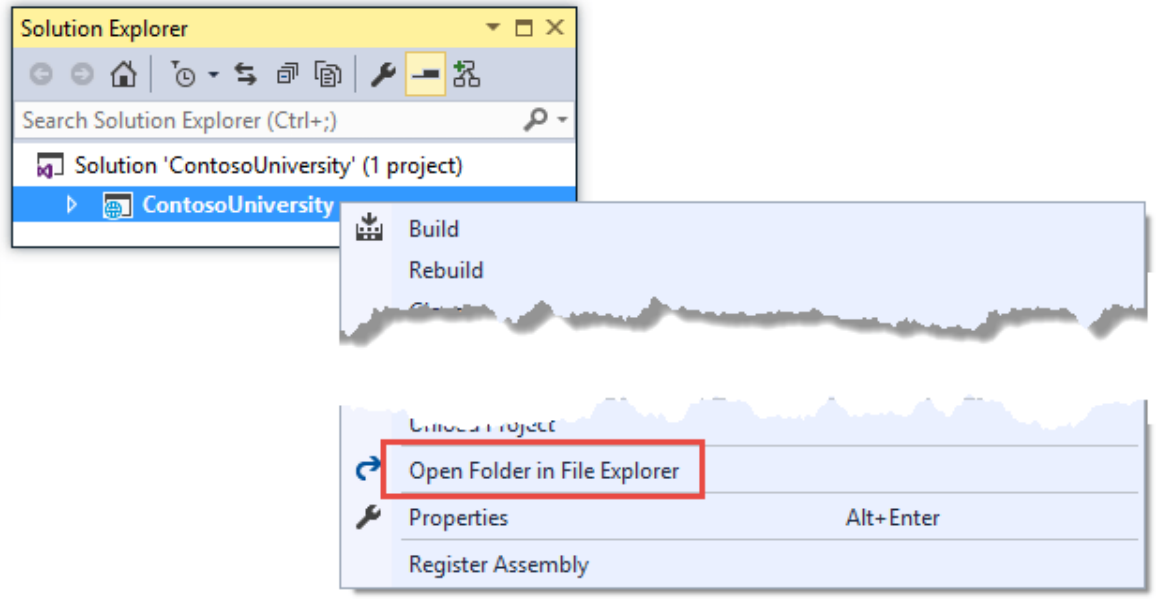
> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

The following section explains how to run CLI commands.

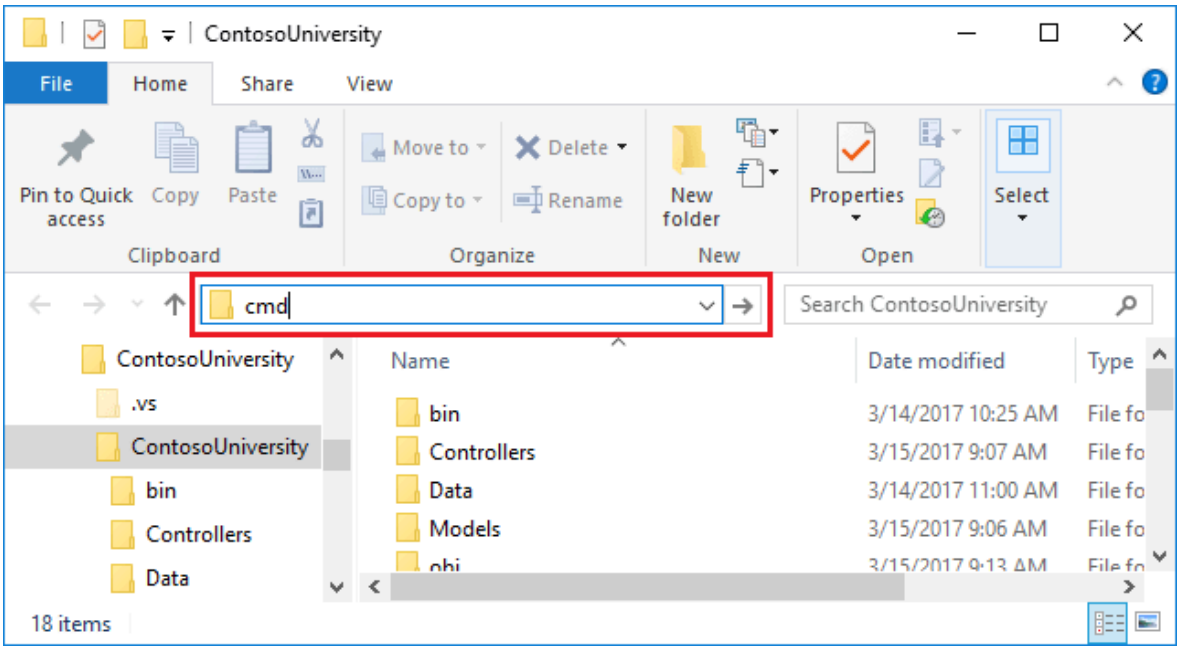
Create an initial migration

Save your changes and build the project. Then open a command window and navigate to the project folder. Here's a quick way to do that:

- In **Solution Explorer**, right-click the project and choose **Open Folder in File Explorer** from the context menu.



- Enter "cmd" in the address bar and press Enter.



Enter the following command in the command window:

.NET CLI

```
dotnet ef migrations add InitialCreate
```

In the preceding commands, output similar to the following is displayed:

Console

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Done. To undo this action, use 'ef migrations remove'
```

If you see an error message "*cannot access the file ... ContosoUniversity.dll because it is being used by another process.*", find the IIS Express icon in the Windows System Tray, and right-click it, then click **ContosoUniversity** > **Stop Site**.

Examine Up and Down methods

When you executed the `migrations add` command, EF generated the code that will create the database from scratch. This code is in the *Migrations* folder, in the file named `<timestamp>_InitialCreate.cs`. The `Up` method of the `InitialCreate` class creates the database tables that correspond to the data model entity sets, and the `Down` method deletes them, as shown in the following example.

C#

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Course",
            columns: table => new
            {
                CourseID = table.Column<int>(nullable: false),
                Credits = table.Column<int>(nullable: false),
                Title = table.Column<string>(nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Course", x => x.CourseID);
            });

        // Additional code not shown
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Enrollment");
        // Additional code not shown
    }
}
```

Migrations calls the `Up` method to implement the data model changes for a migration. When you enter a command to roll back the update, Migrations calls the `Down` method.

This code is for the initial migration that was created when you entered the `migrations add InitialCreate` command. The migration name parameter ("InitialCreate" in the example) is used for the file name and can be whatever you want. It's best to choose a word or phrase that summarizes what is being done in the migration. For example, you might name a later migration "AddDepartmentTable".

If you created the initial migration when the database already exists, the database creation code is generated but it doesn't have to run because the database already matches the data model. When you deploy the app to another environment where the database doesn't exist yet, this code will run to create your database, so it's a good idea to test it first. That's why you dropped the database earlier -- so that migrations can create a new one from scratch.

The data model snapshot

Migrations creates a *snapshot* of the current database schema in `Migrations/SchoolContextModelSnapshot.cs`. When you add a migration, EF determines what changed by comparing the data model to the snapshot file.

Use the dotnet ef migrations remove command to remove a migration. `dotnet ef migrations remove` deletes the migration and ensures the snapshot is correctly reset. If `dotnet ef migrations remove` fails, use `dotnet ef migrations remove -v` to get more information on the failure.

See EF Core Migrations in Team Environments for more information about how the snapshot file is used.

Apply the migration

In the command window, enter the following command to create the database and tables in it.

.NET CLI

```
dotnet ef database update
```

The output from the command is similar to the `migrations add` command, except that you see logs for the SQL commands that set up the database. Most of the logs are omitted in the following sample output. If you prefer not to see this level of detail in log messages, you can change the log level in the `appsettings.Development.json` file. For more information, see Logging in .NET Core and ASP.NET Core.

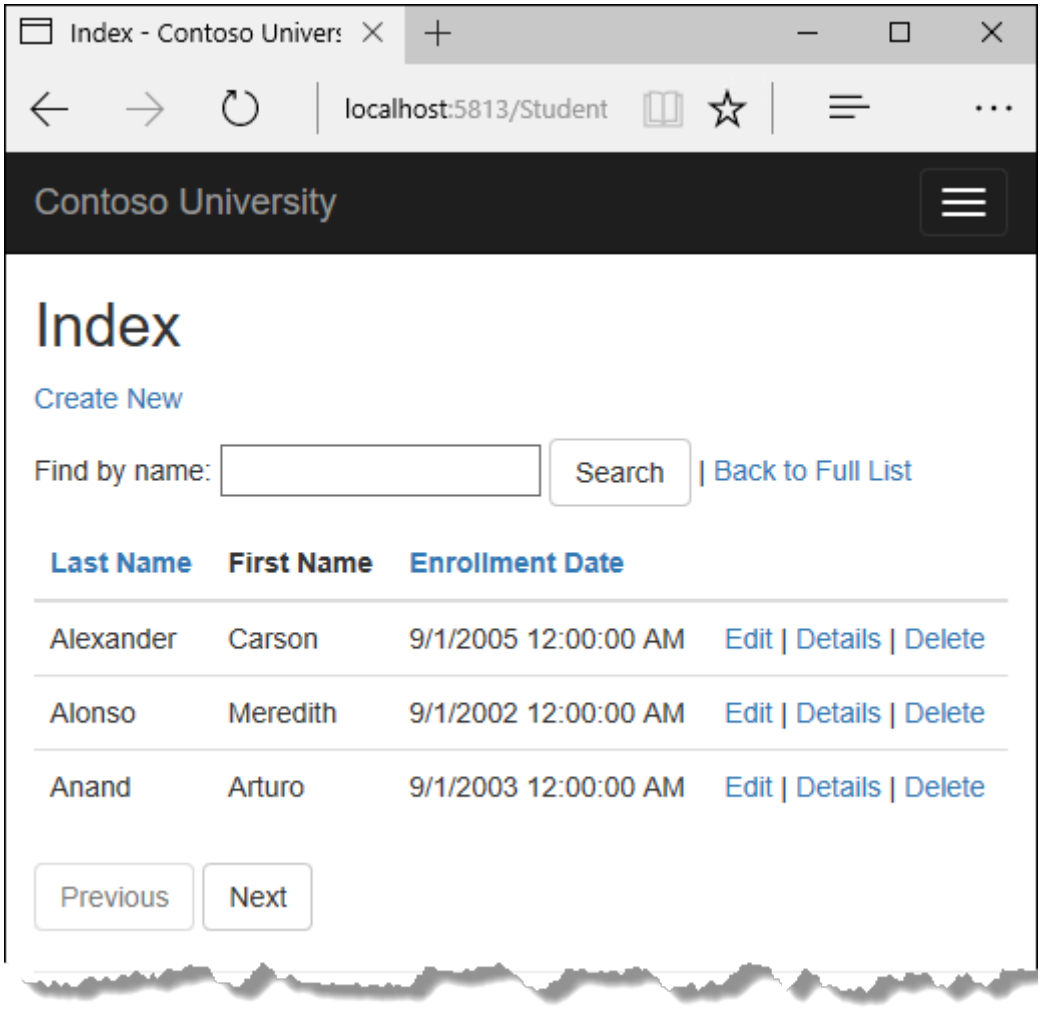
text

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core initialized 'SchoolContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (274ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      CREATE DATABASE [ContosoUniversity2];
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (60ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
      IF SERVERPROPERTY('EngineEdition') <> 5
      BEGIN
          ALTER DATABASE [ContosoUniversity2] SET READ_COMMITTED_SNAPSHOT
ON;
      END;
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (15ms) [Parameters=[], CommandType='Text',
```

```
CommandTimeout='30']
      CREATE TABLE [__EFMigrationsHistory] (
          [MigrationId] nvarchar(150) NOT NULL,
          [ProductVersion] nvarchar(32) NOT NULL,
          CONSTRAINT [PK___EFMigrationsHistory] PRIMARY KEY ([MigrationId])
      );

<logs omitted for brevity>

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
      INSERT INTO [__EFMigrationsHistory] ([MigrationId], [ProductVersion])
      VALUES (N'20190327172701_InitialCreate', N'5.0-rtm');
Done.
```

Use **SQL Server Object Explorer** to inspect the database as you did in the first tutorial. You'll notice the addition of an __EFMigrationsHistory table that keeps track of which migrations have been applied to the database. View the data in that table and you'll see one row for the first migration. (The last log in the preceding CLI output example shows the INSERT statement that creates this row.)

Run the application to verify that everything still works the same as before.



Compare CLI and PMC

The EF tooling for managing migrations is available from .NET CLI commands or from PowerShell cmdlets in the Visual Studio **Package Manager Console** (PMC) window. This tutorial shows how to use the CLI, but you can use the PMC if you prefer.

The EF commands for the PMC commands are in the Microsoft.EntityFrameworkCore.Tools package. This package is included in the Microsoft.AspNetCore.App metapackage, so you don't need to add a package reference if your app has a package reference for `Microsoft.AspNetCore.App`.

Important: This isn't the same package as the one you install for the CLI by editing the `.csproj` file. The name of this one ends in `Tools`, unlike the CLI package name which ends in `Tools.DotNet`.

For more information about the CLI commands, see .NET CLI.

For more information about the PMC commands, see Package Manager Console (Visual Studio).

Get the code

Download or view the completed application.

Next step

Advance to the next tutorial to begin looking at more advanced topics about expanding the data model. Along the way you'll create and apply additional migrations.

Create and apply additional migrations

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

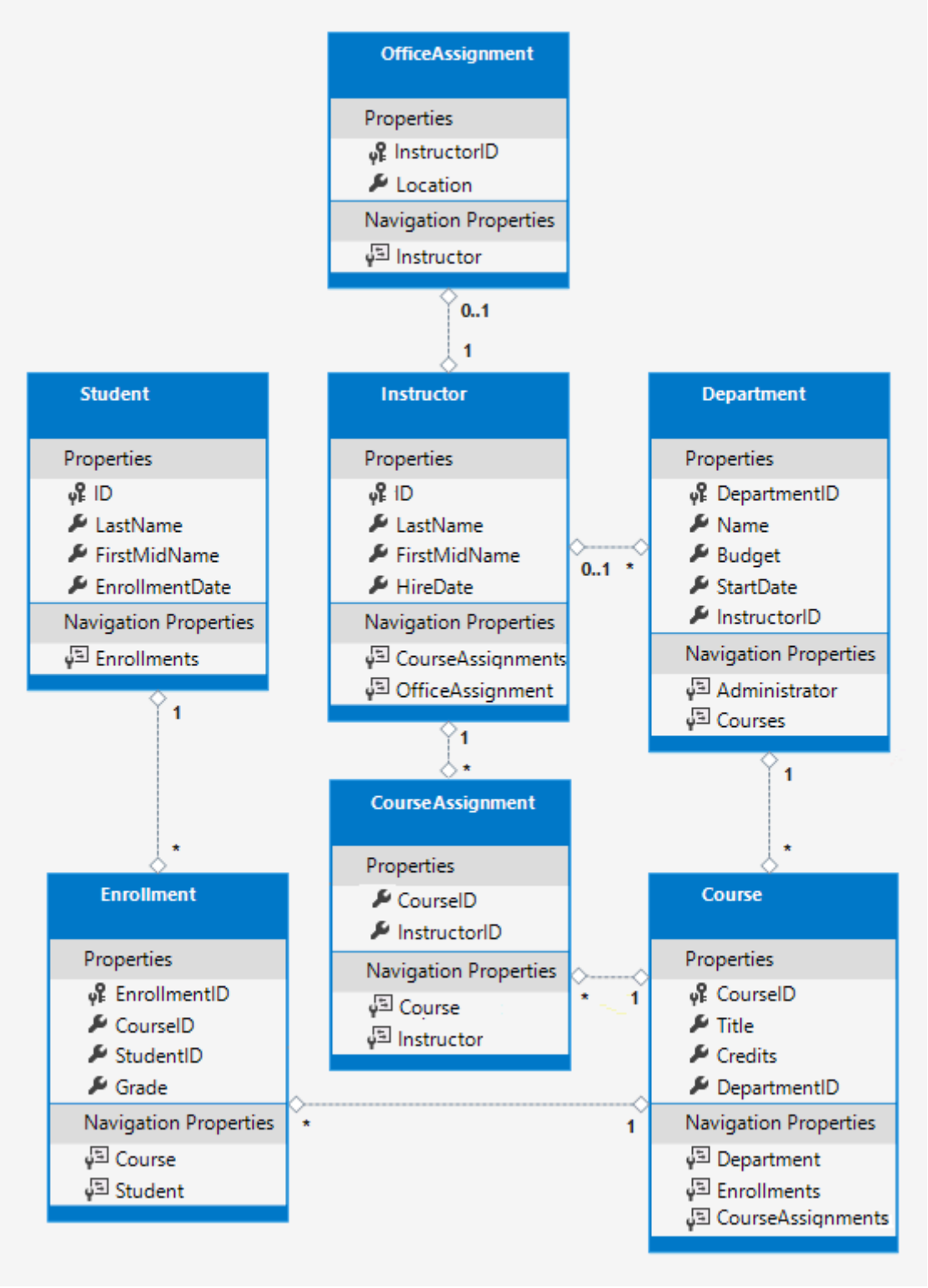
Provide product feedback

Tutorial: Create a complex data model - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorials, you worked with a simple data model that was composed of three entities. In this tutorial, you'll add more entities and relationships and you'll customize the data model by specifying formatting, validation, and database mapping rules.

When you're finished, the entity classes will make up the completed data model that's shown in the following illustration:



In this tutorial, you:

- ✓ Customize the Data model
- ✓ Make changes to Student entity
- ✓ Create Instructor entity
- ✓ Create OfficeAssignment entity
- ✓ Modify Course entity
- ✓ Create Department entity
- ✓ Modify Enrollment entity
- ✓ Update the database context

- ✓ Seed database with test data
- ✓ Add a migration
- ✓ Change the connection string
- ✓ Update the database

Prerequisites

- Using EF Core migrations

Customize the Data model

In this section you'll see how to customize the data model by using attributes that specify formatting, validation, and database mapping rules. Then in several of the following sections you'll create the complete School data model by adding attributes to the classes you already created and creating new classes for the remaining entity types in the model.

The DataType attribute

For student enrollment dates, all of the web pages currently display the time along with the date, although all you care about for this field is the date. By using data annotation attributes, you can make one code change that will fix the display format in every view that shows the data. To see an example of how to do that, you'll add an attribute to the `EnrollmentDate` property in the `Student` class.

In `Models/Student.cs`, add a `using` statement for the `System.ComponentModel.DataAnnotations` namespace and add `DataType` and `DisplayFormat` attributes to the `EnrollmentDate` property, as shown in the following example:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string LastName { get; set; }
        public string FirstMidName { get; set; }
```

```
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type. In this case we only want to keep track of the date, not the date and time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress, and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attribute emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes don't provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's CultureInfo.

The `DisplayFormat` attribute is used to explicitly specify the date format:

C#

```
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields -- for example, for currency values, you might not want the currency symbol in the text box for editing.)

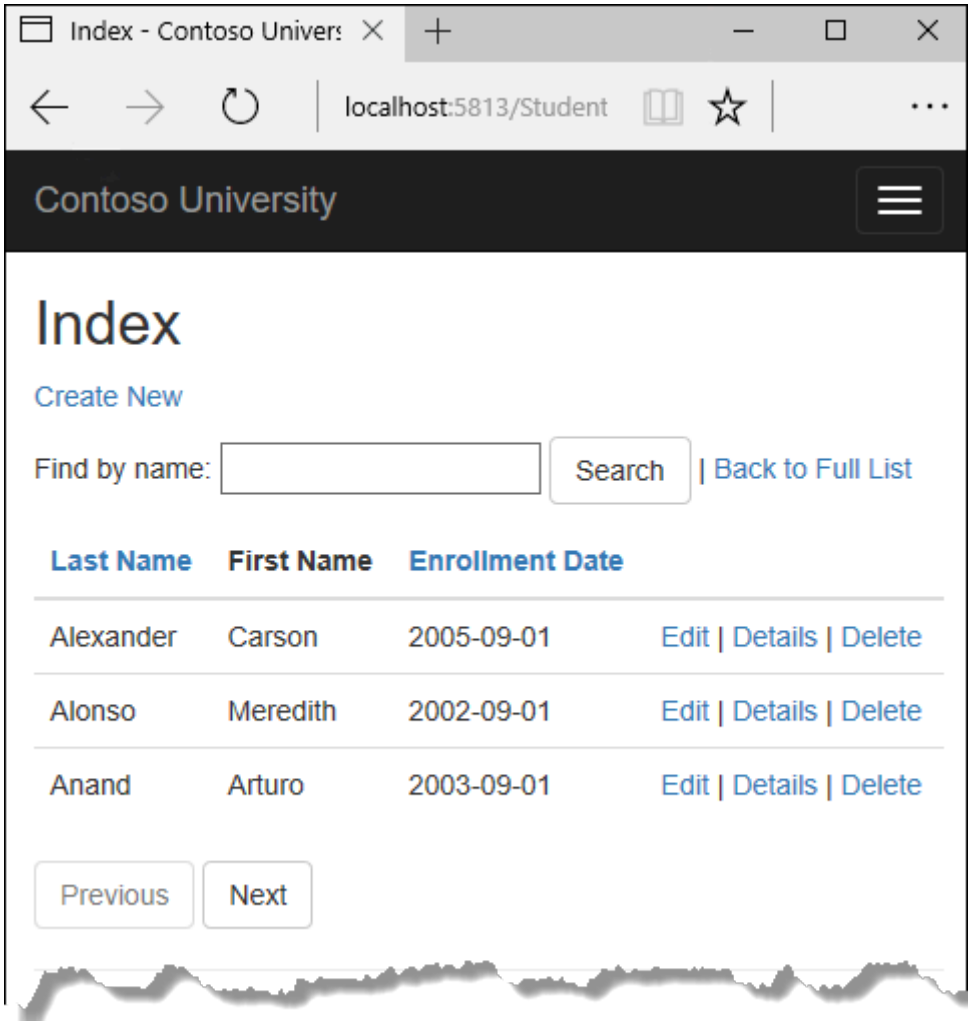
You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute also. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with `DisplayFormat`:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, some client-side input validation, etc.).

- By default, the browser will render data using the correct format based on your locale.

For more information, see the <input> tag helper documentation.

Run the app, go to the Students Index page and notice that times are no longer displayed for the enrollment dates. The same will be true for any view that uses the Student model.



The StringLength attribute

You can also specify data validation rules and validation error messages using attributes. The `StringLength` attribute sets the maximum length in the database and provides client side and server side validation for ASP.NET Core MVC. You can also specify the minimum string length in this attribute, but the minimum value has no impact on the database schema.

Suppose you want to ensure that users don't enter more than 50 characters for a name. To add this limitation, add `StringLength` attributes to the `LastName` and `FirstMidName` properties, as shown in the following example:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The `StringLength` attribute won't prevent a user from entering white space for a name. You can use the `RegularExpression` attribute to apply restrictions to the input. For example, the following code requires the first character to be upper case and the remaining characters to be alphabetical:

C#

```
[RegularExpression(@"^[A-Z]+[a-zA-Z]*$")]
```

The `MaxLength` attribute provides functionality similar to the `StringLength` attribute but doesn't provide client side validation.

The database model has now changed in a way that requires a change in the database schema. You'll use migrations to update the schema without losing any data that you may have added to the database by using the application UI.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands:

.NET CLI

```
dotnet ef migrations add MaxLengthOnNames
```

.NET CLI

```
dotnet ef database update
```

The `migrations add` command warns that data loss may occur, because the change makes the maximum length shorter for two columns. Migrations creates a file named `<timeStamp>_MaxLengthOnNames.cs`. This file contains code in the `Up` method that will update the database to match the current data model. The `database update` command ran that code.

The timestamp prefixed to the migrations file name is used by Entity Framework to order the migrations. You can create multiple migrations before running the update-database command, and then all of the migrations are applied in the order in which they were created.

Run the app, select the **Students** tab, click **Create New**, and try to enter either name longer than 50 characters. The application should prevent you from doing this.

The Column attribute

You can also use attributes to control how your classes and properties are mapped to the database. Suppose you had used the name `FirstMidName` for the first-name field because the field might also contain a middle name. But you want the database column to be named `FirstName`, because users who will be writing ad-hoc queries against the database are accustomed to that name. To make this mapping, you can use the `Column` attribute.

The `Column` attribute specifies that when the database is created, the column of the `Student` table that maps to the `FirstMidName` property will be named `FirstName`. In other words, when your code refers to `Student.FirstMidName`, the data will come from or be updated in the `FirstName` column of the `Student` table. If you don't specify column names, they're given the same name as the property name.

In the `Student.cs` file, add a `using` statement for `System.ComponentModel.DataAnnotations.Schema` and add the column name attribute to the `FirstMidName` property, as shown in the following highlighted code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

```
namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [StringLength(50)]
        public string LastName { get; set; }
        [StringLength(50)]
        [Column("FirstName")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        public DateTime EnrollmentDate { get; set; }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The addition of the `Column` attribute changes the model backing the `SchoolContext`, so it won't match the database.

Save your changes and build the project. Then open the command window in the project folder and enter the following commands to create another migration:

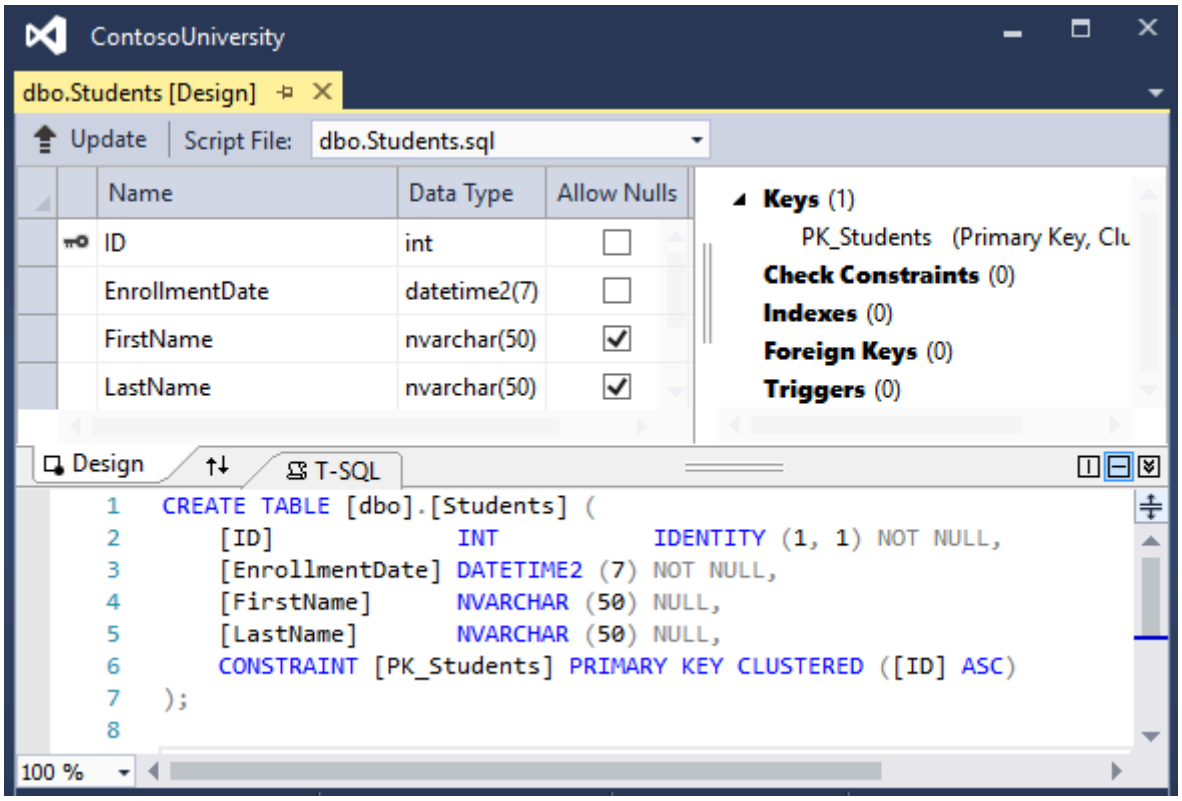
.NET CLI

```
dotnet ef migrations add ColumnFirstName
```

.NET CLI

```
dotnet ef database update
```

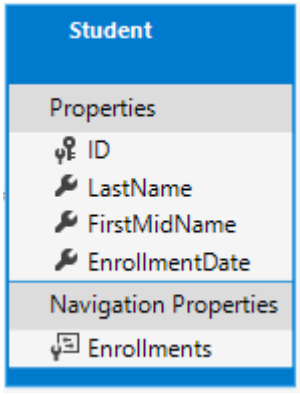
In **SQL Server Object Explorer**, open the Student table designer by double-clicking the **Student** table.



Before you applied the first two migrations, the name columns were of type nvarchar(MAX). They're now nvarchar(50) and the column name has changed from FirstMidName to FirstName.

> ⓘ Note
>
> If you try to compile before you finish creating all of the entity classes in the following sections, you might get compiler errors.

Changes to Student entity



In `Models/Student.cs`, replace the code you added earlier with the following code. The changes are highlighted.

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Student
    {
        public int ID { get; set; }
        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50)]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }

        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

The Required attribute

The `Required` attribute makes the name properties required fields. The `Required` attribute isn't needed for non-nullable types such as value types (DateTime, int, double, float, etc.). Types that can't be null are automatically treated as required fields.

The `Required` attribute must be used with `MinimumLength` for the `MinimumLength` to be enforced.

C#

```
[Display(Name = "Last Name")]
[Required]
```

```
[StringLength(50, MinimumLength=2)]
public string LastName { get; set; }
```

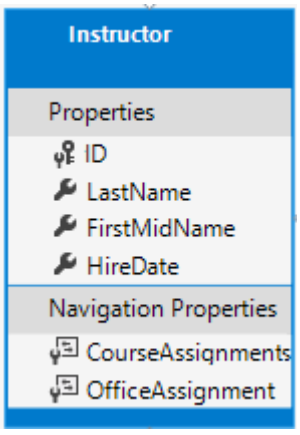
The Display attribute

The `Display` attribute specifies that the caption for the text boxes should be "First Name", "Last Name", "Full Name", and "Enrollment Date" instead of the property name in each instance (which has no space dividing the words).

The FullName calculated property

`FullName` is a calculated property that returns a value that's created by concatenating two other properties. Therefore it has only a get accessor, and no `FullName` column will be generated in the database.

Create Instructor entity



Create `Models/Instructor.cs`, replacing the template code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor
    {
        public int ID { get; set; }

        [Required]
        [Display(Name = "Last Name")]
```

```
        [StringLength(50)]
        public string LastName { get; set; }

        [Required]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        [StringLength(50)]
        public string FirstMidName { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        [Display(Name = "Full Name")]
        public string FullName
        {
            get { return LastName + ", " + FirstMidName; }
        }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Notice that several properties are the same in the Student and Instructor entities. In the Implementing Inheritance tutorial later in this series, you'll refactor this code to eliminate the redundancy.

You can put multiple attributes on one line, so you could also write the `HireDate` attributes as follows:

C#

```
[DataType(DataType.Date),Display(Name = "Hire
Date"),DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
```

The CourseAssignments and OfficeAssignment navigation properties

The `CourseAssignments` and `OfficeAssignment` properties are navigation properties.

An instructor can teach any number of courses, so `CourseAssignments` is defined as a collection.

C#

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

If a navigation property can hold multiple entities, its type must be a list in which entries can be added, deleted, and updated. You can specify `ICollection<T>` or a type such as `List<T>` or `HashSet<T>`. If you specify `ICollection<T>`, EF creates a `HashSet<T>` collection by default.

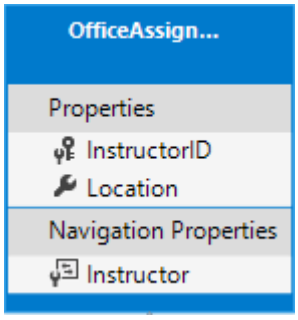
The reason why these are `CourseAssignment` entities is explained below in the section about many-to-many relationships.

Contoso University business rules state that an instructor can only have at most one office, so the `OfficeAssignment` property holds a single OfficeAssignment entity (which may be null if no office is assigned).

C#

```
public OfficeAssignment OfficeAssignment { get; set; }
```

Create OfficeAssignment entity



Create `Models/OfficeAssignment.cs` with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class OfficeAssignment
    {
        [Key]
        public int InstructorID { get; set; }
        [StringLength(50)]
        [Display(Name = "Office Location")]
        public string Location { get; set; }
```

```
        public Instructor Instructor { get; set; }
    }
}
```

The Key attribute

There's a one-to-zero-or-one relationship between the `Instructor` and the `OfficeAssignment` entities. An office assignment only exists in relation to the instructor it's assigned to, and therefore its primary key is also its foreign key to the `Instructor` entity. But the Entity Framework can't automatically recognize `InstructorID` as the primary key of this entity because its name doesn't follow the `ID` or `classnameID` naming convention. Therefore, the `Key` attribute is used to identify it as the key:

C#

```
[Key]
public int InstructorID { get; set; }
```

You can also use the `Key` attribute if the entity does have its own primary key but you want to name the property something other than classnameID or ID.

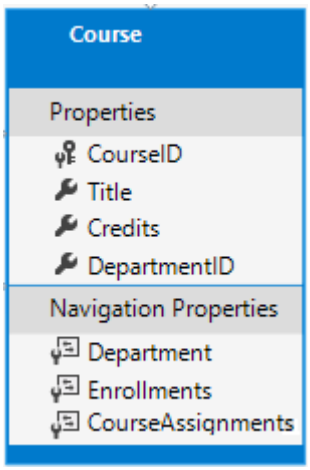
By default, EF treats the key as non-database-generated because the column is for an identifying relationship.

The Instructor navigation property

The Instructor entity has a nullable `OfficeAssignment` navigation property (because an instructor might not have an office assignment), and the OfficeAssignment entity has a non-nullable `Instructor` navigation property (because an office assignment can't exist without an instructor -- `InstructorID` is non-nullable). When an Instructor entity has a related OfficeAssignment entity, each entity will have a reference to the other one in its navigation property.

You could put a `[Required]` attribute on the Instructor navigation property to specify that there must be a related instructor, but you don't have to do that because the `InstructorID` foreign key (which is also the key to this table) is non-nullable.

Modify Course entity



In `Models/Course.cs`, replace the code you added earlier with the following code. The changes are highlighted.

C#

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Course
    {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        [Display(Name = "Number")]
        public int CourseID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Title { get; set; }

        [Range(0, 5)]
        public int Credits { get; set; }

        public int DepartmentID { get; set; }

        public Department Department { get; set; }
        public ICollection<Enrollment> Enrollments { get; set; }
        public ICollection<CourseAssignment> CourseAssignments { get; set; }
    }
}
```

The course entity has a foreign key property `DepartmentID` which points to the related Department entity and it has a `Department` navigation property.

The Entity Framework doesn't require you to add a foreign key property to your data model when you have a navigation property for a related entity. EF automatically creates foreign keys in the database wherever they're needed and creates shadow properties for

them. But having the foreign key in the data model can make updates simpler and more efficient. For example, when you fetch a `Course` entity to edit, the `Department` entity is null if you don't load it, so when you update the `Course` entity, you would have to first fetch the `Department` entity. When the foreign key property `DepartmentID` is included in the data model, you don't need to fetch the `Department` entity before you update.

The DatabaseGenerated attribute

The `DatabaseGenerated` attribute with the `None` parameter on the `CourseID` property specifies that primary key values are provided by the user rather than generated by the database.

C#

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
[Display(Name = "Number")]
public int CourseID { get; set; }
```

By default, Entity Framework assumes that primary key values are generated by the database. That's what you want in most scenarios. However, for `Course` entities, you'll use a user-specified course number such as a 1000 series for one department, a 2000 series for another department, and so on.

The `DatabaseGenerated` attribute can also be used to generate default values, as in the case of database columns used to record the date a row was created or updated. For more information, see Generated Properties.

Foreign key and navigation properties

The foreign key properties and navigation properties in the `Course` entity reflect the following relationships:

A course is assigned to one department, so there's a `DepartmentID` foreign key and a `Department` navigation property for the reasons mentioned above.

C#

```
public int DepartmentID { get; set; }
public Department Department { get; set; }
```

A course can have any number of students enrolled in it, so the `Enrollments` navigation property is a collection:

C#

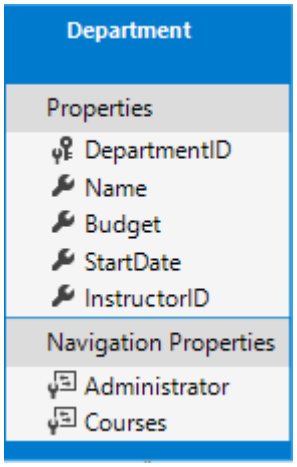
```
public ICollection<Enrollment> Enrollments { get; set; }
```

A course may be taught by multiple instructors, so the `CourseAssignments` navigation property is a collection (the type `CourseAssignment` is explained later):

C#

```
public ICollection<CourseAssignment> CourseAssignments { get; set; }
```

Create Department entity



Create `Models/Department.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
```

```
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The Column attribute

Earlier you used the `Column` attribute to change column name mapping. In the code for the `Department` entity, the `Column` attribute is being used to change SQL data type mapping so that the column will be defined using the SQL Server `money` type in the database:

C#

```
[Column(TypeName="money")]
public decimal Budget { get; set; }
```

Column mapping is generally not required, because the Entity Framework chooses the appropriate SQL Server data type based on the CLR type that you define for the property. The CLR `decimal` type maps to a SQL Server `decimal` type. But in this case you know that the column will be holding currency amounts, and the money data type is more appropriate for that.

Foreign key and navigation properties

The foreign key and navigation properties reflect the following relationships:

A department may or may not have an administrator, and an administrator is always an instructor. Therefore the `InstructorID` property is included as the foreign key to the Instructor entity, and a question mark is added after the `int` type designation to mark the property as nullable. The navigation property is named `Administrator` but holds an Instructor entity:

C#

```
public int? InstructorID { get; set; }
```

```
public Instructor Administrator { get; set; }
```

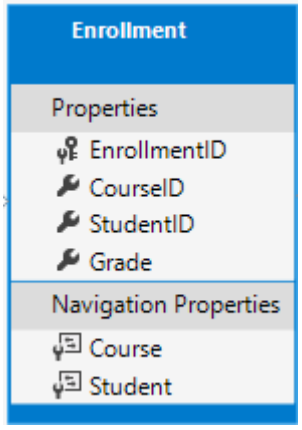
A department may have many courses, so there's a Courses navigation property:

C#

```
public ICollection<Course> Courses { get; set; }
```

> **Note**
>
> By convention, the Entity Framework enables cascade delete for non-nullable foreign keys and for many-to-many relationships. This can result in circular cascade delete rules, which will cause an exception when you try to add a migration. For example, if you didn't define the `Department.InstructorID` property as nullable, EF would configure a cascade delete rule to delete the department when you delete the instructor, which isn't what you want to have happen. If your business rules required the `InstructorID` property to be non-nullable, you would have to use the following fluent API statement to disable cascade delete on the relationship:
>
> C#
>
> ```
> modelBuilder.Entity<Department>()
>     .HasOne(d => d.Administrator)
>     .WithMany()
>     .OnDelete(DeleteBehavior.Restrict)
> ```

Modify Enrollment entity



In `Models/Enrollment.cs`, replace the code you added earlier with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public enum Grade
    {
        A, B, C, D, F
    }

    public class Enrollment
    {
        public int EnrollmentID { get; set; }
        public int CourseID { get; set; }
        public int StudentID { get; set; }
        [DisplayFormat(NullDisplayText = "No grade")]
        public Grade? Grade { get; set; }

        public Course Course { get; set; }
        public Student Student { get; set; }
    }
}
```

Foreign key and navigation properties

The foreign key properties and navigation properties reflect the following relationships:

An enrollment record is for a single course, so there's a `CourseID` foreign key property and a `Course` navigation property:

C#

```
public int CourseID { get; set; }
public Course Course { get; set; }
```

An enrollment record is for a single student, so there's a `StudentID` foreign key property and a `Student` navigation property:

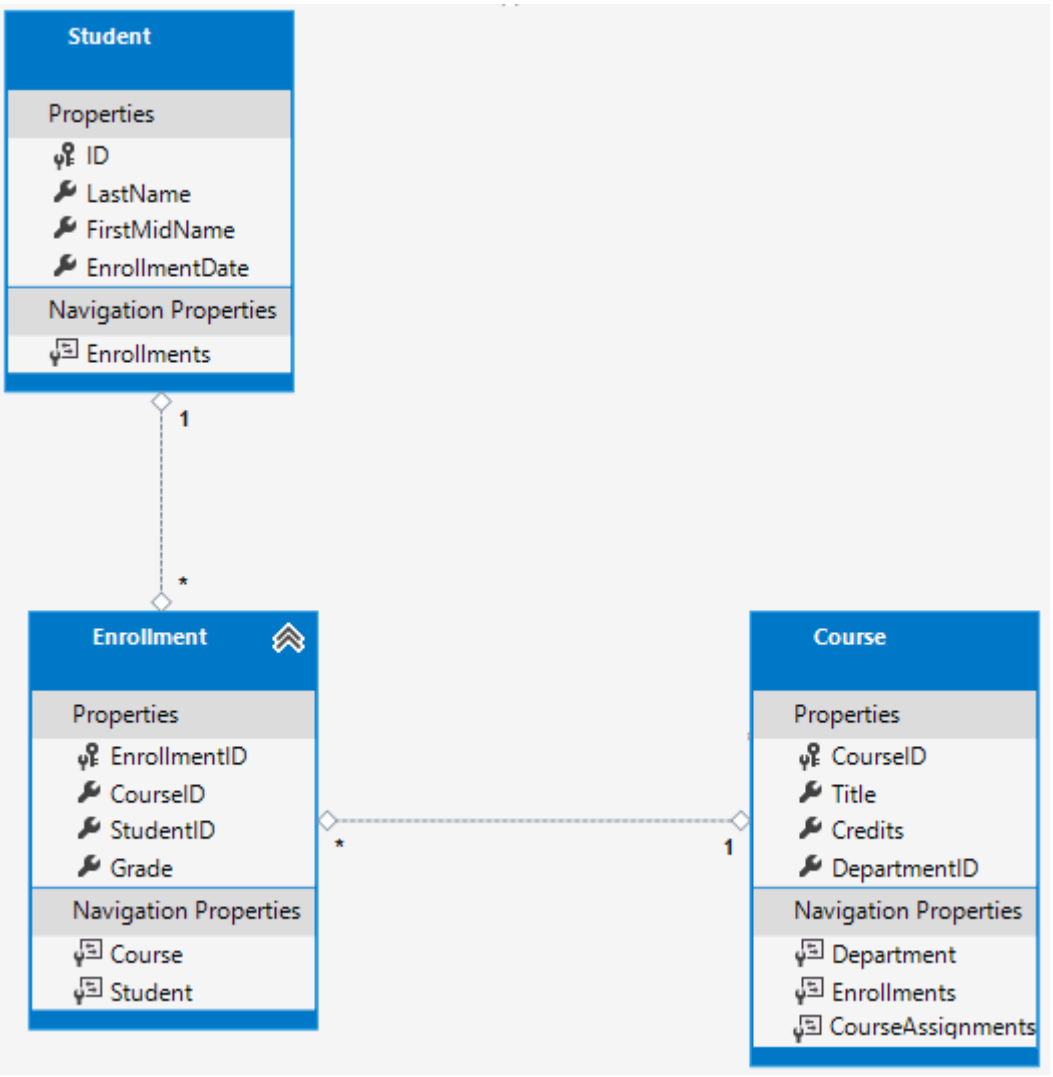
C#

```
public int StudentID { get; set; }
public Student Student { get; set; }
```

Many-to-Many relationships

There's a many-to-many relationship between the `Student` and `Course` entities, and the `Enrollment` entity functions as a many-to-many join table *with payload* in the database. "With payload" means that the `Enrollment` table contains additional data besides foreign keys for the joined tables (in this case, a primary key and a `Grade` property).

The following illustration shows what these relationships look like in an entity diagram. (This diagram was generated using the Entity Framework Power Tools for EF 6.x; creating the diagram isn't part of the tutorial, it's just being used here as an illustration.)



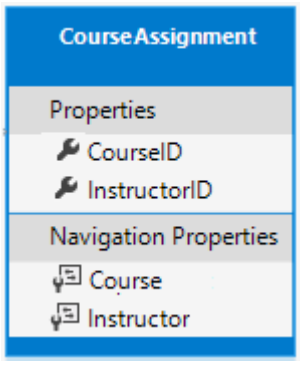
Each relationship line has a 1 at one end and an asterisk (*) at the other, indicating a one-to-many relationship.

If the `Enrollment` table didn't include grade information, it would only need to contain the two foreign keys `CourseID` and `StudentID`. In that case, it would be a many-to-many join table without payload (or a pure join table) in the database. The `Instructor` and `Course` entities have that kind of many-to-many relationship, and your next step is to create an entity class to function as a join table without payload.

EF Core supports implicit join tables for many-to-many relationships, but this tutoral has not been updated to use an implicit join table. See Many-to-Many Relationships, the

Razor Pages version of this tutorial which has been updated.

The CourseAssignment entity



Create `Models/CourseAssignment.cs` with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class CourseAssignment
    {
        public int InstructorID { get; set; }
        public int CourseID { get; set; }
        public Instructor Instructor { get; set; }
        public Course Course { get; set; }
    }
}
```

Join entity names

A join table is required in the database for the Instructor-to-Courses many-to-many relationship, and it has to be represented by an entity set. It's common to name a join entity `EntityName1EntityName2`, which in this case would be `CourseInstructor`. However, we recommend that you choose a name that describes the relationship. Data models start out simple and grow, with no-payload joins frequently getting payloads later. If you start with a descriptive entity name, you won't have to change the name later. Ideally, the join entity would have its own natural (possibly single word) name in the business domain. For example, Books and Customers could be linked through Ratings. For this relationship, `CourseAssignment` is a better choice than `CourseInstructor`.

Composite key

Since the foreign keys are not nullable and together uniquely identify each row of the table, there's no need for a separate primary key. The `InstructorID` and `CourseID` properties should function as a composite primary key. The only way to identify composite primary keys to EF is by using the *fluent API* (it can't be done by using attributes). You'll see how to configure the composite primary key in the next section.

The composite key ensures that while you can have multiple rows for one course, and multiple rows for one instructor, you can't have multiple rows for the same instructor and course. The `Enrollment` join entity defines its own primary key, so duplicates of this sort are possible. To prevent such duplicates, you could add a unique index on the foreign key fields, or configure `Enrollment` with a primary composite key similar to `CourseAssignment`. For more information, see Indexes.

Update the database context

Add the following highlighted code to the `Data/SchoolContext.cs` file:

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
```

```
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>
().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>
().ToTable("CourseAssignment");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This code adds the new entities and configures the CourseAssignment entity's composite primary key.

About a fluent API alternative

The code in the `OnModelCreating` method of the `DbContext` class uses the *fluent API* to configure EF behavior. The API is called "fluent" because it's often used by stringing a series of method calls together into a single statement, as in this example from the EF Core documentation:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .IsRequired();
}
```

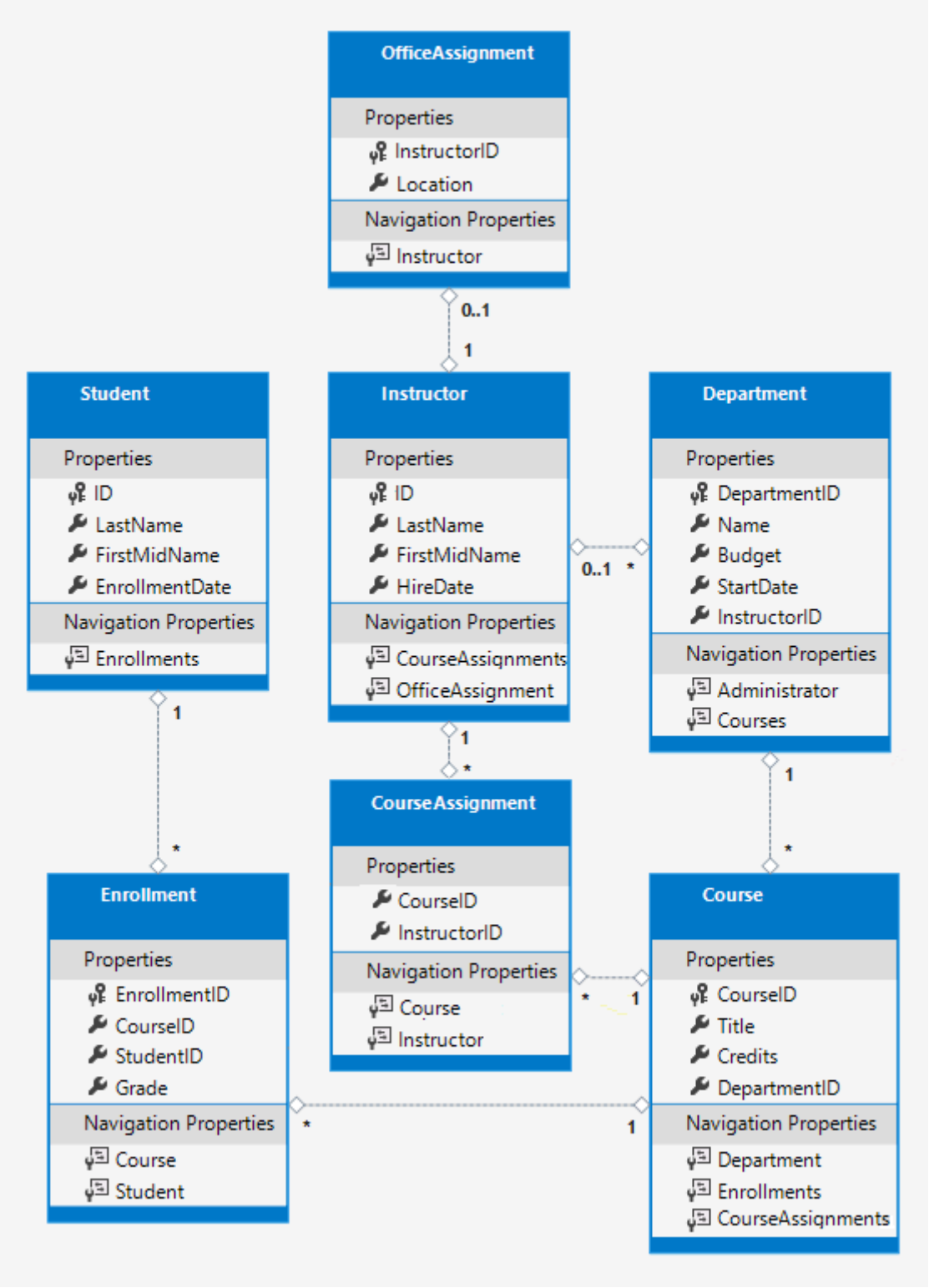
In this tutorial, you're using the fluent API only for database mapping that you can't do with attributes. However, you can also use the fluent API to specify most of the formatting, validation, and mapping rules that you can do by using attributes. Some attributes such as `MinimumLength` can't be applied with the fluent API. As mentioned previously, `MinimumLength` doesn't change the schema, it only applies a client and server side validation rule.

Some developers prefer to use the fluent API exclusively so that they can keep their entity classes "clean." You can mix attributes and fluent API if you want, and there are a few customizations that can only be done by using fluent API, but in general the recommended practice is to choose one of these two approaches and use that consistently as much as possible. If you do use both, note that wherever there's a conflict, Fluent API overrides attributes.

For more information about attributes vs. fluent API, see Methods of configuration.

Entity Diagram Showing Relationships

The following illustration shows the diagram that the Entity Framework Power Tools create for the completed School model.



Besides the one-to-many relationship lines (1 to *), you can see here the one-to-zero-or-one relationship line (1 to 0..1) between the `Instructor` and `OfficeAssignment`

entities and the zero-or-one-to-many relationship line (0..1 to *) between the Instructor and Department entities.

Seed database with test data

Replace the code in the `Data/DbInitializer.cs` file with the following code in order to provide seed data for the new entities you've created.

C#

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using ContosoUniversity.Models;

namespace ContosoUniversity.Data
{
    public static class DbInitializer
    {
        public static void Initialize(SchoolContext context)
        {
            //context.Database.EnsureCreated();

            // Look for any students.
            if (context.Students.Any())
            {
                return;   // DB has been seeded
            }

            var students = new Student[]
            {
                new Student { FirstMidName = "Carson",   LastName =
"Alexander",
                    EnrollmentDate = DateTime.Parse("2010-09-01") },
                new Student { FirstMidName = "Meredith", LastName =
"Alonso",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Arturo",   LastName = "Anand",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
                new Student { FirstMidName = "Gytis",    LastName =
"Barzdukas",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Yan",      LastName = "Li",
                    EnrollmentDate = DateTime.Parse("2012-09-01") },
                new Student { FirstMidName = "Peggy",    LastName =
"Justice",
                    EnrollmentDate = DateTime.Parse("2011-09-01") },
                new Student { FirstMidName = "Laura",    LastName =
"Norman",
                    EnrollmentDate = DateTime.Parse("2013-09-01") },
```

```
                new Student { FirstMidName = "Nino",     LastName =
"Olivetto",
                    EnrollmentDate = DateTime.Parse("2005-09-01") }
            };

            foreach (Student s in students)
            {
                context.Students.Add(s);
            }
            context.SaveChanges();

            var instructors = new Instructor[]
            {
                new Instructor { FirstMidName = "Kim",     LastName =
"Abercrombie",
                    HireDate = DateTime.Parse("1995-03-11") },
                new Instructor { FirstMidName = "Fadi",    LastName =
"Fakhouri",
                    HireDate = DateTime.Parse("2002-07-06") },
                new Instructor { FirstMidName = "Roger",   LastName =
"Harui",
                    HireDate = DateTime.Parse("1998-07-01") },
                new Instructor { FirstMidName = "Candace", LastName =
"Kapoor",
                    HireDate = DateTime.Parse("2001-01-15") },
                new Instructor { FirstMidName = "Roger",   LastName =
"Zheng",
                    HireDate = DateTime.Parse("2004-02-12") }
            };

            foreach (Instructor i in instructors)
            {
                context.Instructors.Add(i);
            }
            context.SaveChanges();

            var departments = new Department[]
            {
                new Department { Name = "English",     Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Abercrombie").ID },
                new Department { Name = "Mathematics", Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Fakhouri").ID },
                new Department { Name = "Engineering", Budget = 350000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Harui").ID },
                new Department { Name = "Economics",   Budget = 100000,
                    StartDate = DateTime.Parse("2007-09-01"),
                    InstructorID  = instructors.Single( i => i.LastName ==
"Kapoor").ID }
            };
```

```
            foreach (Department d in departments)
            {
                context.Departments.Add(d);
            }
            context.SaveChanges();

            var courses = new Course[]
            {
                new Course {CourseID = 1050, Title = "Chemistry",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"Engineering").DepartmentID
                },
                new Course {CourseID = 4022, Title = "Microeconomics",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID
                },
                new Course {CourseID = 4041, Title = "Macroeconomics",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"Economics").DepartmentID
                },
                new Course {CourseID = 1045, Title = "Calculus",
Credits = 4,
                    DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID
                },
                new Course {CourseID = 3141, Title = "Trigonometry",
Credits = 4,
                    DepartmentID = departments.Single( s => s.Name ==
"Mathematics").DepartmentID
                },
                new Course {CourseID = 2021, Title = "Composition",
Credits = 3,
                    DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID
                },
                new Course {CourseID = 2042, Title = "Literature",
Credits = 4,
                    DepartmentID = departments.Single( s => s.Name ==
"English").DepartmentID
                },
            };

            foreach (Course c in courses)
            {
                context.Courses.Add(c);
            }
            context.SaveChanges();

            var officeAssignments = new OfficeAssignment[]
            {
                new OfficeAssignment {
```

```
                    InstructorID = instructors.Single( i => i.LastName ==
"Fakhouri").ID,
                    Location = "Smith 17" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName ==
"Harui").ID,
                    Location = "Gowan 27" },
                new OfficeAssignment {
                    InstructorID = instructors.Single( i => i.LastName ==
"Kapoor").ID,
                    Location = "Thompson 304" },
            };

            foreach (OfficeAssignment o in officeAssignments)
            {
                context.OfficeAssignments.Add(o);
            }
            context.SaveChanges();

            var courseInstructors = new CourseAssignment[]
            {
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Kapoor").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Harui").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title ==
"Microeconomics" ).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Zheng").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title ==
"Macroeconomics" ).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Zheng").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Calculus"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Fakhouri").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Trigonometry"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
```

```
"Harui").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Composition"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Abercrombie").ID
                    },
                new CourseAssignment {
                    CourseID = courses.Single(c => c.Title == "Literature"
).CourseID,
                    InstructorID = instructors.Single(i => i.LastName ==
"Abercrombie").ID
                    },
            };

            foreach (CourseAssignment ci in courseInstructors)
            {
                context.CourseAssignments.Add(ci);
            }
            context.SaveChanges();

            var enrollments = new Enrollment[]
            {
                new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID,
                    Grade = Grade.A
                },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title ==
"Microeconomics" ).CourseID,
                    Grade = Grade.C
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alexander").ID,
                    CourseID = courses.Single(c => c.Title ==
"Macroeconomics" ).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                        StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Calculus"
).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                        StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
```

```
                    CourseID = courses.Single(c => c.Title == "Trigonometry"
).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Alonso").ID,
                    CourseID = courses.Single(c => c.Title == "Composition"
).CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Anand").ID,
                    CourseID = courses.Single(c => c.Title == "Chemistry"
).CourseID
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Anand").ID,
                    CourseID = courses.Single(c => c.Title ==
"Microeconomics").CourseID,
                    Grade = Grade.B
                    },
                new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Barzdukas").ID,
                    CourseID = courses.Single(c => c.Title ==
"Chemistry").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName == "Li").ID,
                    CourseID = courses.Single(c => c.Title ==
"Composition").CourseID,
                    Grade = Grade.B
                    },
                    new Enrollment {
                    StudentID = students.Single(s => s.LastName ==
"Justice").ID,
                    CourseID = courses.Single(c => c.Title ==
"Literature").CourseID,
                    Grade = Grade.B
                    }
            };

            foreach (Enrollment e in enrollments)
            {
                var enrollmentInDataBase = context.Enrollments.Where(
                    s =>
                            s.Student.ID == e.StudentID &&
                            s.Course.CourseID ==
e.CourseID).SingleOrDefault();
                if (enrollmentInDataBase == null)
                {
```

```
                    context.Enrollments.Add(e);
                }
            }
            context.SaveChanges();
        }
    }
}
```

As you saw in the first tutorial, most of this code simply creates new entity objects and loads sample data into properties as required for testing. Notice how the many-to-many relationships are handled: the code creates relationships by creating entities in the `Enrollments` and `CourseAssignment` join entity sets.

Add a migration

Save your changes and build the project. Then open the command window in the project folder and enter the `migrations add` command (don't do the update-database command yet):

.NET CLI

```
dotnet ef migrations add ComplexDataModel
```

You get a warning about possible data loss.

text

```
An operation was scaffolded that may result in the loss of data. Please
review the migration for accuracy.
Done. To undo this action, use 'ef migrations remove'
```

If you tried to run the `database update` command at this point (don't do it yet), you would get the following error:

> The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_dbo.Course_dbo.Department_DepartmentID". The conflict occurred in database "ContosoUniversity", table "dbo.Department", column 'DepartmentID'.

Sometimes when you execute migrations with existing data, you need to insert stub data into the database to satisfy foreign key constraints. The generated code in the `Up` method adds a non-nullable `DepartmentID` foreign key to the `Course` table. If there are already rows in the Course table when the code runs, the `AddColumn` operation fails

because SQL Server doesn't know what value to put in the column that can't be null. For this tutorial you'll run the migration on a new database, but in a production application you'd have to make the migration handle existing data, so the following directions show an example of how to do that.

To make this migration work with existing data you have to change the code to give the new column a default value, and create a stub department named "Temp" to act as the default department. As a result, existing Course rows will all be related to the "Temp" department after the `Up` method runs.

- Open the `{timestamp}_ComplexDataModel.cs` file.
- Comment out the line of code that adds the DepartmentID column to the Course table.

```
C#

migrationBuilder.AlterColumn<string>(
    name: "Title",
    table: "Course",
    maxLength: 50,
    nullable: true,
    oldClrType: typeof(string),
    oldNullable: true);

//migrationBuilder.AddColumn<int>(
//    name: "DepartmentID",
//    table: "Course",
//    nullable: false,
//    defaultValue: 0);
```

- Add the following highlighted code after the code that creates the Department table:

```
C#

migrationBuilder.CreateTable(
    name: "Department",
    columns: table => new
    {
        DepartmentID = table.Column<int>(nullable: false)
            .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
        Budget = table.Column<decimal>(type: "money", nullable: false),
        InstructorID = table.Column<int>(nullable: true),
        Name = table.Column<string>(maxLength: 50, nullable: true),
        StartDate = table.Column<DateTime>(nullable: false)
    },
    constraints: table =>
```

```
    {
        table.PrimaryKey("PK_Department", x => x.DepartmentID);
        table.ForeignKey(
            name: "FK_Department_Instructor_InstructorID",
            column: x => x.InstructorID,
            principalTable: "Instructor",
            principalColumn: "ID",
            onDelete: ReferentialAction.Restrict);
    });

migrationBuilder.Sql("INSERT INTO dbo.Department (Name, Budget,
StartDate) VALUES ('Temp', 0.00, GETDATE())");
// Default value for FK points to department created above, with
// defaultValue changed to 1 in following AddColumn statement.

migrationBuilder.AddColumn<int>(
    name: "DepartmentID",
    table: "Course",
    nullable: false,
    defaultValue: 1);
```

In a production application, you would write code or scripts to add Department rows and relate Course rows to the new Department rows. You would then no longer need the "Temp" department or the default value on the `Course.DepartmentID` column.

Save your changes and build the project.

Change the connection string

You now have new code in the `DbInitializer` class that adds seed data for the new entities to an empty database. To make EF create a new empty database, change the name of the database in the connection string in `appsettings.json` to ContosoUniversity3 or some other name that you haven't used on the computer you're using.

JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=
(localdb)\\mssqllocaldb;Database=ContosoUniversity3;Trusted_Connection=True;
MultipleActiveResultSets=true"
  },
```

Save your change to `appsettings.json`.

> **Note**
>
> As an alternative to changing the database name, you can delete the database. Use **SQL Server Object Explorer** (SSOX) or the `database drop` CLI command:
>
> .NET CLI
>
> ```
> dotnet ef database drop
> ```

Update the database

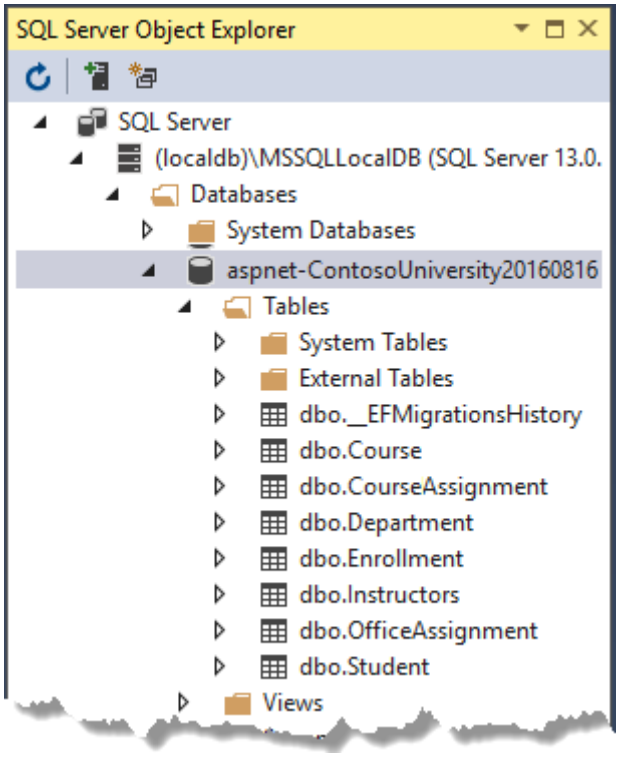
After you have changed the database name or deleted the database, run the `database update` command in the command window to execute the migrations.

.NET CLI

```
dotnet ef database update
```

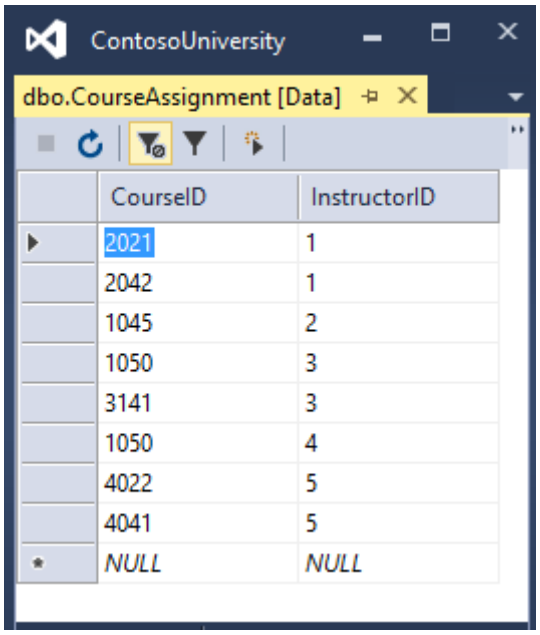
Run the app to cause the `DbInitializer.Initialize` method to run and populate the new database.

Open the database in SSOX as you did earlier, and expand the **Tables** node to see that all of the tables have been created. (If you still have SSOX open from the earlier time, click the **Refresh** button.)



Run the app to trigger the initializer code that seeds the database.

Right-click the **CourseAssignment** table and select **View Data** to verify that it has data in it.



Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized the Data model
- ✓ Made changes to Student entity
- ✓ Created Instructor entity
- ✓ Created OfficeAssignment entity
- ✓ Modified Course entity
- ✓ Created Department entity
- ✓ Modified Enrollment entity
- ✓ Updated the database context
- ✓ Seeded database with test data
- ✓ Added a migration
- ✓ Changed the connection string
- ✓ Updated the database

Advance to the next tutorial to learn more about how to access related data.

Next: Access related data

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

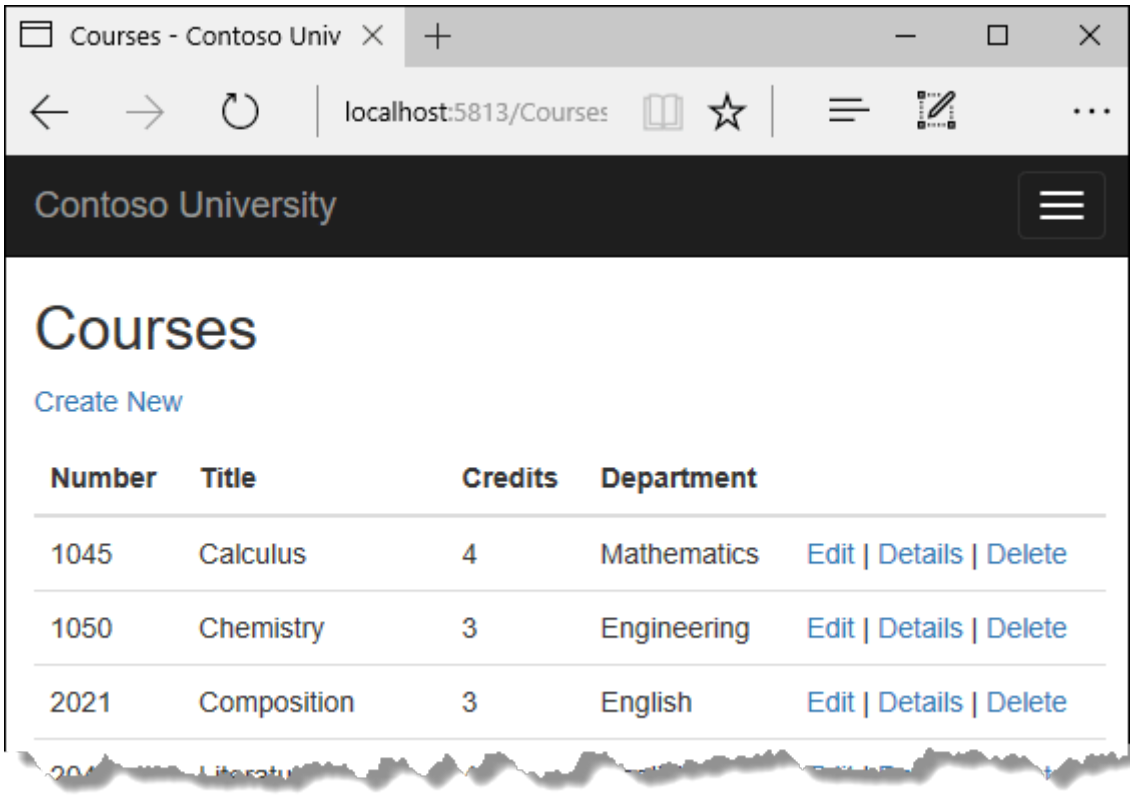
Provide product feedback

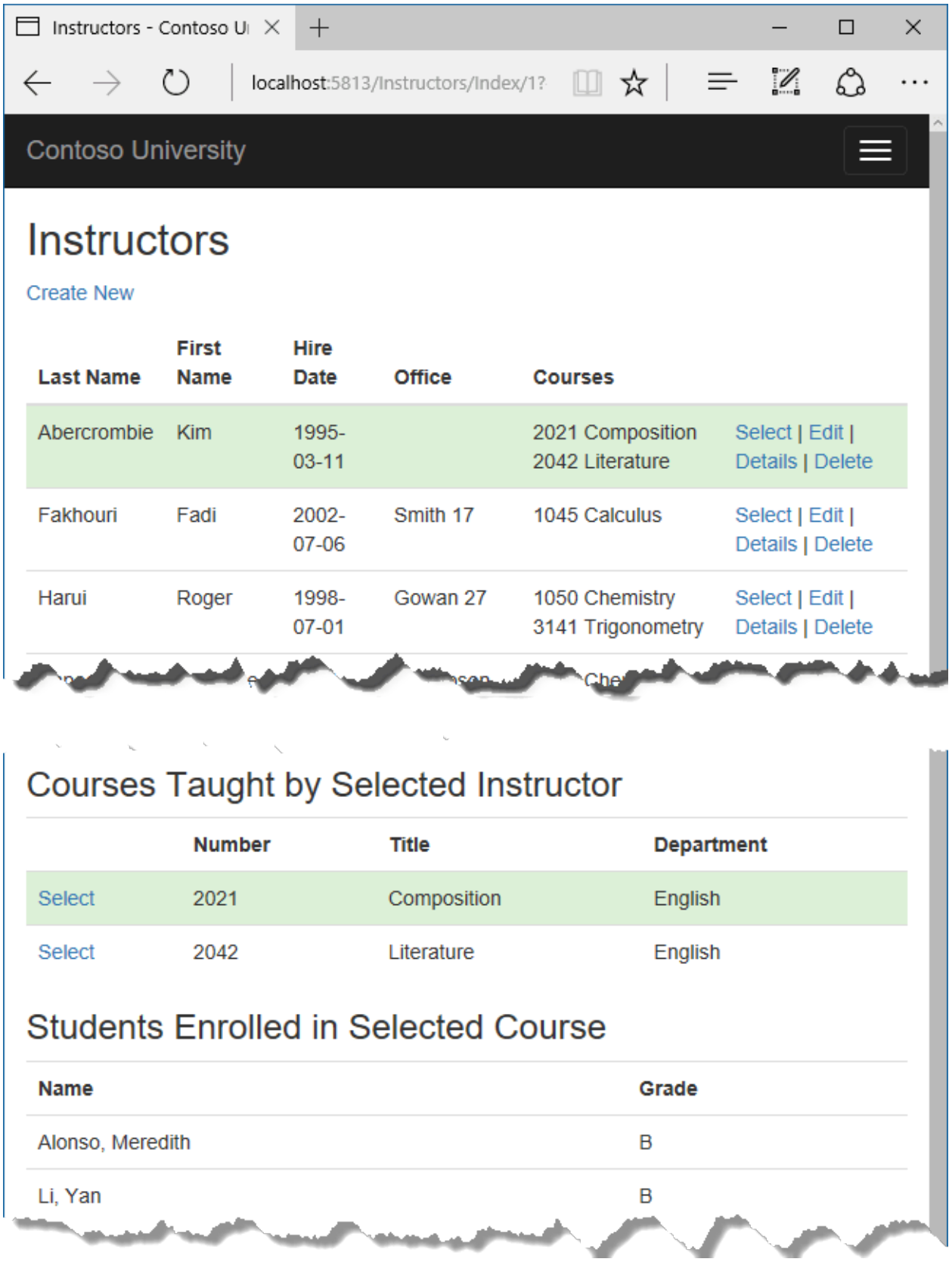
Tutorial: Read related data - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you completed the School data model. In this tutorial, you'll read and display related data -- that is, data that the Entity Framework loads into navigation properties.

The following illustrations show the pages that you'll work with.





In this tutorial, you:

- ✓ Learn how to load related data
- ✓ Create a Courses page
- ✓ Create an Instructors page
- ✓ Learn about explicit loading

Prerequisites

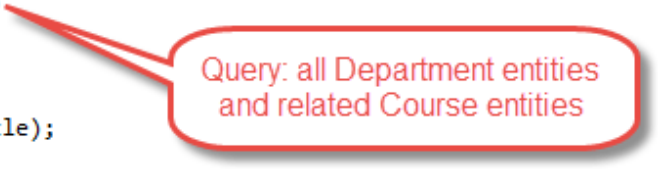
- Create a complex data model

Learn how to load related data

There are several ways that Object-Relational Mapping (ORM) software such as Entity Framework can load related data into the navigation properties of an entity:

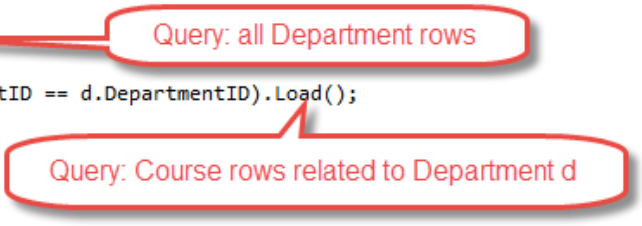
- Eager loading: When the entity is read, related data is retrieved along with it. This typically results in a single join query that retrieves all of the data that's needed. You specify eager loading in Entity Framework Core by using the `Include` and `ThenInclude` methods.

```
var departments = _context.Departments.Include(d => d.Courses);
foreach (Department d in departments)
{
    foreach(Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```

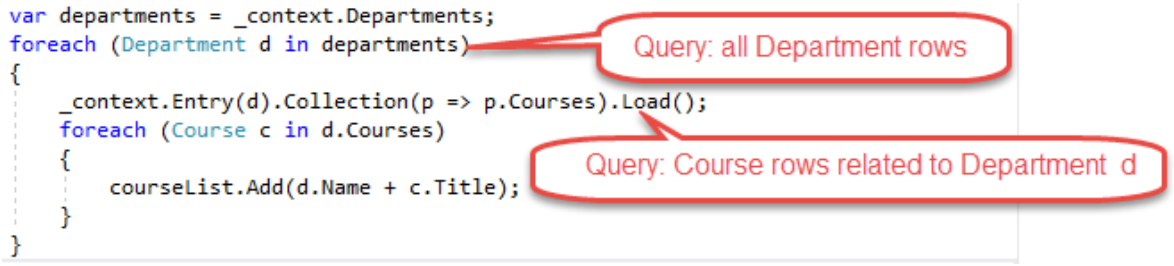


  You can retrieve some of the data in separate queries, and EF "fixes up" the navigation properties. That is, EF automatically adds the separately retrieved entities where they belong in navigation properties of previously retrieved entities. For the query that retrieves related data, you can use the `Load` method instead of a method that returns a list or object, such as `ToList` or `Single`.

```
var departments = _context.Departments;
foreach (Department d in departments)
{
    _context.Courses.Where(c => c.DepartmentID == d.DepartmentID).Load();
    foreach (Course c in d.Courses)
    {
        courseList.Add(d.Name + c.Title);
    }
}
```



- Explicit loading: When the entity is first read, related data isn't retrieved. You write code that retrieves the related data if it's needed. As in the case of eager loading with separate queries, explicit loading results in multiple queries sent to the database. The difference is that with explicit loading, the code specifies the navigation properties to be loaded. In Entity Framework Core 1.1 you can use the `Load` method to do explicit loading. For example:



- Lazy loading: When the entity is first read, related data isn't retrieved. However, the first time you attempt to access a navigation property, the data required for that navigation property is automatically retrieved. A query is sent to the database each time you try to get data from a navigation property for the first time. Entity Framework Core 1.0 doesn't support lazy loading.

Performance considerations

If you know you need related data for every entity retrieved, eager loading often offers the best performance, because a single query sent to the database is typically more efficient than separate queries for each entity retrieved. For example, suppose that each department has ten related courses. Eager loading of all related data would result in just a single (join) query and a single round trip to the database. A separate query for courses for each department would result in eleven round trips to the database. The extra round trips to the database are especially detrimental to performance when latency is high.

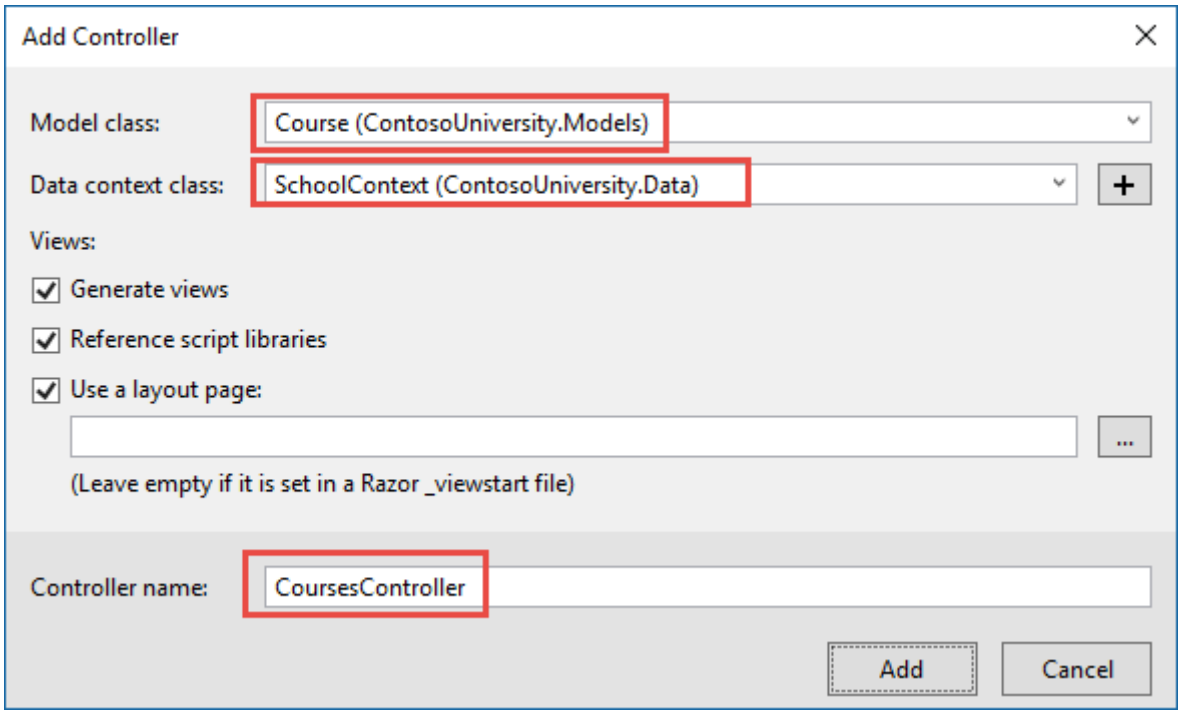
On the other hand, in some scenarios separate queries is more efficient. Eager loading of all related data in one query might cause a very complex join to be generated, which SQL Server can't process efficiently. Or if you need to access an entity's navigation properties only for a subset of a set of the entities you're processing, separate queries might perform better because eager loading of everything up front would retrieve more data than you need. If performance is critical, it's best to test performance both ways in order to make the best choice.

Create a Courses page

The `Course` entity includes a navigation property that contains the `Department` entity of the department that the course is assigned to. To display the name of the assigned department in a list of courses, you need to get the `Name` property from the `Department` entity that's in the `Course.Department` navigation property.

Create a controller named `CoursesController` for the `Course` entity type, using the same options for the **MVC Controller with views, using Entity Framework** scaffolder that you

did earlier for the `StudentsController`, as shown in the following illustration:



Open `CoursesController.cs` and examine the `Index` method. The automatic scaffolding has specified eager loading for the `Department` navigation property by using the `Include` method.

Replace the `Index` method with the following code that uses a more appropriate name for the `IQueryable` that returns Course entities (`courses` instead of `schoolContext`):

C#

```
public async Task<IActionResult> Index()
{
    var courses = _context.Courses
        .Include(c => c.Department)
        .AsNoTracking();
    return View(await courses.ToListAsync());
}
```

Open `Views/Courses/Index.cshtml` and replace the template code with the following code. The changes are highlighted:

CSHTML

```
@model IEnumerable<ContosoUniversity.Models.Course>

@{
    ViewData["Title"] = "Courses";
}
```

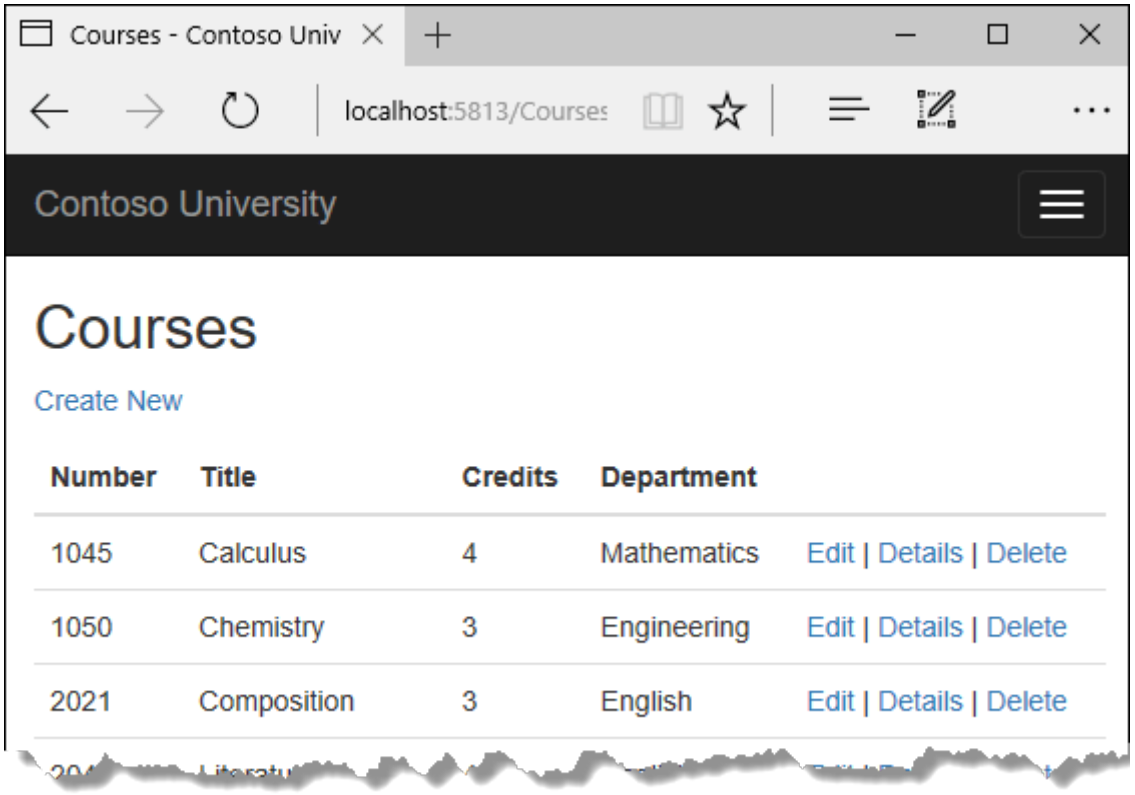
```
<h2>Courses</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.CourseID)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Credits)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Department)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.CourseID)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Credits)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Department.Name)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.CourseID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.CourseID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.CourseID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

You've made the following changes to the scaffolded code:

- Changed the heading from **Index** to **Courses**.
- Added a **Number** column that shows the `CourseID` property value. By default, primary keys aren't scaffolded because normally they're meaningless to end users. However, in this case the primary key is meaningful and you want to show it.
- Changed the **Department** column to display the department name. The code displays the `Name` property of the `Department` entity that's loaded into the `Department` navigation property:

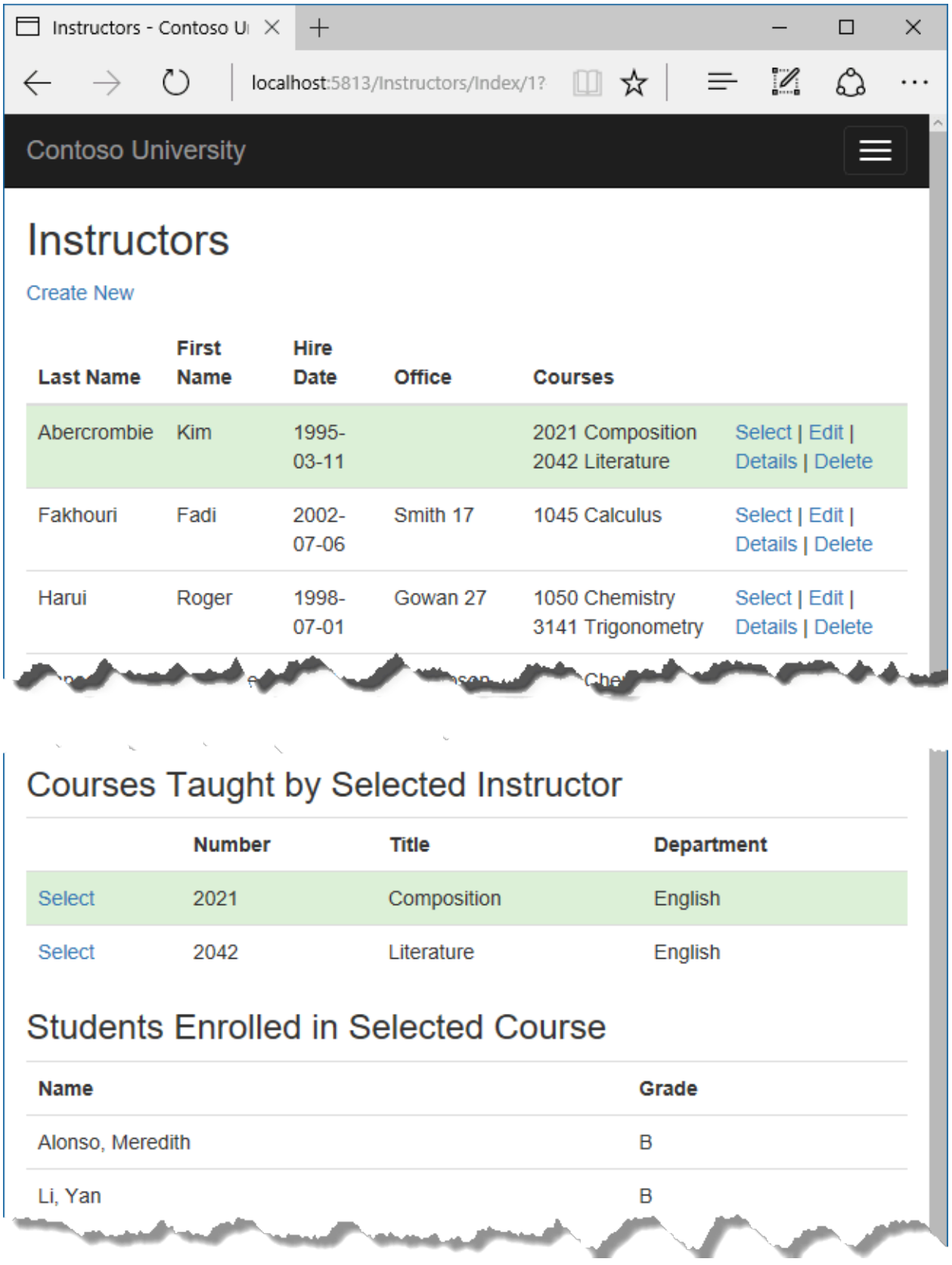
```
HTML

@Html.DisplayFor(modelItem => item.Department.Name)
```

Run the app and select the **Courses** tab to see the list with department names.



Create an Instructors page

In this section, you'll create a controller and view for the `Instructor` entity in order to display the Instructors page:



This page reads and displays related data in the following ways:

- The list of instructors displays related data from the `OfficeAssignment` entity. The `Instructor` and `OfficeAssignment` entities are in a one-to-zero-or-one relationship. You'll use eager loading for the `OfficeAssignment` entities. As explained earlier, eager loading is typically more efficient when you need the

related data for all retrieved rows of the primary table. In this case, you want to display office assignments for all displayed instructors.

- When the user selects an instructor, related `Course` entities are displayed. The `Instructor` and `Course` entities are in a many-to-many relationship. You'll use eager loading for the `Course` entities and their related `Department` entities. In this case, separate queries might be more efficient because you need courses only for the selected instructor. However, this example shows how to use eager loading for navigation properties within entities that are themselves in navigation properties.
- When the user selects a course, related data from the `Enrollments` entity set is displayed. The `Course` and `Enrollment` entities are in a one-to-many relationship. You'll use separate queries for `Enrollment` entities and their related `Student` entities.

Create a view model for the Instructor Index view

The Instructors page shows data from three different tables. Therefore, you'll create a view model that includes three properties, each holding the data for one of the tables.

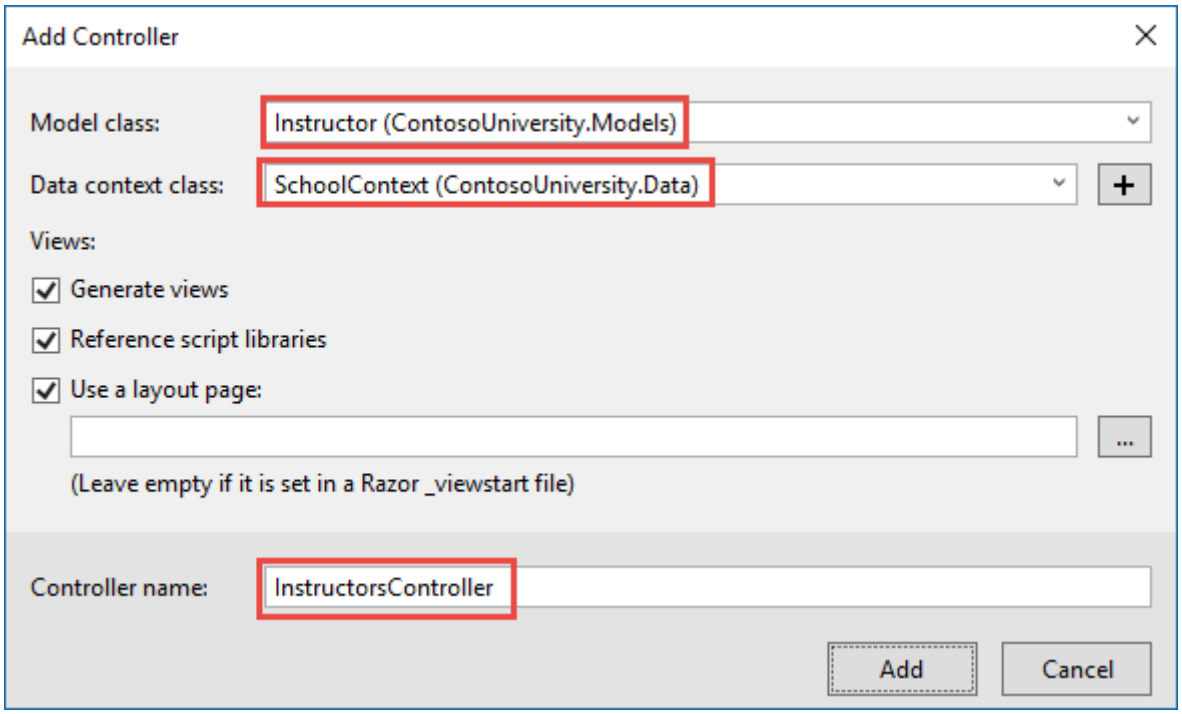
In the *SchoolViewModels* folder, create `InstructorIndexData.cs` and replace the existing code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class InstructorIndexData
    {
        public IEnumerable<Instructor> Instructors { get; set; }
        public IEnumerable<Course> Courses { get; set; }
        public IEnumerable<Enrollment> Enrollments { get; set; }
    }
}
```

Create the Instructor controller and views

Create an Instructors controller with EF read/write actions as shown in the following illustration:



Open `InstructorsController.cs` and add a using statement for the ViewModels namespace:

C#

```
using ContosoUniversity.Models.SchoolViewModels;
```

Replace the Index method with the following code to do eager loading of related data and put it in the view model.

C#

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Enrollments)
                    .ThenInclude(i => i.Student)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .AsNoTracking()
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
```

```
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s =>
s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        viewModel.Enrollments = viewModel.Courses.Where(
            x => x.CourseID == courseID).Single().Enrollments;
    }

    return View(viewModel);
}
```

The method accepts optional route data (`id`) and a query string parameter (`courseID`) that provide the ID values of the selected instructor and selected course. The parameters are provided by the **Select** hyperlinks on the page.

The code begins by creating an instance of the view model and putting in it the list of instructors. The code specifies eager loading for the `Instructor.OfficeAssignment` and the `Instructor.CourseAssignments` navigation properties. Within the `CourseAssignments` property, the `Course` property is loaded, and within that, the `Enrollments` and `Department` properties are loaded, and within each `Enrollment` entity the `Student` property is loaded.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

Since the view always requires the `OfficeAssignment` entity, it's more efficient to fetch that in the same query. Course entities are required when an instructor is selected in the web page, so a single query is better than multiple queries only if the page is displayed more often with a course selected than without.

The code repeats `CourseAssignments` and `Course` because you need two properties from `Course`. The first string of `ThenInclude` calls gets `CourseAssignment.Course`, `Course.Enrollments`, and `Enrollment.Student`.

You can read more about including multiple levels of related data here.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

At that point in the code, another `ThenInclude` would be for navigation properties of `Student`, which you don't need. But calling `Include` starts over with `Instructor` properties, so you have to go through the chain again, this time specifying `Course.Department` instead of `Course.Enrollments`.

C#

```
viewModel.Instructors = await _context.Instructors
      .Include(i => i.OfficeAssignment)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Enrollments)
                .ThenInclude(i => i.Student)
      .Include(i => i.CourseAssignments)
        .ThenInclude(i => i.Course)
            .ThenInclude(i => i.Department)
      .AsNoTracking()
      .OrderBy(i => i.LastName)
      .ToListAsync();
```

The following code executes when an instructor was selected. The selected instructor is retrieved from the list of instructors in the view model. The view model's `Courses` property is then loaded with the `Course` entities from that instructor's `CourseAssignments` navigation property.

C#

```
if (id != null)
{
    ViewData["InstructorID"] = id.Value;
    Instructor instructor = viewModel.Instructors.Where(
        i => i.ID == id.Value).Single();
    viewModel.Courses = instructor.CourseAssignments.Select(s => s.Course);
}
```

The `Where` method returns a collection, but in this case the criteria passed to that method result in only a single Instructor entity being returned. The `Single` method converts the collection into a single `Instructor` entity, which gives you access to that entity's `CourseAssignments` property. The `CourseAssignments` property contains `CourseAssignment` entities, from which you want only the related `Course` entities.

You use the `Single` method on a collection when you know the collection will have only one item. The `Single` method throws an exception if the collection passed to it's empty or if there's more than one item. An alternative is `SingleOrDefault`, which returns a default value (null in this case) if the collection is empty. However, in this case that would still result in an exception (from trying to find a `Courses` property on a null reference), and the exception message would less clearly indicate the cause of the problem. When you call the `Single` method, you can also pass in the Where condition instead of calling the `Where` method separately:

C#

```
.Single(i => i.ID == id.Value)
```

Instead of:

C#

```
.Where(i => i.ID == id.Value).Single()
```

Next, if a course was selected, the selected course is retrieved from the list of courses in the view model. Then the view model's `Enrollments` property is loaded with the Enrollment entities from that course's `Enrollments` navigation property.

C#

```
if (courseID != null)
{
    ViewData["CourseID"] = courseID.Value;
    viewModel.Enrollments = viewModel.Courses.Where(
```

```
        x => x.CourseID == courseID).Single().Enrollments;
}
```

Tracking vs no-tracking

No-tracking queries are useful when the results are used in a read-only scenario. They're generally quicker to execute because there's no need to set up the change tracking information. If the entities retrieved from the database don't need to be updated, then a no-tracking query is likely to perform better than a tracking query.

In some cases a tracking query is more efficient than a no-tracking query. For more information, see Tracking vs. No-Tracking Queries.

Modify the Instructor Index view

In `Views/Instructors/Index.cshtml`, replace the template code with the following code. The changes are highlighted.

CSHTML

```
@model ContosoUniversity.Models.SchoolViewModels.InstructorIndexData

@{
    ViewData["Title"] = "Instructors";
}

<h2>Instructors</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>Last Name</th>
            <th>First Name</th>
            <th>Hire Date</th>
            <th>Office</th>
            <th>Courses</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Instructors)
        {
```

```
            string selectedRow = "";
            if (item.ID == (int?)ViewData["InstructorID"])
            {
                selectedRow = "table-success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.DisplayFor(modelItem => item.LastName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.FirstMidName)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.HireDate)
                </td>
                <td>
                    @if (item.OfficeAssignment != null)
                    {
                        @item.OfficeAssignment.Location
                    }
                </td>
                <td>
                    @foreach (var course in item.CourseAssignments)
                    {
                        @course.Course.CourseID @course.Course.Title <br />
                    }
                </td>
                <td>
                    <a asp-action="Index" asp-route-id="@item.ID">Select</a>
|
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.ID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.ID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

You've made the following changes to the existing code:

- Changed the model class to `InstructorIndexData`.
- Changed the page title from **Index** to **Instructors**.
- Added an **Office** column that displays `item.OfficeAssignment.Location` only if `item.OfficeAssignment` isn't null. (Because this is a one-to-zero-or-one relationship, there might not be a related OfficeAssignment entity.)

CSHTML

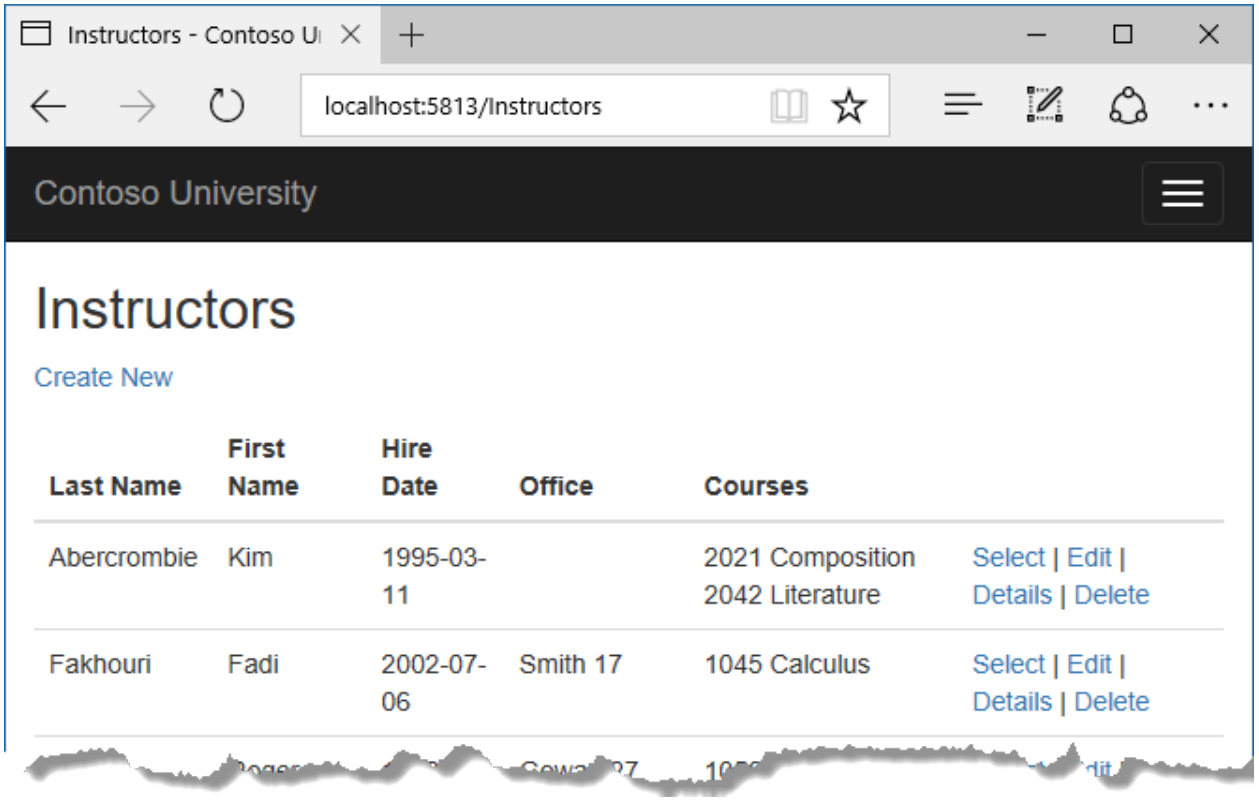
```
@if (item.OfficeAssignment != null)
{
    @item.OfficeAssignment.Location
}
```

- Added a **Courses** column that displays courses taught by each instructor. For more information, see the Explicit line transition section of the Razor syntax article.
- Added code that conditionally adds a Bootstrap CSS class to the `tr` element of the selected instructor. This class sets a background color for the selected row.
- Added a new hyperlink labeled **Select** immediately before the other links in each row, which causes the selected instructor's ID to be sent to the `Index` method.

CSHTML

```
<a asp-action="Index" asp-route-id="@item.ID">Select</a> |
```

Run the app and select the **Instructors** tab. The page displays the Location property of related OfficeAssignment entities and an empty table cell when there's no related OfficeAssignment entity.



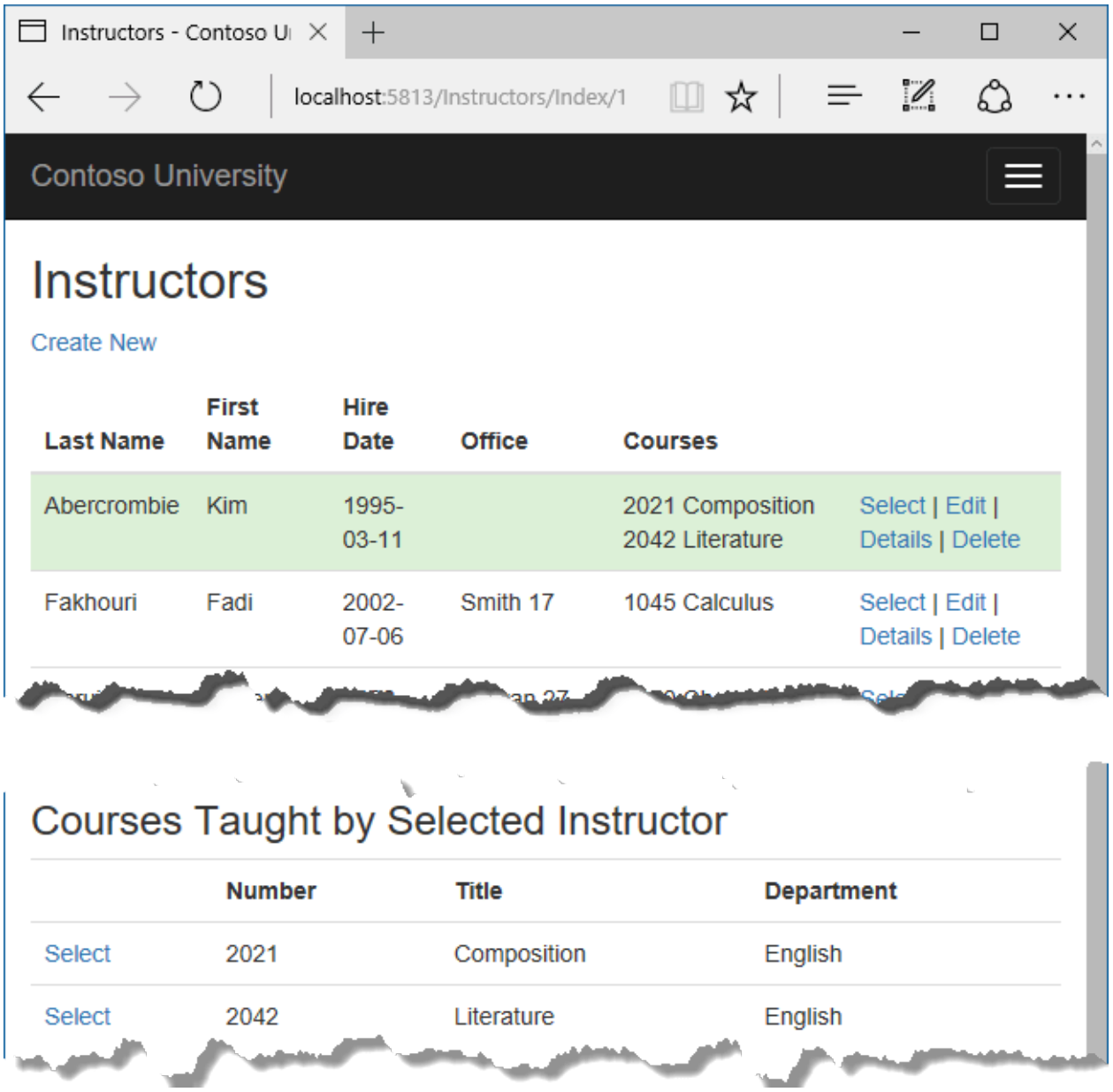
In the `Views/Instructors/Index.cshtml` file, after the closing table element (at the end of the file), add the following code. This code displays a list of courses related to an instructor when an instructor is selected.

CSHTML

```
@if (Model.Courses != null)
{
    <h3>Courses Taught by Selected Instructor</h3>
    <table class="table">
        <tr>
            <th></th>
            <th>Number</th>
            <th>Title</th>
            <th>Department</th>
        </tr>

        @foreach (var item in Model.Courses)
        {
            string selectedRow = "";
            if (item.CourseID == (int?)ViewData["CourseID"])
            {
                selectedRow = "success";
            }
            <tr class="@selectedRow">
                <td>
                    @Html.ActionLink("Select", "Index", new { courseID =
item.CourseID })
                </td>
                <td>
                    @item.CourseID
                </td>
                <td>
                    @item.Title
                </td>
                <td>
                    @item.Department.Name
                </td>
            </tr>
        }

    </table>
}
```

This code reads the `Courses` property of the view model to display a list of courses. It also provides a **Select** hyperlink that sends the ID of the selected course to the `Index` action method.

Refresh the page and select an instructor. Now you see a grid that displays courses assigned to the selected instructor, and for each course you see the name of the assigned department.



After the code block you just added, add the following code. This displays a list of the students who are enrolled in a course when that course is selected.

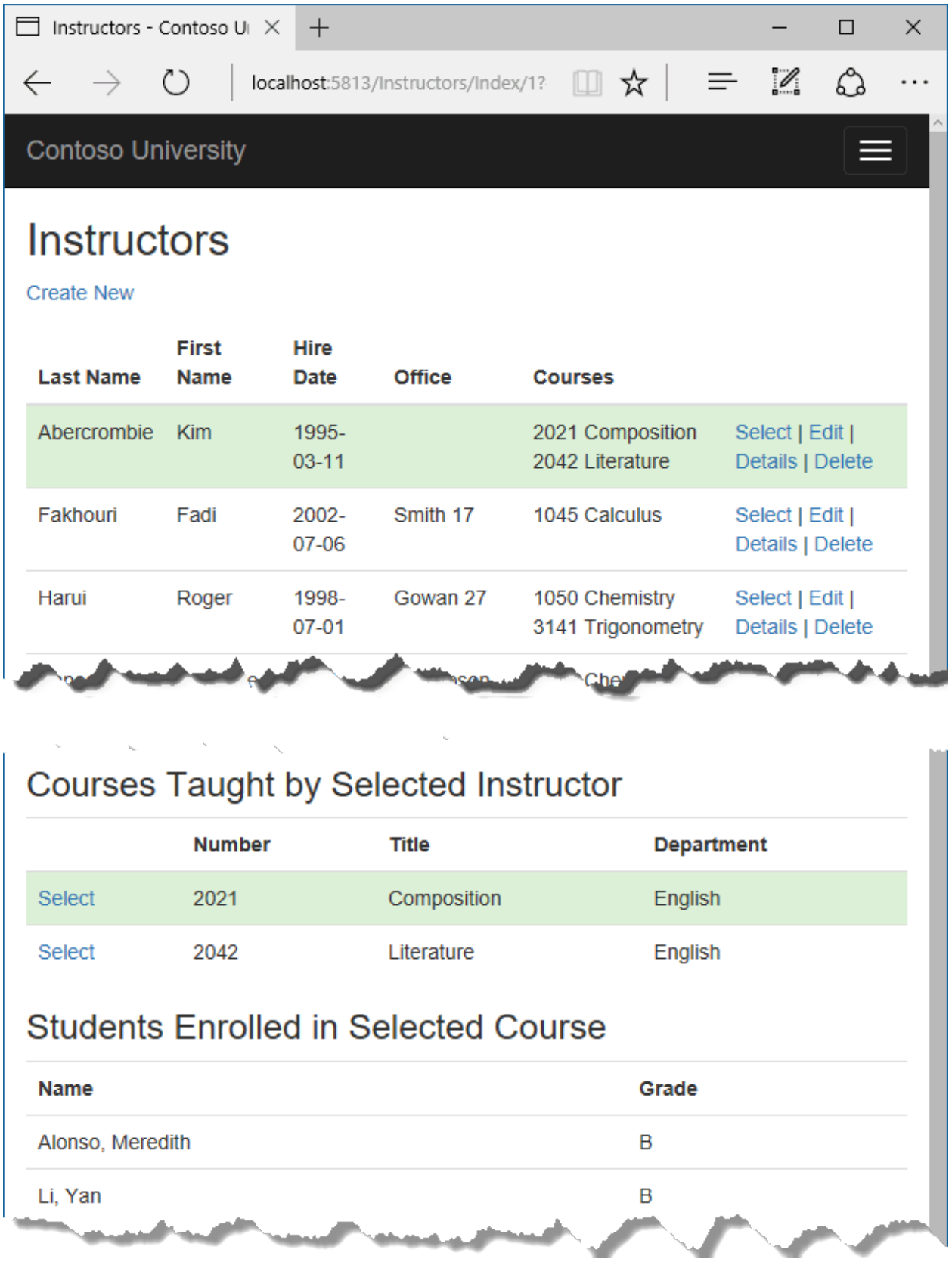
CSHTML

```
@if (Model.Enrollments != null)
{
    <h3>
        Students Enrolled in Selected Course
    </h3>
    <table class="table">
        <tr>
            <th>Name</th>
            <th>Grade</th>
        </tr>
        @foreach (var item in Model.Enrollments)
        {
            <tr>
                <td>
                    @item.Student.FullName
```

```
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Grade)
                </td>
            </tr>
        }
    </table>
}
```

This code reads the `Enrollments` property of the view model in order to display a list of students enrolled in the course.

Refresh the page again and select an instructor. Then select a course to see the list of enrolled students and their grades.



About explicit loading

When you retrieved the list of instructors in `InstructorsController.cs`, you specified eager loading for the `CourseAssignments` navigation property.

Suppose you expected users to only rarely want to see enrollments in a selected instructor and course. In that case, you might want to load the enrollment data only if

it's requested. To see an example of how to do explicit loading, replace the `Index` method with the following code, which removes eager loading for `Enrollments` and loads that property explicitly. The code changes are highlighted.

C#

```
public async Task<IActionResult> Index(int? id, int? courseID)
{
    var viewModel = new InstructorIndexData();
    viewModel.Instructors = await _context.Instructors
          .Include(i => i.OfficeAssignment)
          .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
                .ThenInclude(i => i.Department)
          .OrderBy(i => i.LastName)
          .ToListAsync();

    if (id != null)
    {
        ViewData["InstructorID"] = id.Value;
        Instructor instructor = viewModel.Instructors.Where(
            i => i.ID == id.Value).Single();
        viewModel.Courses = instructor.CourseAssignments.Select(s =>
s.Course);
    }

    if (courseID != null)
    {
        ViewData["CourseID"] = courseID.Value;
        var selectedCourse = viewModel.Courses.Where(x => x.CourseID ==
courseID).Single();
        await _context.Entry(selectedCourse).Collection(x =>
x.Enrollments).LoadAsync();
        foreach (Enrollment enrollment in selectedCourse.Enrollments)
        {
            await _context.Entry(enrollment).Reference(x =>
x.Student).LoadAsync();
        }
        viewModel.Enrollments = selectedCourse.Enrollments;
    }

    return View(viewModel);
}
```

The new code drops the `ThenInclude` method calls for enrollment data from the code that retrieves instructor entities. It also drops `AsNoTracking`. If an instructor and course are selected, the highlighted code retrieves `Enrollment` entities for the selected course, and `Student` entities for each `Enrollment`.

Run the app, go to the Instructors Index page now and you'll see no difference in what's displayed on the page, although you've changed how the data is retrieved.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Learned how to load related data
- ✓ Created a Courses page
- ✓ Created an Instructors page
- ✓ Learned about explicit loading

Advance to the next tutorial to learn how to update related data.

Update related data

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

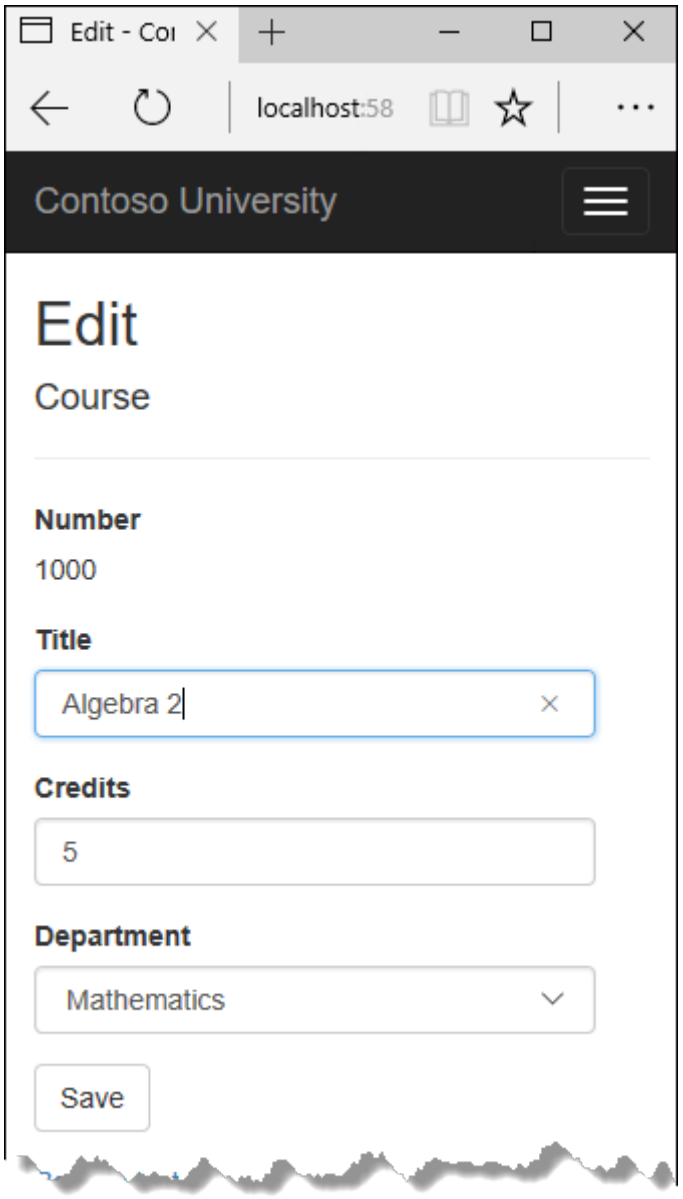
Provide product feedback

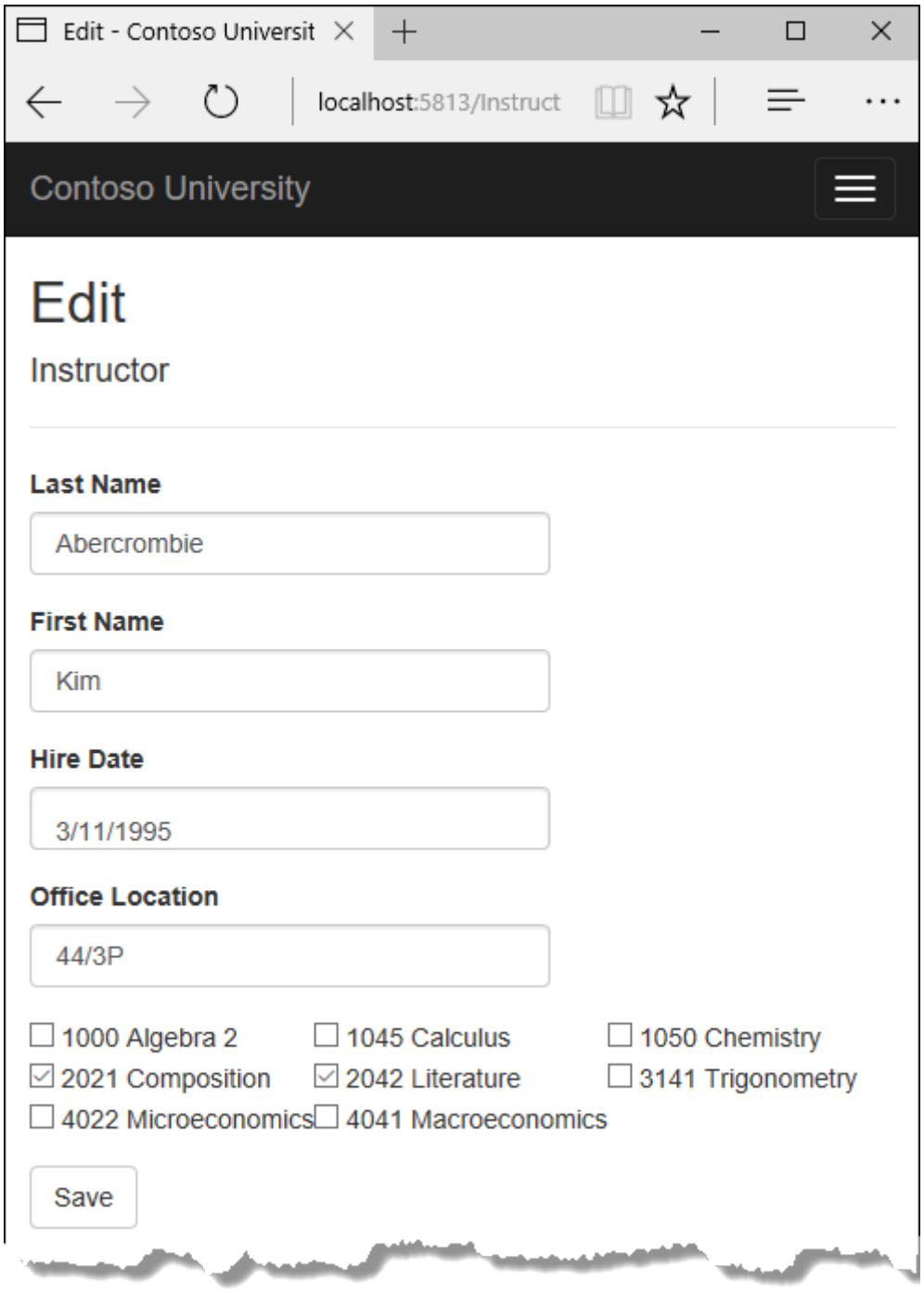
Tutorial: Update related data - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial you displayed related data; in this tutorial you'll update related data by updating foreign key fields and navigation properties.

The following illustrations show some of the pages that you'll work with.





In this tutorial, you:

- ✓ Customize Courses pages
- ✓ Add Instructors Edit page
- ✓ Add courses to Edit page
- ✓ Update Delete page
- ✓ Add office location and courses to Create page

Prerequisites

- Read related data

Customize Courses pages

When a new `Course` entity is created, it must have a relationship to an existing department. To facilitate this, the scaffolded code includes controller methods and Create and Edit views that include a drop-down list for selecting the department. The drop-down list sets the `Course.DepartmentID` foreign key property, and that's all the Entity Framework needs in order to load the `Department` navigation property with the appropriate `Department` entity. You'll use the scaffolded code, but change it slightly to add error handling and sort the drop-down list.

In `CoursesController.cs`, delete the four Create and Edit methods and replace them with the following code:

C#

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("CourseID,Credits,DepartmentID,Title")] Course course)
{
    if (ModelState.IsValid)
    {
        _context.Add(course);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }
```

```
    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var courseToUpdate = await _context.Courses
        .FirstOrDefaultAsync(c => c.CourseID == id);

    if (await TryUpdateModelAsync<Course>(courseToUpdate,
        "",
        c => c.Credits, c => c.DepartmentID, c => c.Title))
    {
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    PopulateDepartmentsDropDownList(courseToUpdate.DepartmentID);
    return View(courseToUpdate);
}
```

After the `Edit` HttpPost method, create a new method that loads department info for the drop-down list.

C#

```
private void PopulateDepartmentsDropDownList(object selectedDepartment =
null)
{
    var departmentsQuery = from d in _context.Departments
                           orderby d.Name
                           select d;
    ViewBag.DepartmentID = new SelectList(departmentsQuery.AsNoTracking(),
"DepartmentID", "Name", selectedDepartment);
}
```

The `PopulateDepartmentsDropDownList` method gets a list of all departments sorted by name, creates a `SelectList` collection for a drop-down list, and passes the collection to the view in `ViewBag`. The method accepts the optional `selectedDepartment` parameter that allows the calling code to specify the item that will be selected when the drop-down list is rendered. The view will pass the name "DepartmentID" to the `<select>` tag helper, and the helper then knows to look in the `ViewBag` object for a `SelectList` named "DepartmentID".

The HttpGet `Create` method calls the `PopulateDepartmentsDropDownList` method without setting the selected item, because for a new course the department isn't established yet:

C#

```
public IActionResult Create()
{
    PopulateDepartmentsDropDownList();
    return View();
}
```

The HttpGet `Edit` method sets the selected item, based on the ID of the department that's already assigned to the course being edited:

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
```

```
    }
    PopulateDepartmentsDropDownList(course.DepartmentID);
    return View(course);
}
```

The HttpPost methods for both `Create` and `Edit` also include code that sets the selected item when they redisplay the page after an error. This ensures that when the page is redisplayed to show the error message, whatever department was selected stays selected.

Add .AsNoTracking to Details and Delete methods

To optimize performance of the Course Details and Delete pages, add `AsNoTracking` calls in the `Details` and HttpGet `Delete` methods.

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

C#

```
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var course = await _context.Courses
        .Include(c => c.Department)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.CourseID == id);
```

```
    if (course == null)
    {
        return NotFound();
    }

    return View(course);
}
```

Modify the Course views

In `Views/Courses/Create.cshtml`, add a "Select Department" option to the **Department** drop-down list, change the caption from **DepartmentID** to **Department**, and add a validation message.

CSHTML

```
<div class="form-group">
    <label asp-for="Department" class="control-label"></label>
    <select asp-for="DepartmentID" class="form-control" asp-
items="ViewBag.DepartmentID">
        <option value="">-- Select Department --</option>
    </select>
    <span asp-validation-for="DepartmentID" class="text-danger" />
</div>
```

In `Views/Courses/Edit.cshtml`, make the same change for the Department field that you just did in `Create.cshtml`.

Also in `Views/Courses/Edit.cshtml`, add a course number field before the **Title** field. Because the course number is the primary key, it's displayed, but it can't be changed.

CSHTML

```
<div class="form-group">
    <label asp-for="CourseID" class="control-label"></label>
    <div>@Html.DisplayFor(model => model.CourseID)</div>
</div>
```

There's already a hidden field (`<input type="hidden">`) for the course number in the Edit view. Adding a `<label>` tag helper doesn't eliminate the need for the hidden field because it doesn't cause the course number to be included in the posted data when the user clicks **Save** on the **Edit** page.

In `Views/Courses/Delete.cshtml`, add a course number field at the top and change department ID to department name.

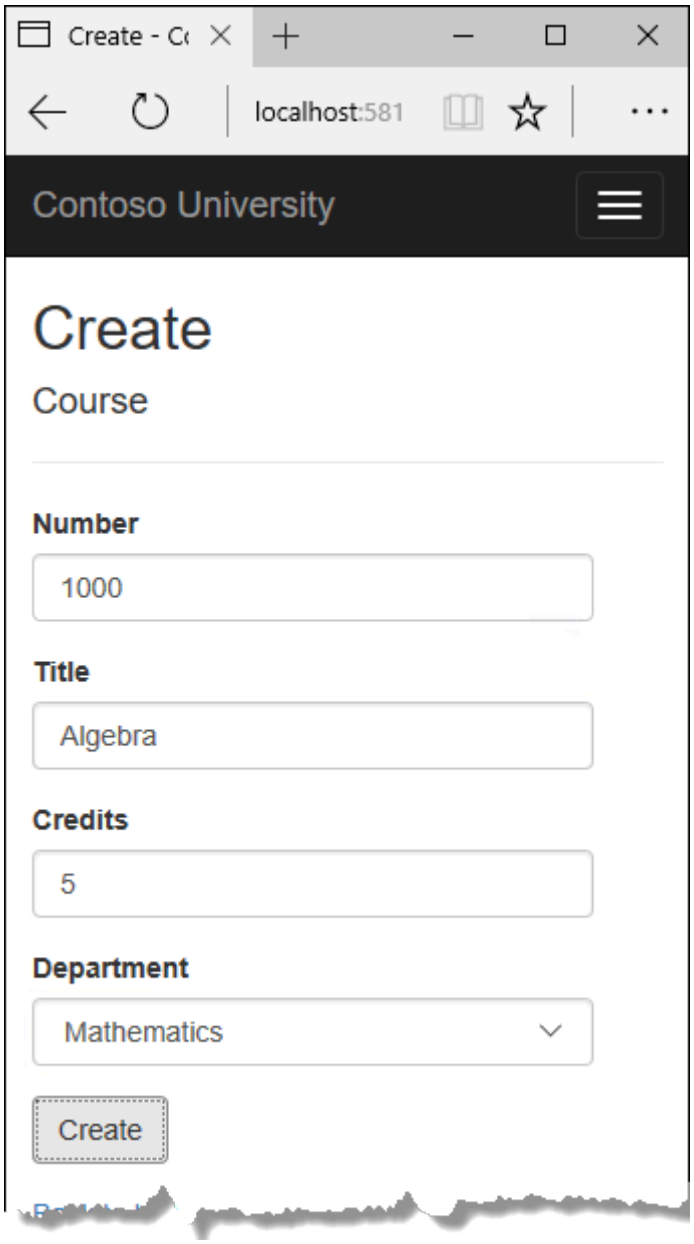
CSHTML

```
@model ContosoUniversity.Models.Course

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Course</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.CourseID)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.CourseID)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Credits)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Credits)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Department)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Department.Name)
        </dd>
    </dl>

    <form asp-action="Delete">
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

In `Views/Courses/Details.cshtml`, make the same change that you just did for `Delete.cshtml`.

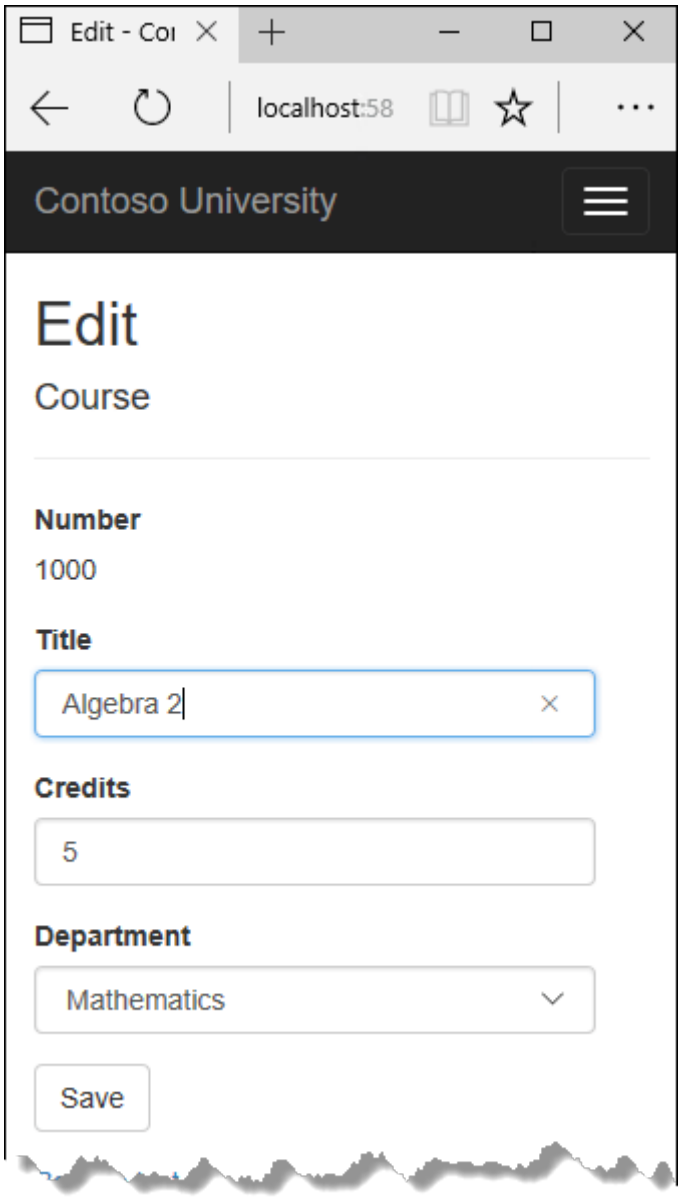
Test the Course pages

Run the app, select the **Courses** tab, click **Create New**, and enter data for a new course:



Click **Create**. The Courses Index page is displayed with the new course added to the list. The department name in the Index page list comes from the navigation property, showing that the relationship was established correctly.

Click **Edit** on a course in the Courses Index page.



Change data on the page and click **Save**. The Courses Index page is displayed with the updated course data.

Add Instructors Edit page

When you edit an instructor record, you want to be able to update the instructor's office assignment. The `Instructor` entity has a one-to-zero-or-one relationship with the `OfficeAssignment` entity, which means your code has to handle the following situations:

- If the user clears the office assignment and it originally had a value, delete the `OfficeAssignment` entity.
- If the user enters an office assignment value and it originally was empty, create a new `OfficeAssignment` entity.
- If the user changes the value of an office assignment, change the value in an existing `OfficeAssignment` entity.

Update the Instructors controller

In `InstructorsController.cs`, change the code in the HttpGet `Edit` method so that it loads the Instructor entity's `OfficeAssignment` navigation property and calls `AsNoTracking`:

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    return View(instructor);
}
```

Replace the HttpPost `Edit` method with the following code to handle office assignment updates:

C#

```
[HttpPost, ActionName("Edit")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> EditPost(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .FirstOrDefaultAsync(s => s.ID == id);

    if (await TryUpdateModelAsync<Instructor>(
        instructorToUpdate,
        "",
        i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
    {
```

```
        if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
        {
            instructorToUpdate.OfficeAssignment = null;
        }
        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateException /* ex */)
        {
            //Log the error (uncomment ex variable name and write a log.)
            ModelState.AddModelError("", "Unable to save changes. " +
                "Try again, and if the problem persists, " +
                "see your system administrator.");
        }
        return RedirectToAction(nameof(Index));
    }
    return View(instructorToUpdate);
}
```

The code does the following:

- Changes the method name to `EditPost` because the signature is now the same as the HttpGet `Edit` method (the `ActionName` attribute specifies that the `/Edit/` URL is still used).
- Gets the current `Instructor` entity from the database using eager loading for the `OfficeAssignment` navigation property. This is the same as what you did in the HttpGet `Edit` method.
- Updates the retrieved `Instructor` entity with values from the model binder. The `TryUpdateModel` overload enables you to declare the properties you want to include. This prevents over-posting, as explained in the second tutorial.

  C#

  ```
  if (await TryUpdateModelAsync<Instructor>(
      instructorToUpdate,
      "",
      i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
  i.OfficeAssignment))
  ```

- If the office location is blank, sets the `Instructor.OfficeAssignment` property to null so that the related row in the `OfficeAssignment` table will be deleted.

  C#

```
if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Locatio
n))
{
    instructorToUpdate.OfficeAssignment = null;
}
```

- Saves the changes to the database.

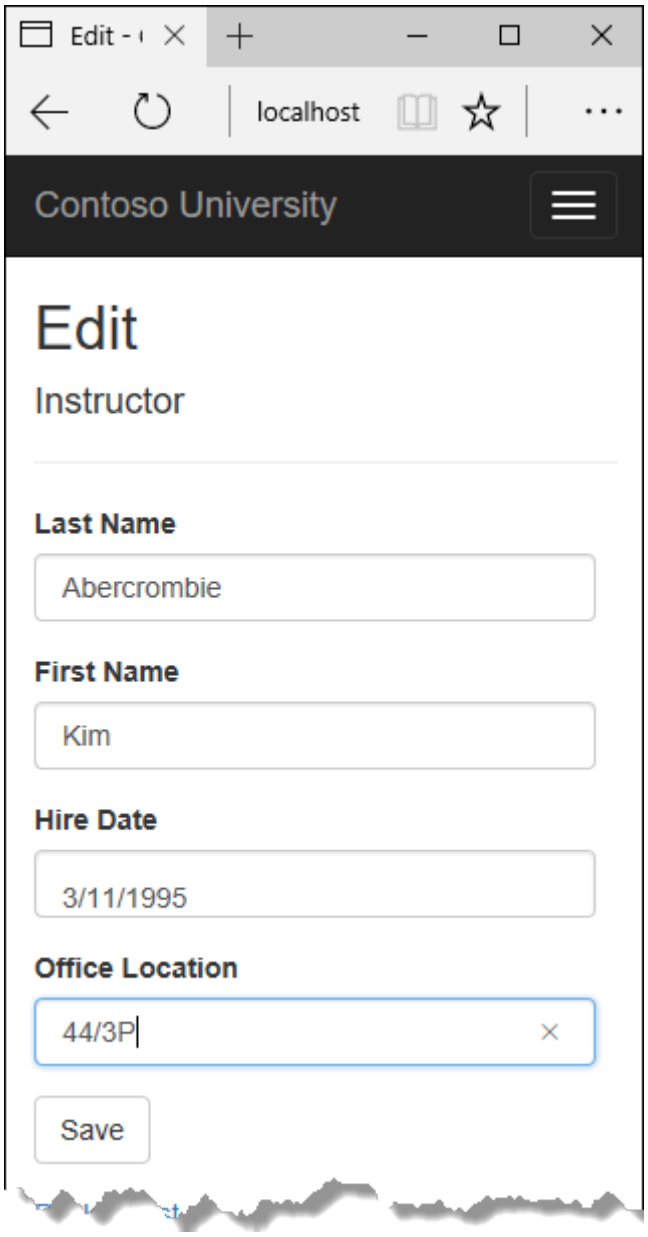
Update the Instructor Edit view

In `Views/Instructors/Edit.cshtml`, add a new field for editing the office location, at the end before the **Save** button:

CSHTML

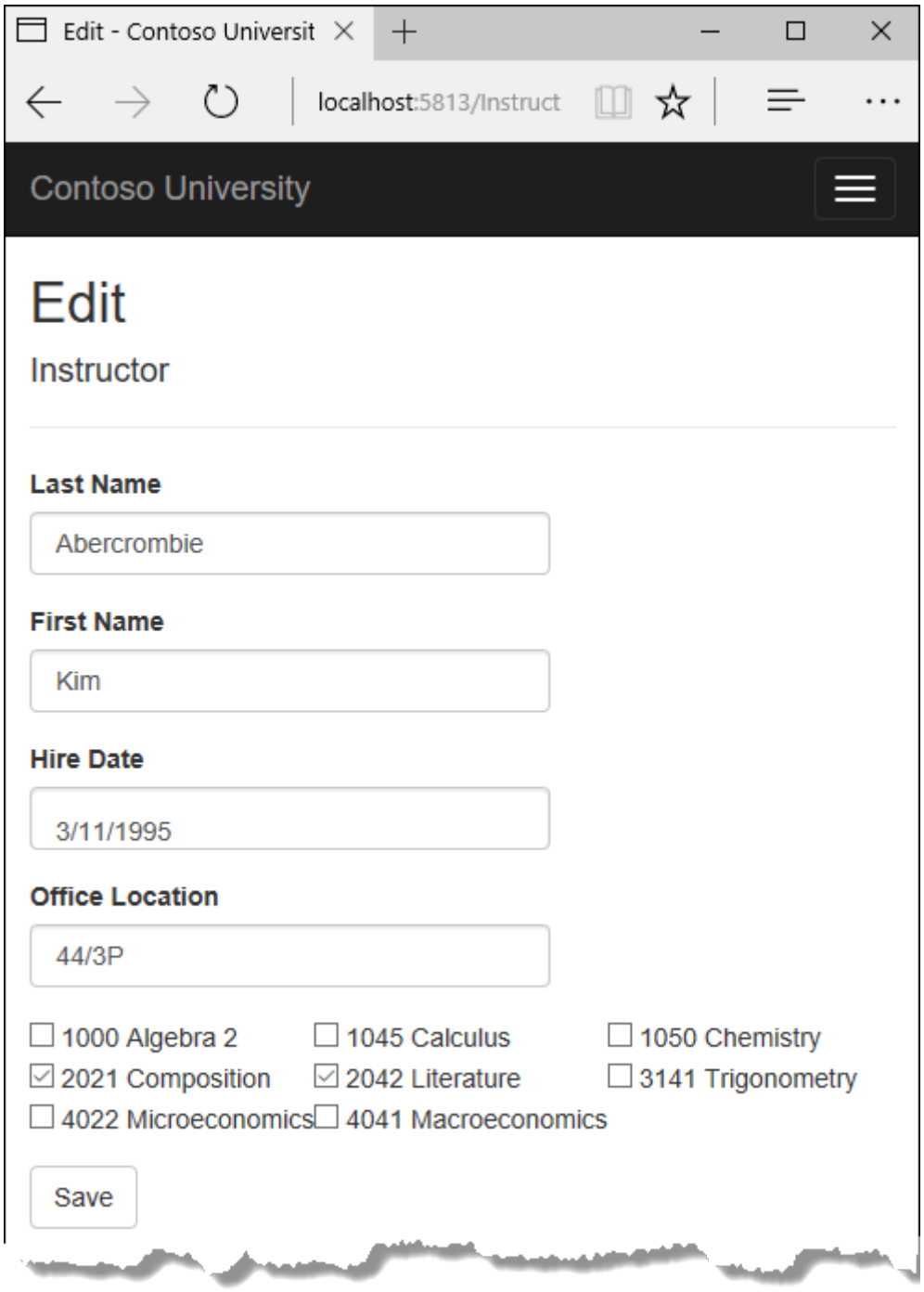
```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label">
</label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger"
/>
</div>
```

Run the app, select the **Instructors** tab, and then click **Edit** on an instructor. Change the **Office Location** and click **Save**.



Add courses to Edit page

Instructors may teach any number of courses. Now you'll enhance the Instructor Edit page by adding the ability to change course assignments using a group of checkboxes, as shown in the following screen shot:



The relationship between the `Course` and `Instructor` entities is many-to-many. To add and remove relationships, you add and remove entities to and from the `CourseAssignments` join entity set.

The UI that enables you to change which courses an instructor is assigned to is a group of checkboxes. A checkbox for every course in the database is displayed, and the ones that the instructor is currently assigned to are selected. The user can select or clear checkboxes to change course assignments. If the number of courses were much greater, you would probably want to use a different method of presenting the data in the view, but you'd use the same method of manipulating a join entity to create or delete relationships.

Update the Instructors controller

To provide data to the view for the list of checkboxes, you'll use a view model class.

Create `AssignedCourseData.cs` in the *SchoolViewModels* folder and replace the existing code with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ContosoUniversity.Models.SchoolViewModels
{
    public class AssignedCourseData
    {
        public int CourseID { get; set; }
        public string Title { get; set; }
        public bool Assigned { get; set; }
    }
}
```

In `InstructorsController.cs`, replace the HttpGet `Edit` method with the following code. The changes are highlighted.

C#

```
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructor = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments).ThenInclude(i => i.Course)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.ID == id);
    if (instructor == null)
    {
        return NotFound();
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}

private void PopulateAssignedCourseData(Instructor instructor)
{
```

```
    var allCourses = _context.Courses;
    var instructorCourses = new HashSet<int>
(instructor.CourseAssignments.Select(c => c.CourseID));
    var viewModel = new List<AssignedCourseData>();
    foreach (var course in allCourses)
    {
        viewModel.Add(new AssignedCourseData
        {
            CourseID = course.CourseID,
            Title = course.Title,
            Assigned = instructorCourses.Contains(course.CourseID)
        });
    }
    ViewData["Courses"] = viewModel;
}
```

The code adds eager loading for the `Courses` navigation property and calls the new `PopulateAssignedCourseData` method to provide information for the checkbox array using the `AssignedCourseData` view model class.

The code in the `PopulateAssignedCourseData` method reads through all `Course` entities in order to load a list of courses using the view model class. For each course, the code checks whether the course exists in the instructor's `Courses` navigation property. To create efficient lookup when checking whether a course is assigned to the instructor, the courses assigned to the instructor are put into a `HashSet` collection. The `Assigned` property is set to true for courses the instructor is assigned to. The view will use this property to determine which checkboxes must be displayed as selected. Finally, the list is passed to the view in `ViewData`.

Next, add the code that's executed when the user clicks **Save**. Replace the `EditPost` method with the following code, and add a new method that updates the `Courses` navigation property of the Instructor entity.

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, string[] selectedCourses)
{
    if (id == null)
    {
        return NotFound();
    }

    var instructorToUpdate = await _context.Instructors
        .Include(i => i.OfficeAssignment)
        .Include(i => i.CourseAssignments)
            .ThenInclude(i => i.Course)
```

```
            .FirstOrDefaultAsync(m => m.ID == id);

        if (await TryUpdateModelAsync<Instructor>(
            instructorToUpdate,
            "",
            i => i.FirstMidName, i => i.LastName, i => i.HireDate, i =>
i.OfficeAssignment))
        {
            if
(String.IsNullOrWhiteSpace(instructorToUpdate.OfficeAssignment?.Location))
            {
                instructorToUpdate.OfficeAssignment = null;
            }
            UpdateInstructorCourses(selectedCourses, instructorToUpdate);
            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateException /* ex */)
            {
                //Log the error (uncomment ex variable name and write a log.)
                ModelState.AddModelError("", "Unable to save changes. " +
                    "Try again, and if the problem persists, " +
                    "see your system administrator.");
            }
            return RedirectToAction(nameof(Index));
        }
        UpdateInstructorCourses(selectedCourses, instructorToUpdate);
        PopulateAssignedCourseData(instructorToUpdate);
        return View(instructorToUpdate);
    }
```

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The method signature is now different from the HttpGet `Edit` method, so the method name changes from `EditPost` back to `Edit`.

Since the view doesn't have a collection of Course entities, the model binder can't automatically update the `CourseAssignments` navigation property. Instead of using the model binder to update the `CourseAssignments` navigation property, you do that in the new `UpdateInstructorCourses` method. Therefore, you need to exclude the `CourseAssignments` property from model binding. This doesn't require any change to the code that calls `TryUpdateModel` because you're using the overload that requires explicit approval and `CourseAssignments` isn't in the include list.

If no checkboxes were selected, the code in `UpdateInstructorCourses` initializes the `CourseAssignments` navigation property with an empty collection and returns:

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

The code then loops through all courses in the database and checks each course against the ones currently assigned to the instructor versus the ones that were selected in the view. To facilitate efficient lookups, the latter two collections are stored in `HashSet` objects.

If the checkbox for a course was selected but the course isn't in the `Instructor.CourseAssignments` navigation property, the course is added to the collection

in the navigation property.

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

If the checkbox for a course wasn't selected, but the course is in the `Instructor.CourseAssignments` navigation property, the course is removed from the navigation property.

C#

```
private void UpdateInstructorCourses(string[] selectedCourses, Instructor
instructorToUpdate)
{
    if (selectedCourses == null)
```

```
    {
        instructorToUpdate.CourseAssignments = new List<CourseAssignment>();
        return;
    }

    var selectedCoursesHS = new HashSet<string>(selectedCourses);
    var instructorCourses = new HashSet<int>
        (instructorToUpdate.CourseAssignments.Select(c =>
c.Course.CourseID));
    foreach (var course in _context.Courses)
    {
        if (selectedCoursesHS.Contains(course.CourseID.ToString()))
        {
            if (!instructorCourses.Contains(course.CourseID))
            {
                instructorToUpdate.CourseAssignments.Add(new
CourseAssignment { InstructorID = instructorToUpdate.ID, CourseID =
course.CourseID });
            }
        }
        else
        {

            if (instructorCourses.Contains(course.CourseID))
            {
                CourseAssignment courseToRemove =
instructorToUpdate.CourseAssignments.FirstOrDefault(i => i.CourseID ==
course.CourseID);
                _context.Remove(courseToRemove);
            }
        }
    }
}
```

Update the Instructor views

In `Views/Instructors/Edit.cshtml`, add a **Courses** field with an array of checkboxes by adding the following code immediately after the `div` elements for the **Office** field and before the `div` element for the **Save** button.

> ⓘ Note
>
> When you paste the code in Visual Studio, line breaks might be changed in a way that breaks the code. If the code looks different after pasting, press Ctrl+Z one time to undo the automatic formatting. This will fix the line breaks so that they look like what you see here. The indentation doesn't have to be perfect, but the `@:</tr><tr>`, `@:<td>`, `@:</td>`, and `@:</tr>` lines must each be on a single line as shown or you'll get a runtime error. With the block of new code selected, press Tab three

times to line up the new code with the existing code. This problem is fixed in Visual Studio 2019.

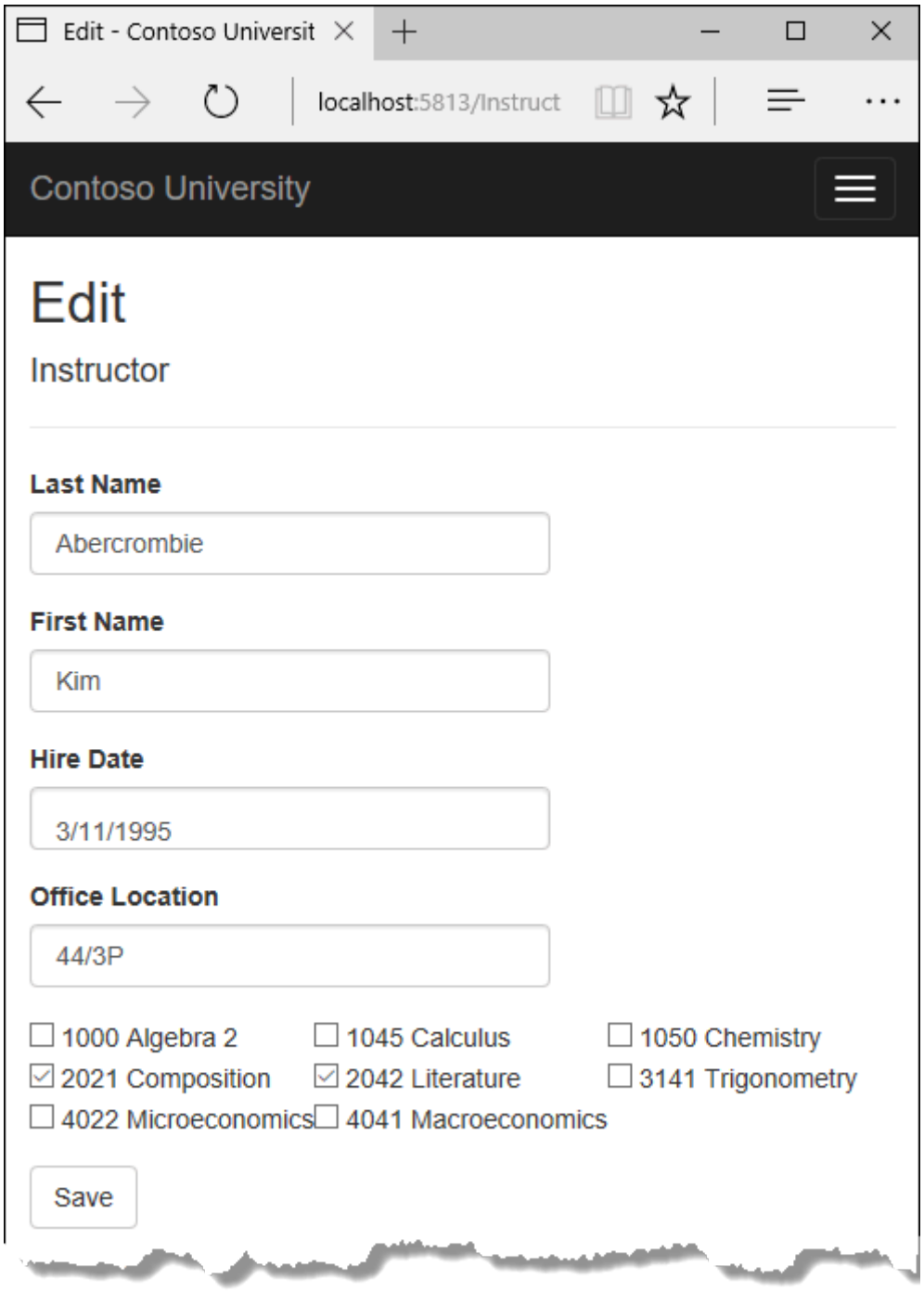
CSHTML

```
<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                        @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

This code creates an HTML table that has three columns. In each column is a checkbox followed by a caption that consists of the course number and title. The checkboxes all have the same name ("selectedCourses"), which informs the model binder that they're to be treated as a group. The value attribute of each checkbox is set to the value of `CourseID`. When the page is posted, the model binder passes an array to the controller that consists of the `CourseID` values for only the checkboxes which are selected.

When the checkboxes are initially rendered, those that are for courses assigned to the instructor have checked attributes, which selects them (displays them checked).

Run the app, select the **Instructors** tab, and click **Edit** on an instructor to see the **Edit** page.



Change some course assignments and click Save. The changes you make are reflected on the Index page.

> ⓘ **Note**
>
> The approach taken here to edit instructor course data works well when there's a limited number of courses. For collections that are much larger, a different UI and a different updating method would be required.

Update Delete page

In `InstructorsController.cs`, delete the `DeleteConfirmed` method and insert the following code in its place.

```
C#

[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    Instructor instructor = await _context.Instructors
        .Include(i => i.CourseAssignments)
        .SingleAsync(i => i.ID == id);

    var departments = await _context.Departments
        .Where(d => d.InstructorID == id)
        .ToListAsync();
    departments.ForEach(d => d.InstructorID = null);

    _context.Instructors.Remove(instructor);

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

This code makes the following changes:

- Does eager loading for the `CourseAssignments` navigation property. You have to include this or EF won't know about related `CourseAssignment` entities and won't delete them. To avoid needing to read them here you could configure cascade delete in the database.
- If the instructor to be deleted is assigned as administrator of any departments, removes the instructor assignment from those departments.

Add office location and courses to Create page

In `InstructorsController.cs`, delete the HttpGet and HttpPost `Create` methods, and then add the following code in their place:

```
C#

public IActionResult Create()
{
    var instructor = new Instructor();
    instructor.CourseAssignments = new List<CourseAssignment>();
    PopulateAssignedCourseData(instructor);
    return View();
}
```

```
// POST: Instructors/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("FirstMidName,HireDate,LastName,OfficeAssignment")] Instructor
instructor, string[] selectedCourses)
{
    if (selectedCourses != null)
    {
        instructor.CourseAssignments = new List<CourseAssignment>();
        foreach (var course in selectedCourses)
        {
            var courseToAdd = new CourseAssignment { InstructorID =
instructor.ID, CourseID = int.Parse(course) };
            instructor.CourseAssignments.Add(courseToAdd);
        }
    }
    if (ModelState.IsValid)
    {
        _context.Add(instructor);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    PopulateAssignedCourseData(instructor);
    return View(instructor);
}
```

This code is similar to what you saw for the `Edit` methods except that initially no courses are selected. The HttpGet `Create` method calls the `PopulateAssignedCourseData` method not because there might be courses selected but in order to provide an empty collection for the `foreach` loop in the view (otherwise the view code would throw a null reference exception).

The HttpPost `Create` method adds each selected course to the `CourseAssignments` navigation property before it checks for validation errors and adds the new instructor to the database. Courses are added even if there are model errors so that when there are model errors (for an example, the user keyed an invalid date), and the page is redisplayed with an error message, any course selections that were made are automatically restored.

Notice that in order to be able to add courses to the `CourseAssignments` navigation property you have to initialize the property as an empty collection:

C#

```
instructor.CourseAssignments = new List<CourseAssignment>();
```

As an alternative to doing this in controller code, you could do it in the `Instructor` model by changing the property getter to automatically create the collection if it doesn't exist, as shown in the following example:

C#

```
private ICollection<CourseAssignment> _courseAssignments;
public ICollection<CourseAssignment> CourseAssignments
{
    get
    {
        return _courseAssignments ?? (_courseAssignments = new
List<CourseAssignment>());
    }
    set
    {
        _courseAssignments = value;
    }
}
```

If you modify the `CourseAssignments` property in this way, you can remove the explicit property initialization code in the controller.

In `Views/Instructor/Create.cshtml`, add an office location text box and checkboxes for courses before the Submit button. As in the case of the Edit page, fix the formatting if Visual Studio reformats the code when you paste it.

CSHTML

```
<div class="form-group">
    <label asp-for="OfficeAssignment.Location" class="control-label">
</label>
    <input asp-for="OfficeAssignment.Location" class="form-control" />
    <span asp-validation-for="OfficeAssignment.Location" class="text-danger"
/>
</div>

<div class="form-group">
    <div class="col-md-offset-2 col-md-10">
        <table>
            <tr>
                @{
                    int cnt = 0;

List<ContosoUniversity.Models.SchoolViewModels.AssignedCourseData> courses =
ViewBag.Courses;

                    foreach (var course in courses)
                    {
                        if (cnt++ % 3 == 0)
```

```
                        {
                            @:</tr><tr>
                        }
                        @:<td>
                            <input type="checkbox"
                                   name="selectedCourses"
                                   value="@course.CourseID"
                                   @(Html.Raw(course.Assigned ?
"checked=\"checked\"" : "")) />
                                   @course.CourseID @:  @course.Title
                            @:</td>
                    }
                    @:</tr>
                }
        </table>
    </div>
</div>
```

Test by running the app and creating an instructor.

Handling Transactions

As explained in the CRUD tutorial, the Entity Framework implicitly implements transactions. For scenarios where you need more control -- for example, if you want to include operations done outside of Entity Framework in a transaction -- see Transactions.

Get the code

Download or view the completed application.

Next steps

In this tutorial, you:

- ✓ Customized Courses pages
- ✓ Added Instructors Edit page
- ✓ Added courses to Edit page
- ✓ Updated Delete page
- ✓ Added office location and courses to Create page

Advance to the next tutorial to learn how to handle concurrency conflicts.

Handle concurrency conflicts

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

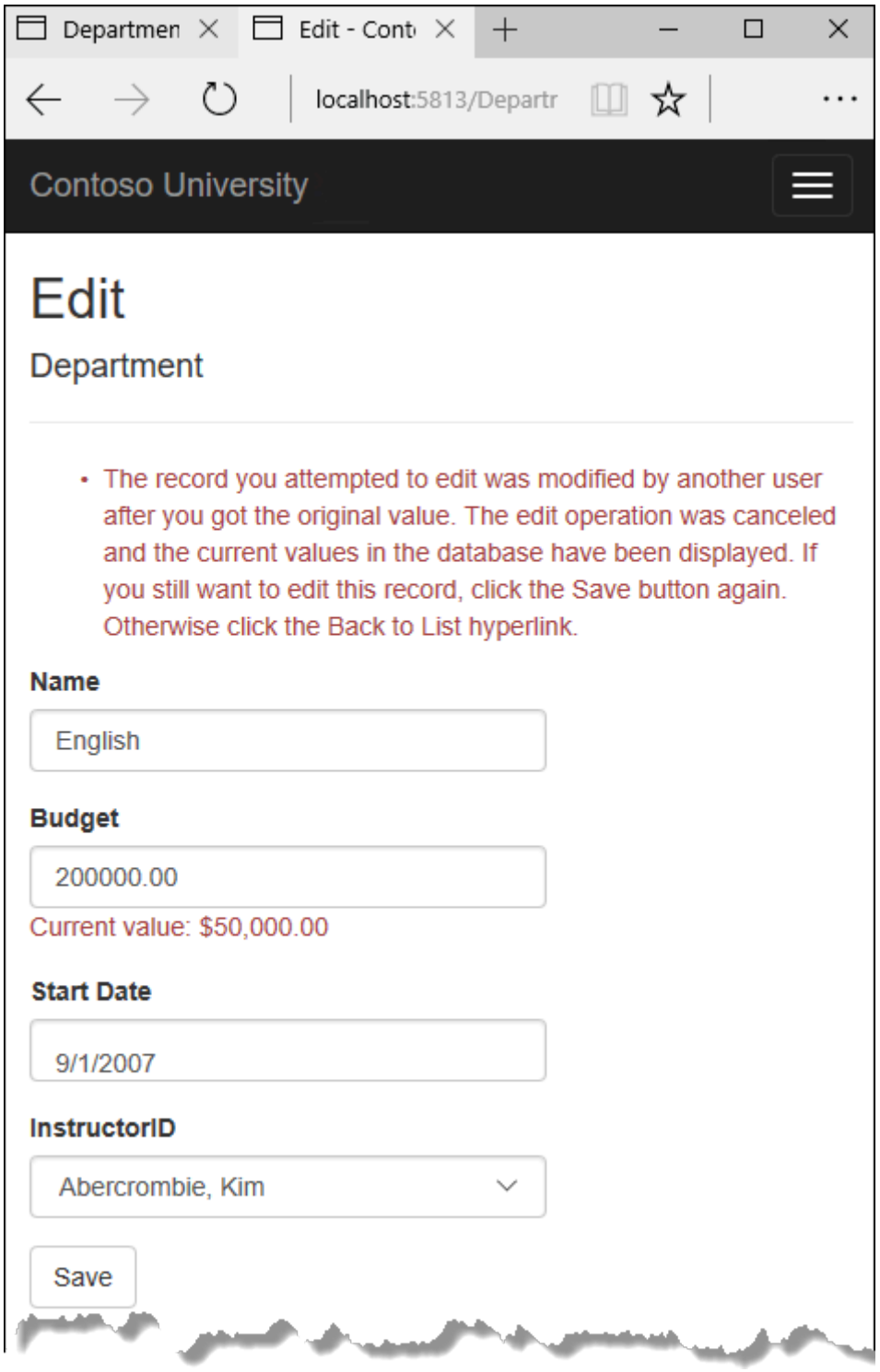
- Open a documentation issue
- Provide product feedback

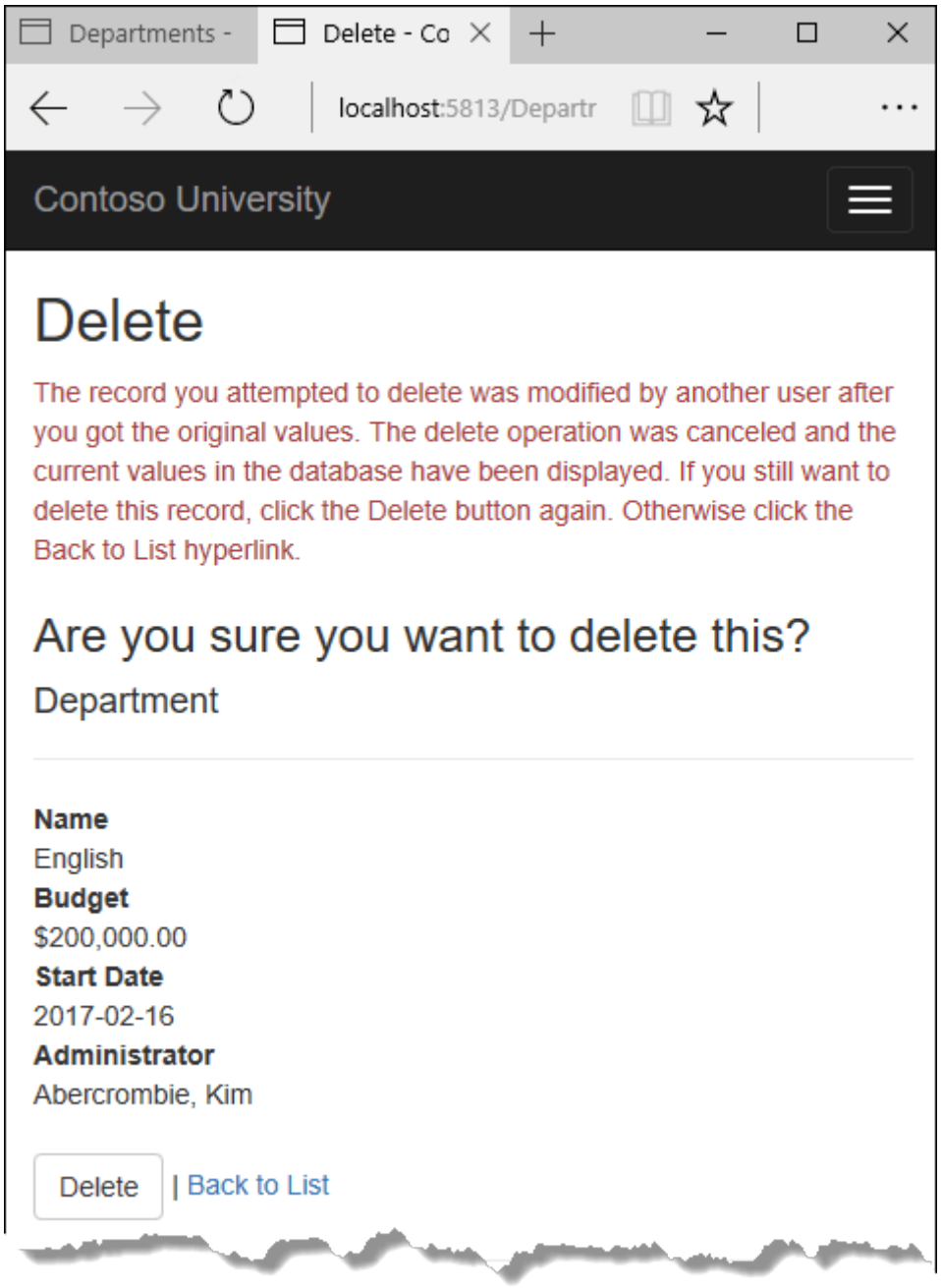
Tutorial: Handle concurrency - ASP.NET MVC with EF Core

Article • 04/11/2023

In earlier tutorials, you learned how to update data. This tutorial shows how to handle conflicts when multiple users update the same entity at the same time.

You'll create web pages that work with the `Department` entity and handle concurrency errors. The following illustrations show the Edit and Delete pages, including some messages that are displayed if a concurrency conflict occurs.


Departmen
Edit - Cont
localhost:5813/Departr
Contoso University
Edit
Department
The record you attempted to edit was modified by another user after you got the original value. The edit operation was canceled and the current values in the database have been displayed. If you still want to edit this record, click the Save button again. Otherwise click the Back to List hyperlink.
Name
English
Budget
200000.00
Current value: $50,000.00
Start Date
9/1/2007
InstructorID
Abercrombie, Kim
Save



In this tutorial, you:

- ✓ Learn about concurrency conflicts
- ✓ Add a tracking property
- ✓ Create Departments controller and views
- ✓ Update Index view
- ✓ Update Edit methods
- ✓ Update Edit view
- ✓ Test concurrency conflicts
- ✓ Update the Delete page
- ✓ Update Details and Create views

Prerequisites

- Update related data

Concurrency conflicts

A concurrency conflict occurs when one user displays an entity's data in order to edit it, and then another user updates the same entity's data before the first user's change is written to the database. If you don't enable the detection of such conflicts, whoever updates the database last overwrites the other user's changes. In many applications, this risk is acceptable: if there are few users, or few updates, or if isn't really critical if some changes are overwritten, the cost of programming for concurrency might outweigh the benefit. In that case, you don't have to configure the application to handle concurrency conflicts.

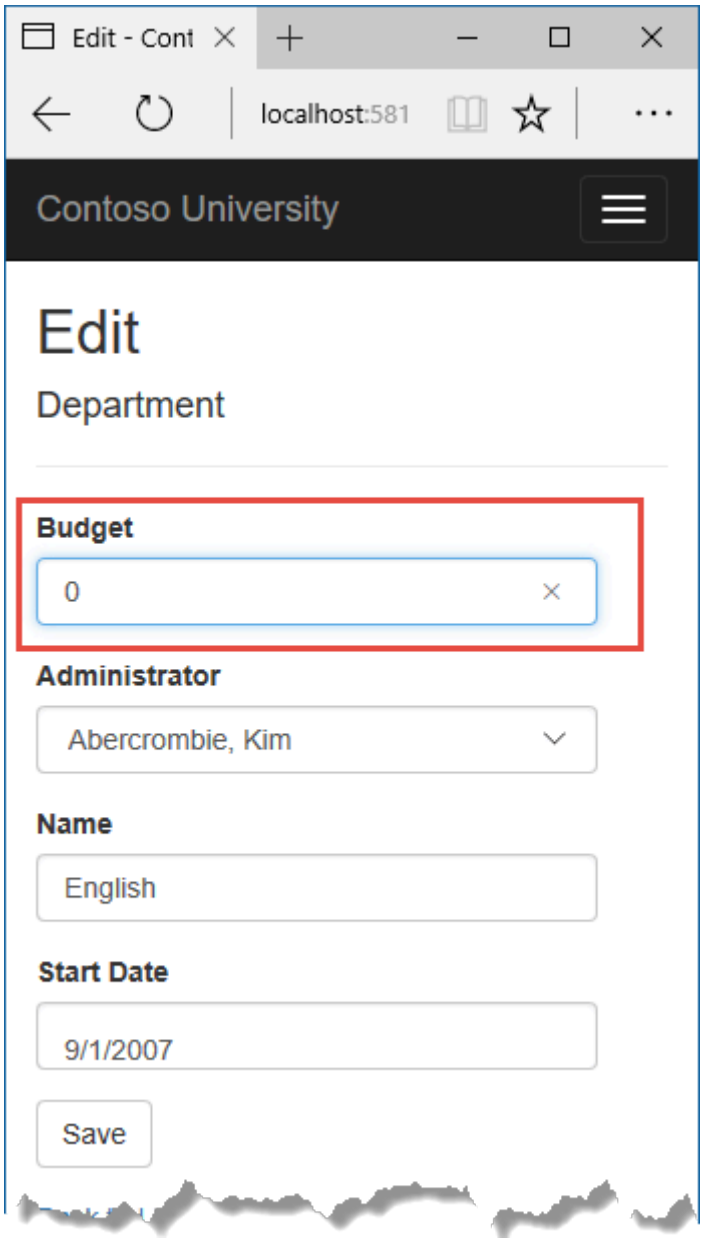
Pessimistic concurrency (locking)

If your application does need to prevent accidental data loss in concurrency scenarios, one way to do that is to use database locks. This is called pessimistic concurrency. For example, before you read a row from a database, you request a lock for read-only or for update access. If you lock a row for update access, no other users are allowed to lock the row either for read-only or update access, because they would get a copy of data that's in the process of being changed. If you lock a row for read-only access, others can also lock it for read-only access but not for update.

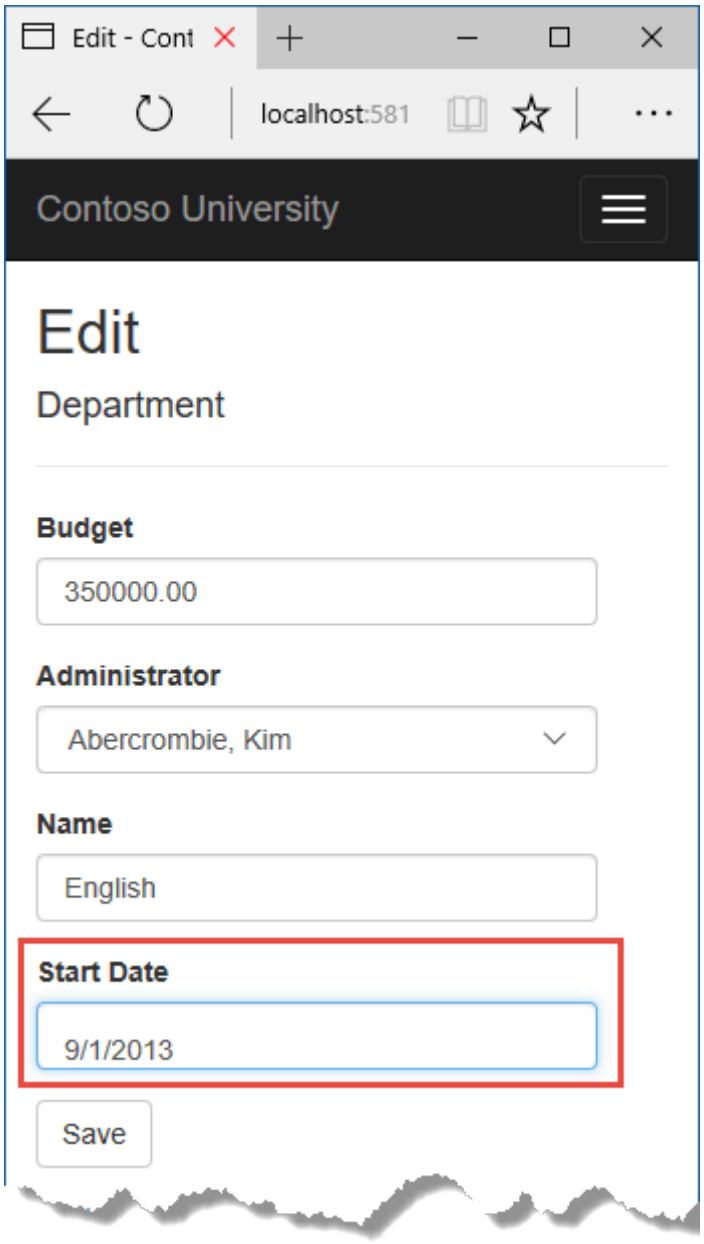
Managing locks has disadvantages. It can be complex to program. It requires significant database management resources, and it can cause performance problems as the number of users of an application increases. For these reasons, not all database management systems support pessimistic concurrency. Entity Framework Core provides no built-in support for it, and this tutorial doesn't show you how to implement it.

Optimistic Concurrency

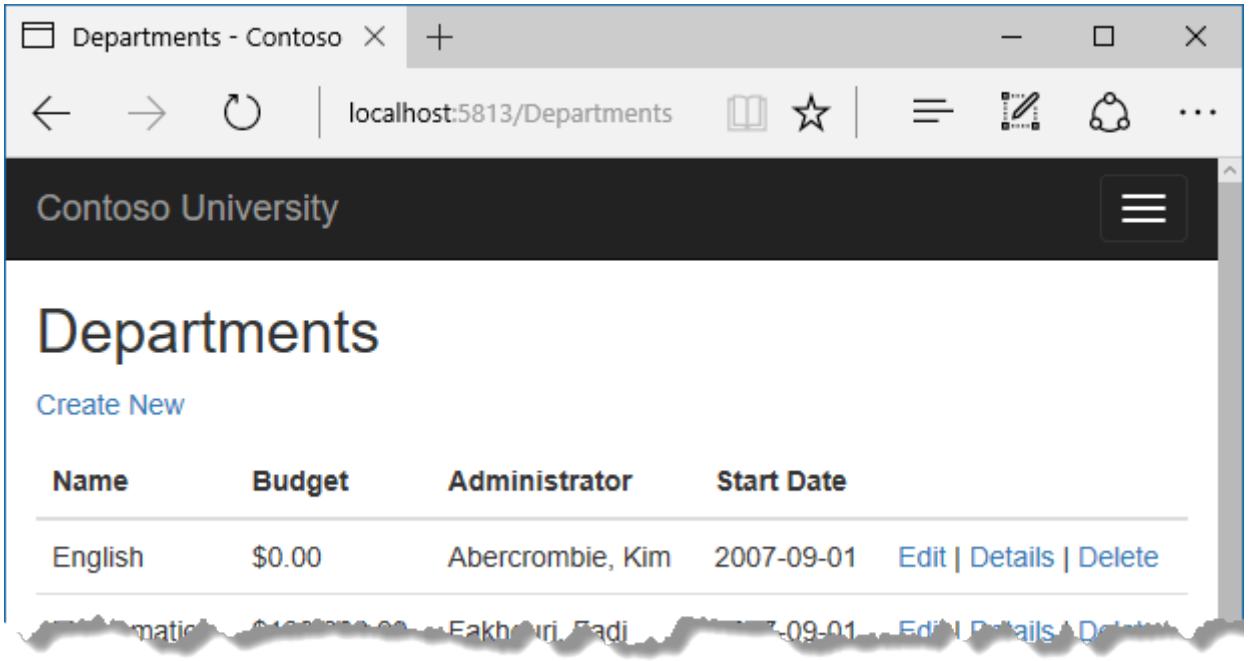
The alternative to pessimistic concurrency is optimistic concurrency. Optimistic concurrency means allowing concurrency conflicts to happen, and then reacting appropriately if they do. For example, Jane visits the Department Edit page and changes the Budget amount for the English department from $350,000.00 to $0.00.



Before Jane clicks **Save**, John visits the same page and changes the Start Date field from 9/1/2007 to 9/1/2013.



Jane clicks **Save** first and sees her change when the browser returns to the Index page.



Then John clicks **Save** on an Edit page that still shows a budget of $350,000.00. What happens next is determined by how you handle concurrency conflicts.

Some of the options include the following:

- You can keep track of which property a user has modified and update only the corresponding columns in the database.

  In the example scenario, no data would be lost, because different properties were updated by the two users. The next time someone browses the English department, they will see both Jane's and John's changes -- a start date of 9/1/2013 and a budget of zero dollars. This method of updating can reduce the number of conflicts that could result in data loss, but it can't avoid data loss if competing changes are made to the same property of an entity. Whether the Entity Framework works this way depends on how you implement your update code. It's often not practical in a web application, because it can require that you maintain large amounts of state in order to keep track of all original property values for an entity as well as new values. Maintaining large amounts of state can affect application performance because it either requires server resources or must be included in the web page itself (for example, in hidden fields) or in a cookie.

- You can let John's change overwrite Jane's change.

  The next time someone browses the English department, they will see 9/1/2013 and the restored $350,000.00 value. This is called a *Client Wins* or *Last in Wins* scenario. (All values from the client take precedence over what's in the data store.) As noted in the introduction to this section, if you don't do any coding for concurrency handling, this will happen automatically.

- You can prevent John's change from being updated in the database.

  Typically, you would display an error message, show him the current state of the data, and allow him to reapply his changes if he still wants to make them. This is called a *Store Wins* scenario. (The data-store values take precedence over the values submitted by the client.) You'll implement the Store Wins scenario in this tutorial. This method ensures that no changes are overwritten without a user being alerted to what's happening.

Detecting concurrency conflicts

You can resolve conflicts by handling `DbConcurrencyException` exceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. Therefore, you must configure the database

and the data model appropriately. Some options for enabling conflict detection include the following:

- In the database table, include a tracking column that can be used to determine when a row has been changed. You can then configure the Entity Framework to include that column in the Where clause of SQL Update or Delete commands.

  The data type of the tracking column is typically `rowversion`. The `rowversion` value is a sequential number that's incremented each time the row is updated. In an Update or Delete command, the Where clause includes the original value of the tracking column (the original row version) . If the row being updated has been changed by another user, the value in the `rowversion` column is different than the original value, so the Update or Delete statement can't find the row to update because of the Where clause. When the Entity Framework finds that no rows have been updated by the Update or Delete command (that is, when the number of affected rows is zero), it interprets that as a concurrency conflict.

- Configure the Entity Framework to include the original values of every column in the table in the Where clause of Update and Delete commands.

  As in the first option, if anything in the row has changed since the row was first read, the Where clause won't return a row to update, which the Entity Framework interprets as a concurrency conflict. For database tables that have many columns, this approach can result in very large Where clauses, and can require that you maintain large amounts of state. As noted earlier, maintaining large amounts of state can affect application performance. Therefore this approach is generally not recommended, and it isn't the method used in this tutorial.

  If you do want to implement this approach to concurrency, you have to mark all non-primary-key properties in the entity you want to track concurrency for by adding the `ConcurrencyCheck` attribute to them. That change enables the Entity Framework to include all columns in the SQL Where clause of Update and Delete statements.

In the remainder of this tutorial you'll add a `rowversion` tracking property to the Department entity, create a controller and views, and test to verify that everything works correctly.

Add a tracking property

In `Models/Department.cs`, add a tracking property named RowVersion:

```
```

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Department
    {
        public int DepartmentID { get; set; }

        [StringLength(50, MinimumLength = 3)]
        public string Name { get; set; }

        [DataType(DataType.Currency)]
        [Column(TypeName = "money")]
        public decimal Budget { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Start Date")]
        public DateTime StartDate { get; set; }

        public int? InstructorID { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public Instructor Administrator { get; set; }
        public ICollection<Course> Courses { get; set; }
    }
}
```

The `Timestamp` attribute specifies that this column will be included in the Where clause of Update and Delete commands sent to the database. The attribute is called `Timestamp` because previous versions of SQL Server used a SQL `timestamp` data type before the SQL `rowversion` replaced it. The .NET type for `rowversion` is a byte array.

If you prefer to use the fluent API, you can use the `IsRowVersion()` method (in `Data/SchoolContext.cs`) to specify the tracking property, as shown in the following example:

C#

```
modelBuilder.Entity<Department>()
    .Property(p => p.RowVersion).IsRowVersion();
```

By adding a property you changed the database model, so you need to do another migration.

Save your changes and build the project, and then enter the following commands in the command window:

.NET CLI

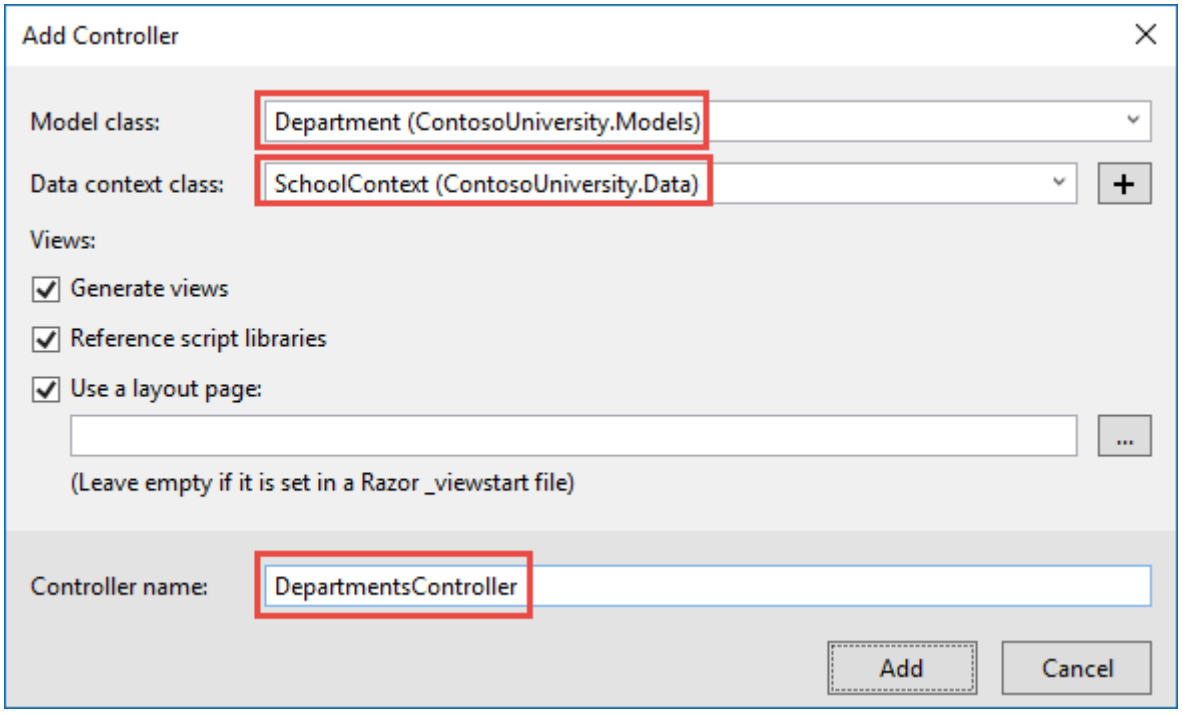
```
dotnet ef migrations add RowVersion
```

.NET CLI

```
dotnet ef database update
```

Create Departments controller and views

Scaffold a Departments controller and views as you did earlier for Students, Courses, and Instructors.



In the `DepartmentsController.cs` file, change all four occurrences of "FirstMidName" to "FullName" so that the department administrator drop-down lists will contain the full name of the instructor rather than just the last name.

C#

```
ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID",
"FullName", department.InstructorID);
```

Update Index view

The scaffolding engine created a `RowVersion` column in the Index view, but that field shouldn't be displayed.

Replace the code in `Views/Departments/Index.cshtml` with the following code.

CSHTML

```
@model IEnumerable<ContosoUniversity.Models.Department>

@{
    ViewData["Title"] = "Departments";
}

<h2>Departments</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Budget)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.StartDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Administrator)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Name)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Budget)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.StartDate)
                </td>
                <td>
```

```
                    @Html.DisplayFor(modelItem =>
item.Administrator.FullName)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-
id="@item.DepartmentID">Edit</a> |
                    <a asp-action="Details" asp-route-
id="@item.DepartmentID">Details</a> |
                    <a asp-action="Delete" asp-route-
id="@item.DepartmentID">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

This changes the heading to "Departments", deletes the `RowVersion` column, and shows full name instead of first name for the administrator.

Update Edit methods

In both the HttpGet `Edit` method and the `Details` method, add `AsNoTracking`. In the HttpGet `Edit` method, add eager loading for the Administrator.

C#

```
var department = await _context.Departments
    .Include(i => i.Administrator)
    .AsNoTracking()
    .FirstOrDefaultAsync(m => m.DepartmentID == id);
```

Replace the existing code for the HttpPost `Edit` method with the following code:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int? id, byte[] rowVersion)
{
    if (id == null)
    {
        return NotFound();
    }

    var departmentToUpdate = await _context.Departments.Include(i =>
i.Administrator).FirstOrDefaultAsync(m => m.DepartmentID == id);

    if (departmentToUpdate == null)
    {
```

```
        Department deletedDepartment = new Department();
        await TryUpdateModelAsync(deletedDepartment);
        ModelState.AddModelError(string.Empty,
            "Unable to save changes. The department was deleted by another
user.");
        ViewData["InstructorID"] = new SelectList(_context.Instructors,
"ID", "FullName", deletedDepartment.InstructorID);
        return View(deletedDepartment);
    }

    _context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue
= rowVersion;

    if (await TryUpdateModelAsync<Department>(
        departmentToUpdate,
        "",
        s => s.Name, s => s.StartDate, s => s.Budget, s => s.InstructorID))
    {
        try
        {
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        catch (DbUpdateConcurrencyException ex)
        {
            var exceptionEntry = ex.Entries.Single();
            var clientValues = (Department)exceptionEntry.Entity;
            var databaseEntry = exceptionEntry.GetDatabaseValues();
            if (databaseEntry == null)
            {
                ModelState.AddModelError(string.Empty,
                    "Unable to save changes. The department was deleted by
another user.");
            }
            else
            {
                var databaseValues = (Department)databaseEntry.ToObject();

                if (databaseValues.Name != clientValues.Name)
                {
                    ModelState.AddModelError("Name", $"Current value:
{databaseValues.Name}");
                }
                if (databaseValues.Budget != clientValues.Budget)
                {
                    ModelState.AddModelError("Budget", $"Current value:
{databaseValues.Budget:c}");
                }
                if (databaseValues.StartDate != clientValues.StartDate)
                {
                    ModelState.AddModelError("StartDate", $"Current value:
{databaseValues.StartDate:d}");
                }
                if (databaseValues.InstructorID !=
clientValues.InstructorID)
```

```
                {
                    Instructor databaseInstructor = await
_context.Instructors.FirstOrDefaultAsync(i => i.ID ==
databaseValues.InstructorID);
                    ModelState.AddModelError("InstructorID", $"Current
value: {databaseInstructor?.FullName}");
                }

                ModelState.AddModelError(string.Empty, "The record you
attempted to edit "
                        + "was modified by another user after you got the
original value. The "
                        + "edit operation was canceled and the current
values in the database "
                        + "have been displayed. If you still want to edit
this record, click "
                        + "the Save button again. Otherwise click the Back
to List hyperlink.");
                departmentToUpdate.RowVersion =
(byte[])databaseValues.RowVersion;
                ModelState.Remove("RowVersion");
            }
        }
    }
    ViewData["InstructorID"] = new SelectList(_context.Instructors, "ID",
"FullName", departmentToUpdate.InstructorID);
    return View(departmentToUpdate);
}
```

The code begins by trying to read the department to be updated. If the `FirstOrDefaultAsync` method returns null, the department was deleted by another user. In that case the code uses the posted form values to create a `Department` entity so that the Edit page can be redisplayed with an error message. As an alternative, you wouldn't have to re-create the `Department` entity if you display only an error message without redisplaying the department fields.

The view stores the original `RowVersion` value in a hidden field, and this method receives that value in the `rowVersion` parameter. Before you call `SaveChanges`, you have to put that original `RowVersion` property value in the `OriginalValues` collection for the entity.

C#

```
_context.Entry(departmentToUpdate).Property("RowVersion").OriginalValue =
rowVersion;
```

Then when the Entity Framework creates a SQL UPDATE command, that command will include a WHERE clause that looks for a row that has the original `RowVersion` value. If no

rows are affected by the UPDATE command (no rows have the original `RowVersion` value), the Entity Framework throws a `DbUpdateConcurrencyException` exception.

The code in the catch block for that exception gets the affected Department entity that has the updated values from the `Entries` property on the exception object.

C#

```
var exceptionEntry = ex.Entries.Single();
```

The `Entries` collection will have just one `EntityEntry` object. You can use that object to get the new values entered by the user and the current database values.

C#

```
var clientValues = (Department)exceptionEntry.Entity;
var databaseEntry = exceptionEntry.GetDatabaseValues();
```

The code adds a custom error message for each column that has database values different from what the user entered on the Edit page (only one field is shown here for brevity).

C#

```
var databaseValues = (Department)databaseEntry.ToObject();

if (databaseValues.Name != clientValues.Name)
{
    ModelState.AddModelError("Name", $"Current value:
{databaseValues.Name}");
```

Finally, the code sets the `RowVersion` value of the `departmentToUpdate` to the new value retrieved from the database. This new `RowVersion` value will be stored in the hidden field when the Edit page is redisplayed, and the next time the user clicks **Save**, only concurrency errors that happen since the redisplay of the Edit page will be caught.

C#

```
departmentToUpdate.RowVersion = (byte[])databaseValues.RowVersion;
ModelState.Remove("RowVersion");
```

The `ModelState.Remove` statement is required because `ModelState` has the old `RowVersion` value. In the view, the `ModelState` value for a field takes precedence over the model property values when both are present.

Update Edit view

In `Views/Departments/Edit.cshtml`, make the following changes:

- Add a hidden field to save the `RowVersion` property value, immediately following the hidden field for the `DepartmentID` property.
- Add a "Select Administrator" option to the drop-down list.

CSHTML

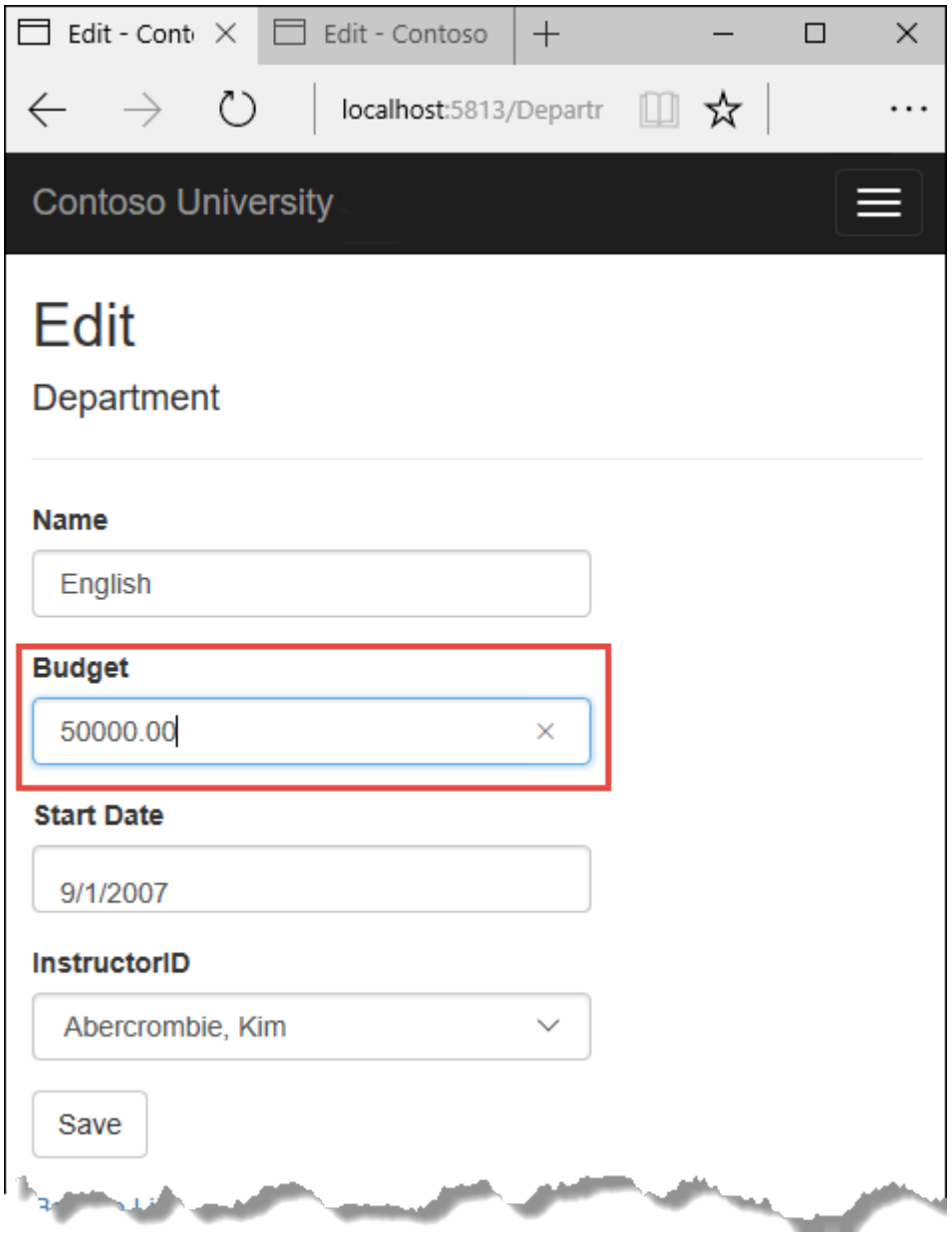
```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Department</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <input type="hidden" asp-for="DepartmentID" />
            <input type="hidden" asp-for="RowVersion" />
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
                <span asp-validation-for="InstructorID" class="text-danger">
```

```
</span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Test concurrency conflicts

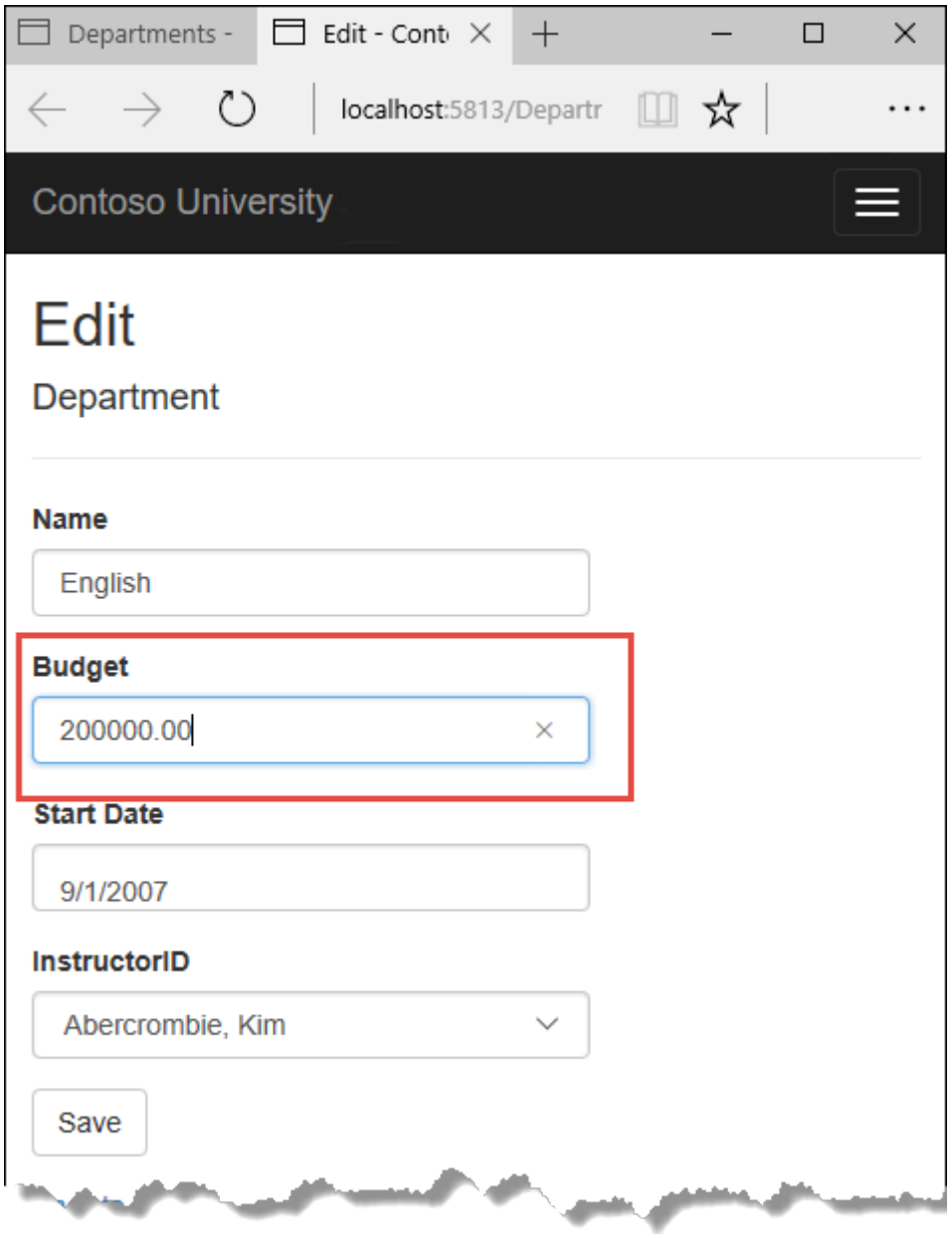
Run the app and go to the Departments Index page. Right-click the **Edit** hyperlink for the English department and select **Open in new tab**, then click the **Edit** hyperlink for the English department. The two browser tabs now display the same information.

Change a field in the first browser tab and click **Save**.

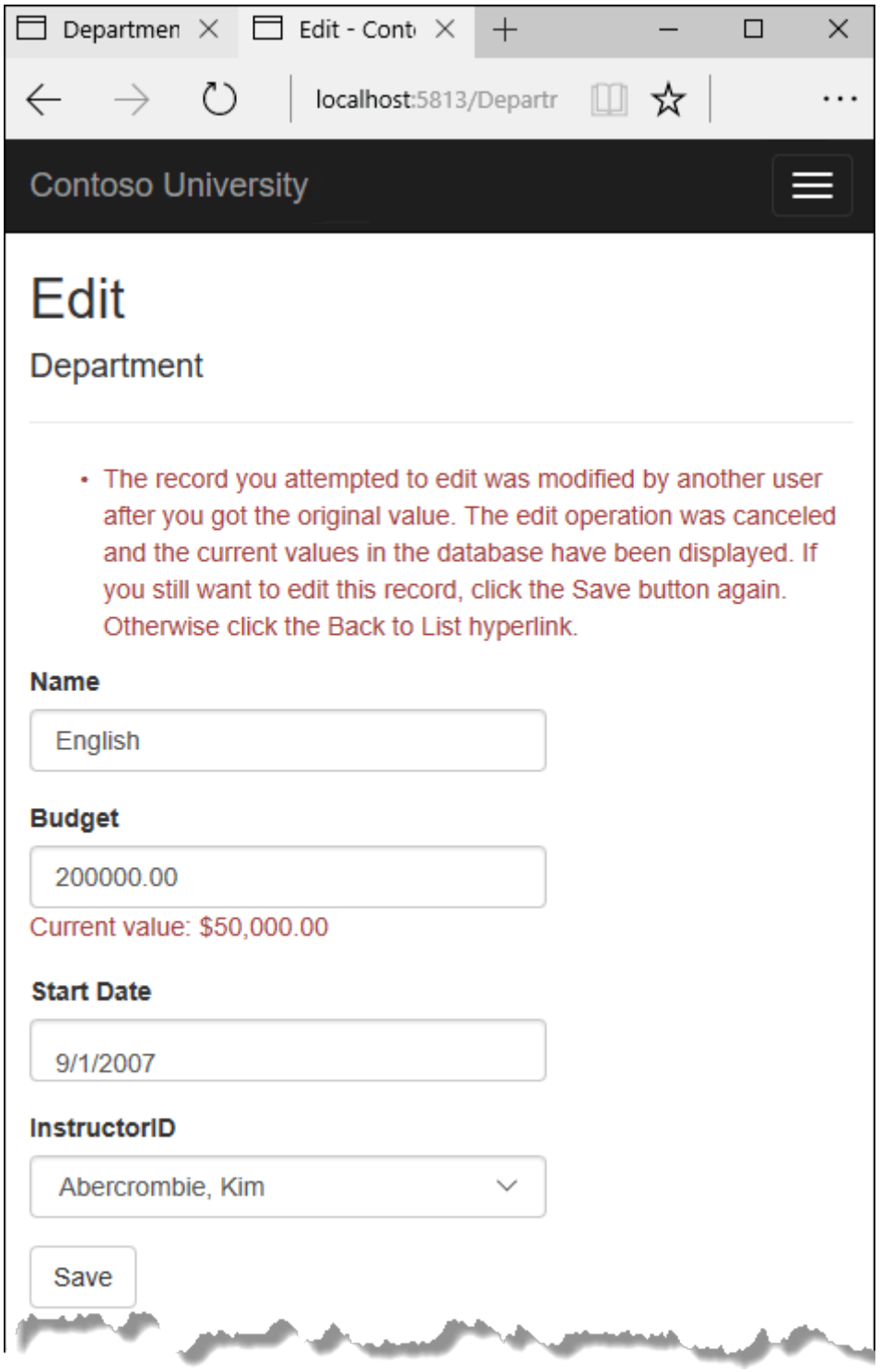


The browser shows the Index page with the changed value.

Change a field in the second browser tab.



Click **Save**. You see an error message:



Click **Save** again. The value you entered in the second browser tab is saved. You see the saved values when the Index page appears.

Update the Delete page

For the Delete page, the Entity Framework detects concurrency conflicts caused by someone else editing the department in a similar manner. When the HttpGet `Delete` method displays the confirmation view, the view includes the original `RowVersion` value in a hidden field. That value is then available to the HttpPost `Delete` method that's called when the user confirms the deletion. When the Entity Framework creates the SQL

DELETE command, it includes a WHERE clause with the original `RowVersion` value. If the command results in zero rows affected (meaning the row was changed after the Delete confirmation page was displayed), a concurrency exception is thrown, and the HttpGet `Delete` method is called with an error flag set to true in order to redisplay the confirmation page with an error message. It's also possible that zero rows were affected because the row was deleted by another user, so in that case no error message is displayed.

Update the Delete methods in the Departments controller

In `DepartmentsController.cs`, replace the HttpGet `Delete` method with the following code:

C#

```
public async Task<IActionResult> Delete(int? id, bool? concurrencyError)
{
    if (id == null)
    {
        return NotFound();
    }

    var department = await _context.Departments
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync(m => m.DepartmentID == id);
    if (department == null)
    {
        if (concurrencyError.GetValueOrDefault())
        {
            return RedirectToAction(nameof(Index));
        }
        return NotFound();
    }

    if (concurrencyError.GetValueOrDefault())
    {
        ViewData["ConcurrencyErrorMessage"] = "The record you attempted to
delete "
            + "was modified by another user after you got the original
values. "
            + "The delete operation was canceled and the current values in
the "
            + "database have been displayed. If you still want to delete
this "
            + "record, click the Delete button again. Otherwise "
            + "click the Back to List hyperlink.";
    }
```

```
    return View(department);
}
```

The method accepts an optional parameter that indicates whether the page is being redisplayed after a concurrency error. If this flag is true and the department specified no longer exists, it was deleted by another user. In that case, the code redirects to the Index page. If this flag is true and the department does exist, it was changed by another user. In that case, the code sends an error message to the view using `ViewData`.

Replace the code in the HttpPost `Delete` method (named `DeleteConfirmed`) with the following code:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(Department department)
{
    try
    {
        if (await _context.Departments.AnyAsync(m => m.DepartmentID ==
department.DepartmentID))
        {
            _context.Departments.Remove(department);
            await _context.SaveChangesAsync();
        }
        return RedirectToAction(nameof(Index));
    }
    catch (DbUpdateConcurrencyException /* ex */)
    {
        //Log the error (uncomment ex variable name and write a log.)
        return RedirectToAction(nameof(Delete), new { concurrencyError =
true, id = department.DepartmentID });
    }
}
```

In the scaffolded code that you just replaced, this method accepted only a record ID:

C#

```
public async Task<IActionResult> DeleteConfirmed(int id)
```

You've changed this parameter to a `Department` entity instance created by the model binder. This gives EF access to the RowVers`ion property value in addition to the record key.

C#

```
public async Task<IActionResult> Delete(Department department)
```

You have also changed the action method name from `DeleteConfirmed` to `Delete`. The scaffolded code used the name `DeleteConfirmed` to give the HttpPost method a unique signature. (The CLR requires overloaded methods to have different method parameters.) Now that the signatures are unique, you can stick with the MVC convention and use the same name for the HttpPost and HttpGet delete methods.

If the department is already deleted, the `AnyAsync` method returns false and the application just goes back to the Index method.

If a concurrency error is caught, the code redisplays the Delete confirmation page and provides a flag that indicates it should display a concurrency error message.

Update the Delete view

In `Views/Departments/Delete.cshtml`, replace the scaffolded code with the following code that adds an error message field and hidden fields for the DepartmentID and RowVersion properties. The changes are highlighted.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Delete";
}

<h2>Delete</h2>

<p class="text-danger">@ViewData["ConcurrencyErrorMessage"]</p>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
```

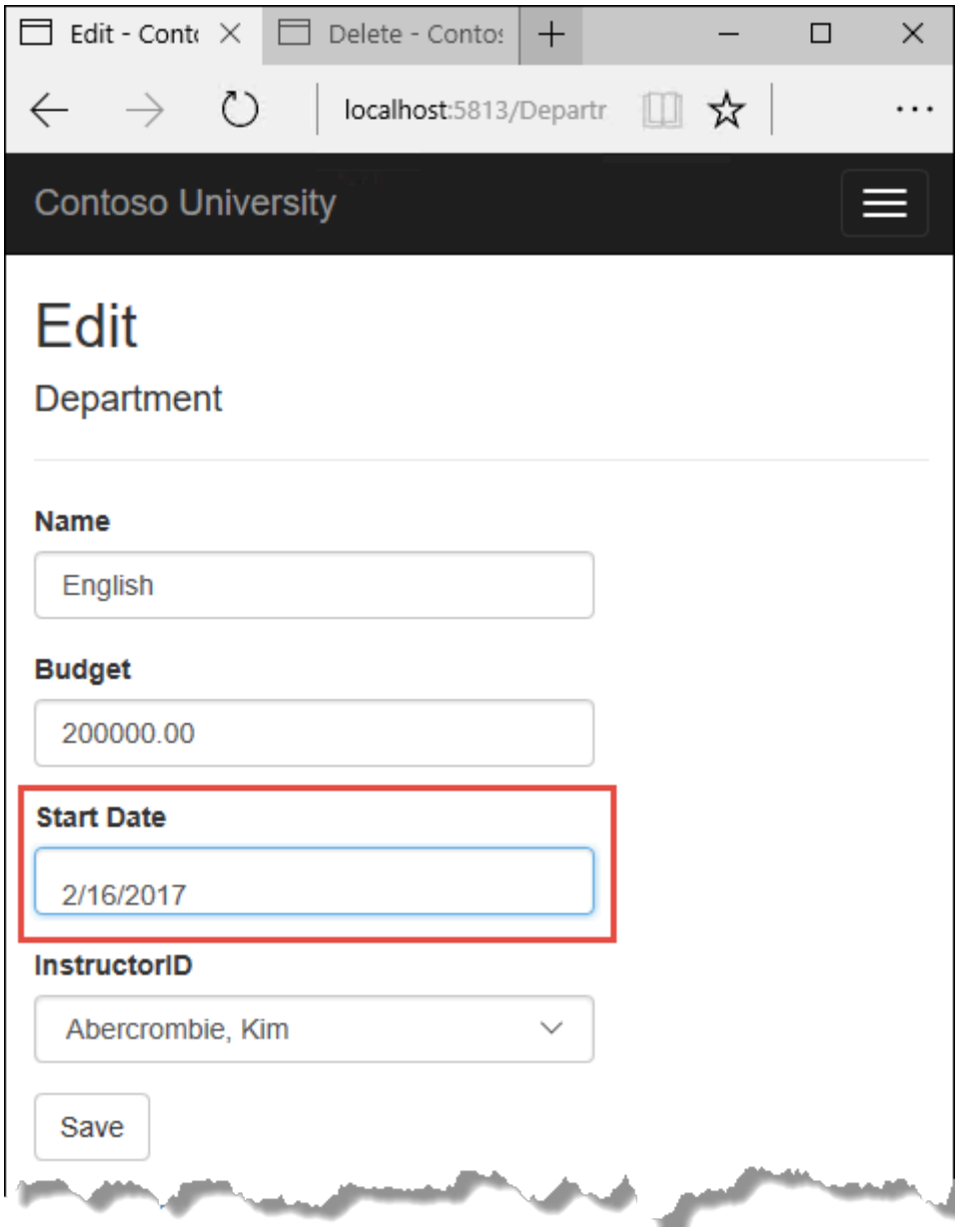
```
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="DepartmentID" />
        <input type="hidden" asp-for="RowVersion" />
        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            <a asp-action="Index">Back to List</a>
        </div>
    </form>
</div>
```

This makes the following changes:

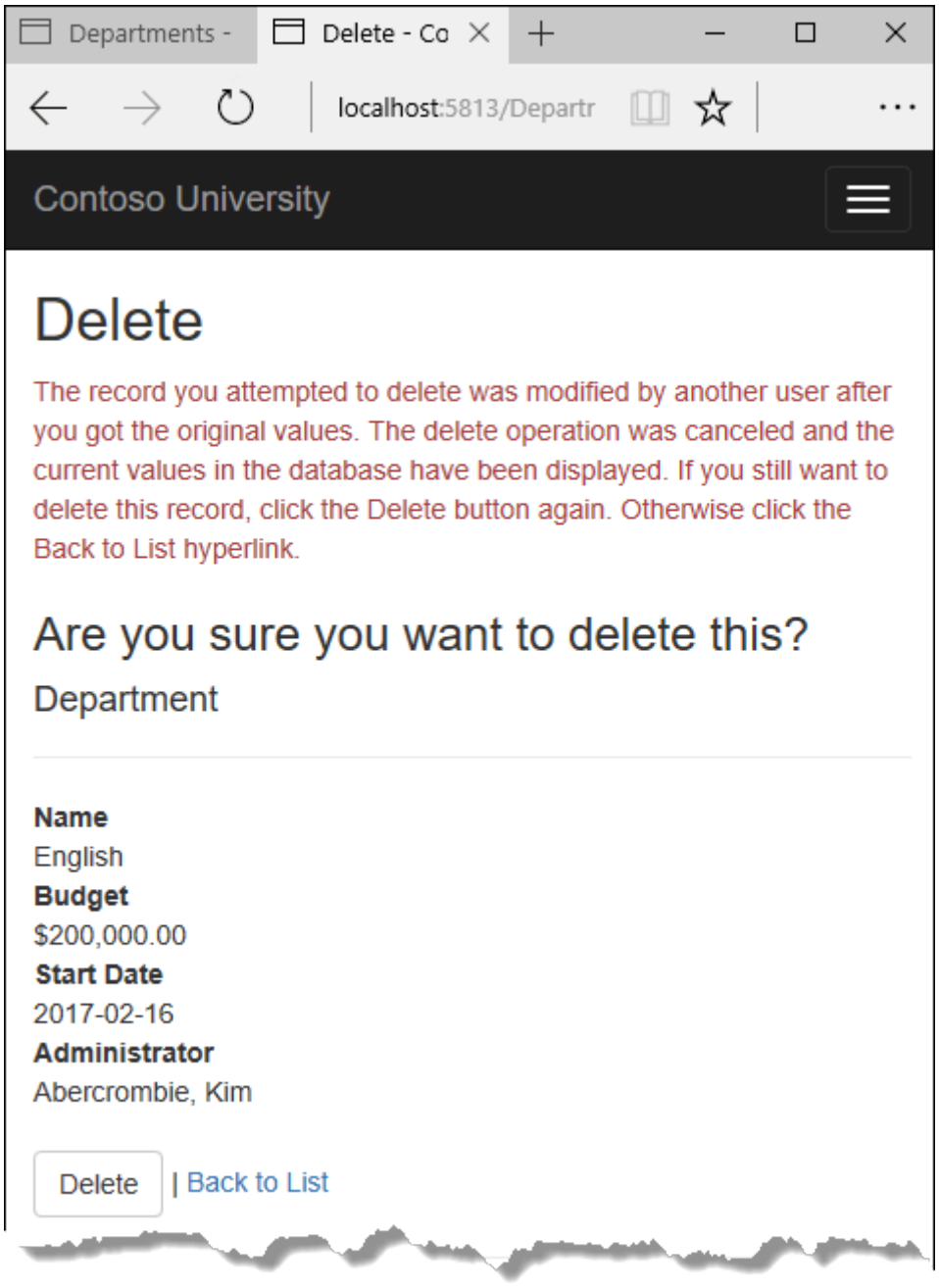
- Adds an error message between the `h2` and `h3` headings.
- Replaces FirstMidName with FullName in the **Administrator** field.
- Removes the RowVersion field.
- Adds a hidden field for the `RowVersion` property.

Run the app and go to the Departments Index page. Right-click the **Delete** hyperlink for the English department and select **Open in new tab**, then in the first tab click the **Edit** hyperlink for the English department.

In the first window, change one of the values, and click **Save**:



In the second tab, click **Delete**. You see the concurrency error message, and the Department values are refreshed with what's currently in the database.



If you click **Delete** again, you're redirected to the Index page, which shows that the department has been deleted.

Update Details and Create views

You can optionally clean up scaffolded code in the Details and Create views.

Replace the code in `Views/Departments/Details.cshtml` to delete the RowVersion column and show the full name of the Administrator.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Details";
```

```
}

<h2>Details</h2>

<div>
    <h4>Department</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Budget)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Budget)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.StartDate)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.StartDate)
        </dd>
        <dt class="col-sm-2">
            @Html.DisplayNameFor(model => model.Administrator)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Administrator.FullName)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.DepartmentID">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

Replace the code in `Views/Departments/Create.cshtml` to add a Select option to the drop-down list.

CSHTML

```
@model ContosoUniversity.Models.Department

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Department</h4>
```

```
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Budget" class="control-label"></label>
                <input asp-for="Budget" class="form-control" />
                <span asp-validation-for="Budget" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="StartDate" class="control-label"></label>
                <input asp-for="StartDate" class="form-control" />
                <span asp-validation-for="StartDate" class="text-danger">
</span>
            </div>
            <div class="form-group">
                <label asp-for="InstructorID" class="control-label"></label>
                <select asp-for="InstructorID" class="form-control" asp-
items="ViewBag.InstructorID">
                    <option value="">-- Select Administrator --</option>
                </select>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default"
/>
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

Get the code

Download or view the completed application.

Additional resources

For more information about how to handle concurrency in EF Core, see Concurrency conflicts.

Next steps

In this tutorial, you:

- ✓ Learned about concurrency conflicts
- ✓ Added a tracking property
- ✓ Created Departments controller and views
- ✓ Updated Index view
- ✓ Updated Edit methods
- ✓ Updated Edit view
- ✓ Tested concurrency conflicts
- ✓ Updated the Delete page
- ✓ Updated Details and Create views

Advance to the next tutorial to learn how to implement table-per-hierarchy inheritance for the Instructor and Student entities.

Next: Implement table-per-hierarchy inheritance

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Implement inheritance - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you handled concurrency exceptions. This tutorial will show you how to implement inheritance in the data model.

In object-oriented programming, you can use inheritance to facilitate code reuse. In this tutorial, you'll change the `Instructor` and `Student` classes so that they derive from a `Person` base class which contains properties such as `LastName` that are common to both instructors and students. You won't add or change any web pages, but you'll change some of the code and those changes will be automatically reflected in the database.

In this tutorial, you:

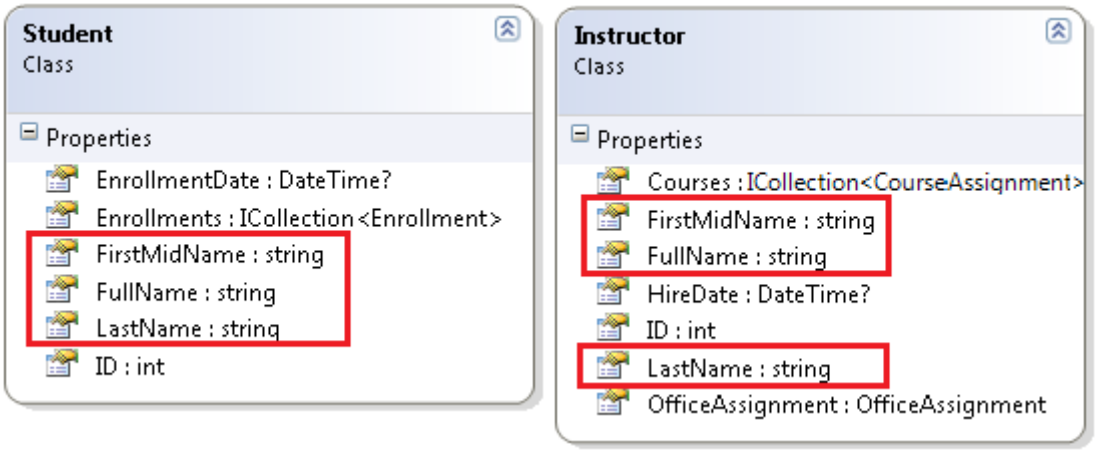
- ✓ Map inheritance to database
- ✓ Create the Person class
- ✓ Update Instructor and Student
- ✓ Add Person to the model
- ✓ Create and update migrations
- ✓ Test the implementation

Prerequisites

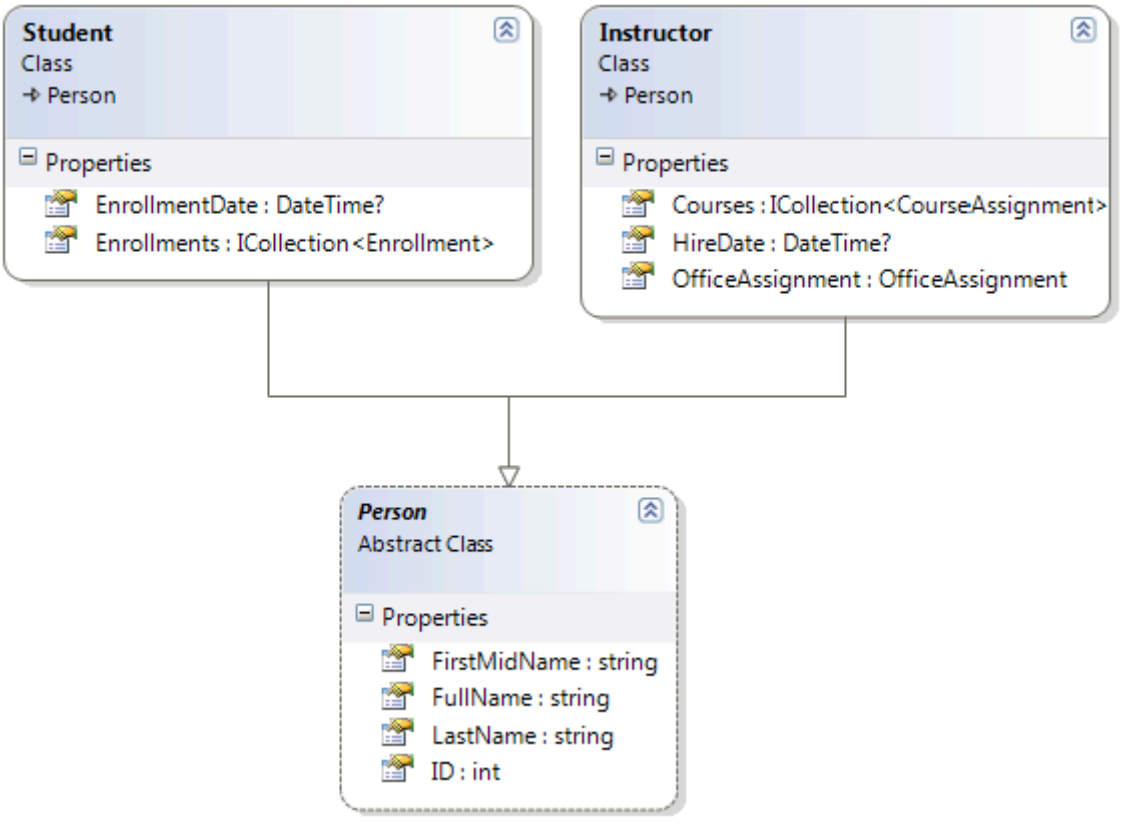
- Handle Concurrency

Map inheritance to database

The `Instructor` and `Student` classes in the School data model have several properties that are identical:

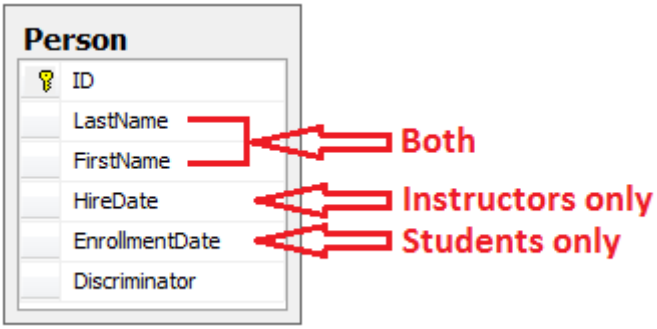


Suppose you want to eliminate the redundant code for the properties that are shared by the `Instructor` and `Student` entities. Or you want to write a service that can format names without caring whether the name came from an instructor or a student. You could create a `Person` base class that contains only those shared properties, then make the `Instructor` and `Student` classes inherit from that base class, as shown in the following illustration:



There are several ways this inheritance structure could be represented in the database. You could have a `Person` table that includes information about both students and instructors in a single table. Some of the columns could apply only to instructors (HireDate), some only to students (EnrollmentDate), some to both (LastName, FirstName). Typically, you'd have a discriminator column to indicate which type each row

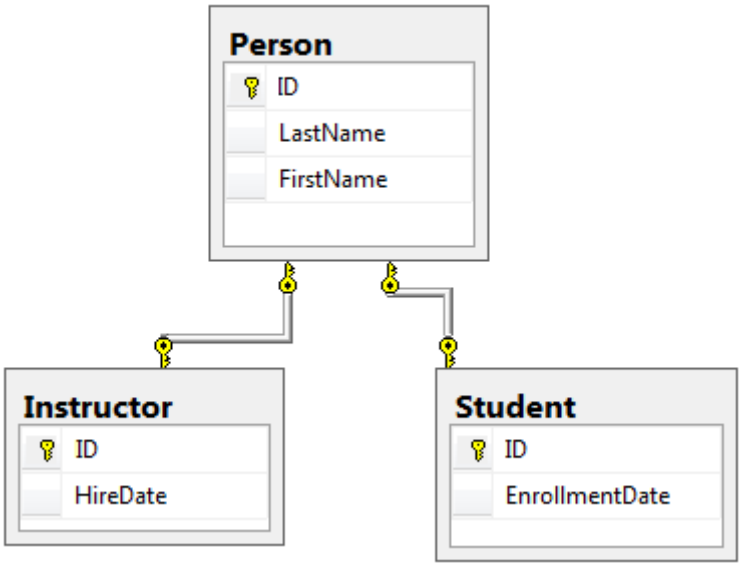
represents. For example, the discriminator column might have "Instructor" for instructors and "Student" for students.



This pattern of generating an entity inheritance structure from a single database table is called *table-per-hierarchy (TPH)* inheritance.

An alternative is to make the database look more like the inheritance structure. For example, you could have only the name fields in the `Person` table and have separate `Instructor` and `Student` tables with the date fields.

> ⚠ **Warning**
>
> Table-Per-Type (TPT) is not supported by EF Core 3.x, however it is has been implemented in EF Core 5.0.



This pattern of making a database table for each entity class is called *table-per-type (TPT)* inheritance.

Yet another option is to map all non-abstract types to individual tables. All properties of a class, including inherited properties, map to columns of the corresponding table. This pattern is called *Table-per-Concrete Class (TPC)* inheritance. If you implemented TPC

inheritance for the `Person`, `Student`, and `Instructor` classes as shown earlier, the `Student` and `Instructor` tables would look no different after implementing inheritance than they did before.

TPC and TPH inheritance patterns generally deliver better performance than TPT inheritance patterns, because TPT patterns can result in complex join queries.

This tutorial demonstrates how to implement TPH inheritance. TPH is the only inheritance pattern that the Entity Framework Core supports. What you'll do is create a `Person` class, change the `Instructor` and `Student` classes to derive from `Person`, add the new class to the `DbContext`, and create a migration.

> **Tip**
>
> Consider saving a copy of the project before making the following changes. Then if you run into problems and need to start over, it will be easier to start from the saved project instead of reversing steps done for this tutorial or going back to the beginning of the whole series.

Create the Person class

In the Models folder, create Person.cs and replace the template code with the following code:

C#

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public abstract class Person
    {
        public int ID { get; set; }

        [Required]
        [StringLength(50)]
        [Display(Name = "Last Name")]
        public string LastName { get; set; }
        [Required]
        [StringLength(50, ErrorMessage = "First name cannot be longer than
50 characters.")]
        [Column("FirstName")]
        [Display(Name = "First Name")]
        public string FirstMidName { get; set; }
```

```
        [Display(Name = "Full Name")]
        public string FullName
        {
            get
            {
                return LastName + ", " + FirstMidName;
            }
        }
    }
}
```

Update Instructor and Student

In `Instructor.cs`, derive the Instructor class from the Person class and remove the key and name fields. The code will look like the following example:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
    public class Instructor : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Hire Date")]
        public DateTime HireDate { get; set; }

        public ICollection<CourseAssignment> CourseAssignments { get; set; }
        public OfficeAssignment OfficeAssignment { get; set; }
    }
}
```

Make the same changes in `Student.cs`.

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ContosoUniversity.Models
{
```

```
    public class Student : Person
    {
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}",
ApplyFormatInEditMode = true)]
        [Display(Name = "Enrollment Date")]
        public DateTime EnrollmentDate { get; set; }


        public ICollection<Enrollment> Enrollments { get; set; }
    }
}
```

Add Person to the model

Add the Person entity type to `SchoolContext.cs`. The new lines are highlighted.

C#

```
using ContosoUniversity.Models;
using Microsoft.EntityFrameworkCore;

namespace ContosoUniversity.Data
{
    public class SchoolContext : DbContext
    {
        public SchoolContext(DbContextOptions<SchoolContext> options) :
base(options)
        {
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Enrollment> Enrollments { get; set; }
        public DbSet<Student> Students { get; set; }
        public DbSet<Department> Departments { get; set; }
        public DbSet<Instructor> Instructors { get; set; }
        public DbSet<OfficeAssignment> OfficeAssignments { get; set; }
        public DbSet<CourseAssignment> CourseAssignments { get; set; }
        public DbSet<Person> People { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Course>().ToTable("Course");
            modelBuilder.Entity<Enrollment>().ToTable("Enrollment");
            modelBuilder.Entity<Student>().ToTable("Student");
            modelBuilder.Entity<Department>().ToTable("Department");
            modelBuilder.Entity<Instructor>().ToTable("Instructor");
            modelBuilder.Entity<OfficeAssignment>
().ToTable("OfficeAssignment");
            modelBuilder.Entity<CourseAssignment>
().ToTable("CourseAssignment");
```

```
            modelBuilder.Entity<Person>().ToTable("Person");

            modelBuilder.Entity<CourseAssignment>()
                .HasKey(c => new { c.CourseID, c.InstructorID });
        }
    }
}
```

This is all that the Entity Framework needs in order to configure table-per-hierarchy inheritance. As you'll see, when the database is updated, it will have a Person table in place of the Student and Instructor tables.

Create and update migrations

Save your changes and build the project. Then open the command window in the project folder and enter the following command:

.NET CLI

```
dotnet ef migrations add Inheritance
```

Don't run the `database update` command yet. That command will result in lost data because it will drop the Instructor table and rename the Student table to Person. You need to provide custom code to preserve existing data.

Open `Migrations/<timestamp>_Inheritance.cs` and replace the `Up` method with the following code:

C#

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropForeignKey(
        name: "FK_Enrollment_Student_StudentID",
        table: "Enrollment");

    migrationBuilder.DropIndex(name: "IX_Enrollment_StudentID", table:
"Enrollment");

    migrationBuilder.RenameTable(name: "Instructor", newName: "Person");
    migrationBuilder.AddColumn<DateTime>(name: "EnrollmentDate", table:
"Person", nullable: true);
    migrationBuilder.AddColumn<string>(name: "Discriminator", table:
"Person", nullable: false, maxLength: 128, defaultValue: "Instructor");
    migrationBuilder.AlterColumn<DateTime>(name: "HireDate", table:
"Person", nullable: true);
    migrationBuilder.AddColumn<int>(name: "OldId", table: "Person",
```

```
nullable: true);

    // Copy existing Student data into new Person table.
    migrationBuilder.Sql("INSERT INTO dbo.Person (LastName, FirstName,
HireDate, EnrollmentDate, Discriminator, OldId) SELECT LastName, FirstName,
null AS HireDate, EnrollmentDate, 'Student' AS Discriminator, ID AS OldId
FROM dbo.Student");
    // Fix up existing relationships to match new PK's.
    migrationBuilder.Sql("UPDATE dbo.Enrollment SET StudentId = (SELECT ID
FROM dbo.Person WHERE OldId = Enrollment.StudentId AND Discriminator =
'Student')");

    // Remove temporary key
    migrationBuilder.DropColumn(name: "OldID", table: "Person");

    migrationBuilder.DropTable(
        name: "Student");

    migrationBuilder.CreateIndex(
         name: "IX_Enrollment_StudentID",
         table: "Enrollment",
         column: "StudentID");

    migrationBuilder.AddForeignKey(
        name: "FK_Enrollment_Person_StudentID",
        table: "Enrollment",
        column: "StudentID",
        principalTable: "Person",
        principalColumn: "ID",
        onDelete: ReferentialAction.Cascade);
}
```

This code takes care of the following database update tasks:

- Removes foreign key constraints and indexes that point to the Student table.
- Renames the Instructor table as Person and makes changes needed for it to store Student data:
- Adds nullable EnrollmentDate for students.
- Adds Discriminator column to indicate whether a row is for a student or an instructor.
- Makes HireDate nullable since student rows won't have hire dates.
- Adds a temporary field that will be used to update foreign keys that point to students. When you copy students into the Person table they will get new primary key values.

- Copies data from the Student table into the Person table. This causes students to get assigned new primary key values.
- Fixes foreign key values that point to students.
- Re-creates foreign key constraints and indexes, now pointing them to the Person table.

(If you had used GUID instead of integer as the primary key type, the student primary key values wouldn't have to change, and several of these steps could have been omitted.)

Run the `database update` command:

.NET CLI

```
dotnet ef database update
```

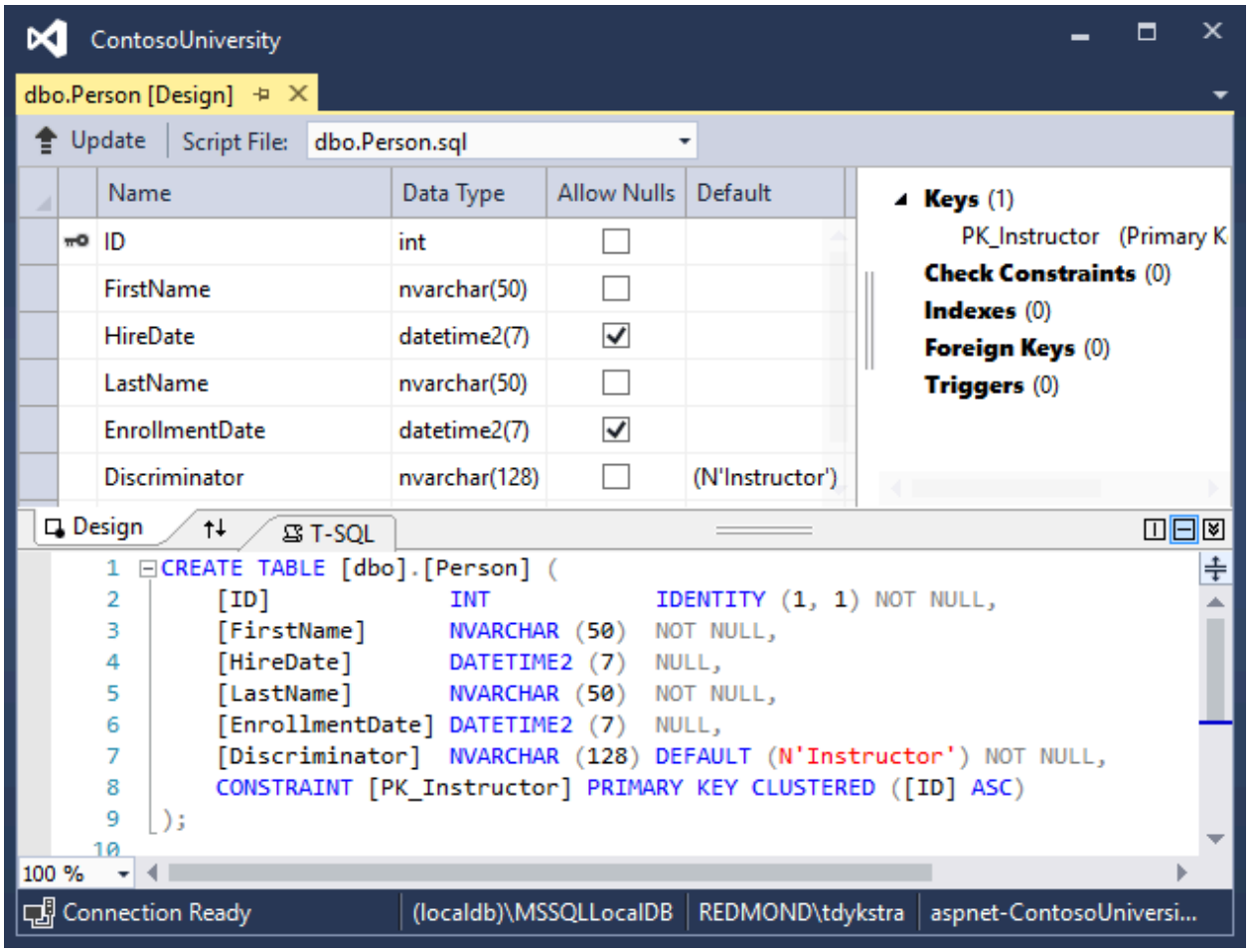
(In a production system you would make corresponding changes to the `Down` method in case you ever had to use that to go back to the previous database version. For this tutorial you won't be using the `Down` method.)

> **Note**
>
> It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors that you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is more likely to complete without errors. To delete the database, use SSOX or run the `database drop` CLI command.

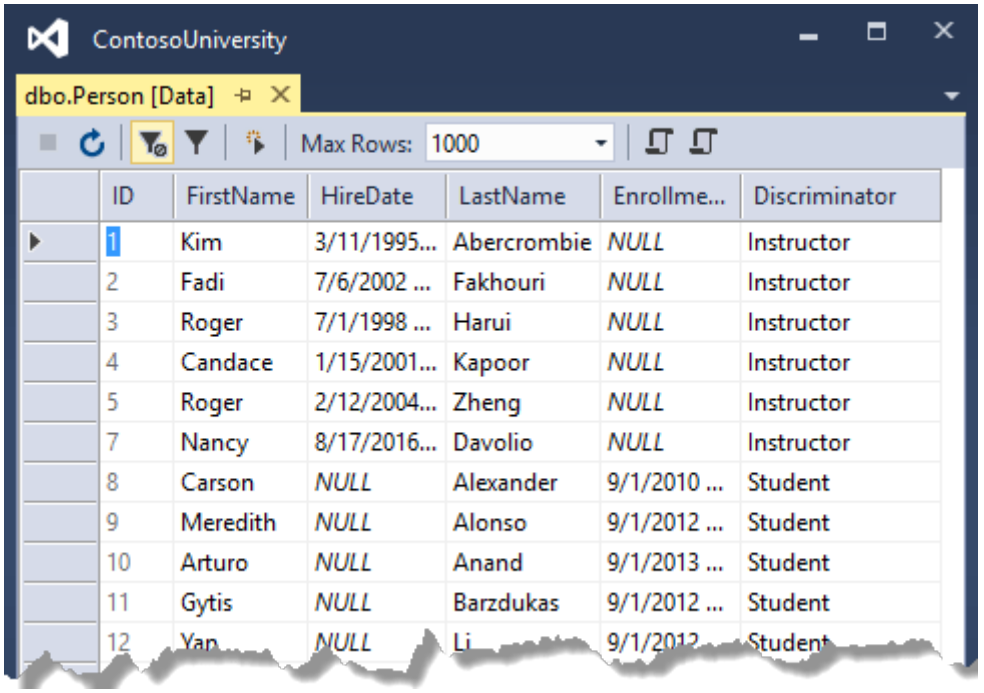
Test the implementation

Run the app and try various pages. Everything works the same as it did before.

In **SQL Server Object Explorer**, expand **Data Connections/SchoolContext** and then **Tables**, and you see that the Student and Instructor tables have been replaced by a Person table. Open the Person table designer and you see that it has all of the columns that used to be in the Student and Instructor tables.



Right-click the Person table, and then click **Show Table Data** to see the discriminator column.



Get the code

Download or view the completed application.

Additional resources

For more information about inheritance in Entity Framework Core, see Inheritance.

Next steps

In this tutorial, you:

- ✓ Mapped inheritance to database
- ✓ Created the Person class
- ✓ Updated Instructor and Student
- ✓ Added Person to the model
- ✓ Created and update migrations
- ✓ Tested the implementation

Advance to the next tutorial to learn how to handle a variety of relatively advanced Entity Framework scenarios.

Next: Advanced topics

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Learn about advanced scenarios - ASP.NET MVC with EF Core

Article • 04/11/2023

In the previous tutorial, you implemented table-per-hierarchy inheritance. This tutorial introduces several topics that are useful to be aware of when you go beyond the basics of developing ASP.NET Core web applications that use Entity Framework Core.

In this tutorial, you:

- ✓ Perform raw SQL queries
- ✓ Call a query to return entities
- ✓ Call a query to return other types
- ✓ Call an update query
- ✓ Examine SQL queries
- ✓ Create an abstraction layer
- ✓ Learn about Automatic change detection
- ✓ Learn about EF Core source code and development plans
- ✓ Learn how to use dynamic LINQ to simplify code

Prerequisites

- Implement Inheritance

Perform raw SQL queries

One of the advantages of using the Entity Framework is that it avoids tying your code too closely to a particular method of storing data. It does this by generating SQL queries and commands for you, which also frees you from having to write them yourself. But there are exceptional scenarios when you need to run specific SQL queries that you have manually created. For these scenarios, the Entity Framework Code First API includes methods that enable you to pass SQL commands directly to the database. You have the following options in EF Core 1.0:

- Use the `DbSet.FromSql` method for queries that return entity types. The returned objects must be of the type expected by the `DbSet` object, and they're automatically tracked by the database context unless you turn tracking off.
- Use the `Database.ExecuteSqlCommand` for non-query commands.

If you need to run a query that returns types that aren't entities, you can use ADO.NET with the database connection provided by EF. The returned data isn't tracked by the database context, even if you use this method to retrieve entity types.

As is always true when you execute SQL commands in a web application, you must take precautions to protect your site against SQL injection attacks. One way to do that is to use parameterized queries to make sure that strings submitted by a web page can't be interpreted as SQL commands. In this tutorial you'll use parameterized queries when integrating user input into a query.

Call a query to return entities

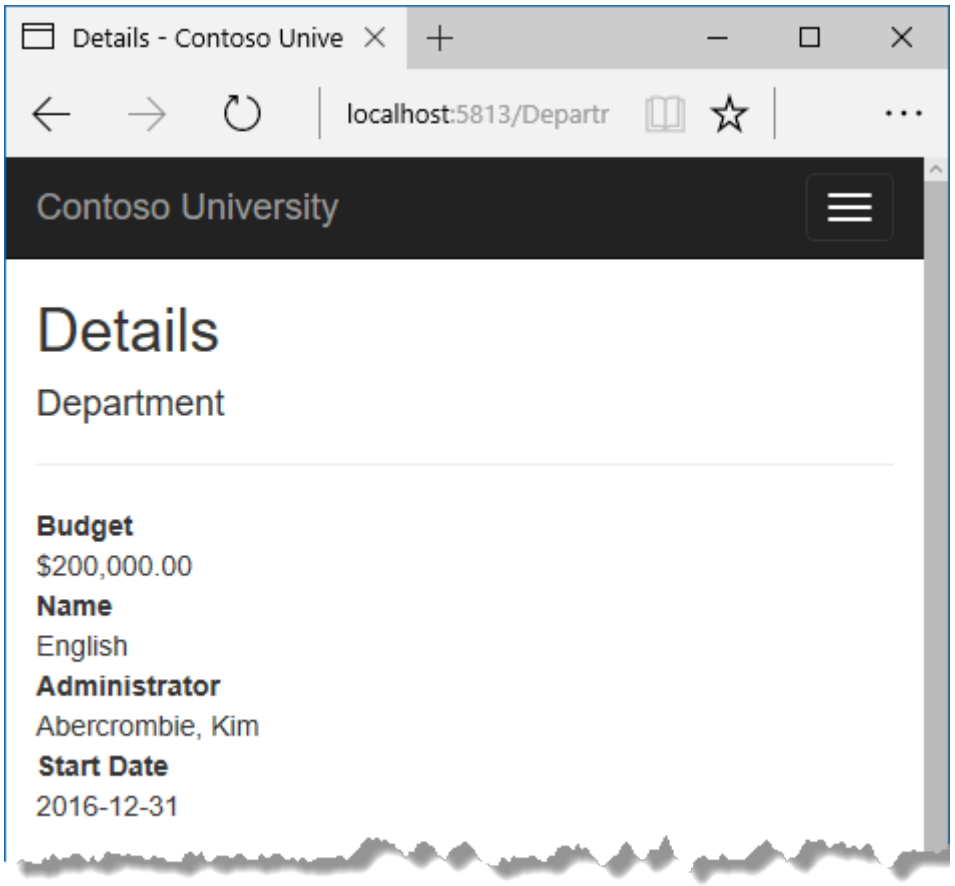
The `DbSet<TEntity>` class provides a method that you can use to execute a query that returns an entity of type `TEntity`. To see how this works you'll change the code in the `Details` method of the Department controller.

In `DepartmentsController.cs`, in the `Details` method, replace the code that retrieves a department with a `FromSql` method call, as shown in the following highlighted code:

C#

```
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    string query = "SELECT * FROM Department WHERE DepartmentID = {0}";
    var department = await _context.Departments
        .FromSql(query, id)
        .Include(d => d.Administrator)
        .AsNoTracking()
        .FirstOrDefaultAsync();

    if (department == null)
    {
        return NotFound();
    }

    return View(department);
}
```

To verify that the new code works correctly, select the **Departments** tab and then **Details** for one of the departments.



Call a query to return other types

Earlier you created a student statistics grid for the About page that showed the number of students for each enrollment date. You got the data from the Students entity set (`_context.Students`) and used LINQ to project the results into a list of `EnrollmentDateGroup` view model objects. Suppose you want to write the SQL itself rather than using LINQ. To do that you need to run a SQL query that returns something other than entity objects. In EF Core 1.0, one way to do that is to write ADO.NET code and get the database connection from EF.

In `HomeController.cs`, replace the `About` method with the following code:

C#

```
public async Task<ActionResult> About()
{
```

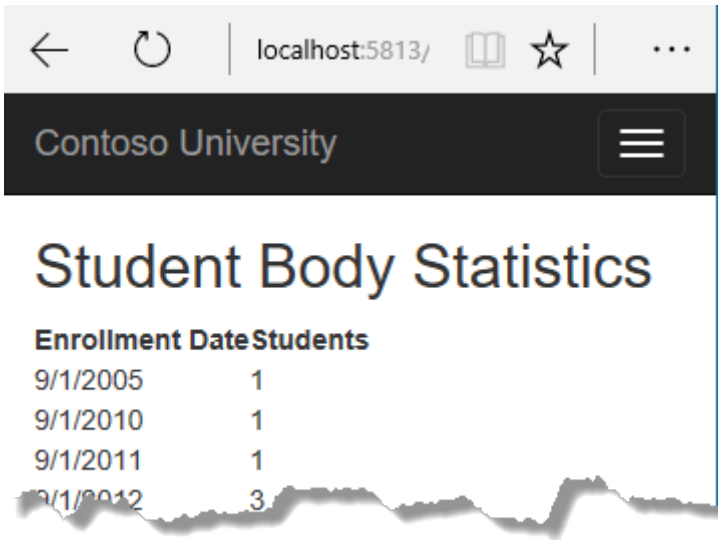
```
    List<EnrollmentDateGroup> groups = new List<EnrollmentDateGroup>();
    var conn = _context.Database.GetDbConnection();
    try
    {
        await conn.OpenAsync();
        using (var command = conn.CreateCommand())
        {
            string query = "SELECT EnrollmentDate, COUNT(*) AS StudentCount
"
                + "FROM Person "
                + "WHERE Discriminator = 'Student' "
                + "GROUP BY EnrollmentDate";
            command.CommandText = query;
            DbDataReader reader = await command.ExecuteReaderAsync();

            if (reader.HasRows)
            {
                while (await reader.ReadAsync())
                {
                    var row = new EnrollmentDateGroup { EnrollmentDate =
reader.GetDateTime(0), StudentCount = reader.GetInt32(1) };
                    groups.Add(row);
                }
            }
            reader.Dispose();
        }
    }
    finally
    {
        conn.Close();
    }
    return View(groups);
}
```

Add a using statement:

C#

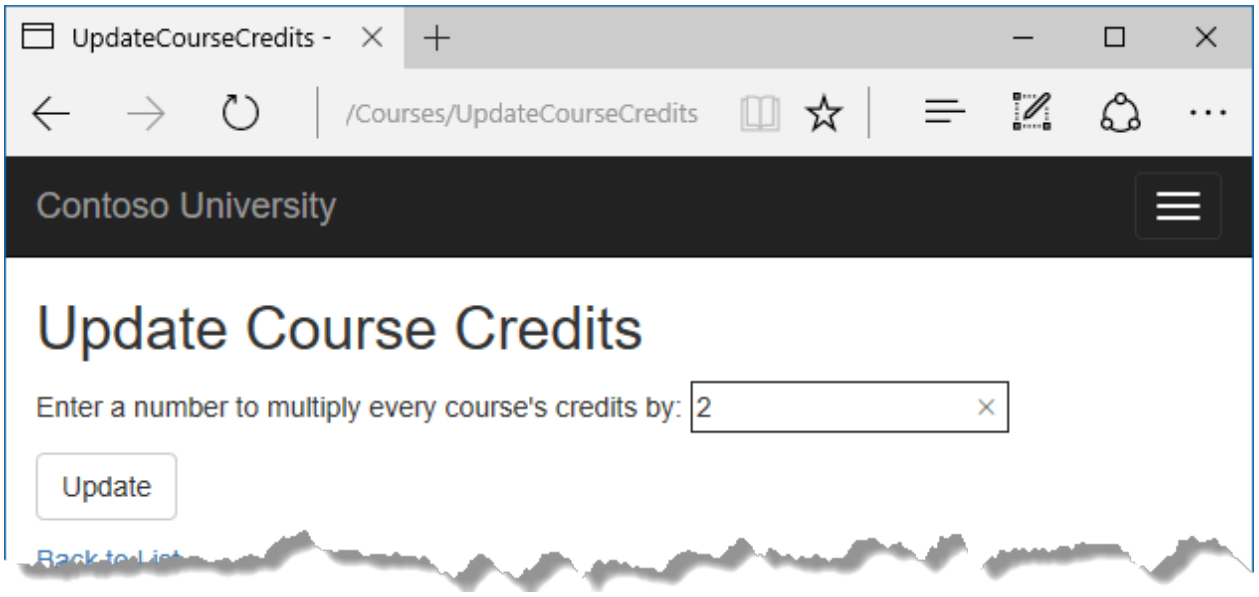
```
using System.Data.Common;
```

Run the app and go to the About page. It displays the same data it did before.



Call an update query

Suppose Contoso University administrators want to perform global changes in the database, such as changing the number of credits for every course. If the university has a large number of courses, it would be inefficient to retrieve them all as entities and change them individually. In this section you'll implement a web page that enables the user to specify a factor by which to change the number of credits for all courses, and you'll make the change by executing a SQL UPDATE statement. The web page will look like the following illustration:



In `CoursesController.cs`, add UpdateCourseCredits methods for HttpGet and HttpPost:

C#

```
public IActionResult UpdateCourseCredits()
{
```

```
    return View();
}
```

C#

```
[HttpPost]
public async Task<IActionResult> UpdateCourseCredits(int? multiplier)
{
    if (multiplier != null)
    {
        ViewData["RowsAffected"] =
            await _context.Database.ExecuteSqlCommandAsync(
                "UPDATE Course SET Credits = Credits * {0}",
                parameters: multiplier);
    }
    return View();
}
```

When the controller processes an HttpGet request, nothing is returned in `ViewData["RowsAffected"]`, and the view displays an empty text box and a submit button, as shown in the preceding illustration.

When the **Update** button is clicked, the HttpPost method is called, and multiplier has the value entered in the text box. The code then executes the SQL that updates courses and returns the number of affected rows to the view in `ViewData`. When the view gets a `RowsAffected` value, it displays the number of rows updated.

In **Solution Explorer**, right-click the *Views/Courses* folder, and then click **Add** > **New Item**.

In the **Add New Item** dialog, click **ASP.NET Core** under **Installed** in the left pane, click **Razor View**, and name the new view `UpdateCourseCredits.cshtml`.

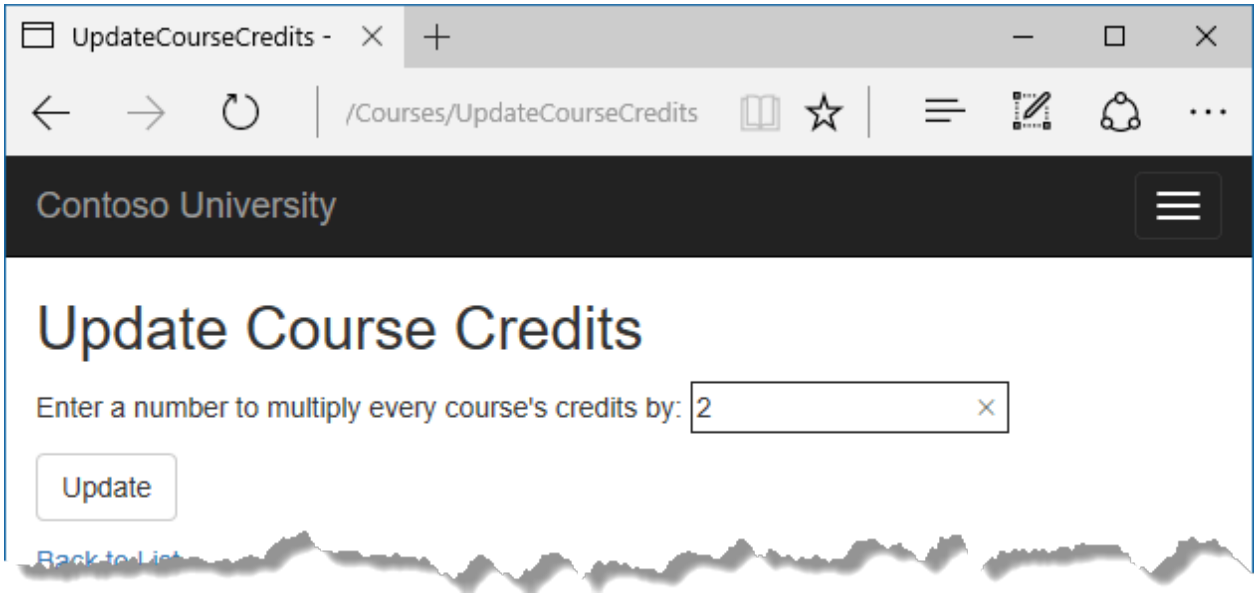
In `Views/Courses/UpdateCourseCredits.cshtml`, replace the template code with the following code:

CSHTML

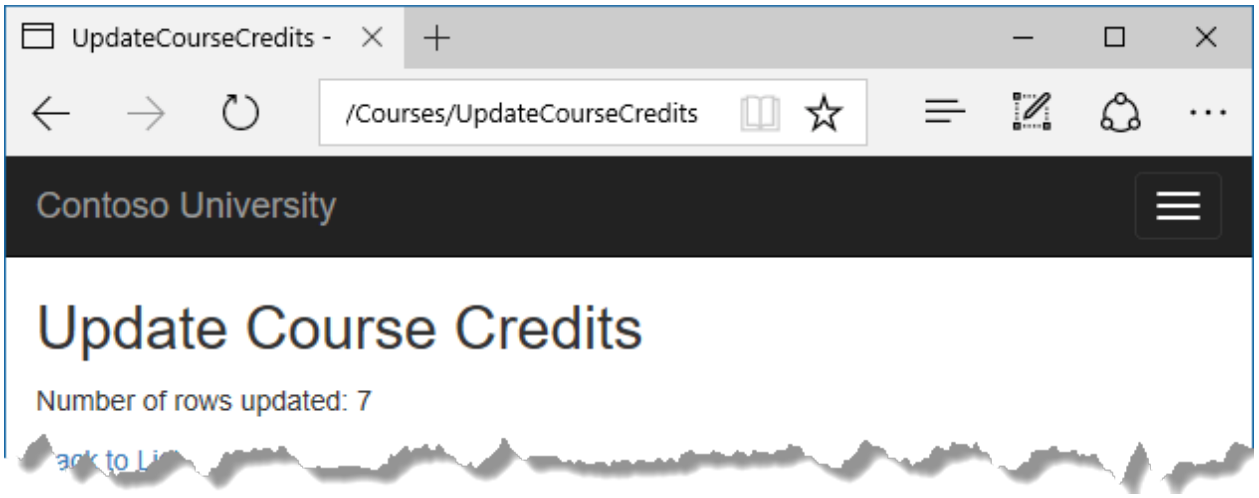
```
@{
    ViewBag.Title = "UpdateCourseCredits";
}

<h2>Update Course Credits</h2>

@if (ViewData["RowsAffected"] == null)
{
    <form asp-action="UpdateCourseCredits">
        <div class="form-actions no-color">
```

```
            <p>
                Enter a number to multiply every course's credits by:
@Html.TextBox("multiplier")
            </p>
            <p>
                <input type="submit" value="Update" class="btn btn-default"
/>
            </p>
        </div>
    </form>
}
@if (ViewData["RowsAffected"] != null)
{
    <p>
        Number of rows updated: @ViewData["RowsAffected"]
    </p>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Run the `UpdateCourseCredits` method by selecting the **Courses** tab, then adding "/UpdateCourseCredits" to the end of the URL in the browser's address bar (for example: `http://localhost:5813/Courses/UpdateCourseCredits`). Enter a number in the text box:



Click **Update**. You see the number of rows affected:



Click **Back to List** to see the list of courses with the revised number of credits.

Note that production code would ensure that updates always result in valid data. The simplified code shown here could multiply the number of credits enough to result in numbers greater than 5. (The `Credits` property has a `[Range(0, 5)]` attribute.) The update query would work but the invalid data could cause unexpected results in other parts of the system that assume the number of credits is 5 or less.

For more information about raw SQL queries, see Raw SQL Queries.

Examine SQL queries

Sometimes it's helpful to be able to see the actual SQL queries that are sent to the database. Built-in logging functionality for ASP.NET Core is automatically used by EF Core to write logs that contain the SQL for queries and updates. In this section you'll see some examples of SQL logging.

Open `StudentsController.cs` and in the `Details` method set a breakpoint on the `if (student == null)` statement.

Run the app in debug mode, and go to the Details page for a student.

Go to the **Output** window showing debug output, and you see the query:

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed
DbCommand (56ms) [Parameters=[@__id_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT TOP(2) [s].[ID], [s].[Discriminator], [s].[FirstName], [s].
[LastName], [s].[EnrollmentDate]
FROM [Person] AS [s]
WHERE ([s].[Discriminator] = N'Student') AND ([s].[ID] = @__id_0)
ORDER BY [s].[ID]
```

```
Microsoft.EntityFrameworkCore.Database.Command:Information: Executed
DbCommand (122ms) [Parameters=[@__id_0='?'], CommandType='Text',
CommandTimeout='30']
SELECT [s.Enrollments].[EnrollmentID], [s.Enrollments].[CourseID],
[s.Enrollments].[Grade], [s.Enrollments].[StudentID], [e.Course].[CourseID],
[e.Course].[Credits], [e.Course].[DepartmentID], [e.Course].[Title]
FROM [Enrollment] AS [s.Enrollments]
INNER JOIN [Course] AS [e.Course] ON [s.Enrollments].[CourseID] =
[e.Course].[CourseID]
INNER JOIN (
    SELECT TOP(1) [s0].[ID]
    FROM [Person] AS [s0]
    WHERE ([s0].[Discriminator] = N'Student') AND ([s0].[ID] = @__id_0)
    ORDER BY [s0].[ID]
) AS [t] ON [s.Enrollments].[StudentID] = [t].[ID]
ORDER BY [t].[ID]
```

You'll notice something here that might surprise you: the SQL selects up to 2 rows (`TOP(2)`) from the Person table. The `SingleOrDefaultAsync` method doesn't resolve to 1 row on the server. Here's why:

- If the query would return multiple rows, the method returns null.
- To determine whether the query would return multiple rows, EF has to check if it returns at least 2.

Note that you don't have to use debug mode and stop at a breakpoint to get logging output in the **Output** window. It's just a convenient way to stop the logging at the point you want to look at the output. If you don't do that, logging continues and you have to scroll back to find the parts you're interested in.

Create an abstraction layer

Many developers write code to implement the repository and unit of work patterns as a wrapper around code that works with the Entity Framework. These patterns are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing or test-driven development (TDD). However, writing additional code to implement these patterns isn't always the best choice for applications that use EF, for several reasons:

- The EF context class itself insulates your code from data-store-specific code.
- The EF context class can act as a unit-of-work class for database updates that you do using EF.
- EF includes features for implementing TDD without writing repository code.

For information about how to implement the repository and unit of work patterns, see the Entity Framework 5 version of this tutorial series.

Entity Framework Core implements an in-memory database provider that can be used for testing. For more information, see Test with InMemory.

Automatic change detection

The Entity Framework determines how an entity has changed (and therefore which updates need to be sent to the database) by comparing the current values of an entity with the original values. The original values are stored when the entity is queried or attached. Some of the methods that cause automatic change detection are the following:

- DbContext.SaveChanges
- DbContext.Entry
- ChangeTracker.Entries

If you're tracking a large number of entities and you call one of these methods many times in a loop, you might get significant performance improvements by temporarily turning off automatic change detection using the `ChangeTracker.AutoDetectChangesEnabled` property. For example:

```
C#

_context.ChangeTracker.AutoDetectChangesEnabled = false;
```

EF Core source code and development plans

The Entity Framework Core source is at https://github.com/dotnet/efcore. The EF Core repository contains nightly builds, issue tracking, feature specs, design meeting notes, and the roadmap for future development. You can file or find bugs, and contribute.

Although the source code is open, Entity Framework Core is fully supported as a Microsoft product. The Microsoft Entity Framework team keeps control over which contributions are accepted and tests all code changes to ensure the quality of each release.

Reverse engineer from existing database

To reverse engineer a data model including entity classes from an existing database, use the scaffold-dbcontext command. See the getting-started tutorial.

Use dynamic LINQ to simplify code

The third tutorial in this series shows how to write LINQ code by hard-coding column names in a `switch` statement. With two columns to choose from, this works fine, but if you have many columns the code could get verbose. To solve that problem, you can use the `EF.Property` method to specify the name of the property as a string. To try out this approach, replace the `Index` method in the `StudentsController` with the following code.

C#

```
 public async Task<IActionResult> Index(
     string sortOrder,
     string currentFilter,
     string searchString,
     int? pageNumber)
 {
     ViewData["CurrentSort"] = sortOrder;
     ViewData["NameSortParm"] =
         String.IsNullOrEmpty(sortOrder) ? "LastName_desc" : "";
     ViewData["DateSortParm"] =
         sortOrder == "EnrollmentDate" ? "EnrollmentDate_desc" :
"EnrollmentDate";

     if (searchString != null)
     {
         pageNumber = 1;
     }
     else
     {
         searchString = currentFilter;
     }

     ViewData["CurrentFilter"] = searchString;

     var students = from s in _context.Students
                    select s;

     if (!String.IsNullOrEmpty(searchString))
     {
         students = students.Where(s => s.LastName.Contains(searchString)
                                || s.FirstMidName.Contains(searchString));
     }

     if (string.IsNullOrEmpty(sortOrder))
     {
         sortOrder = "LastName";
     }
```

```
      bool descending = false;
      if (sortOrder.EndsWith("_desc"))
      {
          sortOrder = sortOrder.Substring(0, sortOrder.Length - 5);
          descending = true;
      }

      if (descending)
      {
          students = students.OrderByDescending(e => EF.Property<object>(e,
sortOrder));
      }
      else
      {
          students = students.OrderBy(e => EF.Property<object>(e,
sortOrder));
      }

      int pageSize = 3;
      return View(await
PaginatedList<Student>.CreateAsync(students.AsNoTracking(),
          pageNumber ?? 1, pageSize));
  }
```

Acknowledgments

Tom Dykstra and Rick Anderson (twitter @RickAndMSFT) wrote this tutorial. Rowan Miller, Diego Vega, and other members of the Entity Framework team assisted with code reviews and helped debug issues that arose while we were writing code for the tutorials. John Parente and Paul Goldman worked on updating the tutorial for ASP.NET Core 2.2.

Troubleshoot common errors

ContosoUniversity.dll used by another process

Error message:

> Cannot open '...bin\Debug\netcoreapp1.0\ContosoUniversity.dll' for writing -- 'The process cannot access the file '...\bin\Debug\netcoreapp1.0\ContosoUniversity.dll' because it is being used by another process.

Solution:

Stop the site in IIS Express. Go to the Windows System Tray, find IIS Express and right-click its icon, select the Contoso University site, and then click **Stop Site**.

Migration scaffolded with no code in Up and Down methods

Possible cause:

The EF CLI commands don't automatically close and save code files. If you have unsaved changes when you run the `migrations add` command, EF won't find your changes.

Solution:

Run the `migrations remove` command, save your code changes and rerun the `migrations add` command.

Errors while running database update

It's possible to get other errors when making schema changes in a database that has existing data. If you get migration errors you can't resolve, you can either change the database name in the connection string or delete the database. With a new database, there's no data to migrate, and the update-database command is much more likely to complete without errors.

The simplest approach is to rename the database in `appsettings.json`. The next time you run `database update`, a new database will be created.

To delete a database in SSOX, right-click the database, click **Delete**, and then in the **Delete Database** dialog box select **Close existing connections** and click **OK**.

To delete a database by using the CLI, run the `database drop` CLI command:

.NET CLI

```
dotnet ef database drop
```

Error locating SQL Server instance

Error Message:

> A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify

> that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: SQL Network Interfaces, error: 26 - Error Locating Server/Instance Specified)

Solution:

Check the connection string. If you have manually deleted the database file, change the name of the database in the construction string to start over with a new database.

Get the code

Download or view the completed application.

Additional resources

For more information about EF Core, see the Entity Framework Core documentation. A book is also available: Entity Framework Core in Action.

For information on how to deploy a web app, see Host and deploy ASP.NET Core.

For information about other topics related to ASP.NET Core MVC, such as authentication and authorization, see Overview of ASP.NET Core.

Next steps

In this tutorial, you:

- ✓ Performed raw SQL queries
- ✓ Called a query to return entities
- ✓ Called a query to return other types
- ✓ Called an update query
- ✓ Examined SQL queries
- ✓ Created an abstraction layer
- ✓ Learned about Automatic change detection
- ✓ Learned about EF Core source code and development plans
- ✓ Learned how to use dynamic LINQ to simplify code

This completes this series of tutorials on using the Entity Framework Core in an ASP.NET Core MVC application. This series worked with a new database; an alternative is to reverse engineer a model from an existing database.

Tutorial: EF Core with MVC, existing database

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core and Entity Framework 6

Article • 04/10/2024

By Patrick Goode

Using Entity Framework 6 with ASP.NET Core

Entity Framework Core should be used for new development. The download sample uses Entity Framework 6 (EF6), which can be used to migrate existing apps to ASP.NET Core.

Additional resources

- Entity Framework - Code-Based Configuration

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

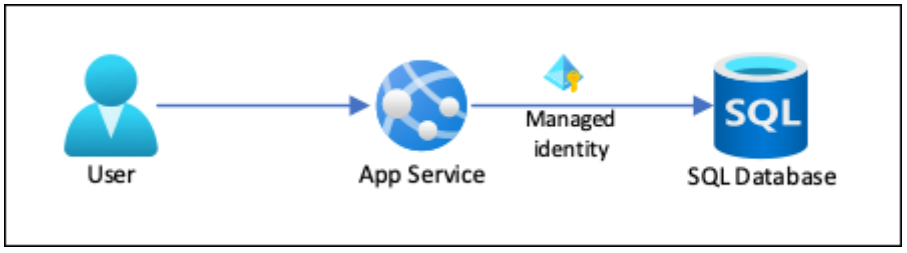
Tutorial: Connect to SQL Database from .NET App Service without secrets using a managed identity

Article • 04/17/2024

App Service provides a highly scalable, self-patching web hosting service in Azure. It also provides a managed identity for your app, which is a turn-key solution for securing access to Azure SQL Database and other Azure services. Managed identities in App Service make your app more secure by eliminating secrets from your app, such as credentials in the connection strings. In this tutorial, you add managed identity to the sample web app you built in one of the following tutorials:

- Tutorial: Build an ASP.NET app in Azure with Azure SQL Database
- Tutorial: Build an ASP.NET Core and Azure SQL Database app in Azure App Service

When you're finished, your sample app will connect to SQL Database securely without the need of username and passwords.



> **Note**
>
> The steps covered in this tutorial support the following versions:
>
> - .NET Framework 4.8 and above
> - .NET 6.0 and above
>
> For guidance for Azure Database for MySQL or Azure Database for PostgreSQL in other language frameworks (Node.js, Python, and Java), see Tutorial: Connect to Azure databases from App Service without secrets using a managed identity.

What you will learn:

- ✓ Enable managed identities
- ✓ Grant SQL Database access to the managed identity

- ✓ Configure Entity Framework to use Microsoft Entra authentication with SQL Database
- ✓ Connect to SQL Database from Visual Studio using Microsoft Entra authentication

> **Note**
>
> Microsoft Entra authentication is *different* from Integrated Windows authentication in on-premises Active Directory (AD DS). AD DS and Microsoft Entra ID use completely different authentication protocols. For more information, see Microsoft Entra Domain Services documentation.

If you don't have an Azure subscription, create an Azure free account before you begin.

Prerequisites

This article continues where you left off in either one of the following tutorials:

- Tutorial: Build an ASP.NET app in Azure with SQL Database
- Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

If you haven't already, follow one of the two tutorials first. Alternatively, you can adapt the steps for your own .NET app with SQL Database.

To debug your app using SQL Database as the back end, make sure that you've allowed client connection from your computer. If not, add the client IP by following the steps at Manage server-level IP firewall rules using the Azure portal.

Prepare your environment for the Azure CLI.

- Use the Bash environment in Azure Cloud Shell. For more information, see Quickstart for Bash in Azure Cloud Shell.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
  - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.

- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
- Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

1. Grant database access to Microsoft Entra user

First, enable Microsoft Entra authentication to SQL Database by assigning a Microsoft Entra user as the admin of the server. This user is different from the Microsoft account you used to sign up for your Azure subscription. It must be a user that you created, imported, synced, or invited into Microsoft Entra ID. For more information on allowed Microsoft Entra users, see Microsoft Entra features and limitations in SQL Database.

1. If your Microsoft Entra tenant doesn't have a user yet, create one by following the steps at Add or delete users using Microsoft Entra ID.
2. Find the object ID of the Microsoft Entra user using the az ad user list and replace *<user-principal-name>*. The result is saved to a variable.

   Azure CLI

   ```
   $azureaduser=(az ad user list --filter "userPrincipalName eq '<user-principal-name>'" --query '[].id' --output tsv)
   ```

   > **Tip**
   >
   > To see the list of all user principal names in Microsoft Entra ID, run `az ad user list --query '[].userPrincipalName'`.

3. Add this Microsoft Entra user as an Active Directory admin using az sql server ad-admin create command in the Cloud Shell. In the following command, replace *<server-name>* with the server name (without the `.database.windows.net` suffix).

   Azure CLI

   ```
   az sql server ad-admin create --resource-group myResourceGroup --server-name <server-name> --display-name ADMIN --object-id $azureaduser
   ```

For more information on adding an Active Directory admin, see Provision a Microsoft Entra administrator for your server

2. Set up your dev environment

Visual Studio Windows

1. Visual Studio for Windows is integrated with Microsoft Entra authentication. To enable development and debugging in Visual Studio, add your Microsoft Entra user in Visual Studio by selecting **File** > **Account Settings** from the menu, and select **Sign in** or **Add**.
2. To set the Microsoft Entra user for Azure service authentication, select **Tools** > **Options** from the menu, then select **Azure Service Authentication** > **Account Selection**. Select the Microsoft Entra user you added and select **OK**.

For more information about setting up your dev environment for Microsoft Entra authentication, see Azure Identity client library for .NET.

You're now ready to develop and debug your app with the SQL Database as the back end, using Microsoft Entra authentication.

3. Modify your project

> **Note**
>
> **Microsoft.Azure.Services.AppAuthentication** is no longer recommended to use with new Azure SDK. It is replaced with new **Azure Identity client library** available for .NET, Java, TypeScript and Python and should be used for all new development. Information about how to migrate to `Azure Identity` can be found here: AppAuthentication to Azure.Identity Migration Guidance.

The steps you follow for your project depends on whether you're using Entity Framework Core (default for ASP.NET Core) or Entity Framework (default for ASP.NET).

Entity Framework Core

1. In Visual Studio, open the Package Manager Console and add the NuGet package Microsoft.Data.SqlClient:

   PowerShell

```
Install-Package Microsoft.Data.SqlClient -Version 5.1.0
```

2. In the ASP.NET Core and SQL Database tutorial, the `MyDbConnection` connection string in *appsettings.json* isn't used at all yet. The local environment and the Azure environment both get connection strings from their respective environment variables in order to keep connection secrets out of the source file. But now with Active Directory authentication, there are no more secrets. In *appsettings.json*, replace the value of the `MyDbConnection` connection string with:

   JSON

   ```
   "Server=tcp:<server-name>.database.windows.net;Authentication=Active Directory Default; Database=<database-name>;"
   ```

   > **Note**
   >
   > The Active Directory Default authentication type can be used both on your local machine and in Azure App Service. The driver attempts to acquire a token from Microsoft Entra ID using various means. If the app is deployed, it gets a token from the app's system-assigned managed identity. It can also authenticate with a user-assigned managed identity if you include: `User Id=<client-id-of-user-assigned-managed-identity>;` in your connection string. If the app is running locally, it tries to get a token from Visual Studio, Visual Studio Code, and Azure CLI.

   That's everything you need to connect to SQL Database. When you debug in Visual Studio, your code uses the Microsoft Entra user you configured in 2. Set up your dev environment. You'll set up SQL Database later to allow connection from the managed identity of your App Service app. The `DefaultAzureCredential` class caches the token in memory and retrieves it from Microsoft Entra ID just before expiration. You don't need any custom code to refresh the token.

3. Type `Ctrl+F5` to run the app again. The same CRUD app in your browser is now connecting to the Azure SQL Database directly, using Microsoft Entra authentication. This setup lets you run database migrations from Visual Studio.

4. Use managed identity connectivity

Next, you configure your App Service app to connect to SQL Database with a system-assigned managed identity.

> ⓘ Note
>
> The instructions in this section are for a system-assigned identity, To use a user-assigned identity, see Tutorial: Connect to Azure databases from App Service without secrets using a managed identity.

Enable managed identity on app

To enable a managed identity for your Azure app, use the az webapp identity assign command in the Cloud Shell. In the following command, replace *<app-name>*.

Azure CLI

```
az webapp identity assign --resource-group myResourceGroup --name <app-name>
```

> ⓘ Note
>
> To enable managed identity for a deployment slot, add `--slot <slot-name>` and use the name of the slot in *<slot-name>*.

Here's an example of the output:


```
{
  "additionalProperties": {},
  "principalId": "aaaaaaaa-bbbb-cccc-1111-222222222222",
  "tenantId": "aaaabbbb-0000-cccc-1111-dddd2222eeee",
  "type": "SystemAssigned"
}
```


Grant permissions to managed identity

> ⓘ Note
>
> If you want, you can add the identity to an Microsoft Entra group, then grant SQL Database access to the Microsoft Entra group instead of the identity. For example,

the following commands add the managed identity from the previous step to a new group called *myAzureSQLDBAccessGroup*:

Azure CLI

```
$groupid=(az ad group create --display-name myAzureSQLDBAccessGroup --
mail-nickname myAzureSQLDBAccessGroup --query objectId --output tsv)
$msiobjectid=(az webapp identity show --resource-group myResourceGroup
--name <app-name> --query principalId --output tsv)
az ad group member add --group $groupid --member-id $msiobjectid
az ad group member list -g $groupid
```

1. In the Cloud Shell, sign in to SQL Database by using the SQLCMD command. Replace <*server-name*> with your server name, <*db-name*> with the database name your app uses, and <*aad-user-name*> and <*aad-password*> with your Microsoft Entra user's credentials.

   Bash

   ```
   sqlcmd -S <server-name>.database.windows.net -d <db-name> -U <aad-user-
   name> -P "<aad-password>" -G -l 30
   ```

2. In the SQL prompt for the database you want, run the following commands to grant the minimum permissions your app needs. For example,

   SQL

   ```
   CREATE USER [<identity-name>] FROM EXTERNAL PROVIDER;
   ALTER ROLE db_datareader ADD MEMBER [<identity-name>];
   ALTER ROLE db_datawriter ADD MEMBER [<identity-name>];
   ALTER ROLE db_ddladmin ADD MEMBER [<identity-name>];
   GO
   ```

   <*identity-name*> is the name of the managed identity in Microsoft Entra ID. If the identity is system-assigned, the name is always the same as the name of your App Service app. For a deployment slot, the name of its system-assigned identity is <*app-name*>/slots/<*slot-name*>. To grant permissions for a Microsoft Entra group, use the group's display name instead (for example, *myAzureSQLDBAccessGroup*).

3. Type `EXIT` to return to the Cloud Shell prompt.

   Note

The back-end services of managed identities also maintains a token cache that updates the token for a target resource only when it expires. If you make a mistake configuring your SQL Database permissions and try to modify the permissions *after* trying to get a token with your app, you don't actually get a new token with the updated permissions until the cached token expires.

> **Note**
>
> Microsoft Entra ID and managed identities are not supported for on-premises SQL Server.

Modify connection string

Remember that the same changes you made in *Web.config* or *appsettings.json* works with the managed identity, so the only thing to do is to remove the existing connection string in App Service, which Visual Studio created deploying your app the first time. Use the following command, but replace <*app-name*> with the name of your app.

Azure CLI

```
az webapp config connection-string delete --resource-group myResourceGroup -
-name <app-name> --setting-names MyDbConnection
```

5. Publish your changes

All that's left now is to publish your changes to Azure.

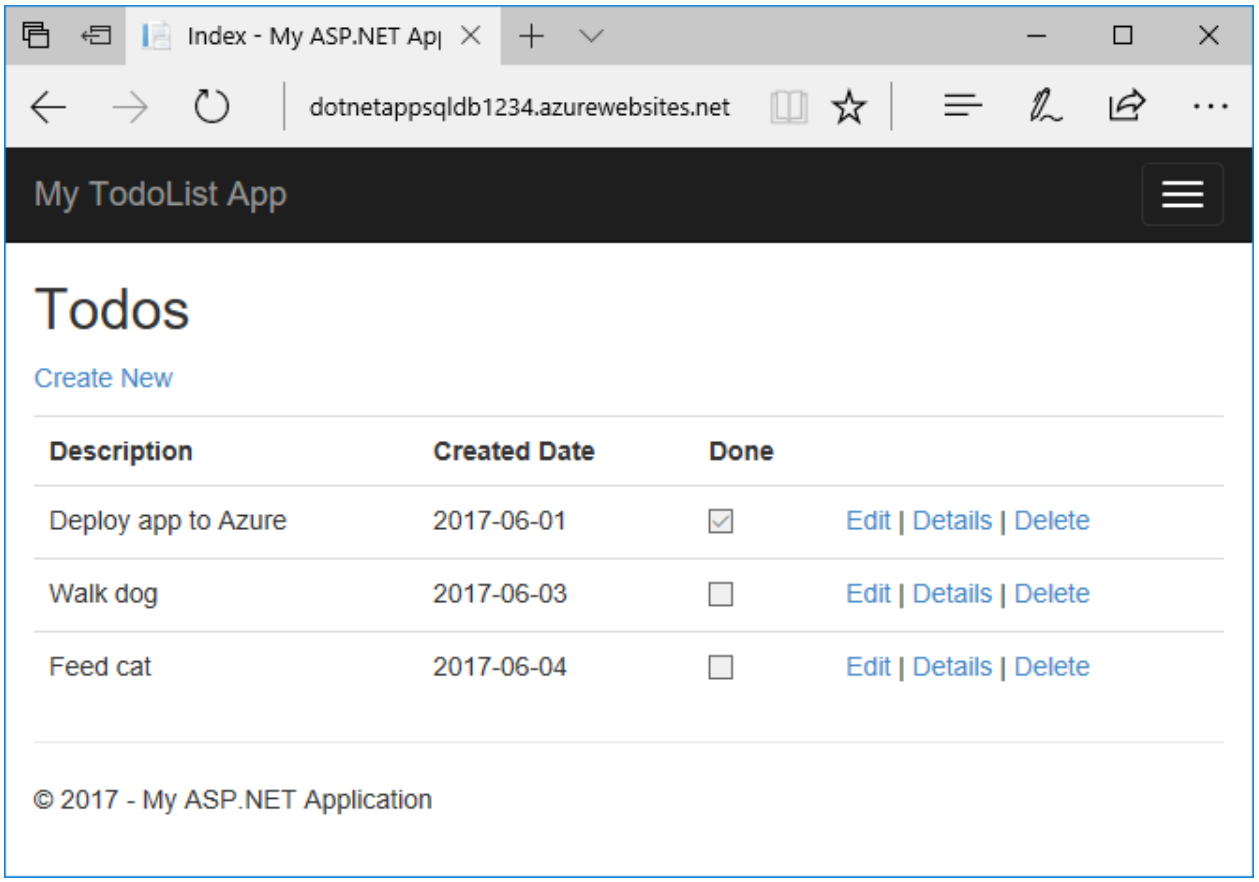
ASP.NET Core

If you came from Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service, publish your changes using Git, with the following commands:

Bash

```
git commit -am "configure managed identity"
git push azure main
```

When the new webpage shows your to-do list, your app is connecting to the database using the managed identity.



You should now be able to edit the to-do list as before.

Clean up resources

In the preceding steps, you created Azure resources in a resource group. If you don't expect to need these resources in the future, delete the resource group by running the following command in the Cloud Shell:

Azure CLI

```
az group delete --name myResourceGroup
```

This command may take a minute to run.

Next steps

What you learned:

- ✓ Enable managed identities
- ✓ Grant SQL Database access to the managed identity
- ✓ Configure Entity Framework to use Microsoft Entra authentication with SQL Database
- ✓ Connect to SQL Database from Visual Studio using Microsoft Entra authentication

Secure with custom domain and certificate

Tutorial: Connect an App Service app to SQL Database on behalf of the signed-in user

Tutorial: Connect to Azure databases from App Service without secrets using a managed identity

Tutorial: Connect to Azure services that don't support managed identities (using Key Vault)

Tutorial: Isolate back-end communication with Virtual Network integration

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Adding Azure Storage by using Visual Studio Connected Services

Article • 10/31/2024

With Visual Studio, you can connect any of the following to Azure Storage by using the **Connected Services** feature:

- .NET Framework console app
- ASP.NET Model-View-Controller (MVC) (.NET Framework)
- ASP.NET Core
- .NET Core (including console app, WPF, Windows Forms, class library)
- .NET Core Worker Role
- Azure Functions
- Universal Windows Platform App
- Xamarin
- Cordova

The connected service functionality adds all the needed references and connection code to your project, and modifies your configuration files appropriately.

Prerequisites

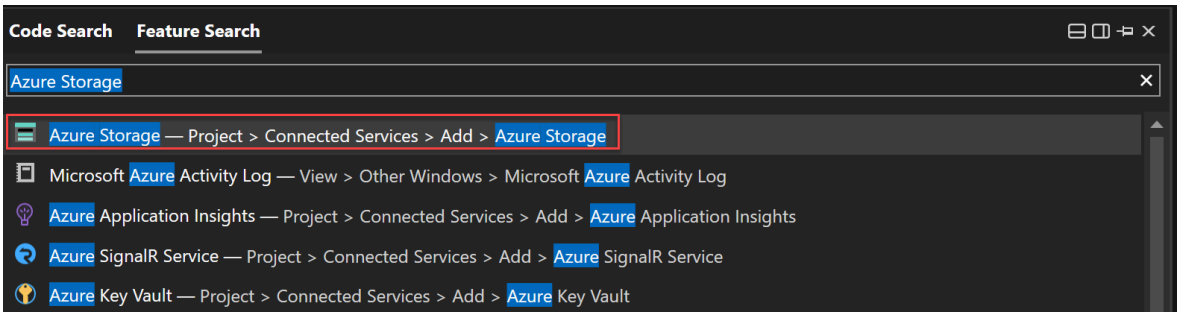
- Visual Studio (see [Visual Studio downloads] (https://visualstudio.microsoft.com/downloads/?cid=learn-onpage-download-cta)) with the **Azure development** workload installed.
- A project of one of the supported types
- An Azure account. If you don't have an Azure account, activate your Azure benefits for Visual Studio subscribers or sign up for a free trial.

Connect to Azure Storage using Connected Services

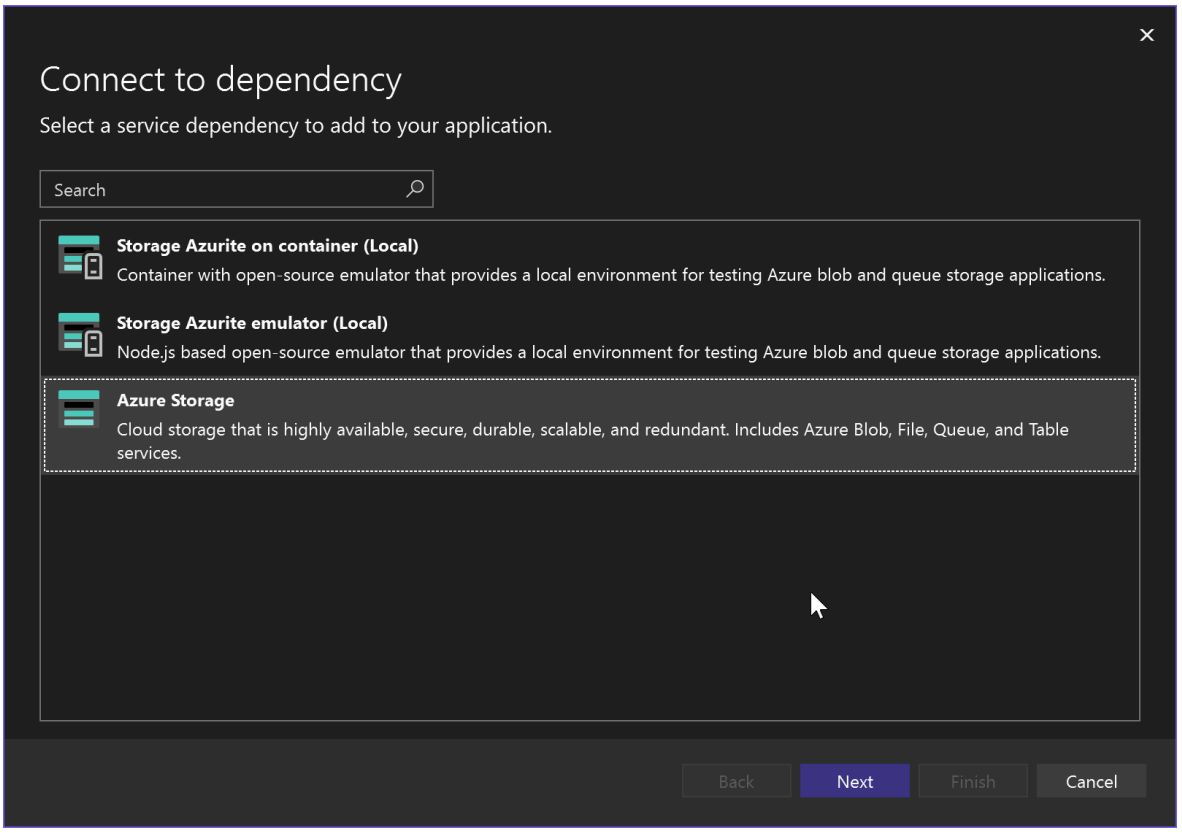
1. Open your project in Visual Studio.

2. Press **Ctrl**+**Q** (or use the **Search** button in the Visual Studio IDE to the right of the main menu bar).

3. In **Feature search**, enter `Azure Storage`, and choose **Azure Storage - Project > Connected Services > Add > Azure Storage**.



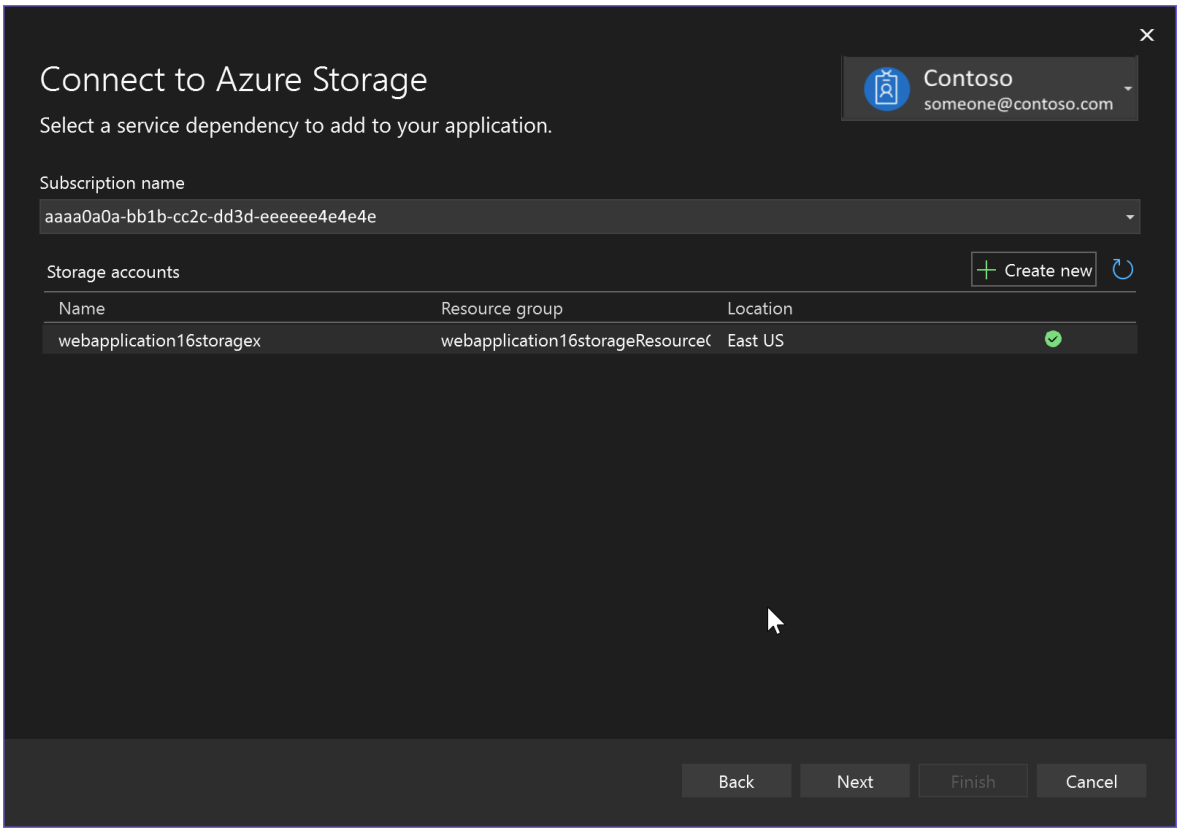
4. In the **Connect to dependency** page, select **Azure Storage**, and then select **Next**.



   If you aren't signed in already, sign in to your Azure account. If you don't have an Azure account, you can sign up for a free trial.

5. In the **Connect to Azure Storage** screen, select an existing storage account, and select **Next**.

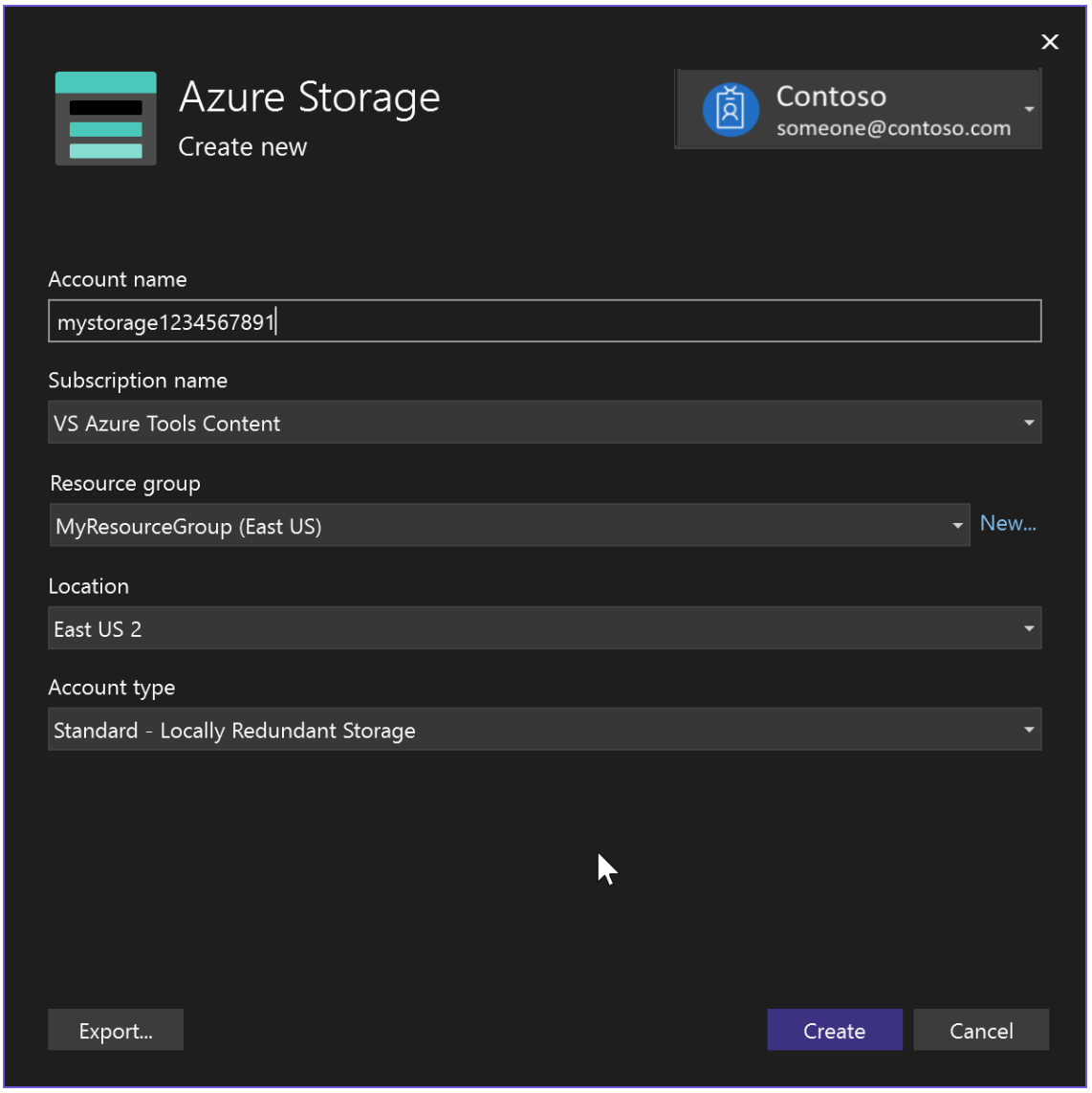
   If you need to create a storage account, go to the next step. Otherwise, skip to the following step.



6. To create a storage account:

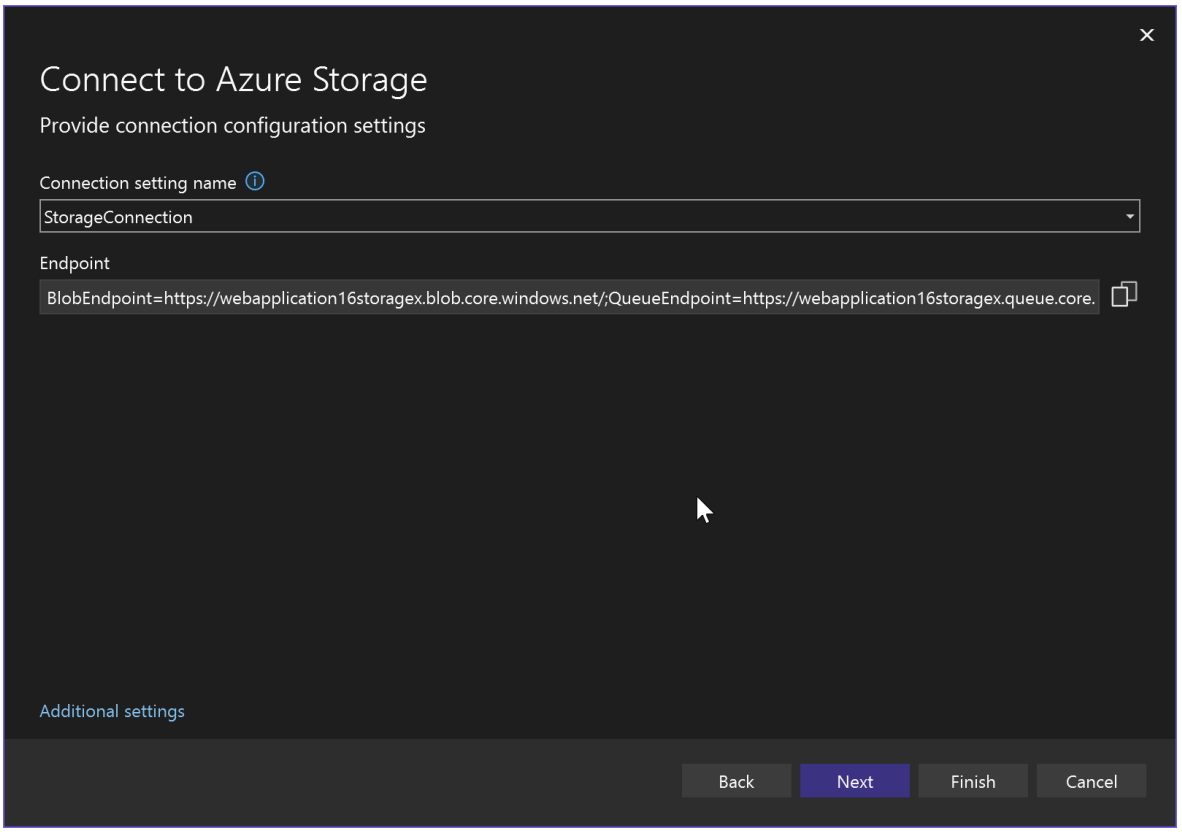
   a. Select **Create new** by the green plus sign.

   b. Fill out the **Azure Storage: Create new** dialog, and select **Create**.

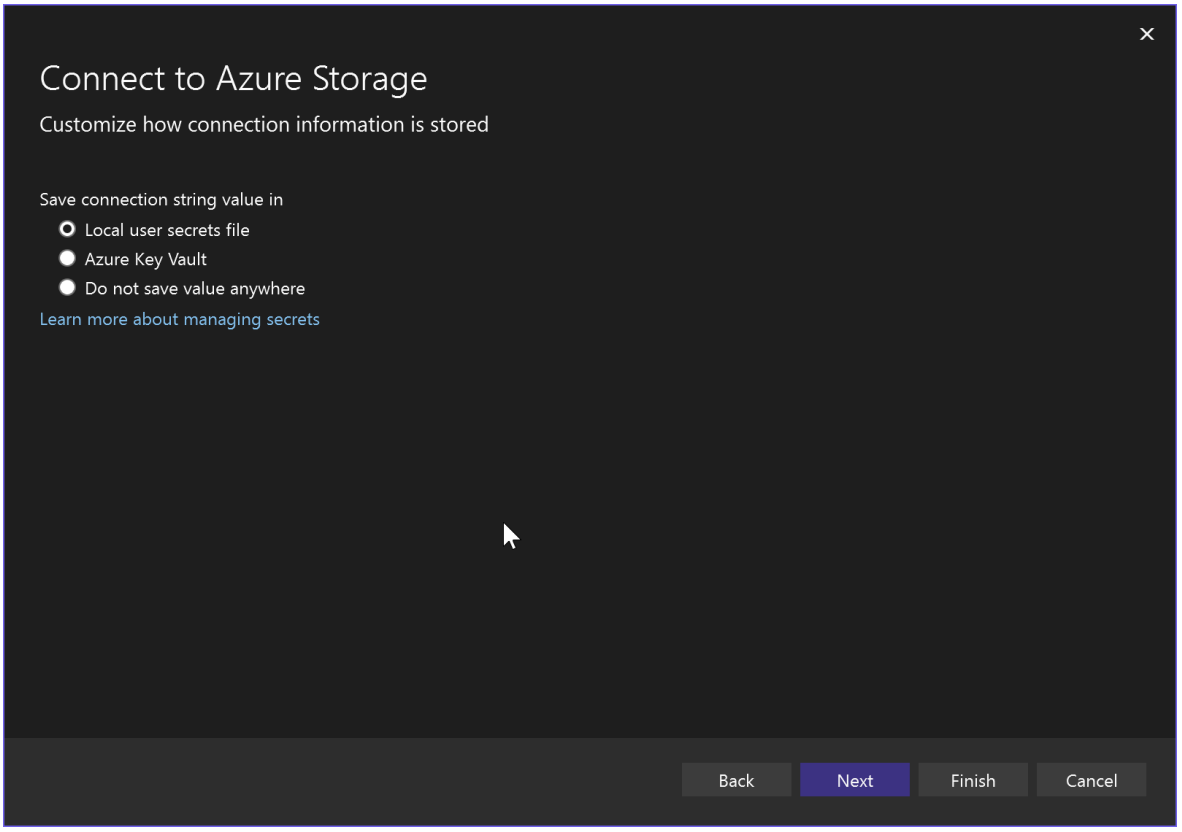


c. When the **Azure Storage** dialog is displayed, the new storage account appears in the list. Select the new storage account in the list, and select **Next**.

7. Enter a connection string setting name. The setting name references the name of the connection string setting as it appears in the *secrets.json* file, or in Azure Key Vault.



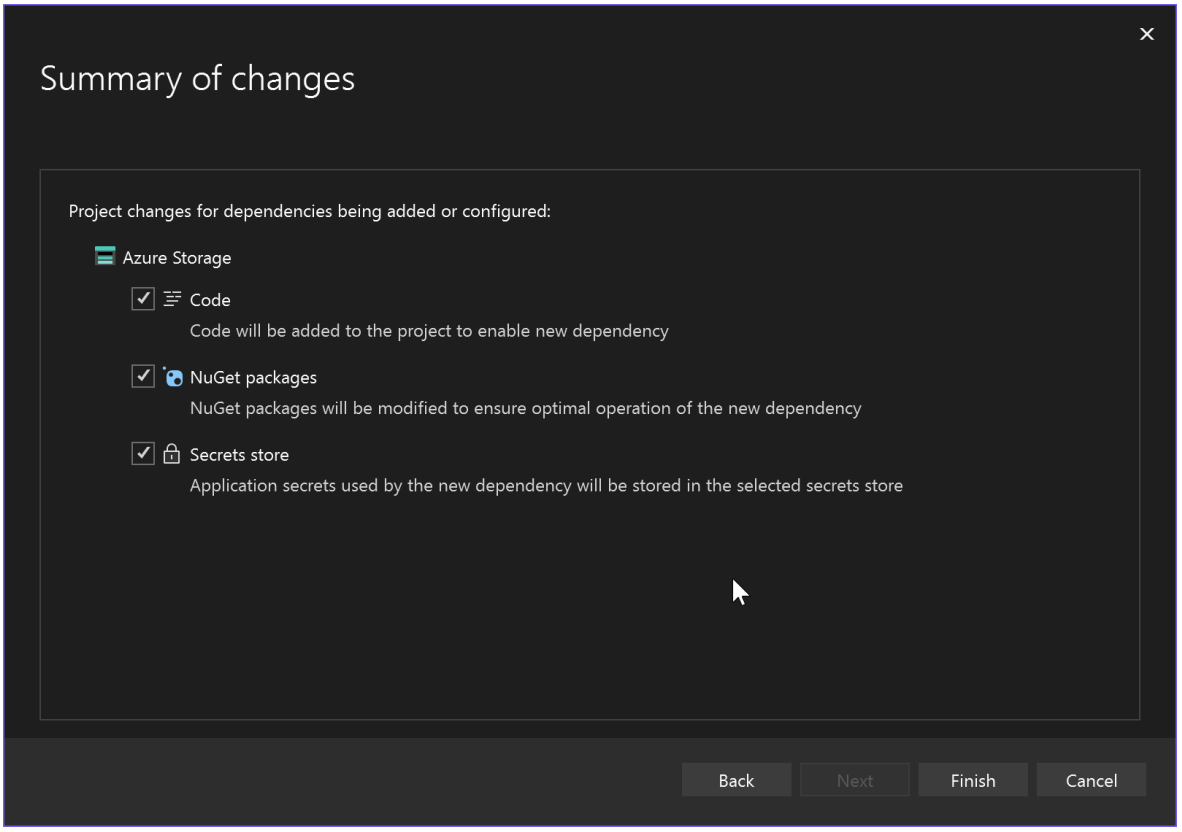
8. Choose whether you want the connection string stored in a local secrets file, in Azure Key Vault, or not stored anywhere.



> **⊗ Caution**
>
> If you're using a version of Visual Studio earlier than Visual Studio 17.12, and you choose to use a *secrets.json* file, you must take security precautions, since

the connection string in the local secrets.json file could be exposed. If you're using Visual Studio 2022 version 17.12 or later, this procedure produces more secure result, because it yields a connection setting value, instead of a connection string with authentication credentials.

9. The **Summary of changes** screen shows all the modifications that will be made to your project if you complete the process. If the changes look OK, choose **Finish**.



10. The storage connected service appears under the **Connected Services** node of your project.

Understand authentication

After you run the previous procedure, your app is set up to use authentication to access the storage account. The connection information for this authentication are stored locally, if you chose the *secrets.json* method, or in your Azure key vault.

If you used the *secrets.json* file, open the file by using the three dots next to **Secrets.json** on the **Connected Services** tab to open a menu, and choose **Manage user secrets**. With Visual Studio 2022 version 17.12 and later, this file contains settings that reference a URI to obtain the secure connection string, rather than the connection string itself.

JSON

```
{
  "StorageConnection:blobServiceUri":
"https://webapplication16storagex.blob.core.windows.net/",
  "StorageConnection:queueServiceUri":
"https://webapplication16storagex.queue.core.windows.net/",
  "StorageConnection:tableServiceUri":
"https://webapplication16storagex.table.core.windows.net/"
}
```

With these settings in Visual Studio 17.12 and later, authentication is automatic and flexible. When you run or debug locally from Visual Studio, your Azure credentials saved by Visual Studio are used to access the Azure Storage account. If you launch your app from the command-line, you first need to sign in using the Azure CLI, and those credentials are automatically detected and used. But when your app is deployed to Azure and runs in Azure, it uses managed identity, without any code changes. The authentication works in all hosting environments because the Azure Identity APIs check for all chained credentials in sequence and use them when they're found. See DefaultAzureCredential.

Next steps

Azure Storage supports blobs and queues, as well as other features.

To learn about working with blobs, you can continue with the quickstart for blob storage, but instead of starting at the beginning, you can start at Azure blobs quickstart (.NET) - Code examples.

To learn about working with queues, start at Azure Queue Storage quickstart (.NET) - Code examples.

Related content

- Azure Storage forum
- Azure Storage documentation

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Quickstart: Azure Blob Storage client library for .NET

Article • 09/13/2024

> **Note**
>
> The **Build from scratch** option walks you step by step through the process of creating a new project, installing packages, writing the code, and running a basic console app. This approach is recommended if you want to understand all the details involved in creating an app that connects to Azure Blob Storage. If you prefer to automate deployment tasks and start with a completed project, choose Start with a template.

Get started with the Azure Blob Storage client library for .NET. Azure Blob Storage is Microsoft's object storage solution for the cloud, and is optimized for storing massive amounts of unstructured data.

In this article, you follow steps to install the package and try out example code for basic tasks.

API reference documentation | Library source code | Package (NuGet) | Samples

This video shows you how to start using the Azure Blob Storage client library for .NET. https://learn-video.azurefd.net/vod/player?id=cdae65e7-1892-48fe-934a-70edfbe147be&locale=en-us&embedUrl=%2Fazure%2Fstorage%2Fblobs%2Fstorage-quickstart-blobs-dotnet

The steps in the video are also described in the following sections.

Prerequisites

- Azure subscription - create one for free
- Azure storage account - create a storage account
- Latest .NET SDK for your operating system. Be sure to get the SDK and not the runtime.

Setting up

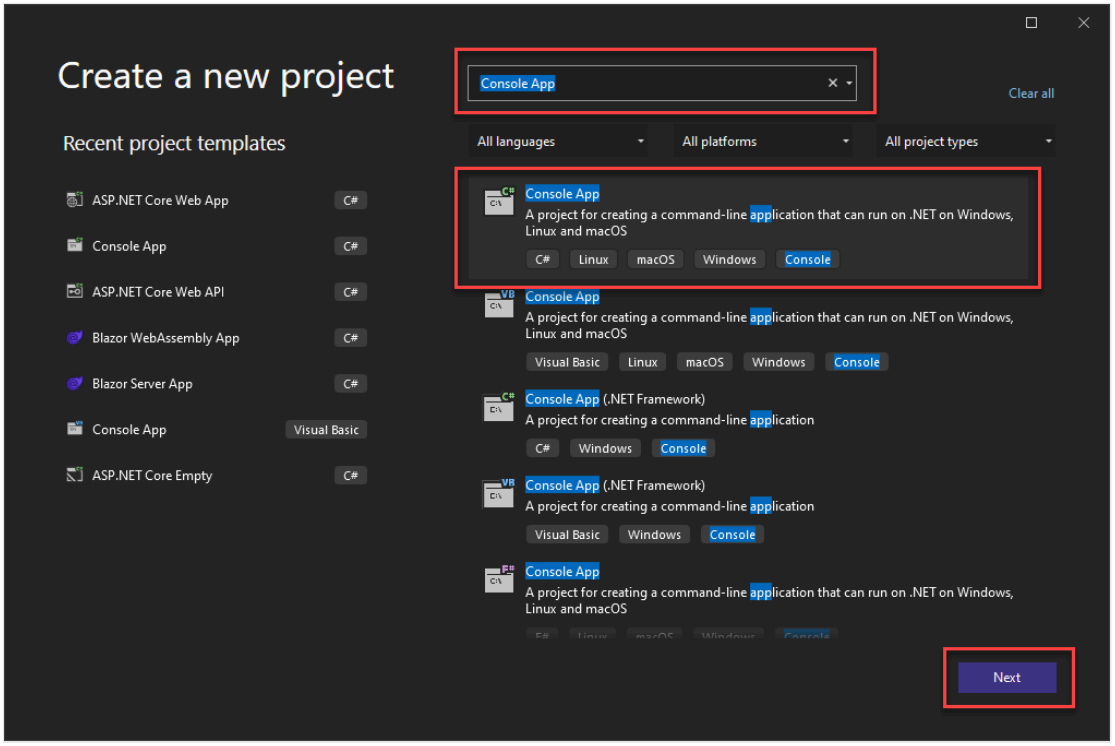
This section walks you through preparing a project to work with the Azure Blob Storage client library for .NET.

Create the project

Create a .NET console app using either the .NET CLI or Visual Studio 2022.

Visual Studio 2022

1. At the top of Visual Studio, navigate to **File** > **New** > **Project...**
2. In the dialog window, enter *console app* into the project template search box and select the first result. Choose **Next** at the bottom of the dialog.



3. For the **Project Name**, enter *BlobQuickstart*. Leave the default values for the rest of the fields and select **Next**.
4. For the **Framework**, ensure the latest installed version of .NET is selected. Then choose **Create**. The new project opens inside the Visual Studio environment.

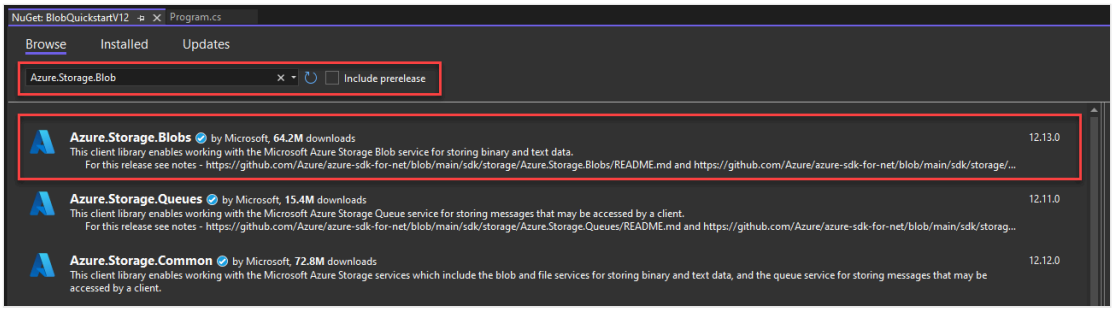
Install the package

To interact with Azure Blob Storage, install the Azure Blob Storage client library for .NET.

Visual Studio 2022

1. In **Solution Explorer**, right-click the **Dependencies** node of your project. Select **Manage NuGet Packages**.

2. In the resulting window, search for *Azure.Storage.Blobs*. Select the appropriate result, and select **Install**.



Set up the app code

Replace the starting code in the `Program.cs` file so that it matches the following example, which includes the necessary `using` statements for this exercise.

C#

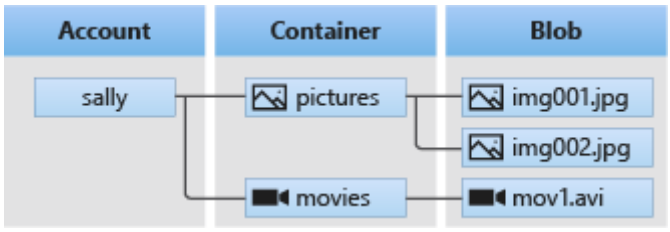
```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using System;
using System.IO;

// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Object model

Azure Blob Storage is optimized for storing massive amounts of unstructured data. Unstructured data doesn't adhere to a particular data model or definition, such as text or binary data. Blob storage offers three types of resources:

- The storage account
- A container in the storage account
- A blob in the container

The following diagram shows the relationship between these resources.



Use the following .NET classes to interact with these resources:

- BlobServiceClient: The `BlobServiceClient` class allows you to manipulate Azure Storage resources and blob containers.
- BlobContainerClient: The `BlobContainerClient` class allows you to manipulate Azure Storage containers and their blobs.
- BlobClient: The `BlobClient` class allows you to manipulate Azure Storage blobs.

Code examples

The sample code snippets in the following sections demonstrate how to perform the following tasks with the Azure Blob Storage client library for .NET:

- Authenticate to Azure and authorize access to blob data
- Create a container
- Upload a blob to a container
- List blobs in a container
- Download a blob
- Delete a container

> ⓘ **Important**
>
> Make sure you've installed the correct NuGet packages and added the necessary using statements in order for the code samples to work, as described in the **setting up** section.

Authenticate to Azure and authorize access to blob data

Application requests to Azure Blob Storage must be authorized. Using the `DefaultAzureCredential` class provided by the Azure Identity client library is the recommended approach for implementing passwordless connections to Azure services in your code, including Blob Storage.

You can also authorize requests to Azure Blob Storage by using the account access key. However, this approach should be used with caution. Developers must be diligent to

never expose the access key in an unsecure location. Anyone who has the access key is able to authorize requests against the storage account, and effectively has access to all the data. `DefaultAzureCredential` offers improved management and security benefits over the account key to allow passwordless authentication. Both options are demonstrated in the following example.

Passwordless (Recommended)

`DefaultAzureCredential` is a class provided by the Azure Identity client library for .NET, which you can learn more about on the DefaultAzureCredential overview. `DefaultAzureCredential` supports multiple authentication methods and determines which method should be used at runtime. This approach enables your app to use different authentication methods in different environments (local vs. production) without implementing environment-specific code.

The order and locations in which `DefaultAzureCredential` looks for credentials can be found in the Azure Identity library overview.

For example, your app can authenticate using your Visual Studio sign-in credentials with when developing locally. Your app can then use a managed identity once it has been deployed to Azure. No code changes are required for this transition.

Assign roles to your Microsoft Entra user account

When developing locally, make sure that the user account that is accessing blob data has the correct permissions. You'll need **Storage Blob Data Contributor** to read and write blob data. To assign yourself this role, you'll need to be assigned the **User Access Administrator** role, or another role that includes the **Microsoft.Authorization/roleAssignments/write** action. You can assign Azure RBAC roles to a user using the Azure portal, Azure CLI, or Azure PowerShell. You can learn more about the available scopes for role assignments on the scope overview page.

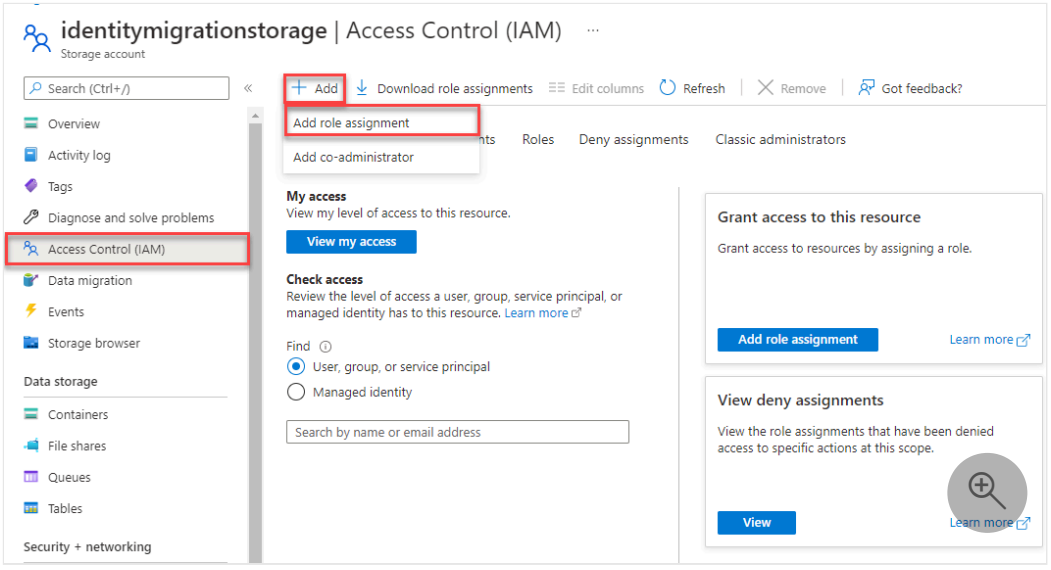
In this scenario, you'll assign permissions to your user account, scoped to the storage account, to follow the Principle of Least Privilege. This practice gives users only the minimum permissions needed and creates more secure production environments.

The following example will assign the **Storage Blob Data Contributor** role to your user account, which provides both read and write access to blob data in your storage account.

> **Important**
>
> In most cases it will take a minute or two for the role assignment to propagate in Azure, but in rare cases it may take up to eight minutes. If you receive authentication errors when you first run your code, wait a few moments and try again.

Azure portal

1. In the Azure portal, locate your storage account using the main search bar or left navigation.
2. On the storage account overview page, select **Access control (IAM)** from the left-hand menu.
3. On the **Access control (IAM)** page, select the **Role assignments** tab.
4. Select **+ Add** from the top menu and then **Add role assignment** from the resulting drop-down menu.



5. Use the search box to filter the results to the desired role. For this example, search for *Storage Blob Data Contributor* and select the matching result and then choose **Next**.
6. Under **Assign access to**, select **User, group, or service principal**, and then choose **+ Select members**.
7. In the dialog, search for your Microsoft Entra username (usually your *user@domain* email address) and then choose **Select** at the bottom of the

dialog.

8. Select **Review + assign** to go to the final page, and then **Review + assign** again to complete the process.

Sign in and connect your app code to Azure using DefaultAzureCredential

You can authorize access to data in your storage account using the following steps:

1. For local development, make sure you're authenticated with the same Microsoft Entra account you assigned the role to. You can authenticate via popular development tools, such as the Azure CLI or Azure PowerShell. The development tools with which you can authenticate vary across languages.

   Azure CLI

   Sign-in to Azure through the Azure CLI using the following command:

   Azure CLI

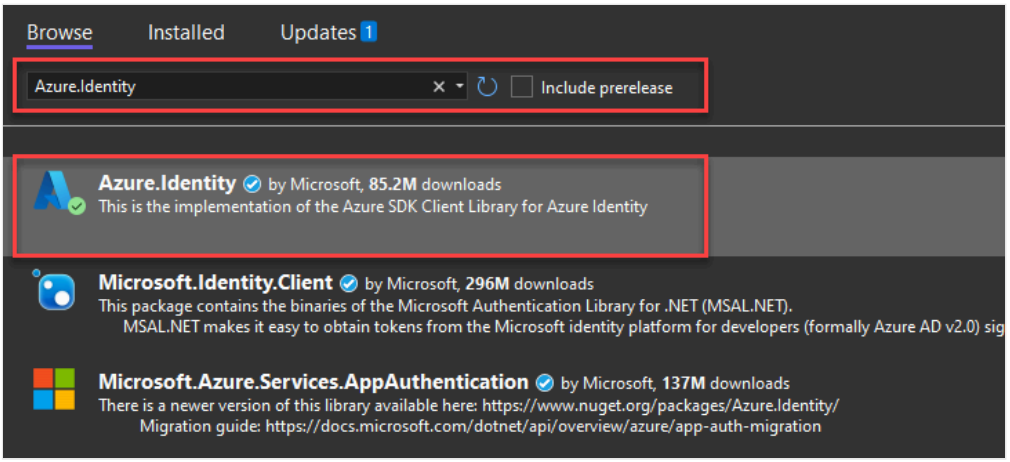
   ```
   az login
   ```

2. To use `DefaultAzureCredential`, add the **Azure.Identity** package to your application.

   Visual Studio

   a. In **Solution Explorer**, right-click the **Dependencies** node of your project. Select **Manage NuGet Packages**.

   b. In the resulting window, search for *Azure.Identity*. Select the appropriate result, and select **Install**.

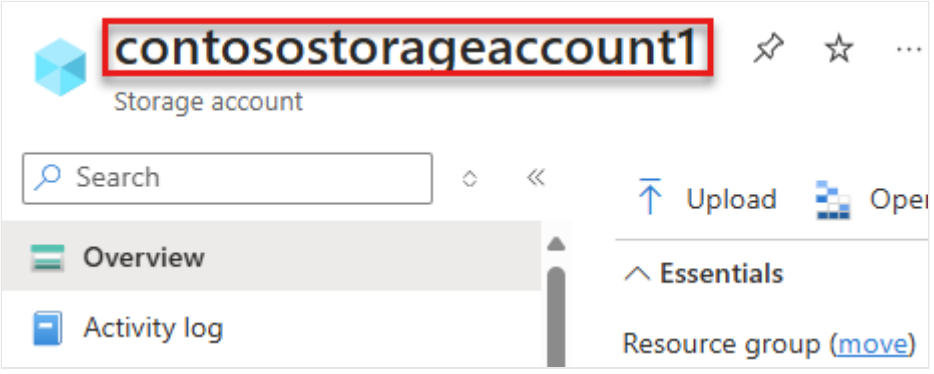


3. Update your *Program.cs* code to match the following example. When the code is run on your local workstation during development, it will use the developer credentials of the prioritized tool you're logged into to authenticate to Azure, such as the Azure CLI or Visual Studio.

C#

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using System;
using System.IO;
using Azure.Identity;

// TODO: Replace <storage-account-name> with your actual storage
account name
var blobServiceClient = new BlobServiceClient(
        new Uri("https://<storage-account-
name>.blob.core.windows.net"),
        new DefaultAzureCredential());
```

4. Make sure to update the storage account name in the URI of your `BlobServiceClient`. The storage account name can be found on the overview page of the Azure portal.



ⓘ Note

When deployed to Azure, this same code can be used to authorize requests to Azure Storage from an application running in Azure. However, you'll need to enable managed identity on your app in Azure. Then configure your storage account to allow that managed identity to connect. For detailed instructions on configuring this connection between Azure services, see the Auth from Azure-hosted apps tutorial.

Create a container

Create a new container in your storage account by calling the CreateBlobContainerAsync method on the `blobServiceClient` object. In this example, the code appends a GUID value to the container name to ensure that it's unique.

Add the following code to the end of the `Program.cs` file:

C#

```
// TODO: Replace <storage-account-name> with your actual storage account name
var blobServiceClient = new BlobServiceClient(
        new Uri("https://<storage-account-name>.blob.core.windows.net"),
        new DefaultAzureCredential());

//Create a unique name for the container
string containerName = "quickstartblobs" + Guid.NewGuid().ToString();

// Create the container and return a container client object
BlobContainerClient containerClient = await
blobServiceClient.CreateBlobContainerAsync(containerName);
```

To learn more about creating a container, and to explore more code samples, see Create a blob container with .NET.

ⓘ Important

Container names must be lowercase. For more information about naming containers and blobs, see Naming and Referencing Containers, Blobs, and Metadata.

Upload a blob to a container

Upload a blob to a container using UploadAsync. The example code creates a text file in the local *data* directory to upload to the container.

Add the following code to the end of the `Program.cs` file:

C#

```
// Create a local file in the ./data/ directory for uploading and
downloading
string localPath = "data";
Directory.CreateDirectory(localPath);
string fileName = "quickstart" + Guid.NewGuid().ToString() + ".txt";
string localFilePath = Path.Combine(localPath, fileName);

// Write text to the file
await File.WriteAllTextAsync(localFilePath, "Hello, World!");

// Get a reference to a blob
BlobClient blobClient = containerClient.GetBlobClient(fileName);

Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n",
blobClient.Uri);

// Upload data from the local file, overwrite the blob if it already exists
await blobClient.UploadAsync(localFilePath, true);
```

To learn more about uploading blobs, and to explore more code samples, see Upload a blob with .NET.

List blobs in a container

List the blobs in the container by calling the GetBlobsAsync method.

Add the following code to the end of the `Program.cs` file:

C#

```
Console.WriteLine("Listing blobs...");

// List all blobs in the container
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine("\t" + blobItem.Name);
}
```

To learn more about listing blobs, and to explore more code samples, see List blobs with .NET.

Download a blob

Download the blob we created earlier by calling the DownloadToAsync method. The example code appends the string "DOWNLOADED" to the file name so that you can see both files in local file system.

Add the following code to the end of the `Program.cs` file:

C#

```
// Download the blob to a local file
// Append the string "DOWNLOADED" before the .txt extension
// so you can compare the files in the data directory
string downloadFilePath = localFilePath.Replace(".txt", "DOWNLOADED.txt");

Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);

// Download the blob's contents and save it to a file
await blobClient.DownloadToAsync(downloadFilePath);
```

To learn more about downloading blobs, and to explore more code samples, see Download a blob with .NET.

Delete a container

The following code cleans up the resources the app created by deleting the container using DeleteAsync. The code example also deletes the local files created by the app.

The app pauses for user input by calling `Console.ReadLine` before it deletes the blob, container, and local files. This is a good chance to verify that the resources were created correctly, before they're deleted.

Add the following code to the end of the `Program.cs` file:

C#

```
// Clean up
Console.Write("Press any key to begin clean up");
Console.ReadLine();

Console.WriteLine("Deleting blob container...");
await containerClient.DeleteAsync();

Console.WriteLine("Deleting the local source and downloaded files...");
File.Delete(localFilePath);
File.Delete(downloadFilePath);
```

```
Console.WriteLine("Done");
```

To learn more about deleting a container, and to explore more code samples, see Delete and restore a blob container with .NET.

The completed code

After completing these steps, the code in your `Program.cs` file should now resemble the following:

Passwordless (Recommended)

C#

```
using Azure.Storage.Blobs;
using Azure.Storage.Blobs.Models;
using Azure.Identity;

// TODO: Replace <storage-account-name> with your actual storage account
name
var blobServiceClient = new BlobServiceClient(
        new Uri("https://<storage-account-name>.blob.core.windows.net"),
        new DefaultAzureCredential());

//Create a unique name for the container
string containerName = "quickstartblobs" + Guid.NewGuid().ToString();

// Create the container and return a container client object
BlobContainerClient containerClient = await
blobServiceClient.CreateBlobContainerAsync(containerName);

// Create a local file in the ./data/ directory for uploading and
downloading
string localPath = "data";
Directory.CreateDirectory(localPath);
string fileName = "quickstart" + Guid.NewGuid().ToString() + ".txt";
string localFilePath = Path.Combine(localPath, fileName);

// Write text to the file
await File.WriteAllTextAsync(localFilePath, "Hello, World!");

// Get a reference to a blob
BlobClient blobClient = containerClient.GetBlobClient(fileName);

Console.WriteLine("Uploading to Blob storage as blob:\n\t {0}\n",
blobClient.Uri);

// Upload data from the local file
await blobClient.UploadAsync(localFilePath, true);
```

```
Console.WriteLine("Listing blobs...");

// List all blobs in the container
await foreach (BlobItem blobItem in containerClient.GetBlobsAsync())
{
    Console.WriteLine("\t" + blobItem.Name);
}

// Download the blob to a local file
// Append the string "DOWNLOADED" before the .txt extension
// so you can compare the files in the data directory
string downloadFilePath = localFilePath.Replace(".txt",
"DOWNLOADED.txt");

Console.WriteLine("\nDownloading blob to\n\t{0}\n", downloadFilePath);

// Download the blob's contents and save it to a file
await blobClient.DownloadToAsync(downloadFilePath);

// Clean up
Console.Write("Press any key to begin clean up");
Console.ReadLine();

Console.WriteLine("Deleting blob container...");
await containerClient.DeleteAsync();

Console.WriteLine("Deleting the local source and downloaded files...");
File.Delete(localFilePath);
File.Delete(downloadFilePath);

Console.WriteLine("Done");
```

Run the code

This app creates a test file in your local *data* folder and uploads it to Blob storage. The example then lists the blobs in the container and downloads the file with a new name so that you can compare the old and new files.

If you're using Visual Studio, press F5 to build and run the code and interact with the console app. If you're using the .NET CLI, navigate to your application directory, then build and run the application.

Console

```
dotnet build
```

Console

```
dotnet run
```

The output of the app is similar to the following example (GUID values omitted for readability):

Output

```
Azure Blob Storage - .NET quickstart sample

Uploading to Blob storage as blob:

https://mystorageacct.blob.core.windows.net/quickstartblobsGUID/quickstartGU
ID.txt

Listing blobs...
        quickstartGUID.txt

Downloading blob to
        ./data/quickstartGUIDDOWNLOADED.txt

Press any key to begin clean up
Deleting blob container...
Deleting the local source and downloaded files...
Done
```

Before you begin the clean-up process, check your *data* folder for the two files. You can open them and observe that they're identical.

Clean up resources

After you verify the files and finish testing, press the **Enter** key to delete the test files along with the container you created in the storage account. You can also use Azure CLI to delete resources.

Next step

Azure Storage samples and developer guides for .NET

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Azure Queue storage documentation

Azure Queue storage is a service for storing large numbers of messages. Access messages from anywhere in the world via HTTP or HTTPS.

About Azure queues

OVERVIEW

What are Azure queues?

Get started

QUICKSTART

Create a queue and add a message with the Azure portal

Develop with queues

QUICKSTART

Azure Queue storage for .NET

Azure Queue storage for Java

Azure Queue storage for Python

Azure Queue storage for JavaScript

HOW-TO GUIDE

Get started with Azure Queue storage using .NET

How to use Queue storage from Java

How to use Queue storage from Node.js

How to use Queue Storage from C++

How to use Queue storage from Python

Manage queues

HOW-TO GUIDE

Perform Azure Queue storage operations with Azure PowerShell

Tutorial

TUTORIAL

Work with Azure storage queues (.NET)

What is Azure Table storage?

Article • 11/29/2022

> **Tip**
>
> The content in this article applies to the original Azure Table storage. However, the same concepts apply to the newer Azure Cosmos DB for Table, which offers higher performance and availability, global distribution, and automatic secondary indexes. It is also available in a consumption-based **serverless** mode. There are some **feature differences** between Table API in Azure Cosmos DB and Azure Table storage. For more information, see **Azure Cosmos DB for Table**. For ease of development, we now provide a unified **Azure Tables SDK** that can be used to target both Azure Table storage and Azure Cosmos DB for Table.

Azure Table storage is a service that stores non-relational structured data (also known as structured NoSQL data) in the cloud, providing a key/attribute store with a schemaless design. Because Table storage is schemaless, it's easy to adapt your data as the needs of your application evolve. Access to Table storage data is fast and cost-effective for many types of applications, and is typically lower in cost than traditional SQL for similar volumes of data.

You can use Table storage to store flexible datasets like user data for web applications, address books, device information, or other types of metadata your service requires. You can store any number of entities in a table, and a storage account may contain any number of tables, up to the capacity limit of the storage account.

What is Table storage

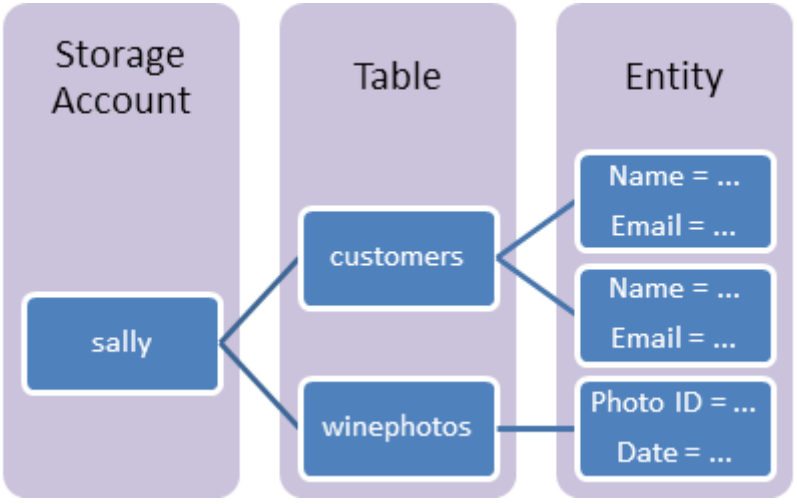
Azure Table storage stores large amounts of structured data. The service is a NoSQL datastore which accepts authenticated calls from inside and outside the Azure cloud. Azure tables are ideal for storing structured, non-relational data. Common uses of Table storage include:

- Storing TBs of structured data capable of serving web scale applications
- Storing datasets that don't require complex joins, foreign keys, or stored procedures and can be denormalized for fast access
- Quickly querying data using a clustered index
- Accessing data using the OData protocol and LINQ queries with WCF Data Service .NET Libraries

You can use Table storage to store and query huge sets of structured, non-relational data, and your tables will scale as demand increases.

Table storage concepts

Table storage contains the following components:



- **URL format:** Azure Table Storage accounts use this format: `http://<storage account>.table.core.windows.net/<table>`

  You can address Azure tables directly using this address with the OData protocol. For more information, see OData.org.

- **Accounts:** All access to Azure Storage is done through a storage account. For more information about storage accounts, see Storage account overview.

  All access to Azure Cosmos DB is done through an Azure Cosmos DB for Table account. For more information, see Create an Azure Cosmos DB for Table account.

- **Table**: A table is a collection of entities. Tables don't enforce a schema on entities, which means a single table can contain entities that have different sets of properties.

- **Entity**: An entity is a set of properties, similar to a database row. An entity in Azure Storage can be up to 1MB in size. An entity in Azure Cosmos DB can be up to 2MB in size.

- **Properties**: A property is a name-value pair. Each entity can include up to 252 properties to store data. Each entity also has three system properties that specify a partition key, a row key, and a timestamp. Entities with the same partition key can be queried more quickly, and inserted/updated in atomic operations. An entity's row key is its unique identifier within a partition.

For details about naming tables and properties, see Understanding the Table Service Data Model.

Next steps

- Microsoft Azure Storage Explorer is a free, standalone app from Microsoft that enables you to work visually with Azure Storage data on Windows, macOS, and Linux.
- Get started with Azure Table Storage in .NET
- View the Table service reference documentation for complete details about available APIs:
  - Storage Client Library for .NET reference
  - REST API reference

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Host and deploy ASP.NET Core

Article • 09/18/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

In general, to deploy an ASP.NET Core app to a hosting environment:

- Deploy the published app to a folder on the hosting server.
- Set up a process manager that starts the app when requests arrive and restarts the app after it crashes or the server reboots.
- For configuration of a reverse proxy, set up a reverse proxy to forward requests to the app.

For Blazor host and deploy guidance, which adds to or supersedes the guidance in this node, see Host and deploy ASP.NET Core Blazor.

Publish to a folder

The dotnet publish command compiles app code and copies the files required to run the app into a *publish* folder. When deploying from Visual Studio, the `dotnet publish` step occurs automatically before the files are copied to the deployment destination.

Run the published app locally

To run the published app locally, run `dotnet <ApplicationName>.dll` from the *publish* folder.

Publish settings files

`*.json` files are published by default. To publish other settings files, specify them in an `<ItemGroup><Content Include= ... />` element in the project file. The following example publishes XML files:

XML

```
<ItemGroup>
  <Content Include="**\*.xml" Exclude="bin\**\*;obj\**\*"
```

```
    CopyToOutputDirectory="PreserveNewest" />
</ItemGroup>
```

Folder contents

The *publish* folder contains one or more app assembly files, dependencies, and optionally the .NET runtime.

A .NET Core app can be published as *self-contained deployment* or *framework-dependent deployment*. If the app is self-contained, the assembly files that contain the .NET runtime are included in the *publish* folder. If the app is framework-dependent, the .NET runtime files aren't included because the app has a reference to a version of .NET that's installed on the server. The default deployment model is framework-dependent. For more information, see .NET Core application deployment.

In addition to *.exe* and *.dll* files, the *publish* folder for an ASP.NET Core app typically contains configuration files, static assets, and MVC views. For more information, see ASP.NET Core directory structure.

Set up a process manager

An ASP.NET Core app is a console app that must be started when a server boots and restarted if it crashes. To automate starts and restarts, a process manager is required. The most common process managers for ASP.NET Core are:

- Linux
  - Nginx
- Windows
  - IIS
  - Windows Service

Set up a reverse proxy

If the app uses the Kestrel server, Nginx, or IIS can be used as a reverse proxy server. A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.

Either configuration—with or without a reverse proxy server—is a supported hosting configuration. For more information, see When to use Kestrel with a reverse proxy.

Proxy server and load balancer scenarios

Additional configuration might be required for apps hosted behind proxy servers and load balancers. Without additional configuration, an app might not have access to the scheme (HTTP/HTTPS) and the remote IP address where a request originated. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Use Visual Studio and MSBuild to automate deployments

Deployment often requires additional tasks besides copying the output from dotnet publish to a server. For example, extra files might be required or excluded from the *publish* folder. Visual Studio uses MSBuild for web deployment, and MSBuild can be customized to do many other tasks during deployment. For more information, see Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment and the Using MSBuild and Team Foundation Build book.

By using the Publish Web feature apps can be deployed directly from Visual Studio to the Azure App Service. Azure DevOps Services supports continuous deployment to Azure App Service. For more information, see DevOps for ASP.NET Core Developers.

Publish to Azure

See Publish an ASP.NET Core app to Azure with Visual Studio for instructions on how to publish an app to Azure using Visual Studio. An additional example is provided by Create an ASP.NET Core web app in Azure.

Publish with MSDeploy on Windows

See Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment for instructions on how to publish an app with a Visual Studio publish profile, including from a Windows command prompt using the dotnet msbuild command.

Internet Information Services (IIS)

For deployments to Internet Information Services (IIS) with configuration provided by the *web.config* file, see the articles under Host ASP.NET Core on Windows with IIS.

Host in a web farm

For information on configuration for hosting ASP.NET Core apps in a web farm environment (for example, deployment of multiple instances of your app for scalability), see Host ASP.NET Core in a web farm.

Host on Docker

For more information, see Host ASP.NET Core in Docker containers.

Perform health checks

Use Health Check Middleware to perform health checks on an app and its dependencies. For more information, see Health checks in ASP.NET Core.

Additional resources

- .NET application publishing overview
- Troubleshoot and debug ASP.NET Core projects
- ASP.NET Hosting

Deploy ASP.NET Core apps to Azure App Service

Article • 06/21/2024

Azure App Service is a Microsoft cloud computing platform service for hosting web apps, including ASP.NET Core.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Useful resources

App Service Documentation is the home for Azure Apps documentation, tutorials, samples, how-to guides, and other resources. Two notable tutorials that pertain to hosting ASP.NET Core apps are:

Create an ASP.NET Core web app in Azure
Use Visual Studio to create and deploy an ASP.NET Core web app to Azure App Service on Windows.

Create an ASP.NET Core app in App Service on Linux
Use the command line to create and deploy an ASP.NET Core web app to Azure App Service on Linux.

See the ASP.NET Core on App Service Dashboard for the version of ASP.NET Core available on Azure App service.

Subscribe to the App Service Announcements repository and monitor the issues. The App Service team regularly posts announcements and scenarios arriving in App Service.

The following articles are available in ASP.NET Core documentation:

Publish an ASP.NET Core app to Azure with Visual Studio
Learn how to publish an ASP.NET Core app to Azure App Service using Visual Studio.

Create your first pipeline
Set up a CI build for an ASP.NET Core app, then create a continuous deployment release to Azure App Service.

Azure Web App sandbox
Discover Azure App Service runtime execution limitations enforced by the Azure Apps platform.

Troubleshoot and debug ASP.NET Core projects
Understand and troubleshoot warnings and errors with ASP.NET Core projects.

Application configuration

Platform

The platform architecture (x86/x64) of an App Services app is set in the app's settings in the Azure Portal for apps that are hosted on an A-series compute (Basic) or higher hosting tier. Confirm that the app's publish settings (for example, in the Visual Studio publish profile (.pubxml)) match the setting in the app's service configuration in the Azure Portal.

ASP.NET Core apps can be published framework-dependent because the runtimes for 64-bit (x64) and 32-bit (x86) apps are present on Azure App Service. The .NET Core SDK available on App Service is 32-bit, but you can deploy 64-bit apps built locally using the Kudu console or the publish process in Visual Studio. For more information, see the Publish and deploy the app section.

For more information on .NET Core framework components and distribution methods, such as information on the .NET Core runtime and the .NET Core SDK, see About .NET Core: Composition.

Packages

Include the following NuGet packages to provide automatic logging features for apps deployed to Azure App Service:

- Microsoft.AspNetCore.AzureAppServices.HostingStartup uses IHostingStartup to provide ASP.NET Core logging integration with Azure App Service. The added logging features are provided by the `Microsoft.AspNetCore.AzureAppServicesIntegration` package.
- Microsoft.AspNetCore.AzureAppServicesIntegration executes AddAzureWebAppDiagnostics to add Azure App Service diagnostics logging providers in the `Microsoft.Extensions.Logging.AzureAppServices` package.
- Microsoft.Extensions.Logging.AzureAppServices provides logger implementations to support Azure App Service diagnostics logs and log streaming

features.

The preceding packages must be explicitly referenced in the app's project file.

Override app configuration using the Azure Portal

App settings in the Azure Portal permit you to set environment variables for the app. Environment variables can be consumed by the Environment Variables Configuration Provider.

When an app setting is created or modified in the Azure Portal and the **Save** button is selected, the Azure App is restarted. The environment variable is available to the app after the service restarts.

Environment variables are loaded into the app's configuration when CreateBuilder is called to build the host. For more information, see the Environment Variables Configuration Provider.

Proxy server and load balancer scenarios

The IIS Integration Middleware, which configures Forwarded Headers Middleware when hosting out-of-process, and the ASP.NET Core Module are configured to forward the scheme (HTTP/HTTPS) and the remote IP address where the request originated. Additional configuration might be required for apps hosted behind additional proxy servers and load balancers. For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Monitoring and logging

ASP.NET Core apps deployed to App Service automatically receive an App Service extension, **ASP.NET Core Logging Integration**. The extension enables logging integration for ASP.NET Core apps on Azure App Service.

For monitoring, logging, and troubleshooting information, see the following articles:

Monitor apps in Azure App Service
Learn how to review quotas and metrics for apps and App Service plans.

Enable diagnostics logging for apps in Azure App Service
Discover how to enable and access diagnostic logging for HTTP status codes, failed

requests, and web server activity.

Handle errors in ASP.NET Core
Understand common approaches to handling errors in ASP.NET Core apps.

Troubleshoot ASP.NET Core on Azure App Service and IIS
Learn how to diagnose issues with Azure App Service deployments with ASP.NET Core apps.

Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
See the common deployment configuration errors for apps hosted by Azure App Service/IIS with troubleshooting advice.

Data Protection key ring and deployment slots

Data Protection keys are persisted to the *%HOME%\ASP.NET\DataProtection-Keys* folder. This folder is backed by network storage and is synchronized across all machines hosting the app. Keys aren't protected at rest. This folder supplies the key ring to all instances of an app in a single deployment slot. Separate deployment slots, such as Staging and Production, don't share a key ring.

When swapping between deployment slots, any system using data protection won't be able to decrypt stored data using the key ring inside the previous slot. ASP.NET Cookie Middleware uses data protection to protect its cookies. This leads to users being signed out of an app that uses the standard ASP.NET Cookie Middleware. For a slot-independent key ring solution, use an external key ring provider, such as:

- Azure Blob Storage
- Azure Key Vault
- SQL store
- Redis cache

For more information, see Key storage providers in ASP.NET Core.

Deploy an ASP.NET Core app that uses a .NET Core preview

To deploy an app that uses a preview release of .NET Core, see the following resources. These approaches are also used when the runtime is available but the SDK hasn't been installed on Azure App Service.

- Specify the .NET Core SDK Version using Azure Pipelines

- Deploy a self-contained preview app
- Use Docker with Web Apps for containers
- Install the preview site extension

See the ASP.NET Core on App Service Dashboard for the version of ASP.NET Core available on Azure App service.

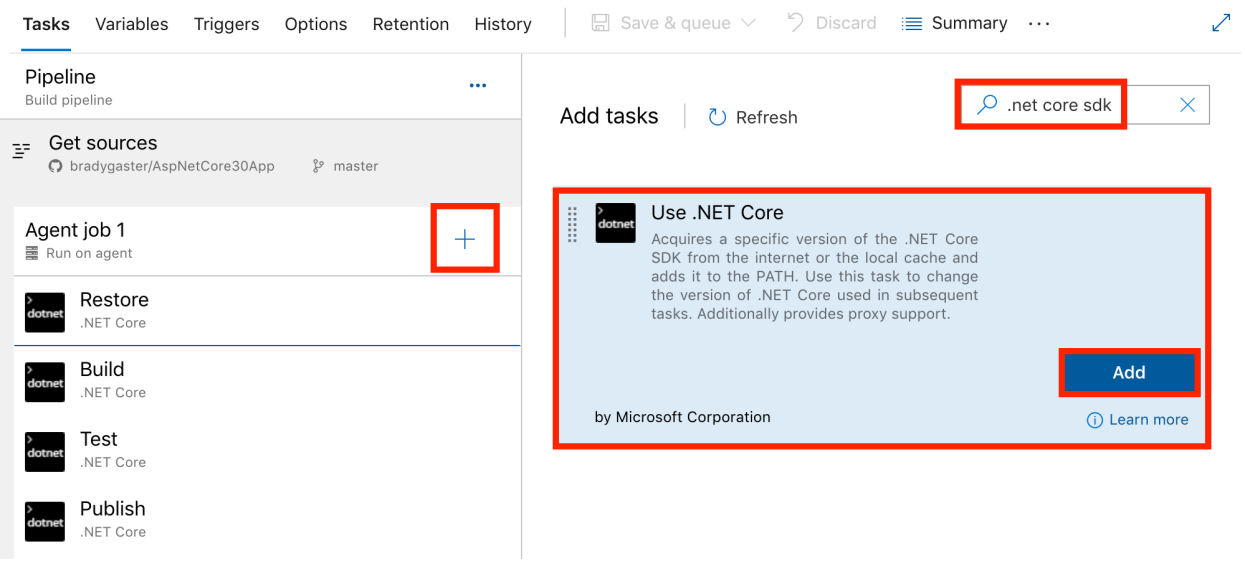
See Select the .NET Core version to use for information on selecting the version of the .NET SDK for self-contained deployments.

Specify the .NET Core SDK Version using Azure Pipelines

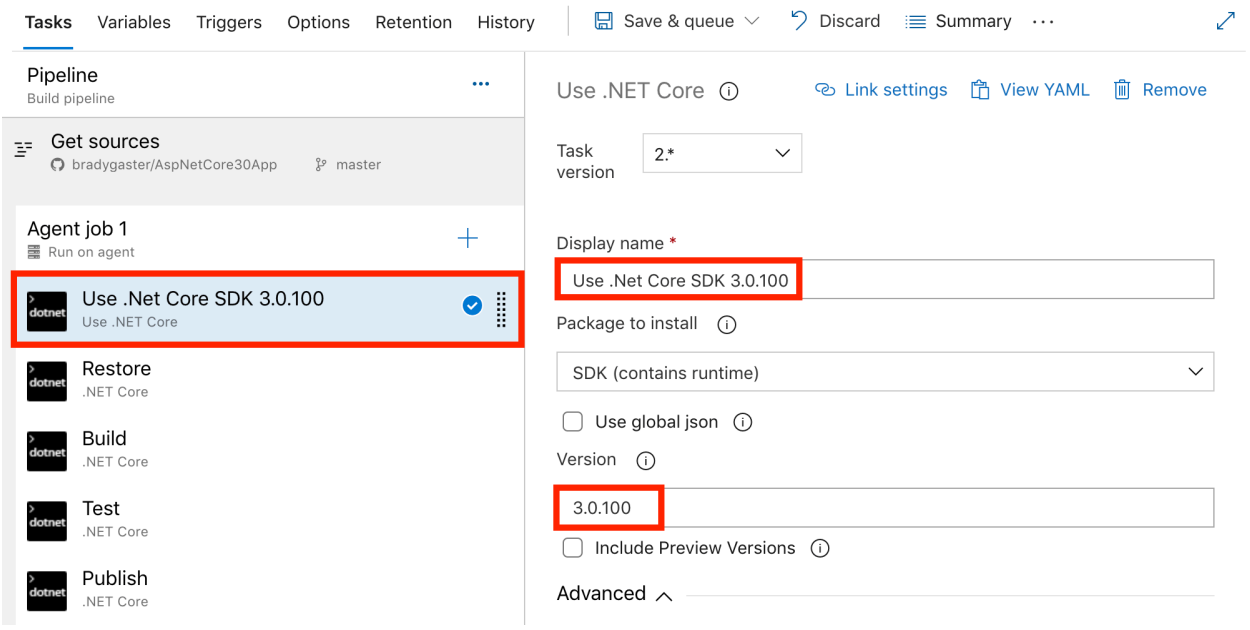
Use Azure App Service CI/CD scenarios to set up a continuous integration build with Azure DevOps. After the Azure DevOps build is created, optionally configure the build to use a specific SDK version.

Specify the .NET Core SDK version

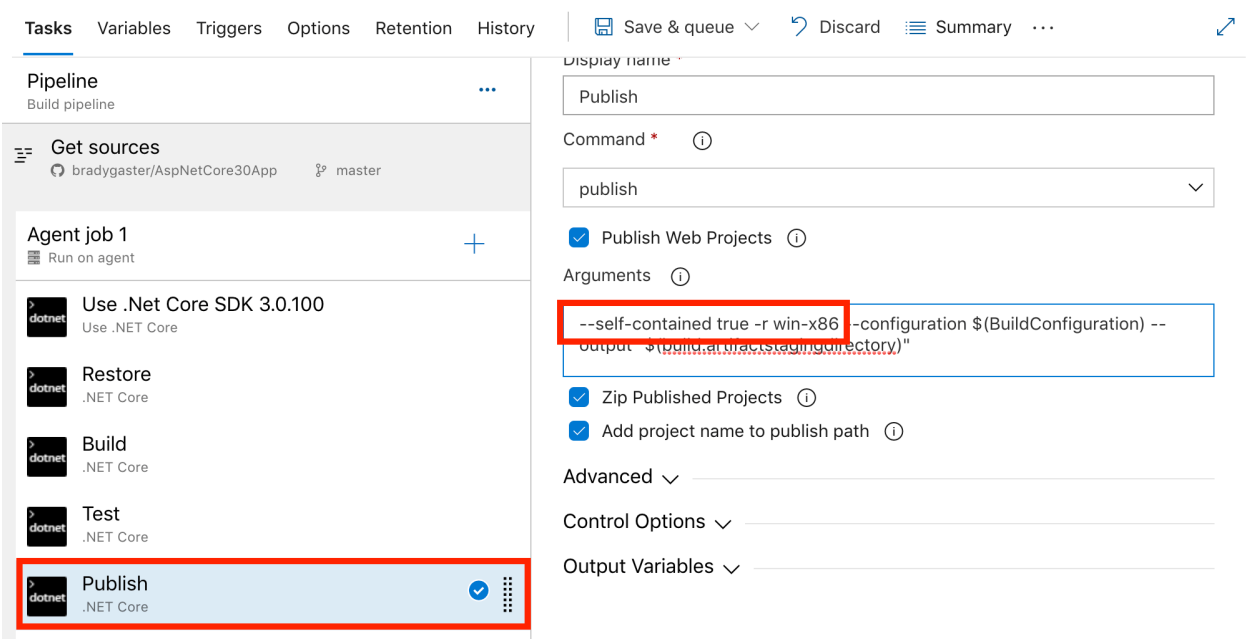
When using the App Service deployment center to create an Azure DevOps build, the default build pipeline includes steps for `Restore`, `Build`, `Test`, and `Publish`. To specify the SDK version, select the **Add (+)** button in the Agent job list to add a new step. Search for **.NET Core SDK** in the search bar.



Move the step into the first position in the build so that the steps following it use the specified version of the .NET Core SDK. Specify the version of the .NET Core SDK. In this example, the SDK is set to `3.0.100`.



To publish a self-contained deployment (SCD), configure SCD in the `Publish` step and provide the Runtime Identifier (RID).



Deploy a self-contained preview app

A self-contained deployment (SCD) that targets a preview runtime carries the preview runtime in the deployment.

When deploying a self-contained app:

- The site in Azure App Service doesn't require the preview site extension.
- The app must be published following a different approach than when publishing for a framework-dependent deployment (FDD).

Follow the guidance in the Deploy the app self-contained section.

Use Docker with Web Apps for containers

The Docker Hub at `https://hub.docker.com/_/microsoft-dotnet` contains the latest preview Docker images. The images can be used as a base image. Use the image and deploy to Web Apps for Containers normally.

Install the preview site extension

If a problem occurs using the preview site extension, open an dotnet/AspNetCore issue.

1. From the Azure Portal, navigate to the App Service.
2. Select the web app.
3. Type "ex" in the search box to filter for "Extensions" or scroll down the list of management tools.
4. Select **Extensions**.
5. Select **Add**.
6. Select the **ASP.NET Core {X.Y} ({x64|x86}) Runtime** extension from the list, where `{X.Y}` is the ASP.NET Core preview version and `{x64|x86}` specifies the platform.
7. Select **OK** to accept the legal terms.
8. Select **OK** to install the extension.

When the operation completes, the latest .NET Core preview is installed. Verify the installation:

1. Select **Advanced Tools**.
2. Select **Go** in **Advanced Tools**.
3. Select the **Debug console** > **PowerShell** menu item.
4. At the PowerShell prompt, execute the following command. Substitute the ASP.NET Core runtime version for `{X.Y}` and the platform for `{PLATFORM}` in the command:

   PowerShell

   ```
   Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.{PLATFORM}\
   ```

   The command returns `True` when the x64 preview runtime is installed.

> **Note**
>
> The platform architecture (x86/x64) of an App Services app is set in the app's settings in the Azure Portal for apps that are hosted on an A-series compute (Basic) or higher hosting tier. Confirm that the app's publish settings (for example, in the Visual Studio publish profile (.pubxml)) match the setting in the app's service configuration in the Azure portal.
>
> If the app is run in in-process mode and the platform architecture is configured for 64-bit (x64), the ASP.NET Core Module uses the 64-bit preview runtime, if present. Install the **ASP.NET Core {X.Y} (x64) Runtime** extension using the Azure Portal.
>
> After installing the x64 preview runtime, run the following command in the Azure Kudu PowerShell command window to verify the installation. Substitute the ASP.NET Core runtime version for `{X.Y}` in the following command:
>
> PowerShell
>
> ```
> Test-Path D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64\
> ```
>
> The command returns `True` when the x64 preview runtime is installed.

Use the preview site extension with an ARM template

If an ARM template is used to create and deploy apps, the `Microsoft.Web/sites/siteextensions` resource type can be used to add the site extension to a web app. In the following example, the ASP.NET Core 5.0 (x64) Runtime site extension (`AspNetCoreRuntime.5.0.x64`) is added to the app:

JSON

```
{
    ...
    "parameters": {
        "site_name": {
            "defaultValue": "{SITE NAME}",
            "type": "String"
        },
        ...
    },
    ...
    "resources": [
        ...
        {
            "type": "Microsoft.Web/sites/siteextensions",
```

```
            "apiVersion": "2018-11-01",
            "name": "[concat(parameters('site_name'),
'/AspNetCoreRuntime.5.0.x64')]",
            "location": "[resourceGroup().location]",
            "dependsOn": [
                "[resourceId('Microsoft.Web/sites',
parameters('site_name'))]"
            ]
        }
    ]
}
```

For the placeholder `{SITE NAME}`, use the app's name in Azure App Service (for example, `contoso`).

Publish and deploy the app

For a 64-bit deployment:

- Use a 64-bit .NET Core SDK to build a 64-bit app.
- Set the **Platform** to **64 Bit** in the App Service's **Configuration** > **General settings**. The app must use a Basic or higher service plan to enable the choice of platform bitness.

Deploy the app framework-dependent

Apps published as framework-dependent are cross-platform and don't include the .NET runtime in the deployment. Azure App Service includes the .NET runtime.

Visual Studio

1. Right-click the project in **Solution Explorer** and select **Publish**. Alternatively, select **Build** > **Publish {Application Name}** from the Visual Studio toolbar.
2. In the **Publish** dialog, select **Azure** > **Next**.
3. Select the Azure service.
4. Select **Advanced**. The **Publish** dialog opens.
5. Select a Resource group and Hosting plan, or create new ones.
6. Select **Finish**.
7. In the **Publish** page:

    - For **Configuration**, select the pen icon **Edit Configuration**:
        - Confirm that the **Release** configuration is selected.

    - In the **Deployment Mode** drop-down list, select **Framework-Dependent**.
    - In the **Target Runtime** drop-down list, select the desired runtime. The default is `win-x86`.
- To remove additional files upon deployment, open **File Publish Options** and select the checkbox to remove additional files at the destination.
- Select **Save**.
- Select **Publish**.

Deploy the app self-contained

Publishing an app as self-contained produces a platform-specific executable. The output publishing folder contains all components of the app, including the .NET libraries and target runtime. For more information, see [Publish self-contained]/dotnet/core/deploying/#publish-self-contained). Use Visual Studio or the .NET CLI for a self-contained deployment (SCD).

Visual Studio

1. Right-click the project in **Solution Explorer** and select **Publish**. Alternatively, select **Build** > **Publish {Application Name}** from the Visual Studio toolbar.
2. In the **Publish** dialog, select **Azure** > **Next**.
3. Select the Azure service.
4. Select **Advanced**. The **Publish** dialog opens.
5. Select a Resource group and Hosting plan, or create new ones.
6. Select **Finish**.
7. In the **Publish** page:

    - For **Configuration**, select the pen icon **Edit Configuration**:
        - Confirm that the **Release** configuration is selected.
        - In the **Deployment Mode** drop-down list, select **Self-Contained**.
        - In the **Target Runtime** drop-down list, select the desired runtime. The default is `win-x86`.
    - To remove additional files upon deployment, open **File Publish Options** and select the checkbox to remove additional files at the destination.
    - Select **Save**.
    - Select **Publish**.

Protocol settings (HTTPS)

Secure protocol bindings allow specifying a certificate to use when responding to requests over HTTPS. Binding requires a valid private certificate (*.pfx*) issued for the specific hostname. For more information, see Tutorial: Bind an existing custom SSL certificate to Azure App Service.

Transform web.config

If you need to transform *web.config* on publish (for example, set environment variables based on the configuration, profile, or environment), see Transform web.config.

Additional resources

- App Service overview
- Azure App Service diagnostics overview
- Host ASP.NET Core in a web farm
- Tutorial: Connect to SQL Database from .NET App Service without secrets using a managed identity

Azure App Service on Windows Server uses Internet Information Services (IIS). Kestrel and YARP on the front end provides the load balancer. The following topics pertain to the underlying IIS technology:

- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module (ANCM) for IIS
- IIS modules with ASP.NET Core
- Windows Server - IT administrator content for current and previous releases

Publish an ASP.NET Core app to Azure with Visual Studio

Article • 11/01/2023

> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see **Deploy ASP.NET Core preview release to Azure App Service**.

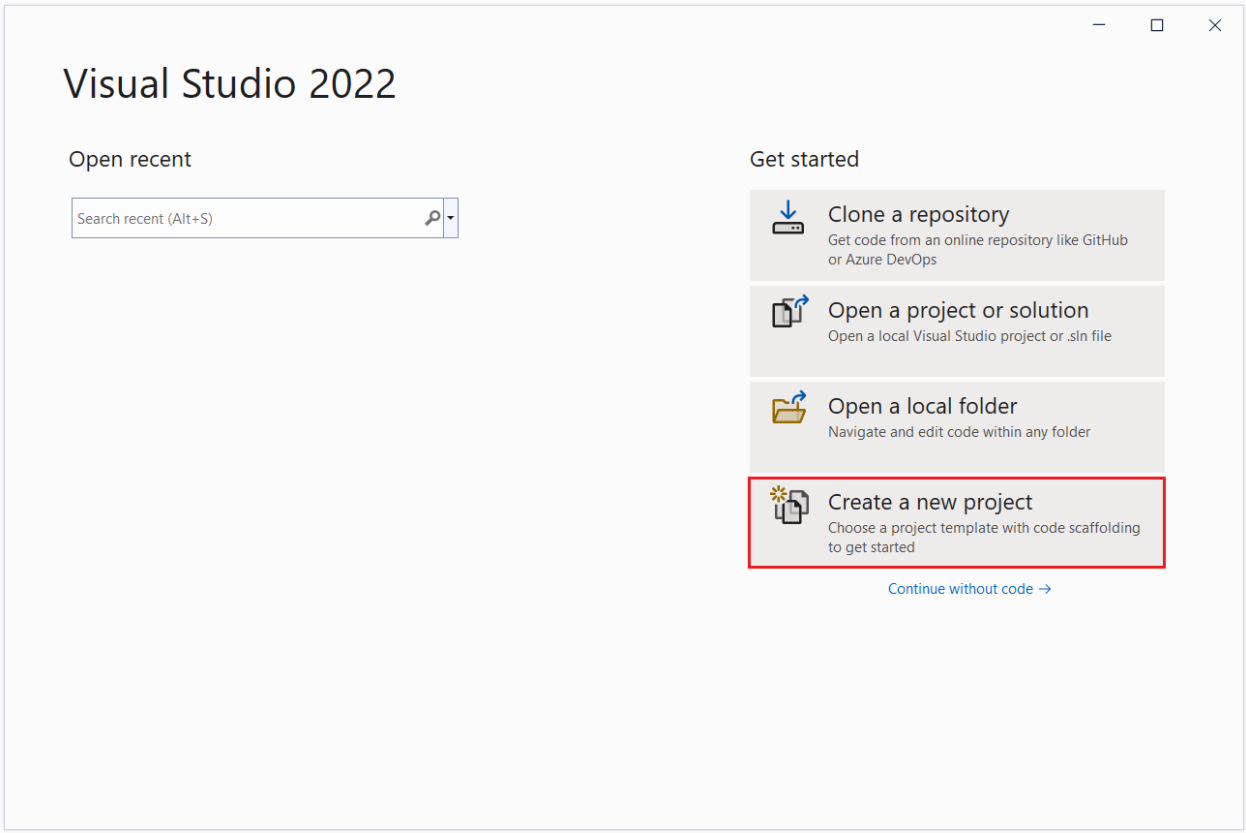
To troubleshoot an App Service deployment issue, see Troubleshoot ASP.NET Core on Azure App Service and IIS.

Set up

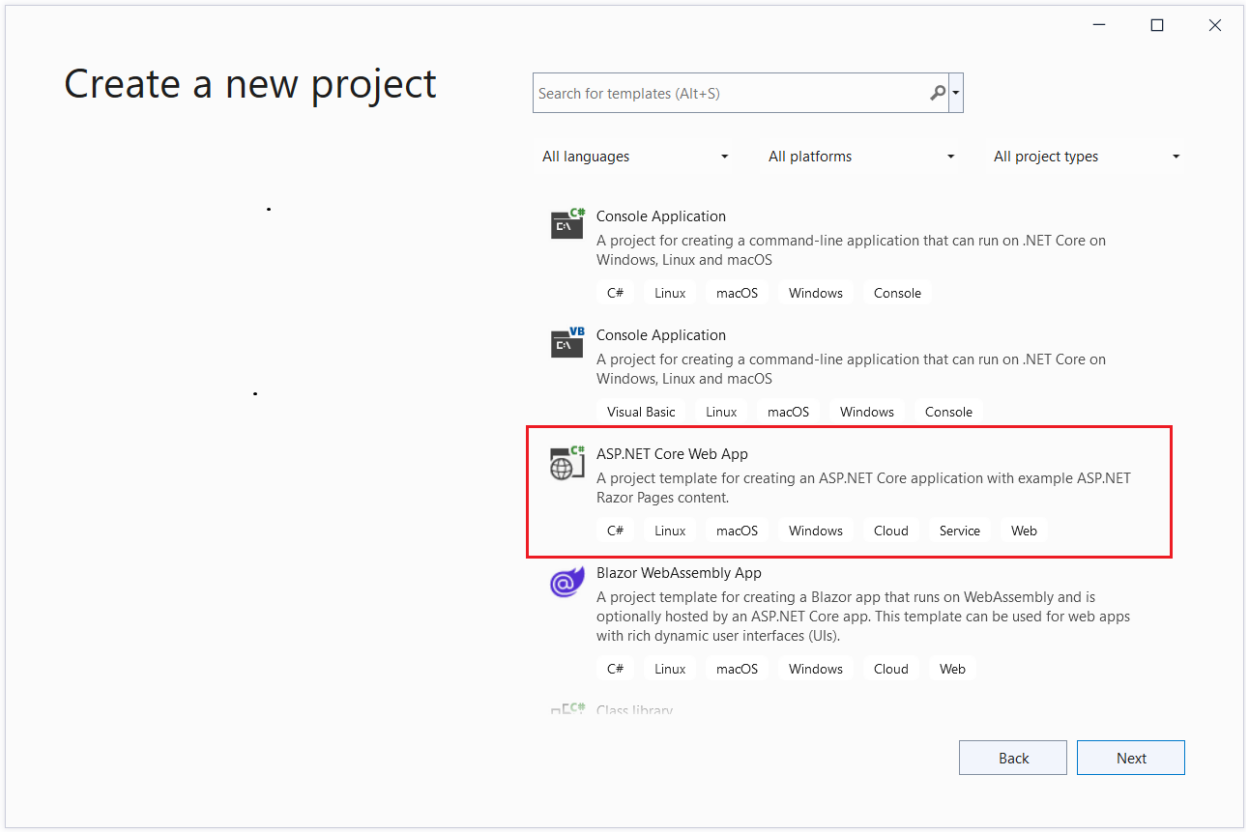
Open a free Azure account if you don't have one.

Create a web app

Start Visual Studio 2022 and select **Create a new project**.



In the **Create a new project** dialog, select **ASP.NET Core Web App**, and then select **Next**.

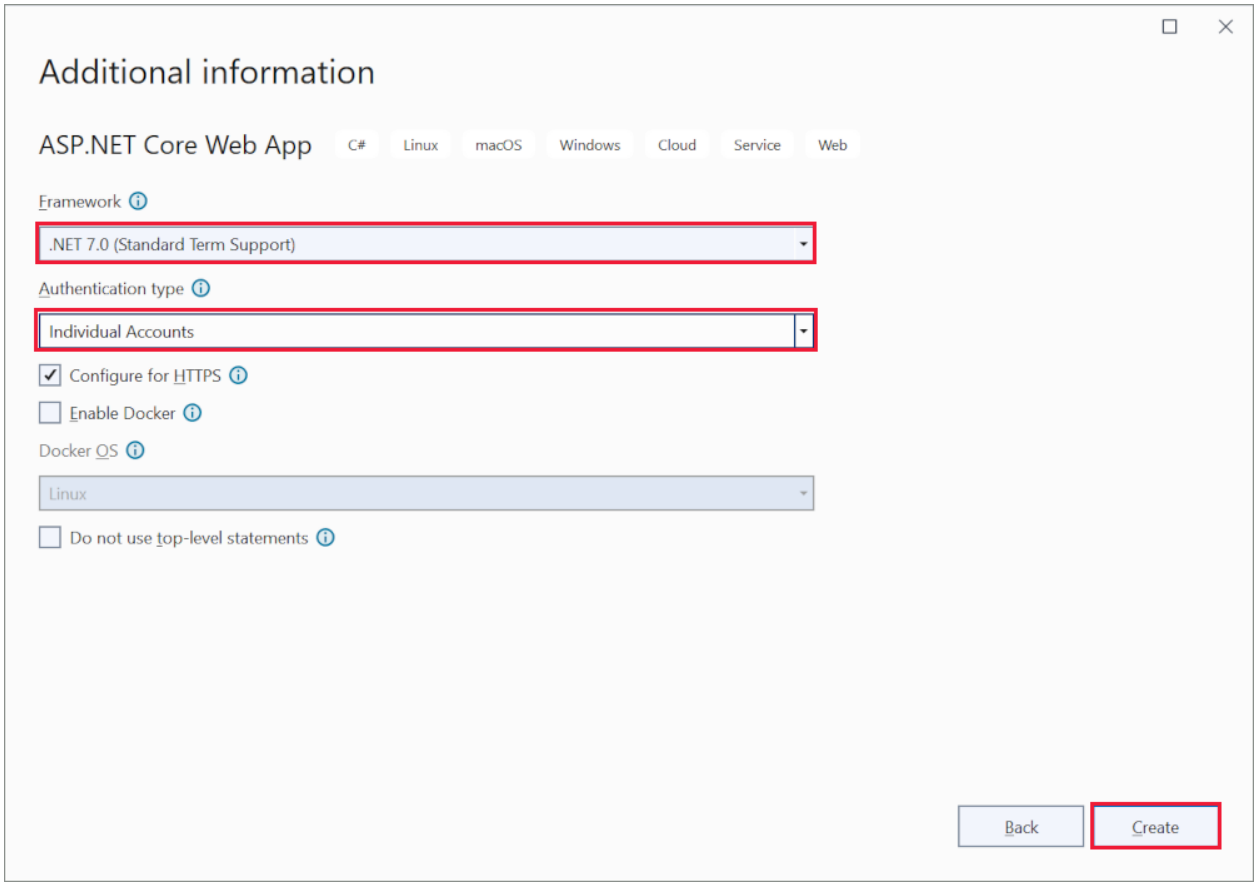


In the **Configure your new project** dialog, name your project, and then select **Next**.

In the **Additional information** dialog:

- In the **Framework** input, select **.NET 7.0 (Standard Term Support)**.

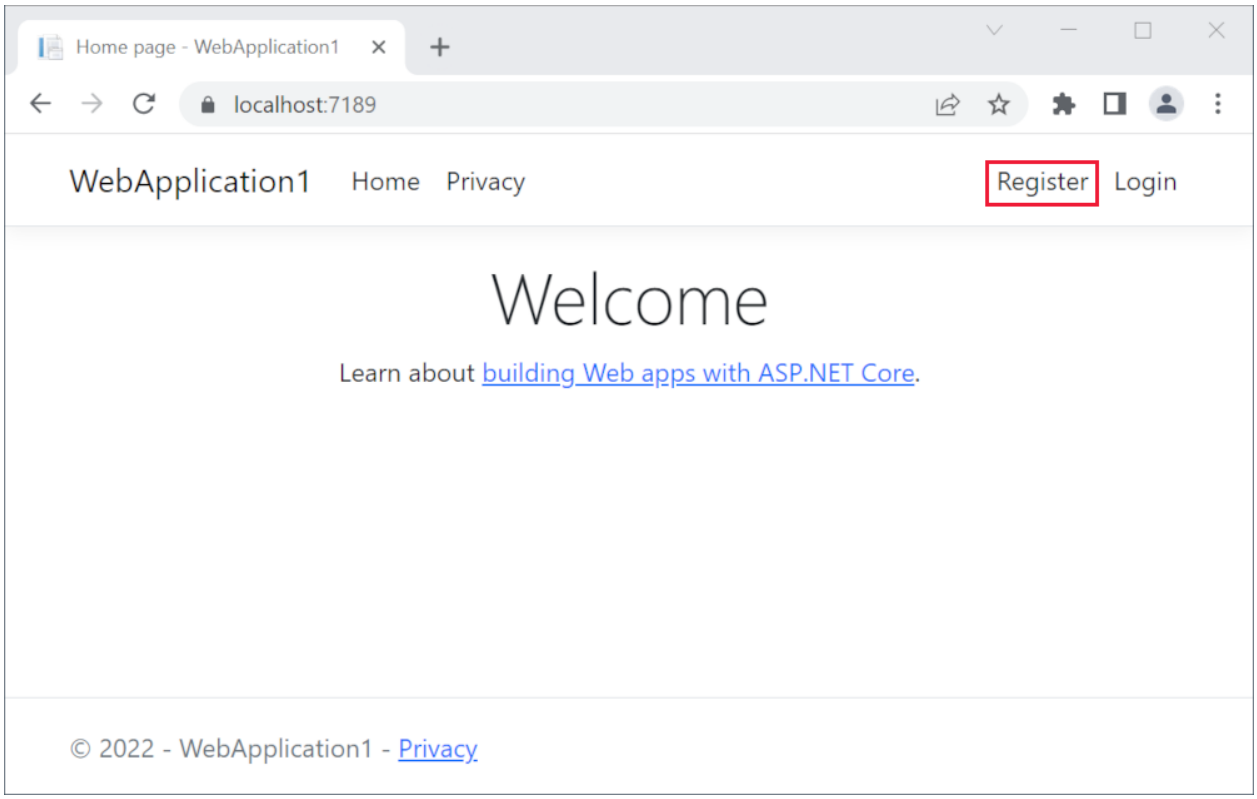
- In the **Authentication type** input, select **Individual Accounts**, and then select **Create**.



Visual Studio creates the solution.

Run the app

- Press F5 to run the project.

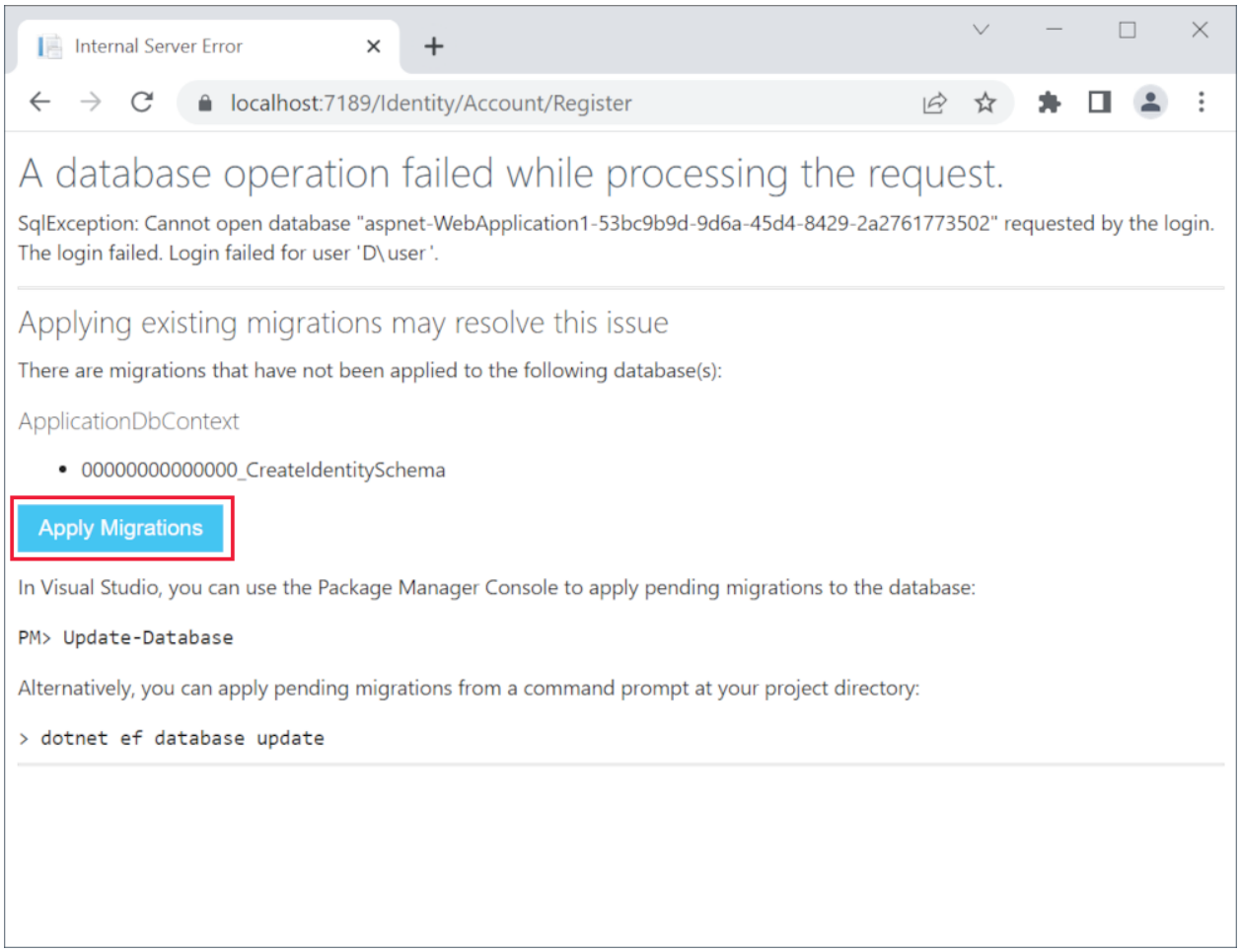


Register a user

- Select **Register** and register a new user. You can use a fictitious email address. When you submit, the page displays the following error:

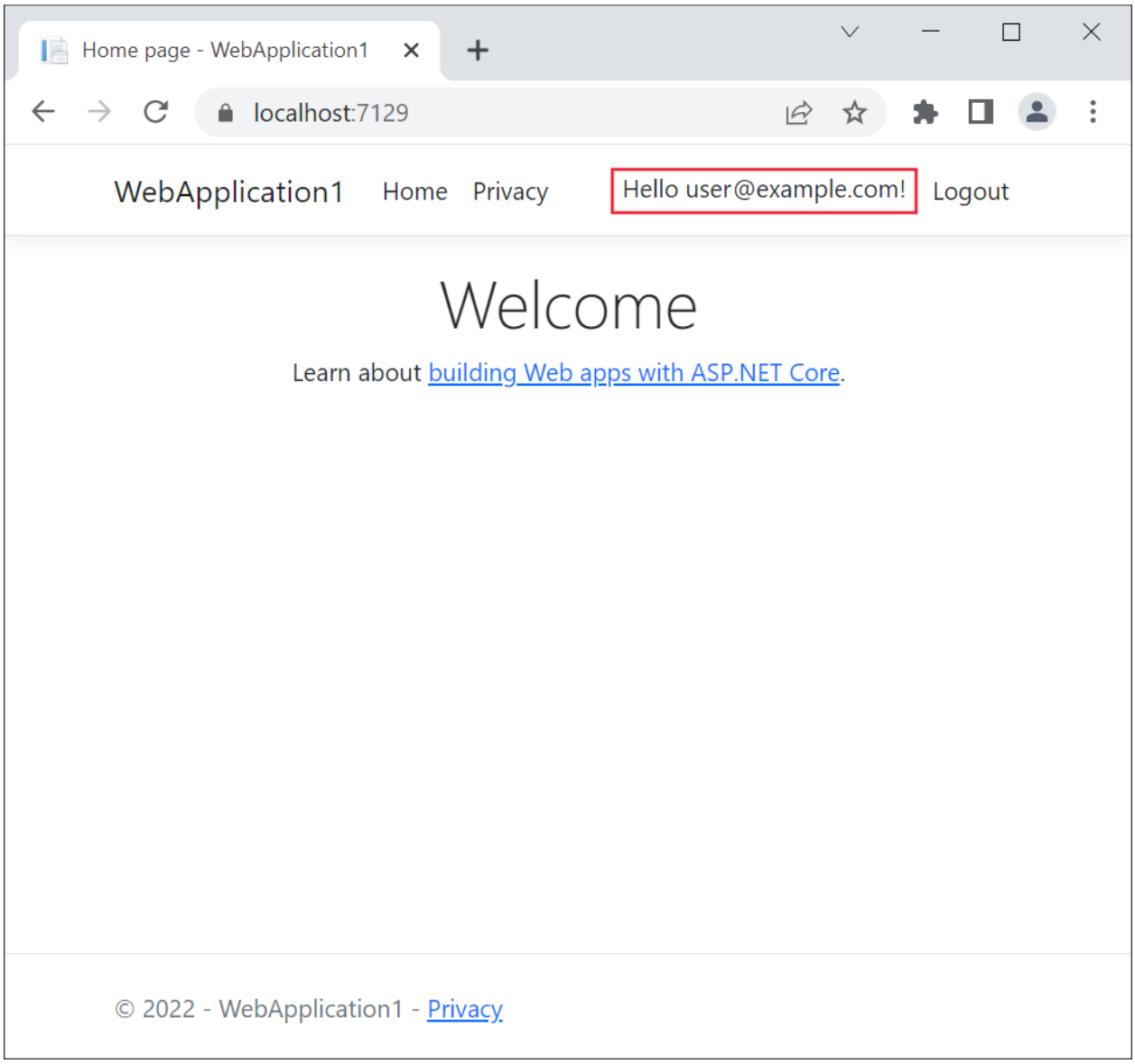
  "A database operation failed while processing the request. Applying existing migrations may resolve this issue"

- Select **Apply Migrations** and, once the page updates, refresh the page.



- A **Register confirmation** page is displayed. Select **Click here to confirm your account**.
- A **Confirm email** page is displayed.
- Log in as the new user.

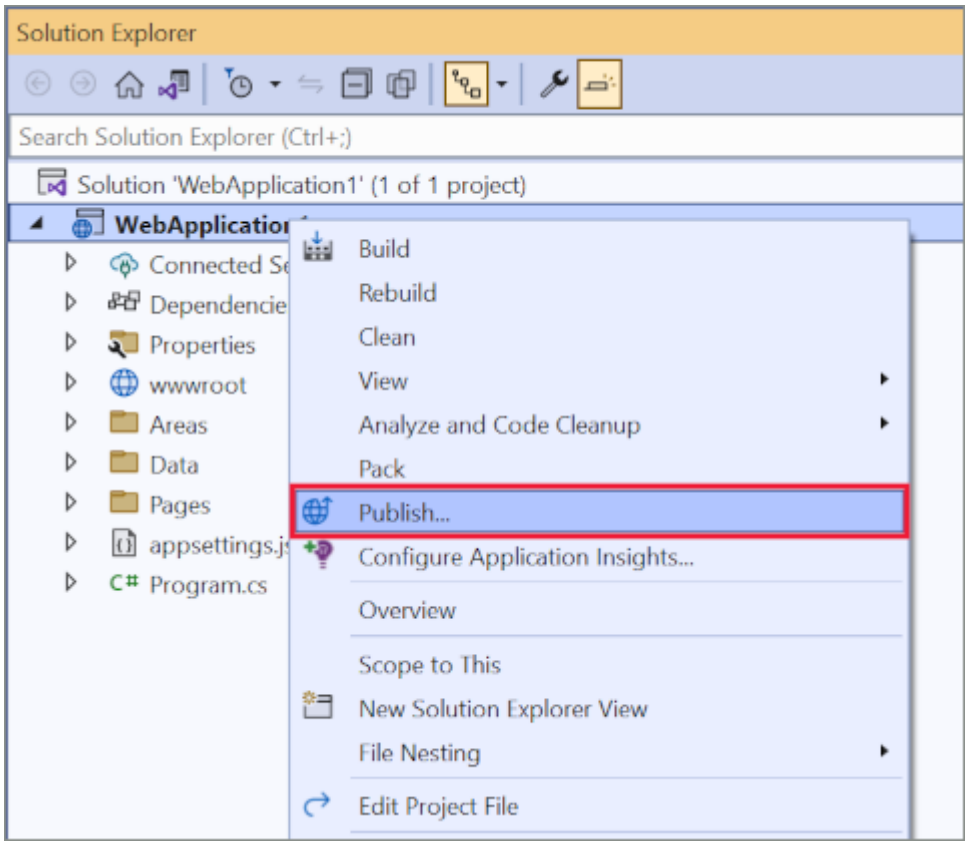
The app displays the email used to register the new user and a **Logout** link.



- Stop the application by closing the browser, or in Visual Studio select **Debug** > **Stop Debugging**.
- In Visual Studio select **Build** > **Clean Solution** to clean project items and avoid file contention.

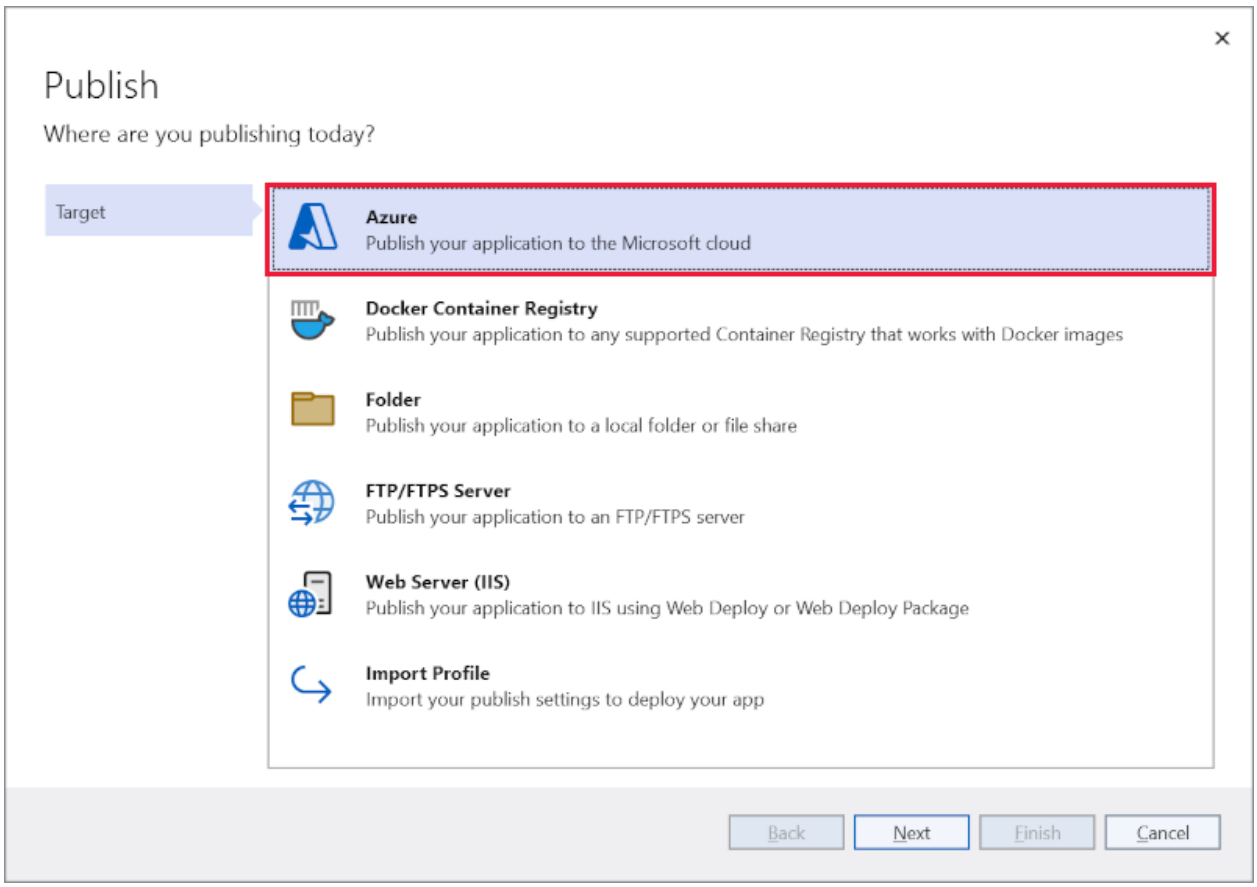
Deploy the app to Azure

Right-click on the project in Solution Explorer and select **Publish**.



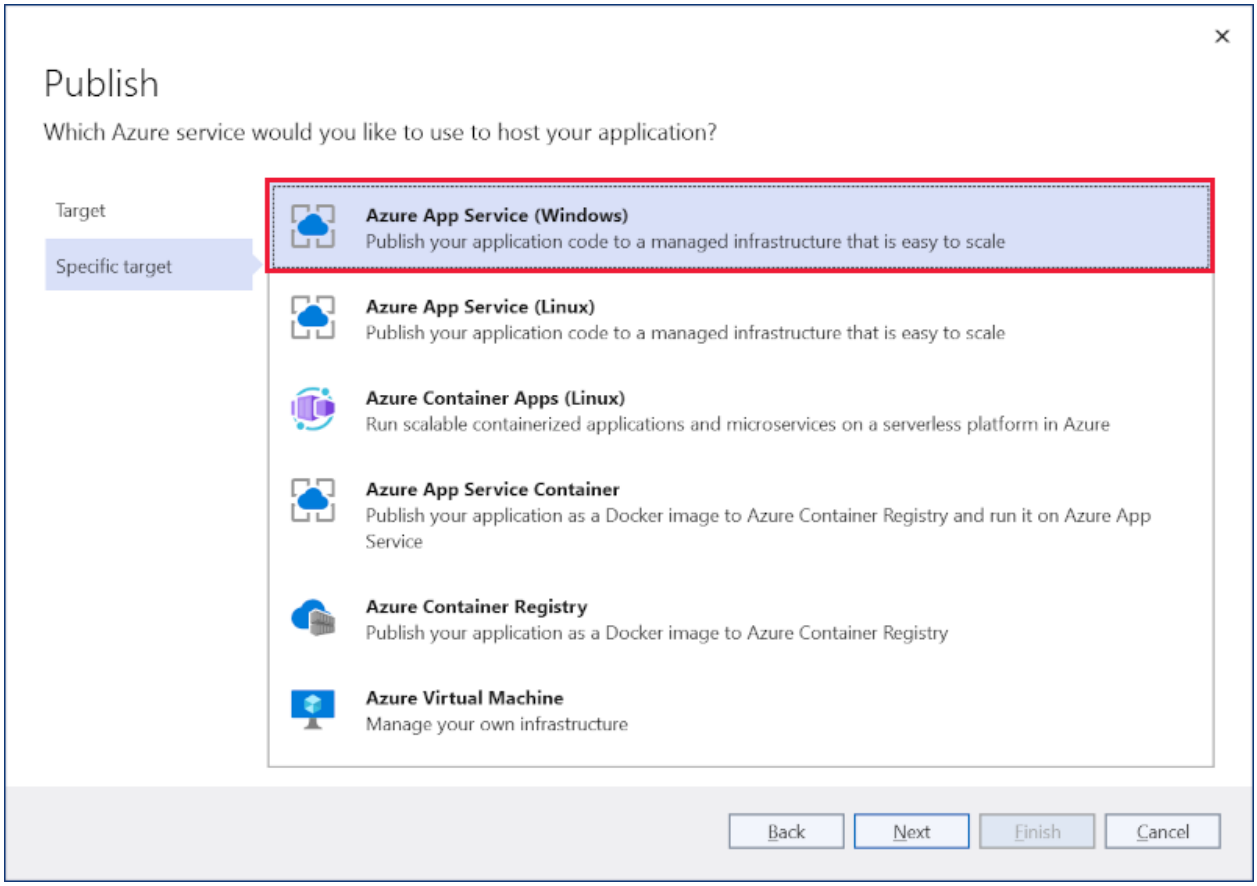
In the **Publish** dialog:

- Select **Azure**.
- Select **Next**.

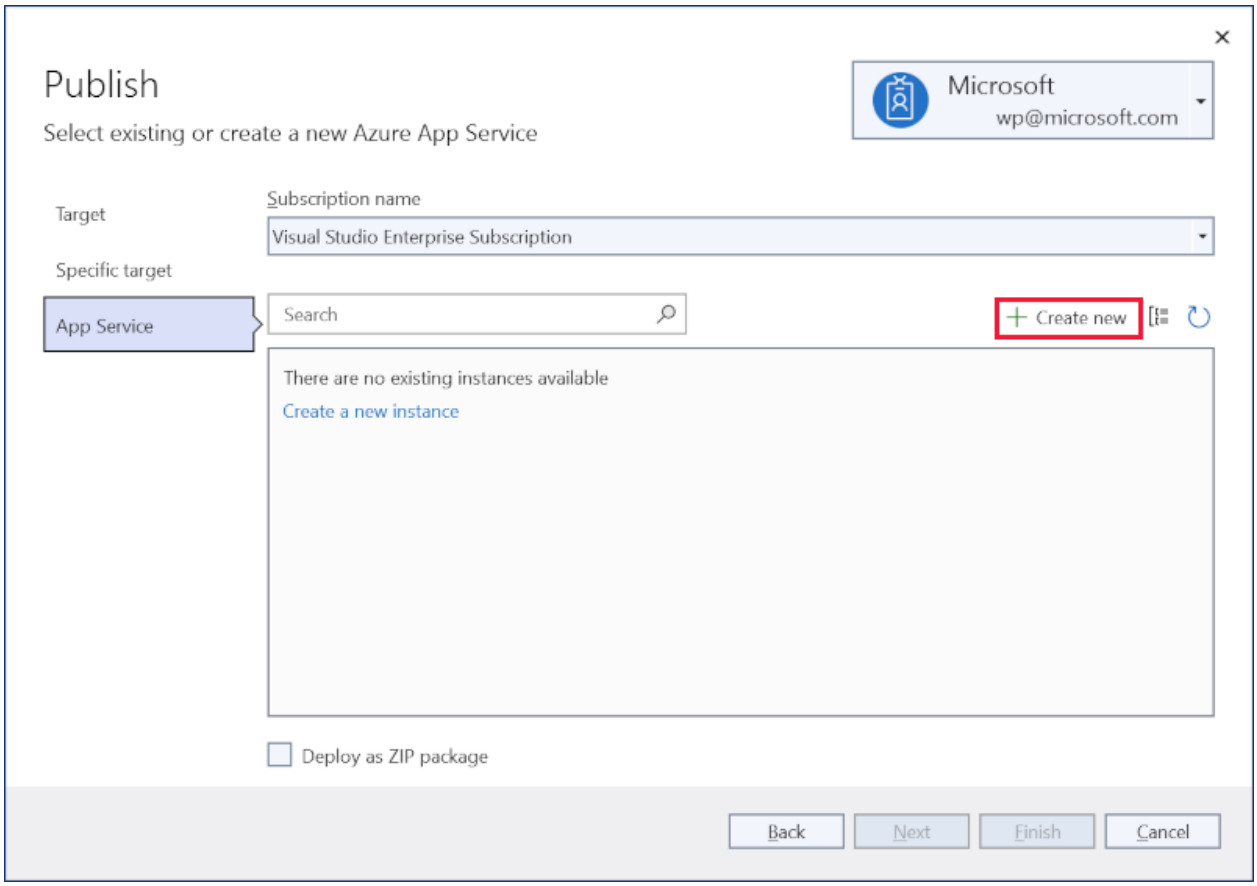


In the **Publish** dialog:

- Select **Azure App Service (Windows)**.
- Select **Next**.

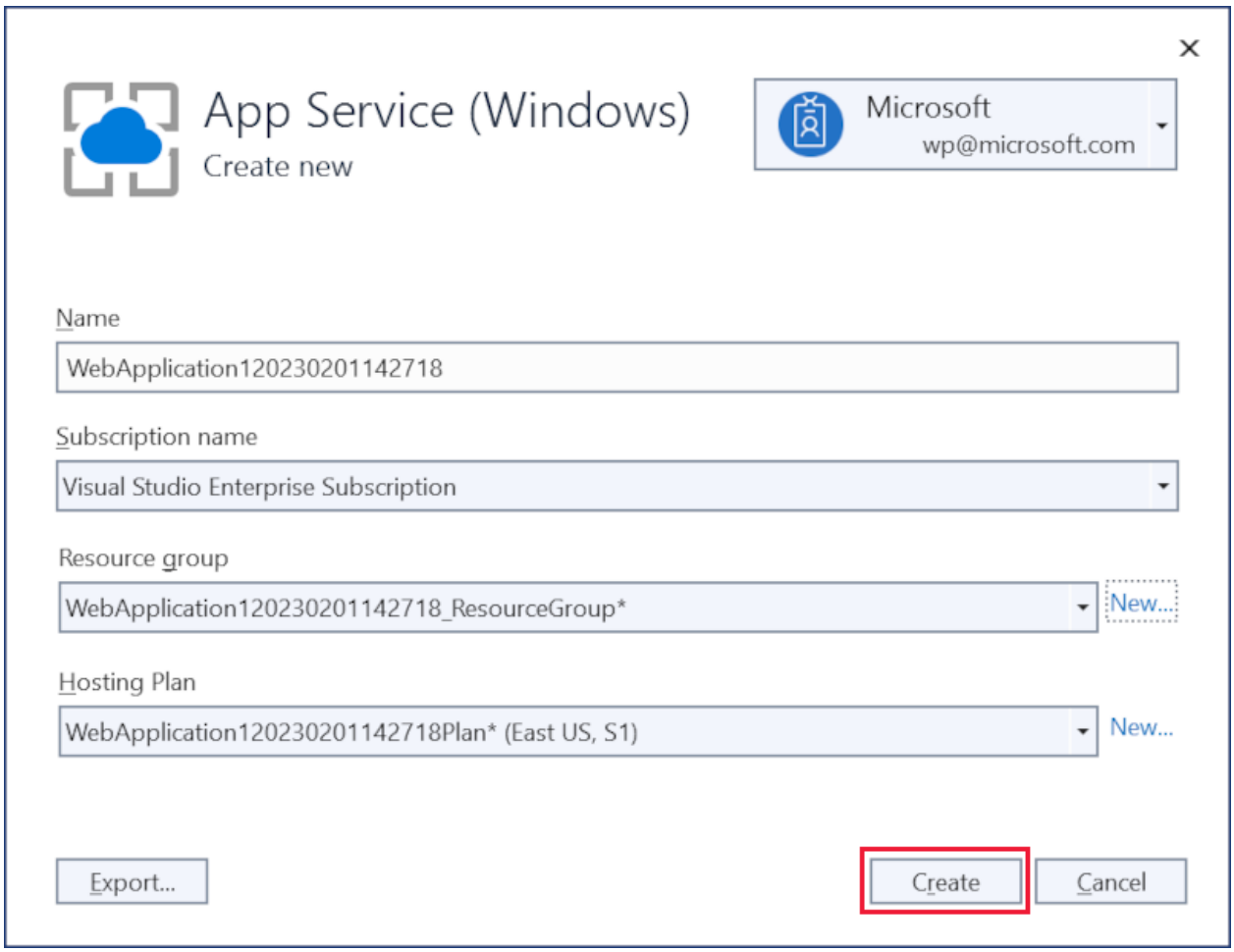


In the **Publish** dialog, in the **App Service** tab, select **Create new**.



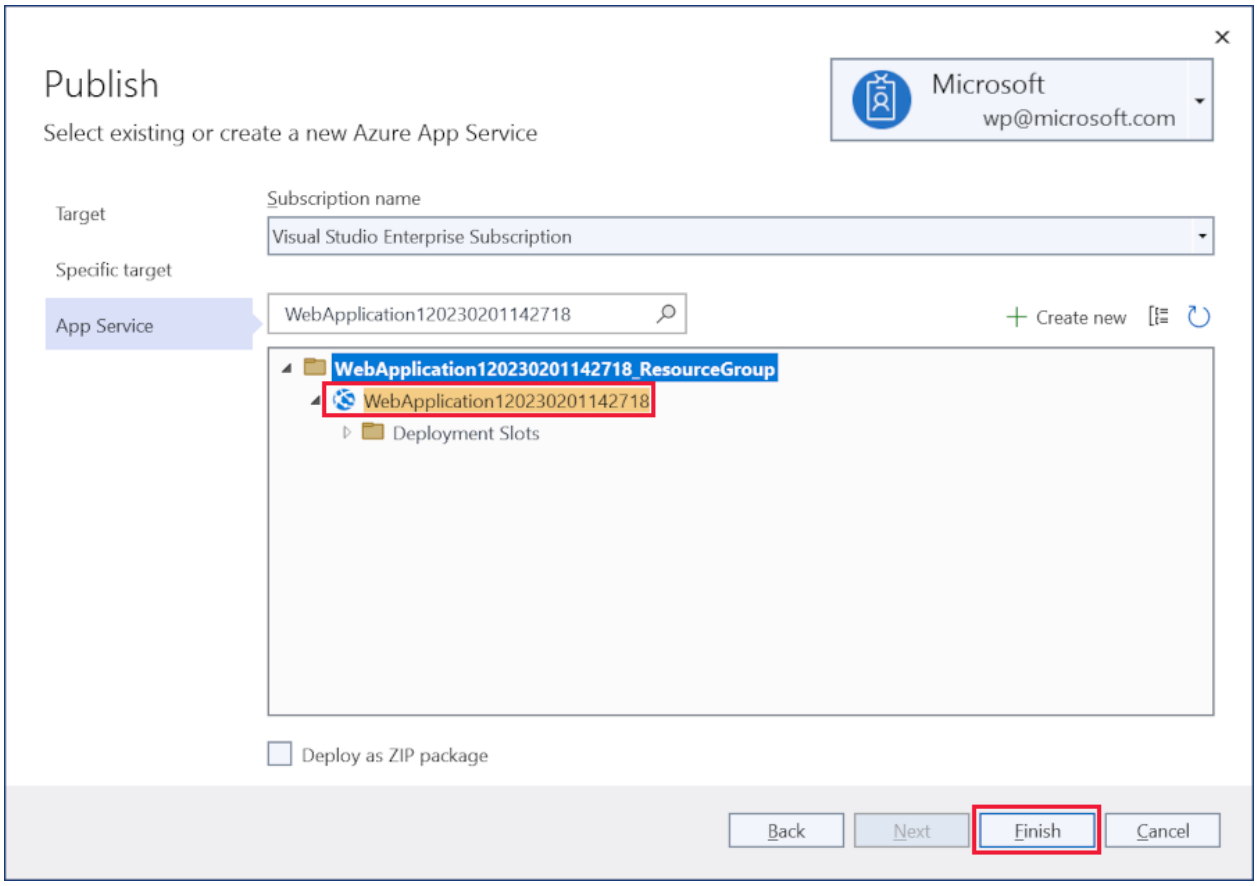
The **Create App Service** dialog appears:

- The **Name**, **Resource Group**, and **Hosting Plan** entry fields are populated. You can keep these names or change them.
- Select **Create**.



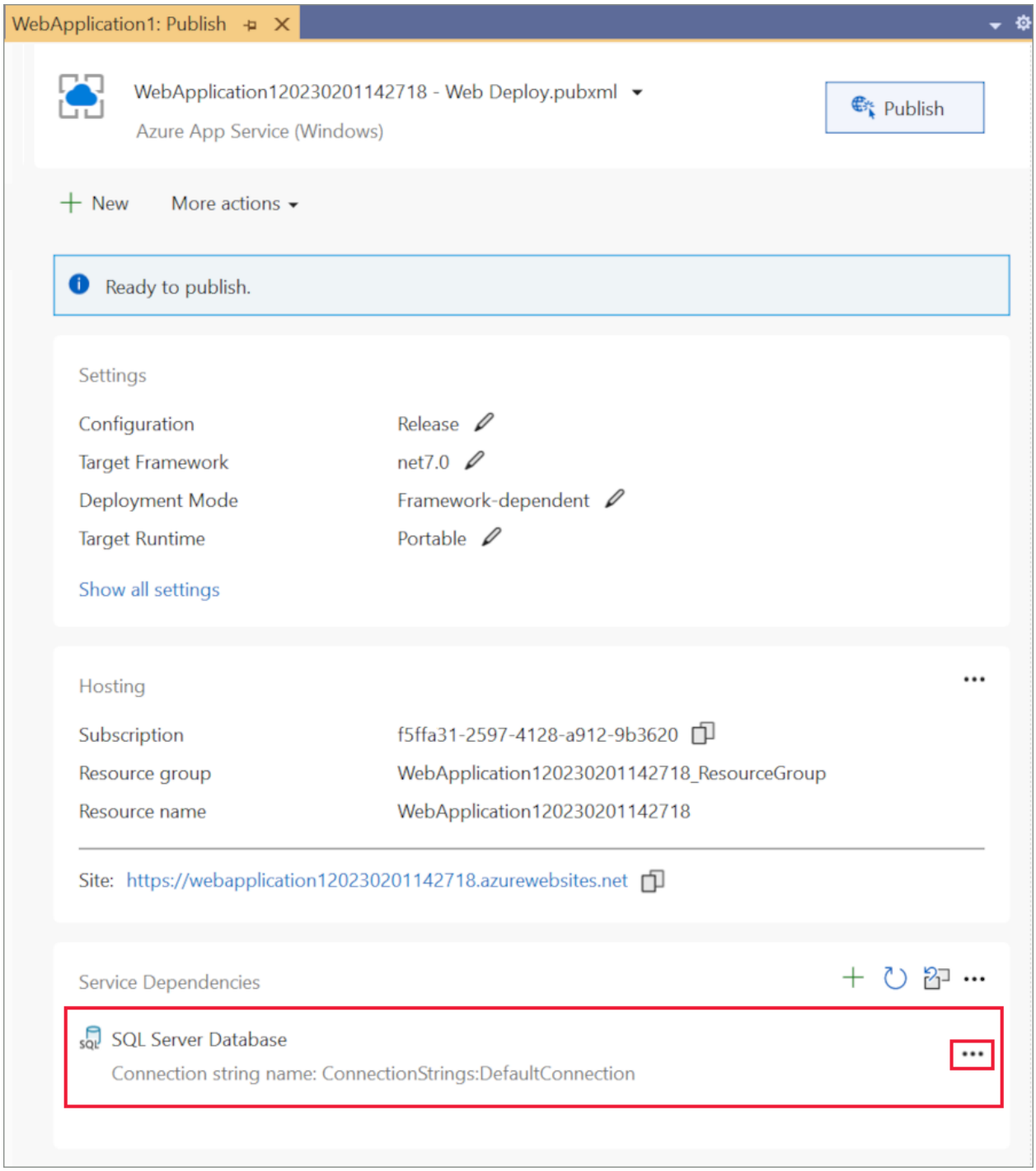
After creation is completed the dialog is automatically closed and the **Publish** dialog gets focus again:

- The new instance that was just created is automatically selected.
- Select **Finish**.



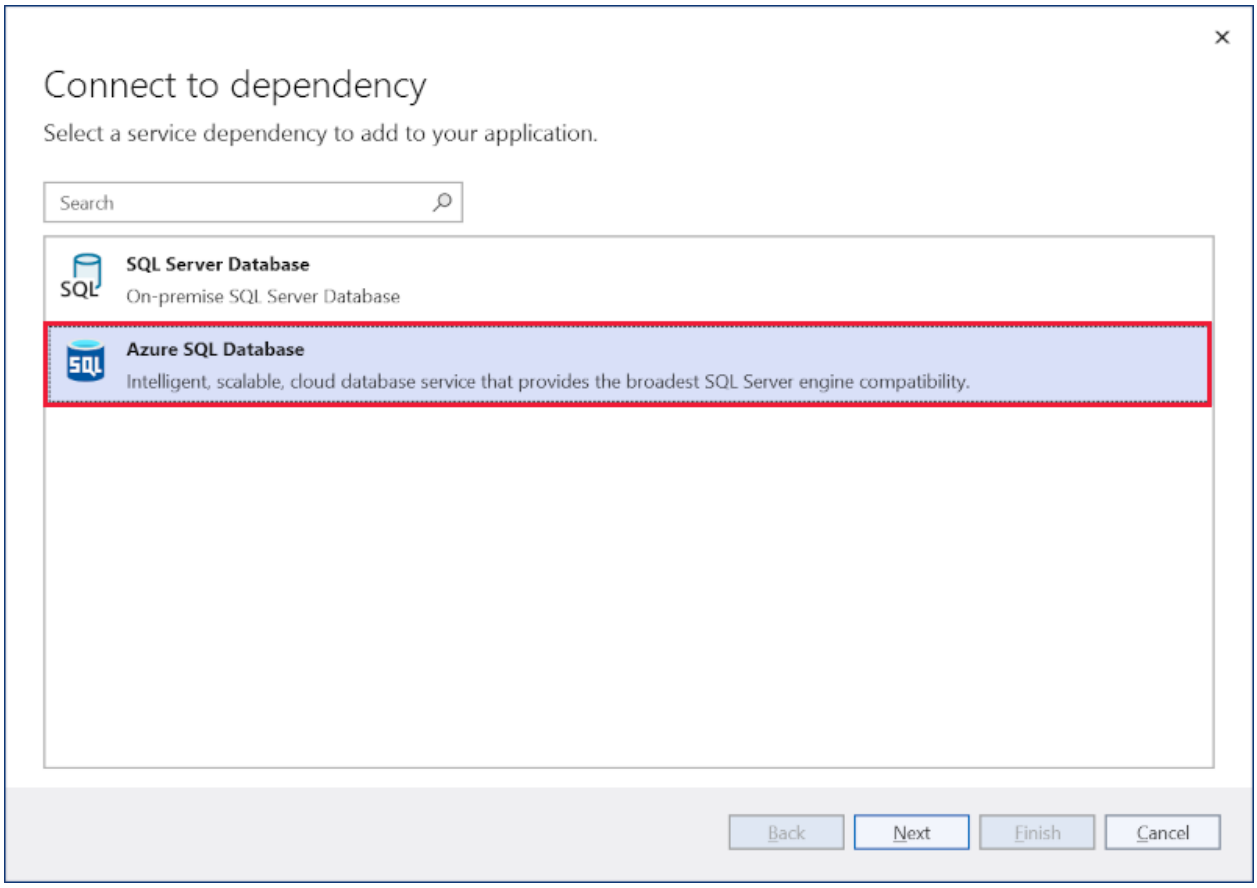
The **Publish profile creation progress** dialog confirms the publish profile was created. Select **Close**.

Next you see the **Publish Profile summary** page. Visual Studio has detected that this application requires a SQL Server database which it has listed in the Service Dependencies pane. Select the ellipsis (...) and then **Connect**.

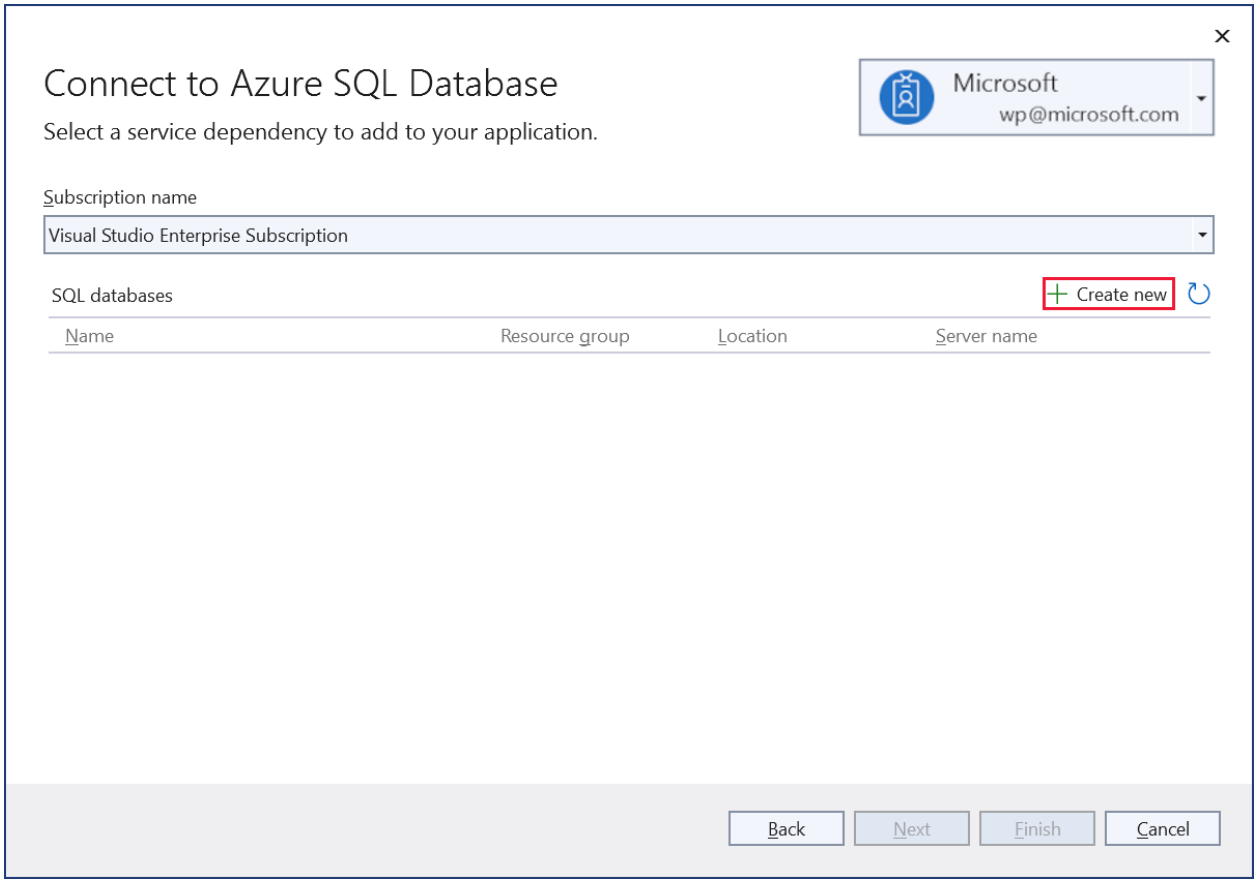


The **Connect to dependency** dialog appears:

- Select **Azure SQL Database**.
- Select **Next**.

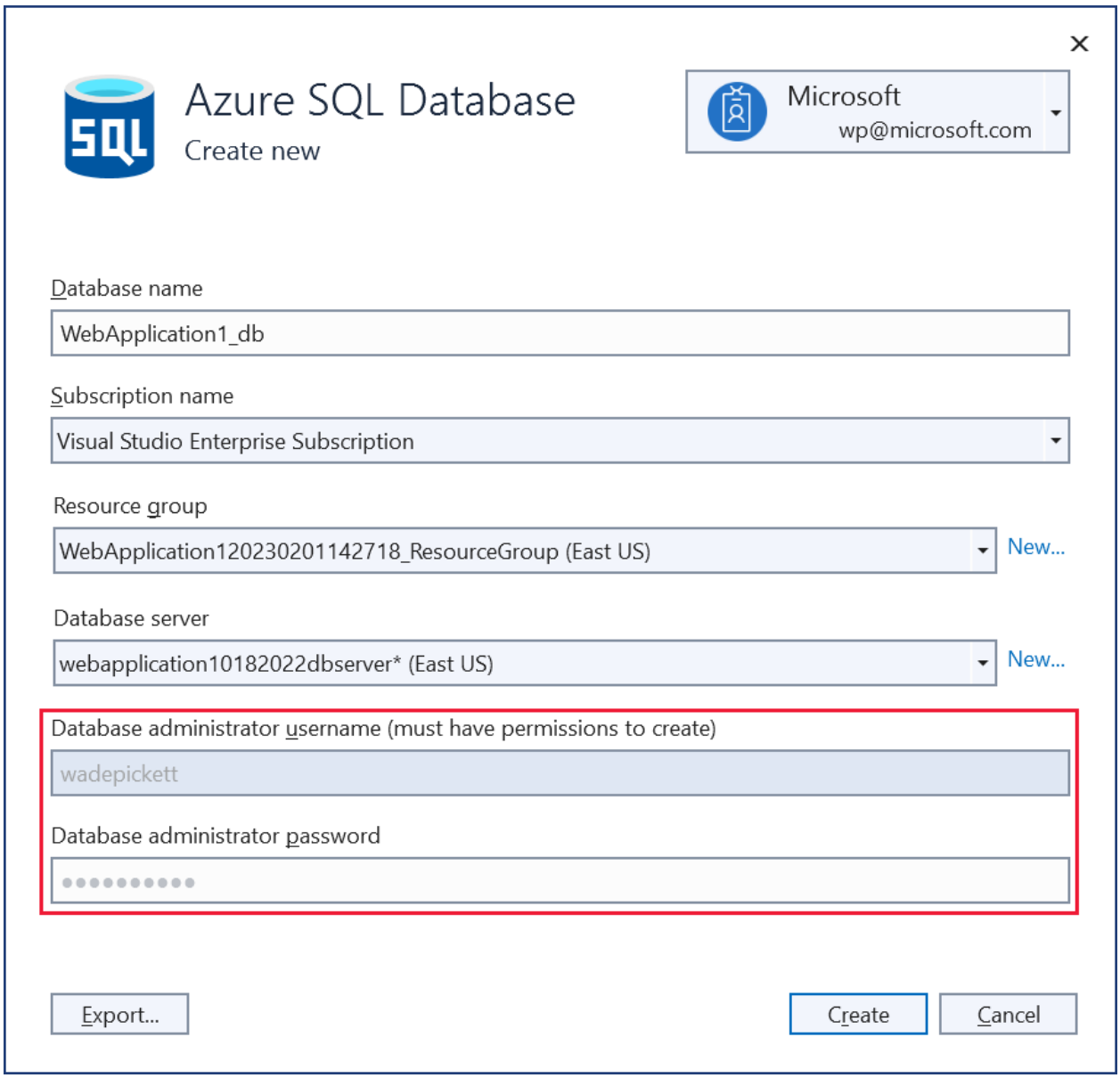


In the **Connect to Azure SQL database** dialog, select **Create new**.



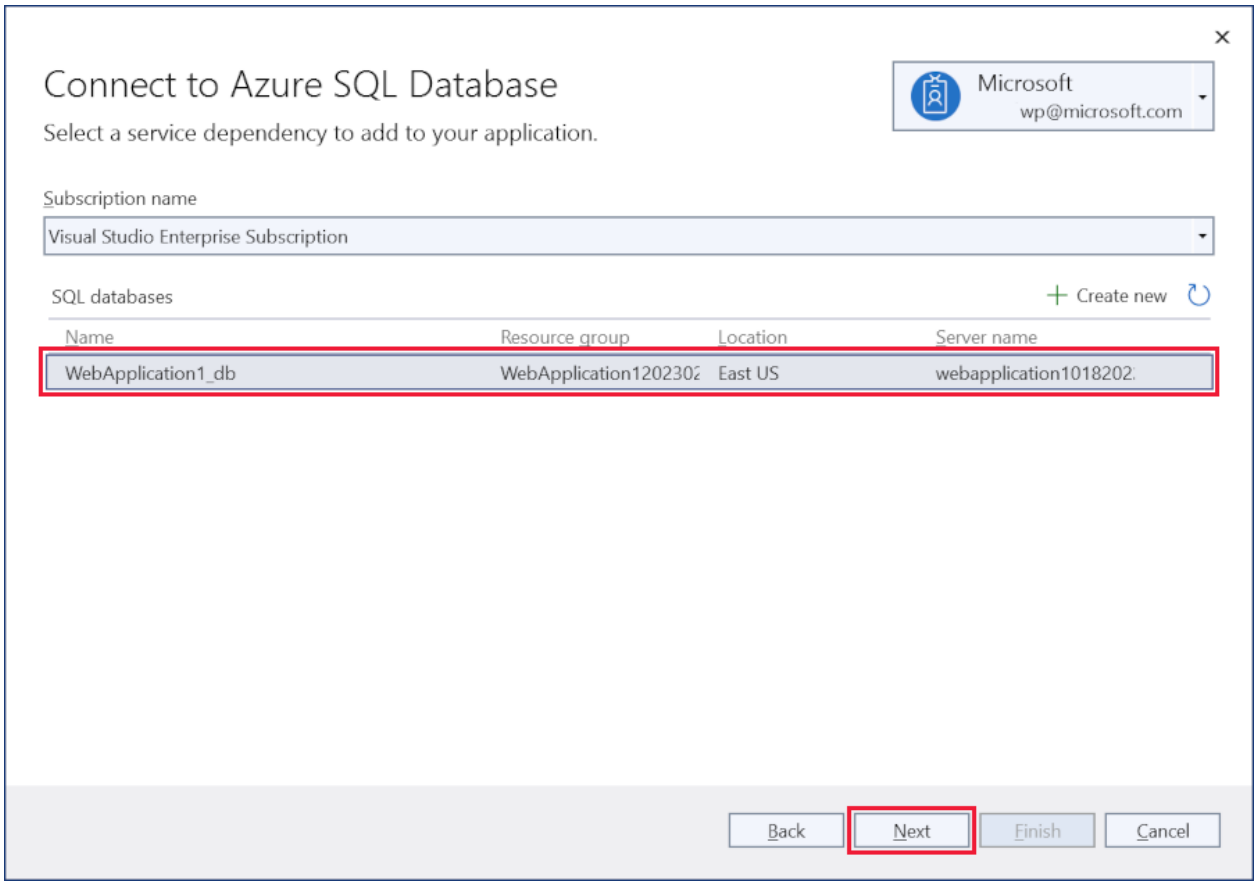
The **Create Azure SQL Database** appears:

- The **Database name**, **Resource Group**, **Database server** and **App Service Plan** entry fields are populated. You can keep these values or change them.
- Enter the **Database administrator username** and **Database administrator password** for the selected **Database server** (note the account you use must have the necessary permissions to create the new Azure SQL database)
- Select **Create**.



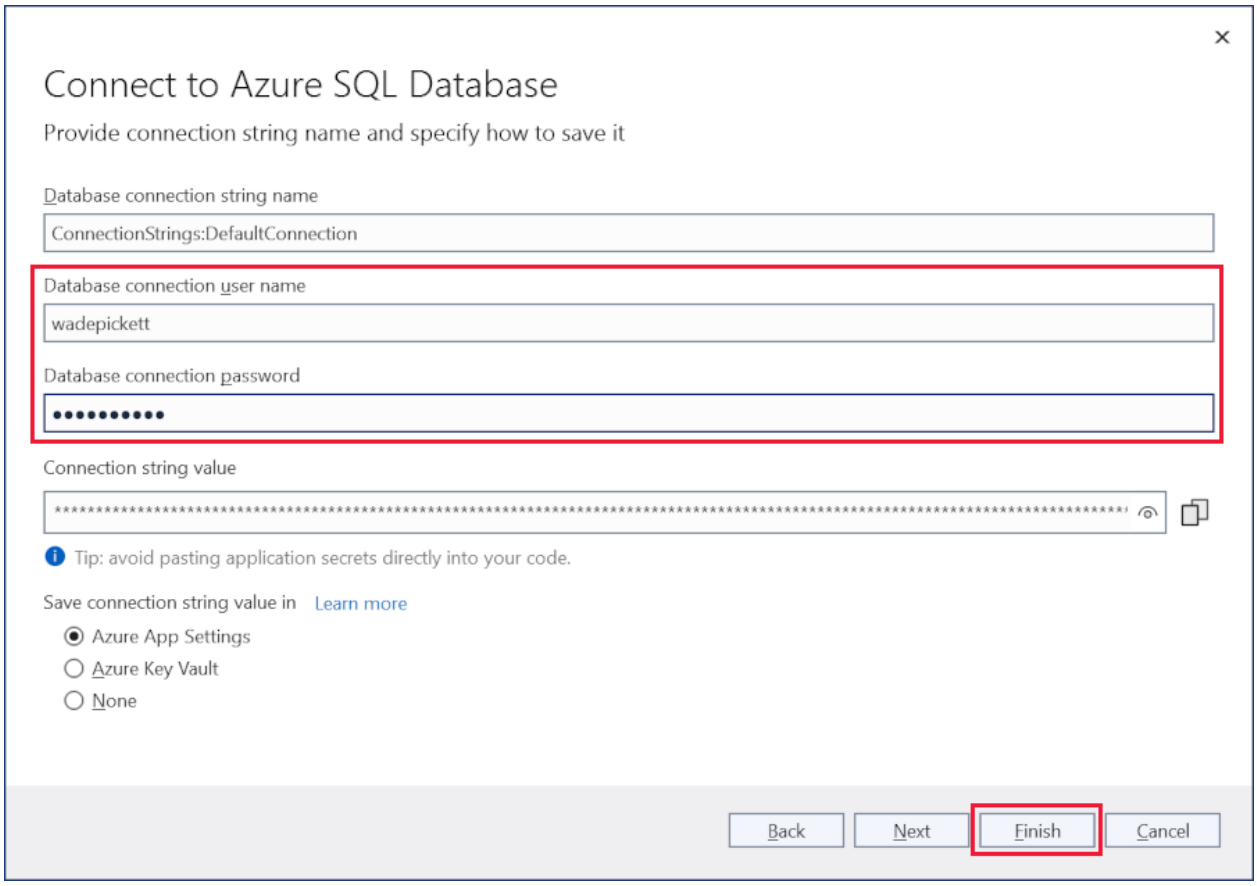
After creation is completed the dialog is automatically closed and the **Connect to Azure SQL Database** dialog gets focus again:

- The new instance that was just created is automatically selected.
- Select **Next**.



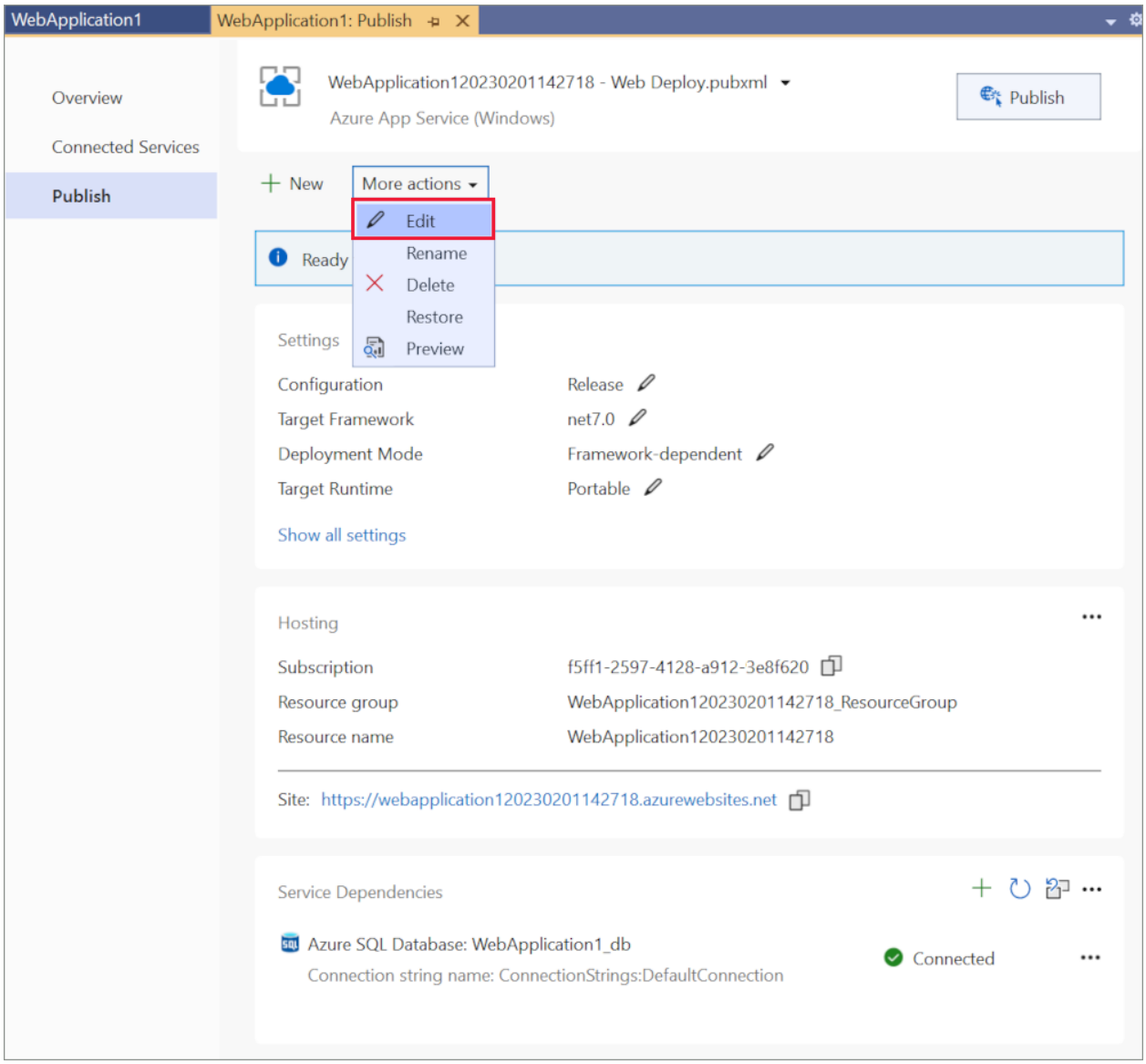
In the next step of the **Connect to Azure SQL Database** dialog:

- Enter the **Database connection user name** and **Database connection password** fields. These are the details your application will use to connect to the database at runtime. Best practice is to avoid using the same details as the admin username and password used in the previous step.
- Select **Finish**.



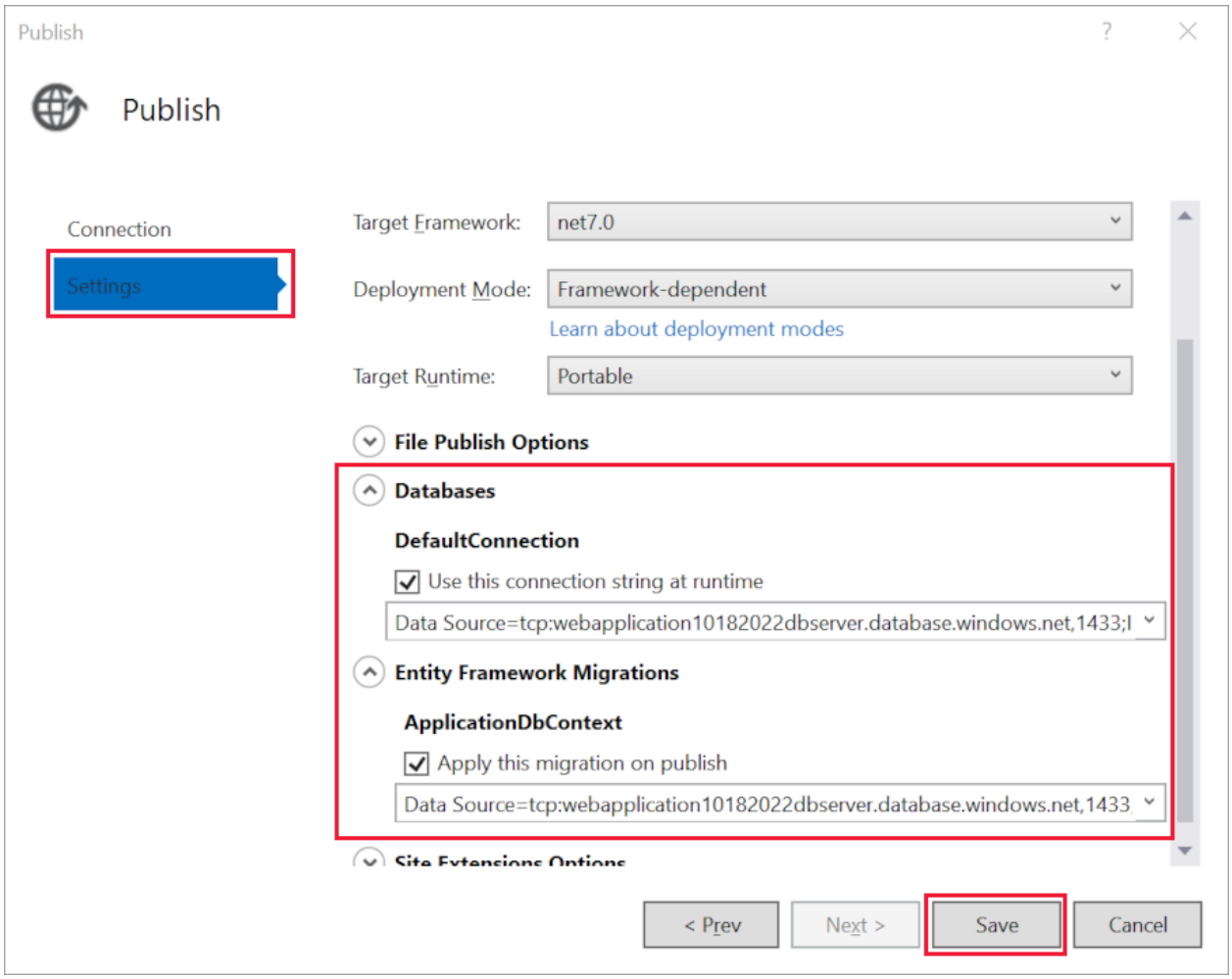
The **Dependency configuration progress** dialog confirms the Azure SQL Database is configured. Select **Close**.

In the **Publish Profile summary** page select **More actions** > **Edit**:

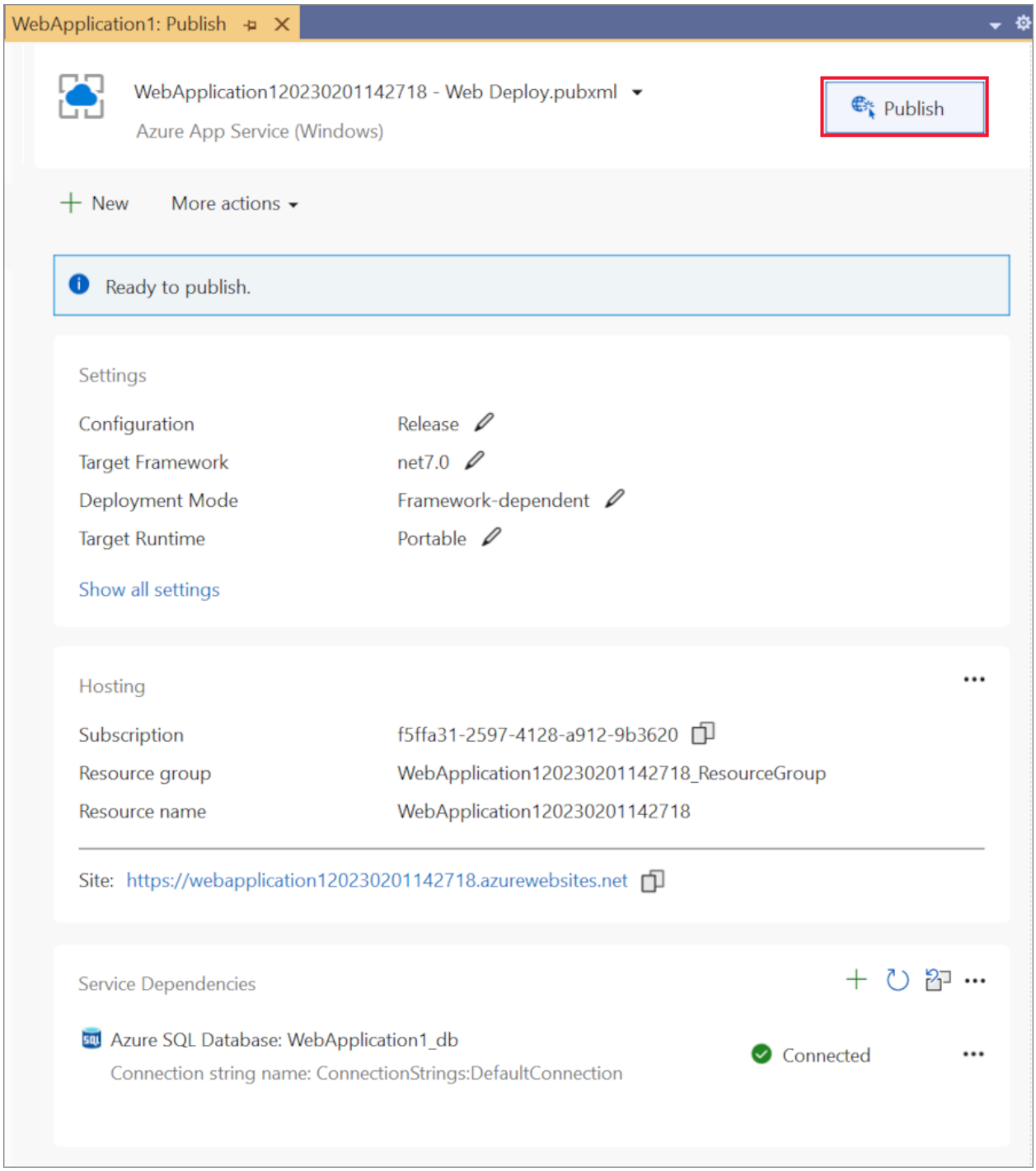


On the **Settings** tab of the **Publish** dialog:

- Expand **Databases** and check **Use this connection string at runtime**.
- Expand **Entity Framework Migrations** and select **Apply this migration on publish**.
- Select **Save**. Visual Studio returns to the **Publish** dialog.



Click **Publish**. Visual Studio publishes your app to Azure. When the deployment completes.



The app is opened in a browser. Register a new user and log in as the new user to validate the database deployment and run-time connection.

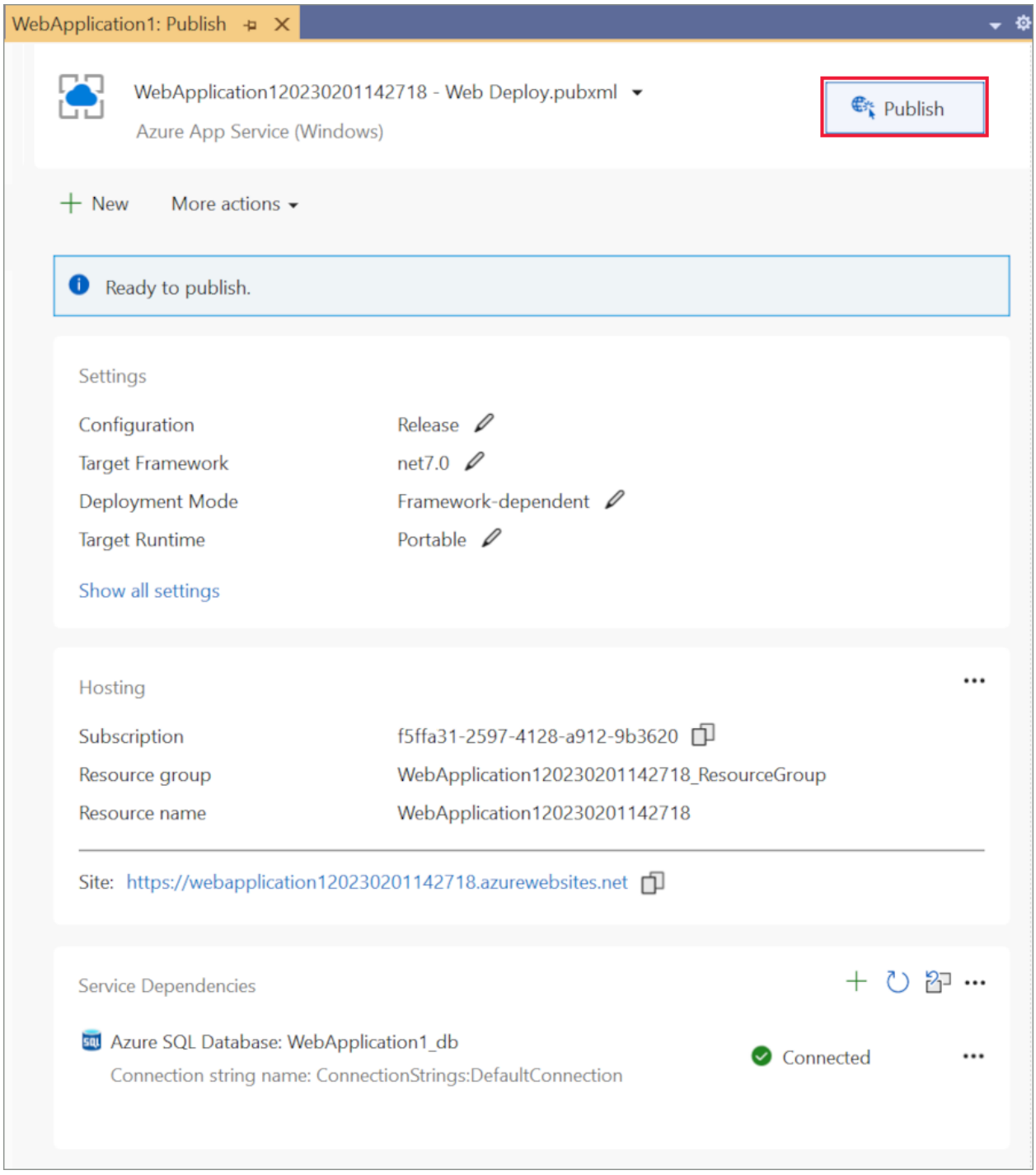
Update the app

- Edit the `Pages/Index.cshtml` Razor page and change its contents, then save the changes. For example, you can modify the paragraph to say "Hello ASP.NET Core!":

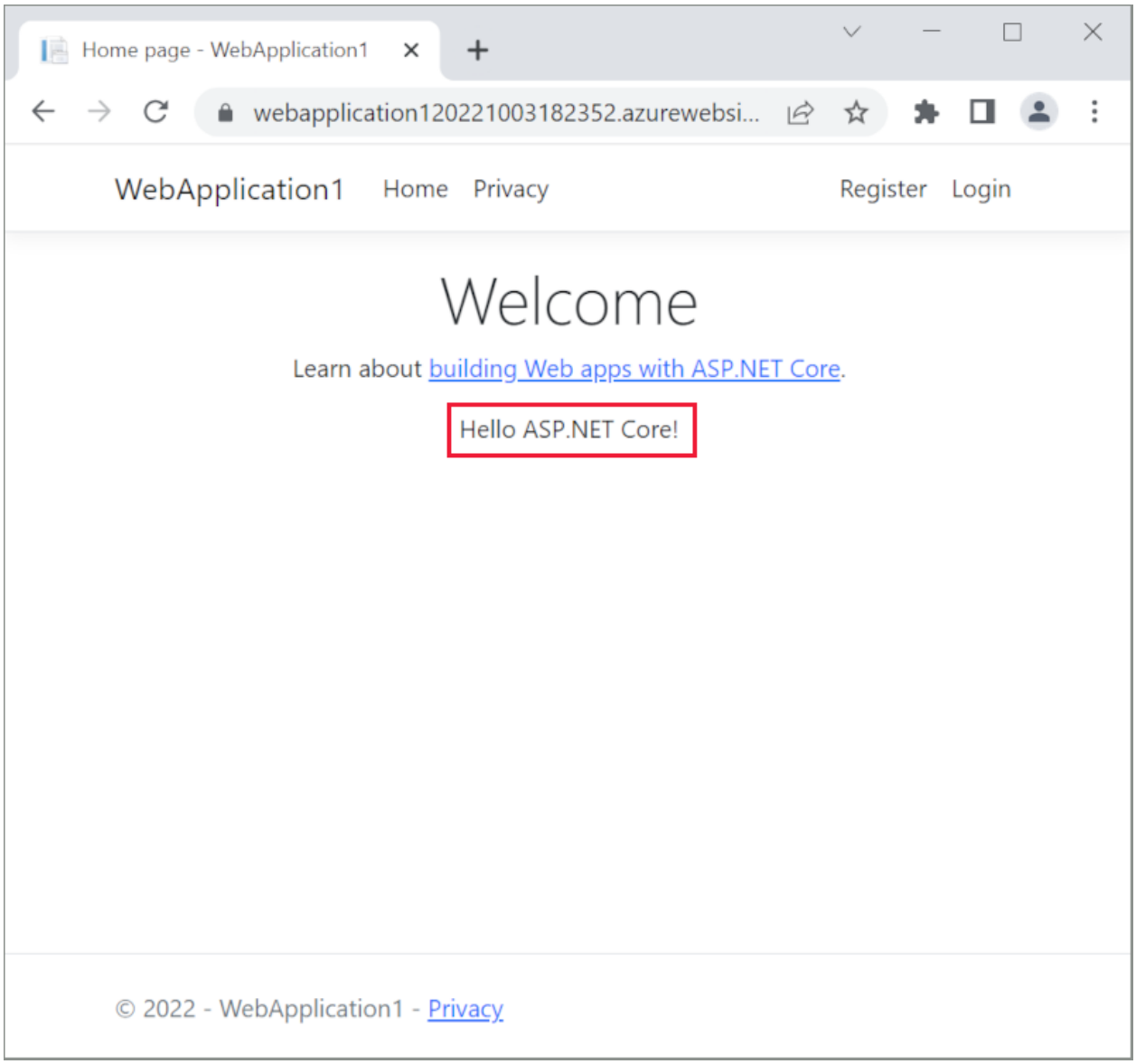
```
HTML

@page
@model IndexModel
@{
```

```
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a
href="https://docs.microsoft.com/aspnet/core">building Web apps with
ASP.NET Core</a>.</p>
    <p>Hello ASP.NET Core!</p>
</div>
```

- Select **Publish** from the **Publish Profile summary** page again.



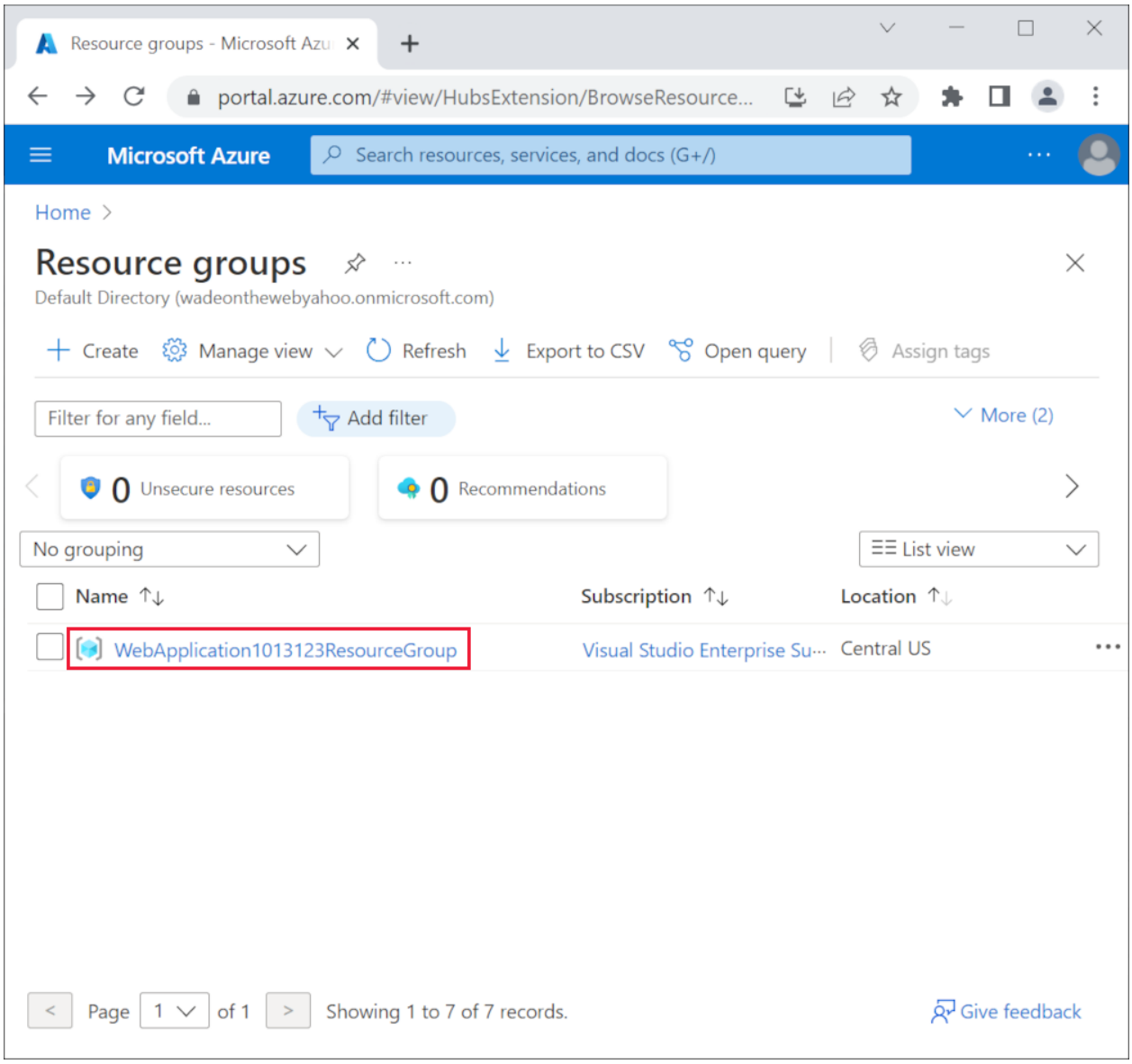
- After the app is published, refresh the page and verify that the changes you made are available on Azure.



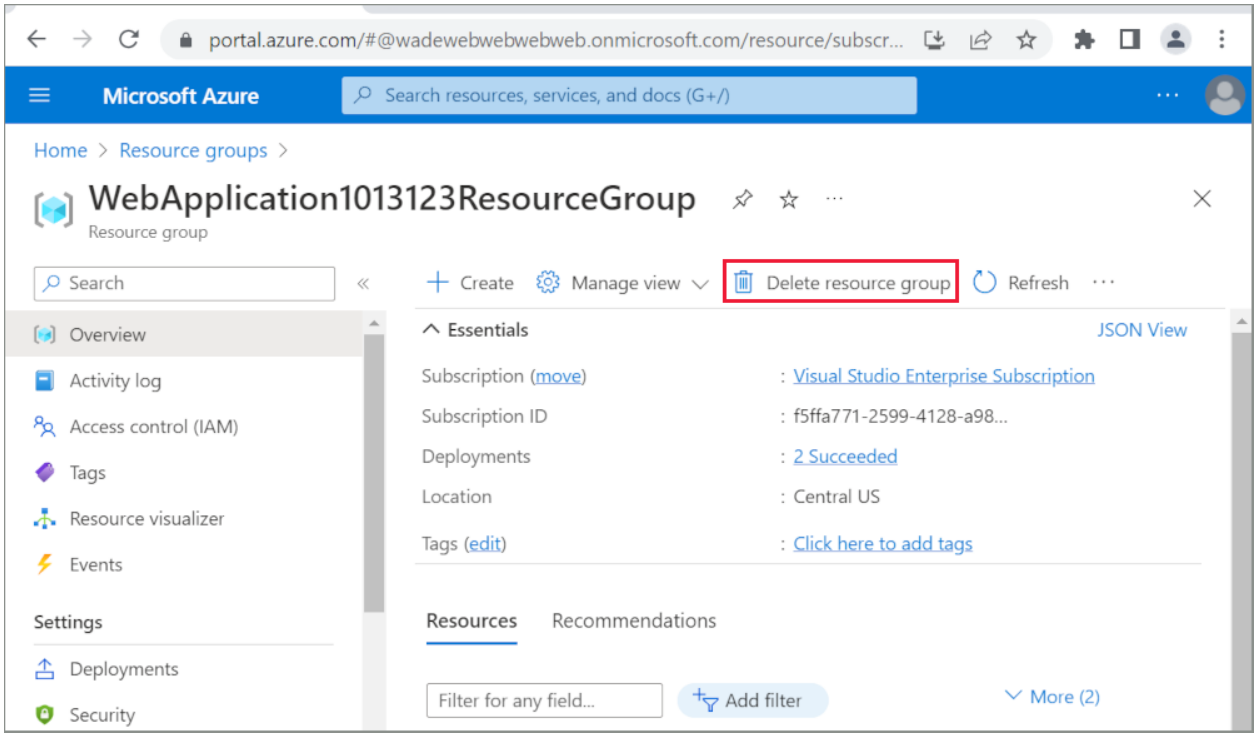
Clean up

When you have finished testing the app, go to the Azure portal and delete the app.

- Select **Resource groups**, then select the resource group you created.



- In the **Resource group** page, select **Delete resource group**.



- Enter the name of the resource group and select **Delete**. Your app and all other resources created in this tutorial are now deleted from Azure.

Additional resources

- Azure App Service
- Azure resource groups
- Azure SQL Database
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment
- Troubleshoot ASP.NET Core on Azure App Service and IIS

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Deploy an ASP.NET Core and Azure SQL Database app to Azure App Service

Article • 09/06/2024

In this tutorial, you learn how to deploy a data-driven ASP.NET Core app to Azure App Service and connect to an Azure SQL Database. You'll also deploy an Azure Cache for Redis to enable the caching code in your application. Azure App Service is a highly scalable, self-patching, web-hosting service that can easily deploy apps on Windows or Linux. Although this tutorial uses an ASP.NET Core 8.0 app, the process is the same for other versions of ASP.NET Core.

In this tutorial, you learn how to:

- ✓ Create a secure-by-default App Service, SQL Database, and Redis cache architecture.
- ✓ Secure connection secrets using a managed identity and Key Vault references.
- ✓ Deploy a sample ASP.NET Core app to App Service from a GitHub repository.
- ✓ Access App Service connection strings and app settings in the application code.
- ✓ Make updates and redeploy the application code.
- ✓ Generate database schema by uploading a migrations bundle.
- ✓ Stream diagnostic logs from Azure.
- ✓ Manage the app in the Azure portal.
- ✓ Provision the same architecture and deploy by using Azure Developer CLI.
- ✓ Optimize your development workflow with GitHub Codespaces and GitHub Copilot.

Prerequisites

- An Azure account with an active subscription. If you don't have an Azure account, you can create one for free.
- A GitHub account. you can also get one for free.
- Knowledge of ASP.NET Core development.
- **(Optional)** To try GitHub Copilot, a GitHub Copilot account. A 30-day free trial is available.

Skip to the end

You can quickly deploy the sample app in this tutorial and see it running in Azure. Just run the following commands in the Azure Cloud Shell, and follow the prompt:

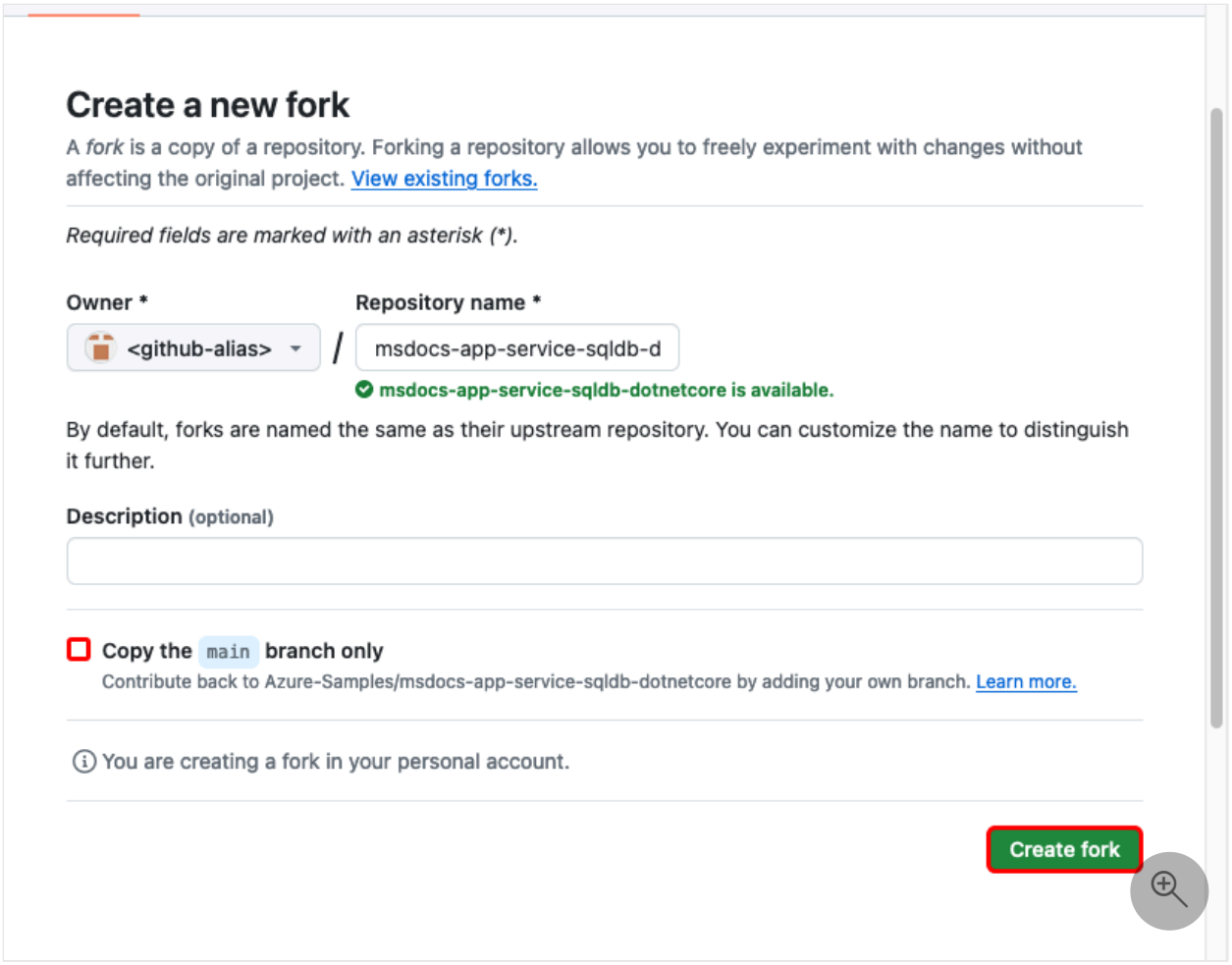
```bash
dotnet tool install --global dotnet-ef
mkdir msdocs-app-service-sqldb-dotnetcore
cd msdocs-app-service-sqldb-dotnetcore
azd init --template msdocs-app-service-sqldb-dotnetcore
azd up
```

1. Run the sample

First, you set up a sample data-driven app as a starting point. For your convenience, the sample repository, includes a dev container configuration. The dev container has everything you need to develop an application, including the database, cache, and all environment variables needed by the sample application. The dev container can run in a GitHub codespace, which means you can run the sample on any computer with a web browser.

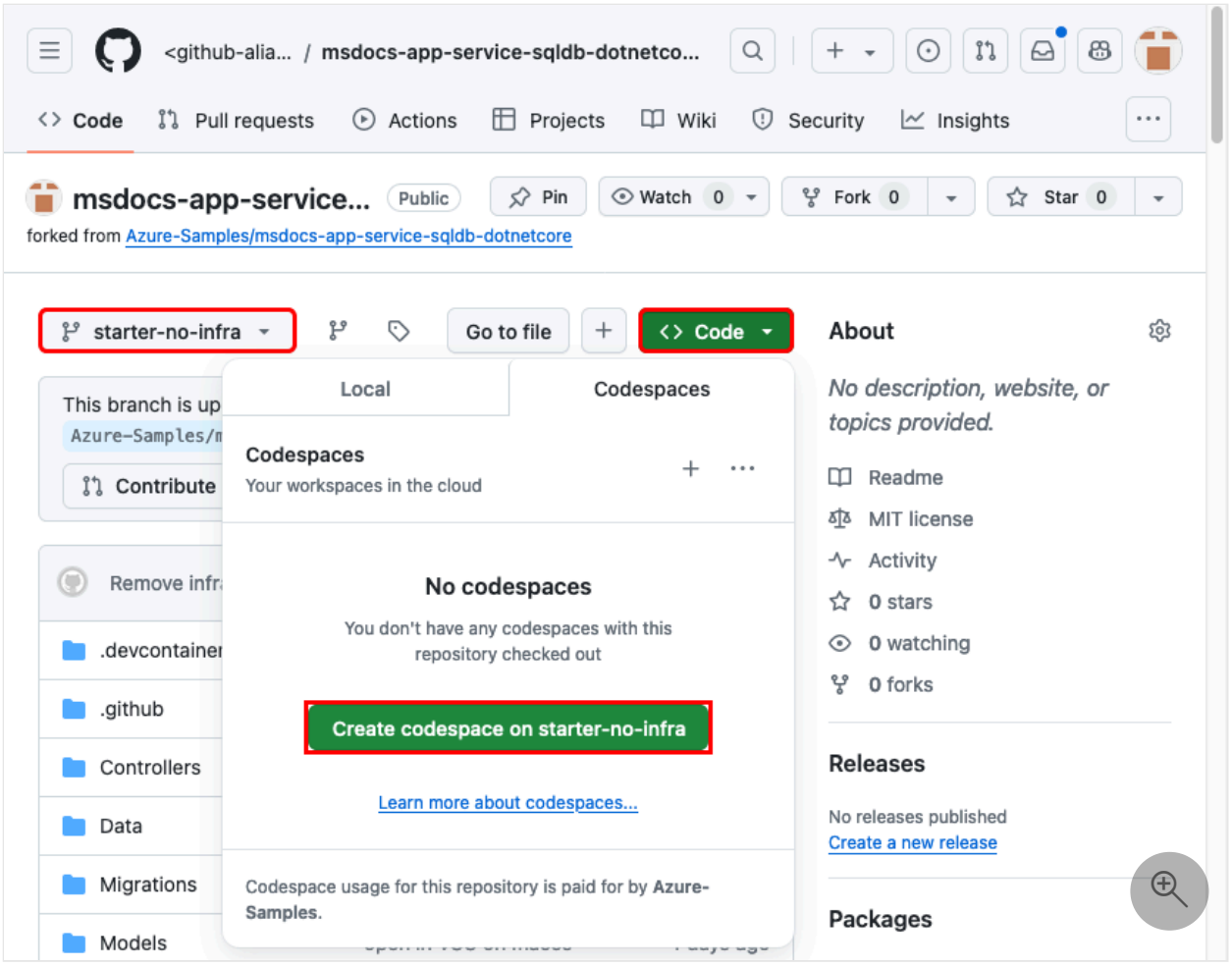
Step 1: In a new browser window:

1. Sign in to your GitHub account.
2. Navigate to https://github.com/Azure-Samples/msdocs-app-service-sqldb-dotnetcore/fork.
3. Unselect **Copy the main branch only**. You want all the branches.
4. Select **Create fork**.



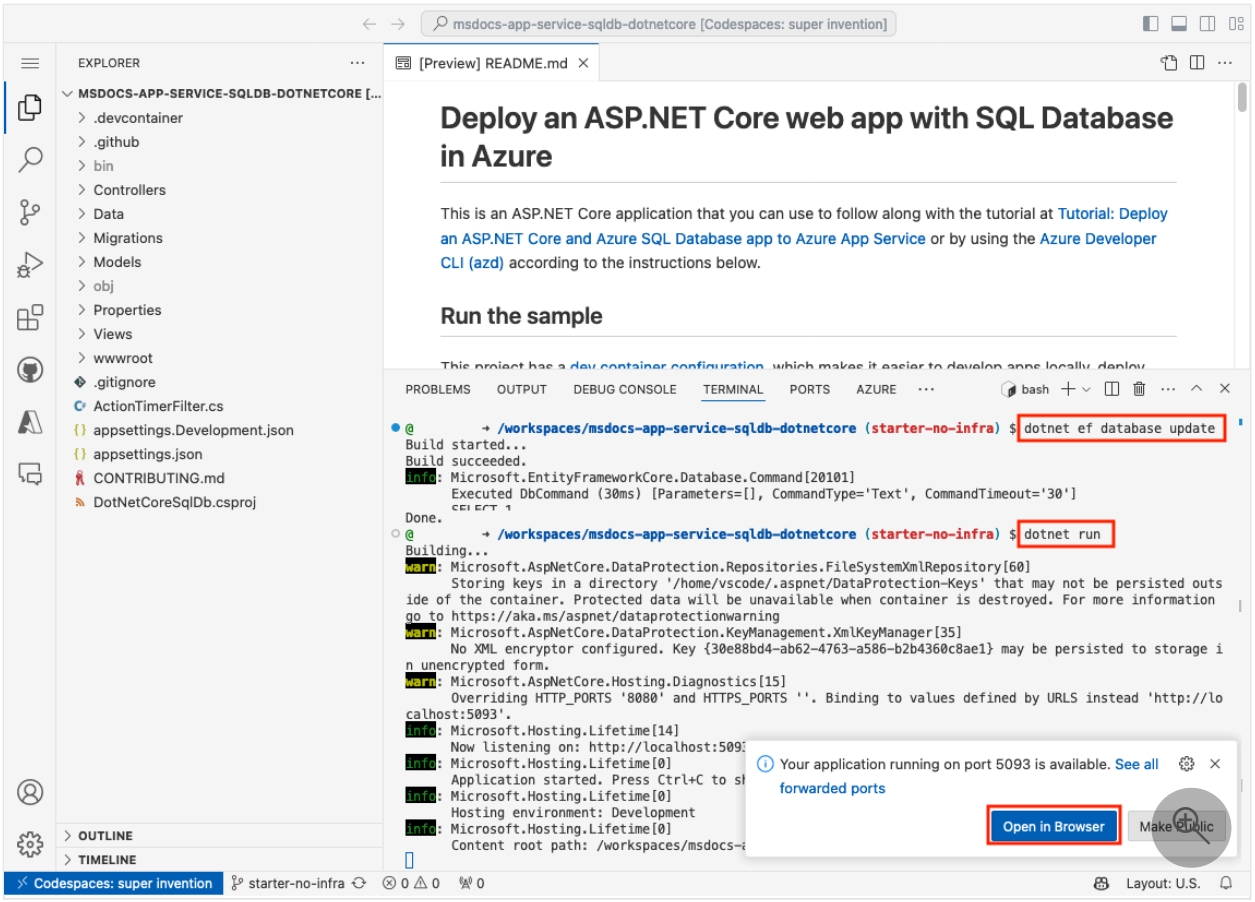
Step 2: In the GitHub fork:

1. Select **main** > **starter-no-infra** for the starter branch. This branch contains just the sample project and no Azure-related files or configuration.
2. Select **Code** > **Create codespace on starter-no-infra**. The codespace takes a few minutes to set up.



Step 3: In the codespace terminal:

1. Run database migrations with `dotnet ef database update`.
2. Run the app with `dotnet run`.
3. When you see the notification `Your application running on port 5093 is available.`, select **Open in Browser**. You should see the sample application in a new browser tab. To stop the application, type `Ctrl` + `C`.



> **Tip**
>
> You can ask GitHub Copilot about this repository. For example:
>
> - *@workspace What does this project do?*
> - *@workspace What does the .devcontainer folder do?*

Having issues? Check the Troubleshooting section.

2. Create App Service, database, and cache

In this step, you create the Azure resources. The steps used in this tutorial create a set of secure-by-default resources that include App Service, Azure SQL Database, and Azure Cache. For the creation process, you'll specify:

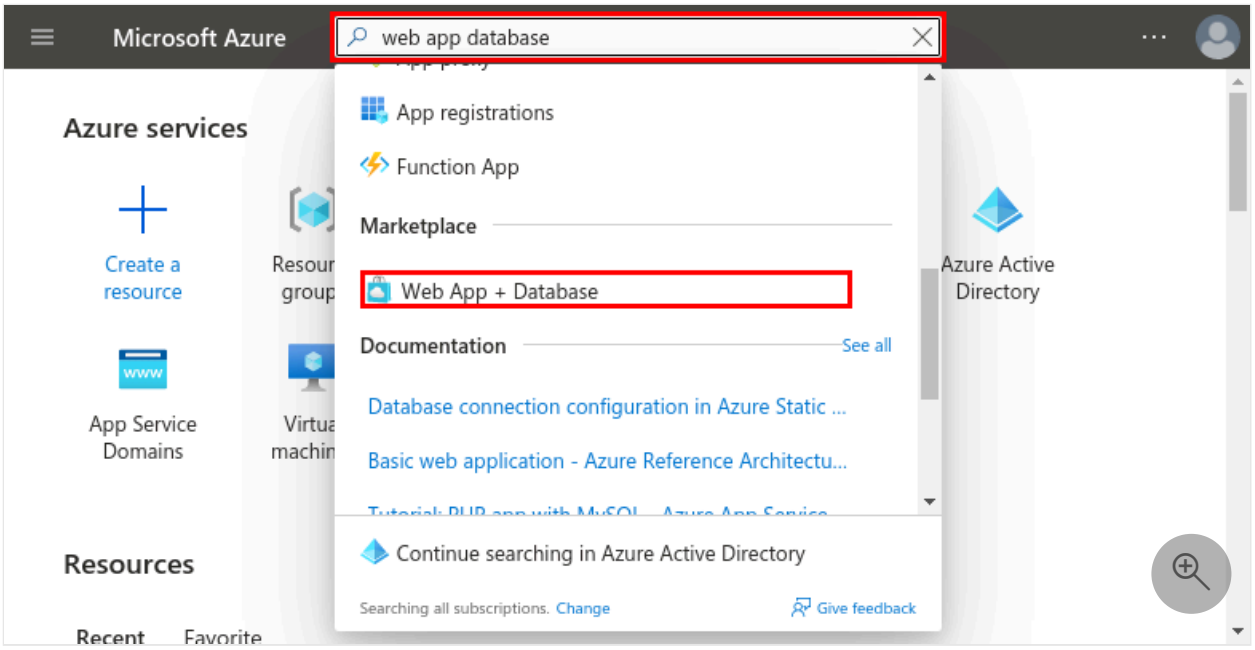
- The **Name** for the web app. It's used as part of the DNS name for your app in the form of `https://<app-name>-<hash>.<region>.azurewebsites.net`.
- The **Region** to run the app physically in the world. It's also used as part of the DNS name for your app.
- The **Runtime stack** for the app. It's where you select the .NET version to use for your app.

- The **Hosting plan** for the app. It's the pricing tier that includes the set of features and scaling capacity for your app.
- The **Resource Group** for the app. A resource group lets you group (in a logical container) all the Azure resources needed for the application.

Sign in to the Azure portal and follow these steps to create your Azure App Service resources.

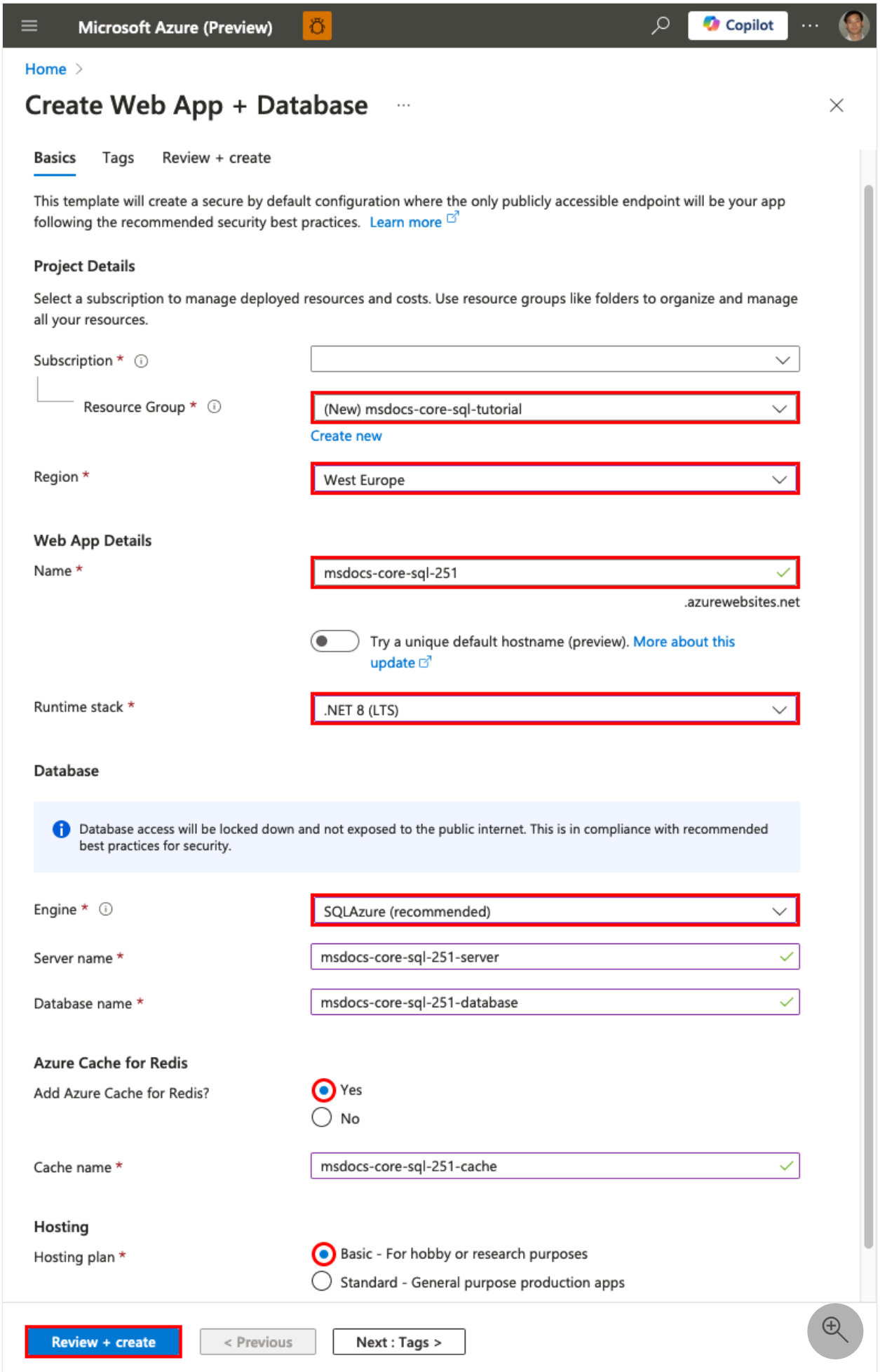
Step 1: In the Azure portal:

1. Enter "web app database" in the search bar at the top of the Azure portal.
2. Select the item labeled **Web App + Database** under the **Marketplace** heading. You can also navigate to the creation wizard directly.



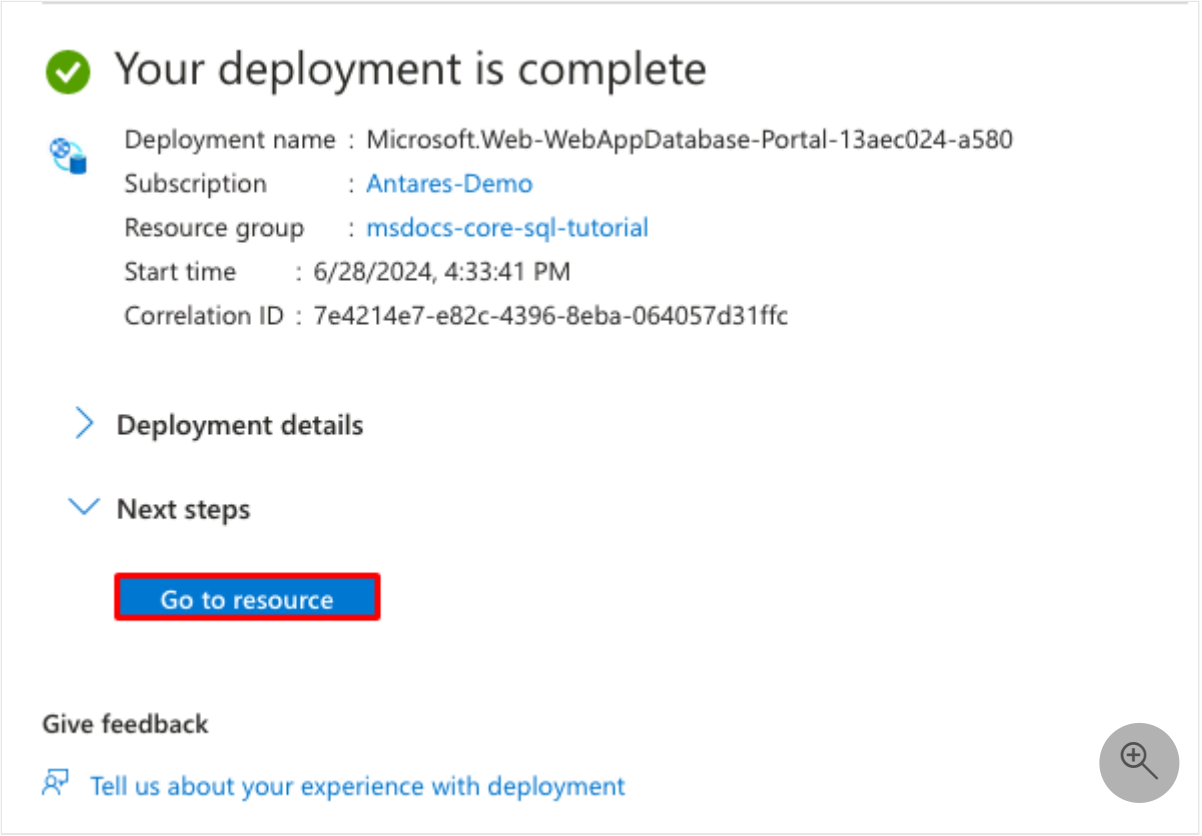
Step 2: In the **Create Web App + Database** page, fill out the form as follows.

1. *Resource Group*: Select **Create new** and use a name of **msdocs-core-sql-tutorial**.
2. *Region*: Any Azure region near you.
3. *Name*: **msdocs-core-sql-XYZ** where *XYZ* is any three random characters. This name must be unique across Azure.
4. *Runtime stack*: **.NET 8 (LTS)**.
5. *Engine*: **SQLAzure**. Azure SQL Database is a fully managed platform as a service (PaaS) database engine that's always running on the latest stable version of the SQL Server.
6. *Add Azure Cache for Redis?*: **Yes**.
7. *Hosting plan*: **Basic**. When you're ready, you can scale up to a production pricing tier.
8. Select **Review + create**.
9. After validation completes, select **Create**.


Microsoft Azure (Preview)
Copilot
Home
Create Web App + Database
Basics
Tags
Review + create
This template will create a secure by default configuration where the only publicly accessible endpoint will be your app following the recommended security best practices. Learn more
Project Details
Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.
Subscription *
Resource Group *
(New) msdocs-core-sql-tutorial
Create new
Region *
West Europe
Web App Details
Name *
msdocs-core-sql-251
.azurewebsites.net
Try a unique default hostname (preview). More about this update
Runtime stack *
.NET 8 (LTS)
Database
Database access will be locked down and not exposed to the public internet. This is in compliance with recommended best practices for security.
Engine *
SQLAzure (recommended)
Server name *
msdocs-core-sql-251-server
Database name *
msdocs-core-sql-251-database
Azure Cache for Redis
Add Azure Cache for Redis?
Yes
No
Cache name *
msdocs-core-sql-251-cache
Hosting
Hosting plan *
Basic - For hobby or research purposes
Standard - General purpose production apps
Review + create
< Previous
Next : Tags >

Step 3: The deployment takes a few minutes to complete. Once deployment completes, select the **Go to resource** button. You're taken directly to the App Service app, but the following resources are created:

- **Resource group**: The container for all the created resources.
- **App Service plan**: Defines the compute resources for App Service. A Linux plan in the *Basic* tier is created.
- **App Service**: Represents your app and runs in the App Service plan.
- **Virtual network**: Integrated with the App Service app and isolates back-end network traffic.
- **Private endpoints**: Access endpoints for the key vault, the database server, and the Redis cache in the virtual network.
- **Network interfaces**: Represents private IP addresses, one for each of the private endpoints.
- **Azure SQL Database server**: Accessible only from behind its private endpoint.
- **Azure SQL Database**: A database and a user are created for you on the server.
- **Azure Cache for Redis**: Accessible only from behind its private endpoint.
- **Key vault**: Accessible only from behind its private endpoint. Used to manage secrets for the App Service app.
- **Private DNS zones**: Enable DNS resolution of the key vault, the database server, and the Redis cache in the virtual network.



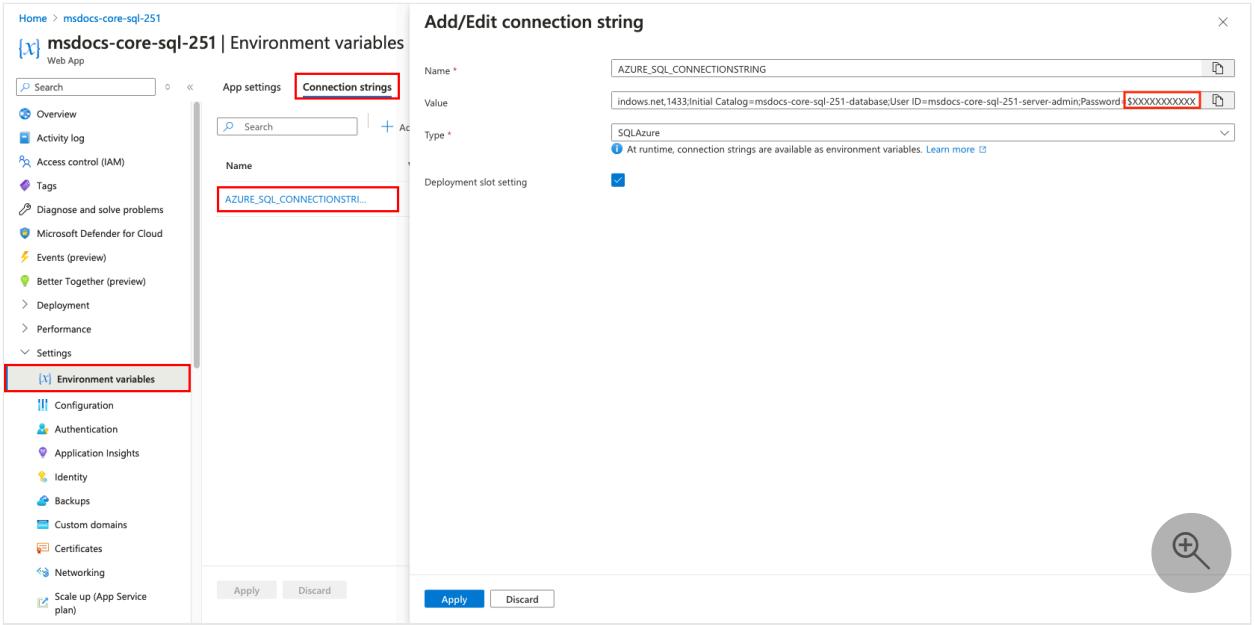
3. Secure connection secrets

The creation wizard generated the connectivity string for you already as .NET connection strings and app settings. However, the security best practice is to keep secrets out of App Service completely. You'll move your secrets to a key vault and change your app setting to Key Vault references with the help of Service Connectors.

> **Tip**
>
> To use passwordless authentication, see **How do I change the SQL Database connection to use a managed identity instead?**

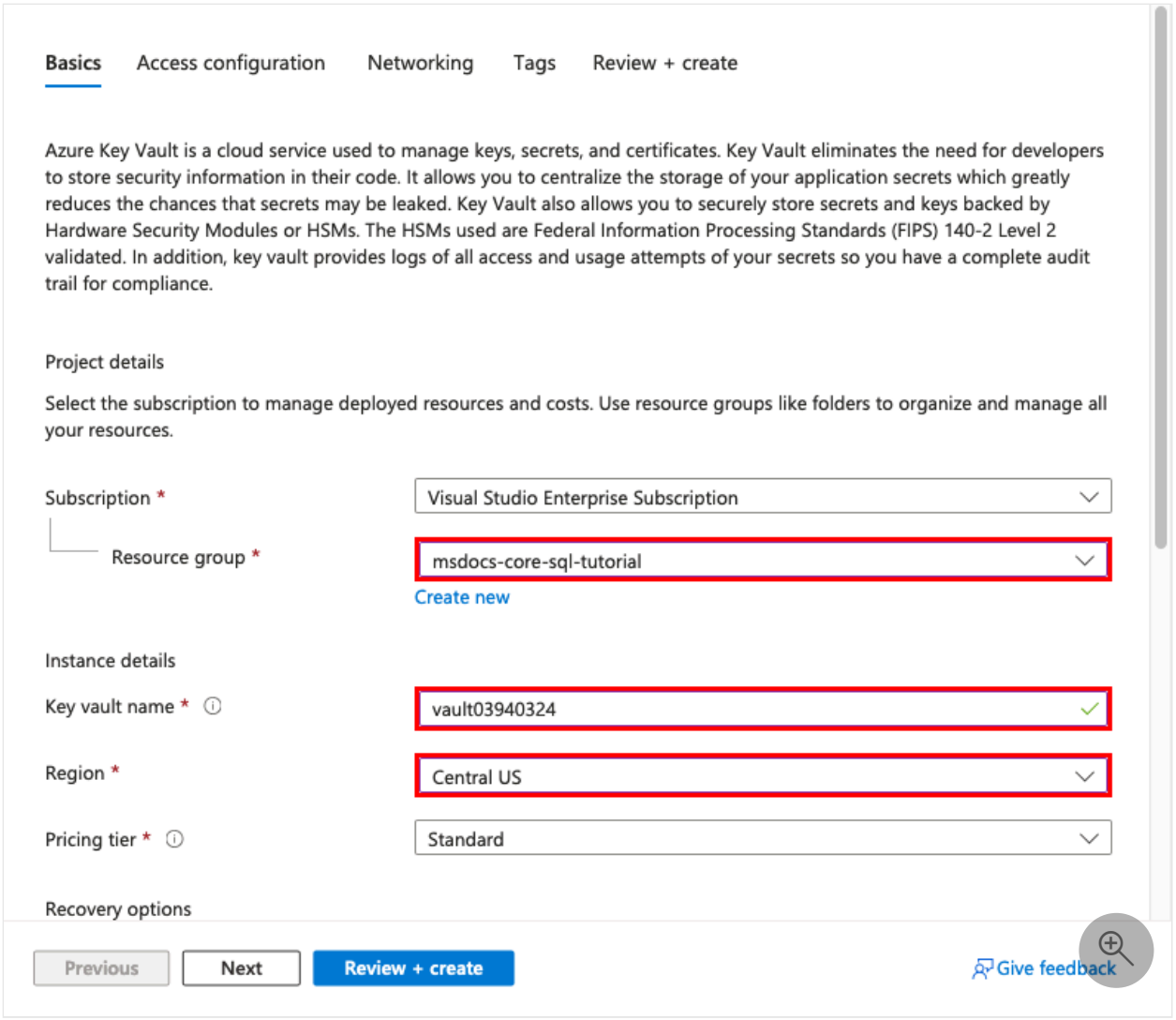
Step 1: In the App Service page:

1. In the left menu, select **Settings** > **Environment variables** > **Connection strings**.
2. Select **AZURE_SQL_CONNECTIONSTRING**.
3. In **Add/Edit connection string**, in the **Value** field, find the *Password*= part at the end of the string.
4. Copy the password string after *Password*= for use later. This connection string lets you connect to the SQL database secured behind a private endpoint. The password is saved directly in the App Service app, which isn't the best. Likewise, the Redis cache connection string in the **App settings** tab contains a secret. You'll change this.



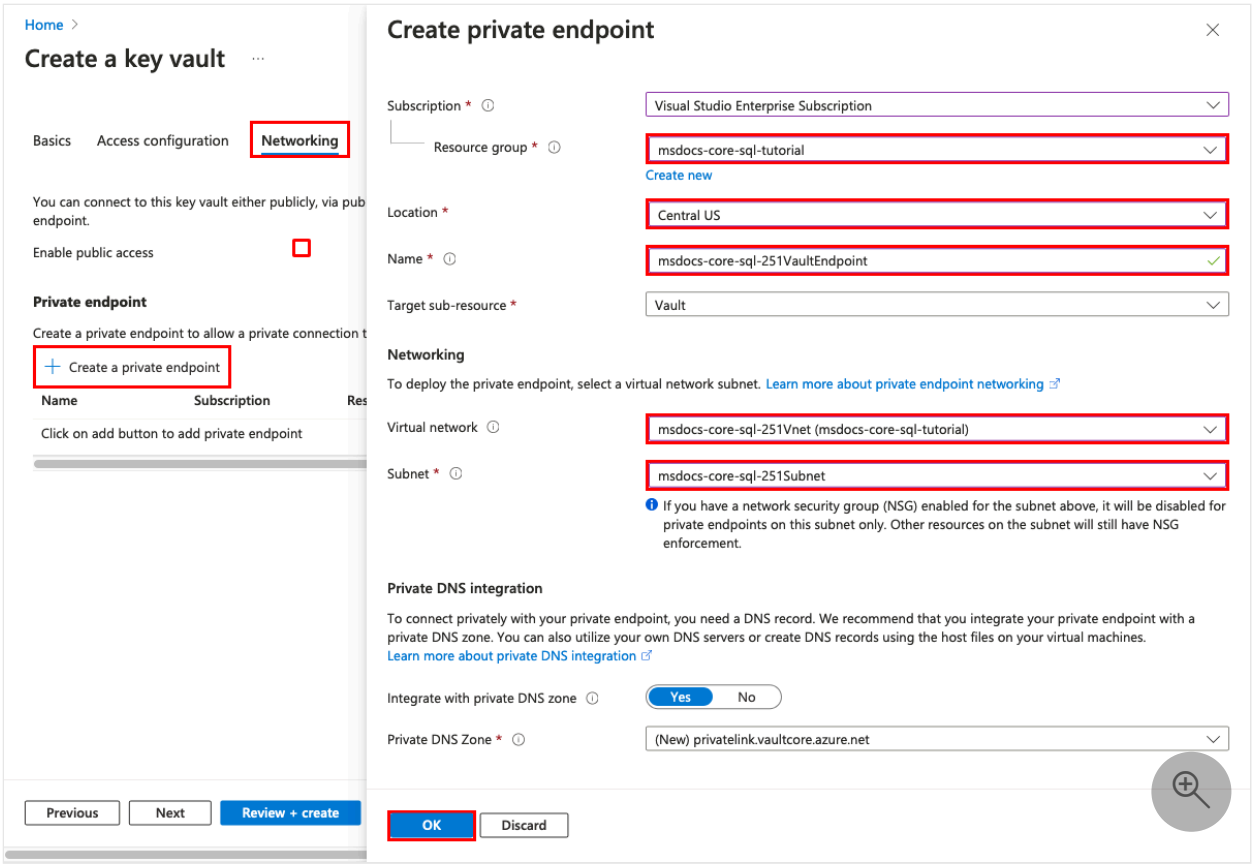
Step 2: Create a key vault for secure management of secrets.

1. In the top search bar, type "*key vault*", then select **Marketplace** > **Key Vault**.
2. In **Resource Group**, select **msdocs-core-sql-tutorial**.
3. In **Key vault name**, type a name that consists of only letters and numbers.
4. In **Region**, set it to the sample location as the resource group.



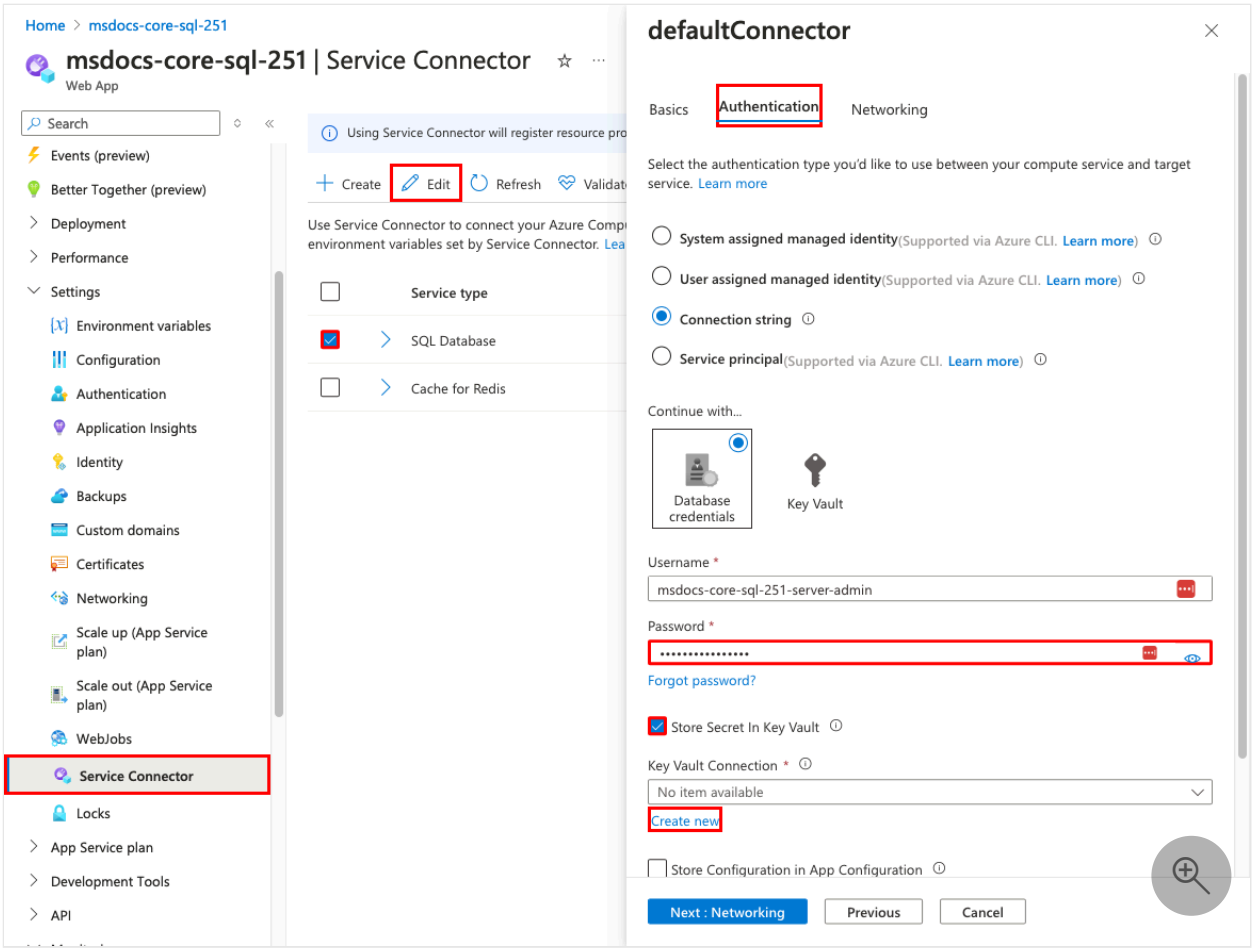
Step 3:

1. Select the **Networking** tab.
2. Unselect **Enable public access**.
3. Select **Create a private endpoint**.
4. In **Resource Group**, select **msdocs-core-sql-tutorial**.
5. In **Key vault name**, type a name that consists of only letters and numbers.
6. In **Region**, set it to the sample location as the resource group.
7. In the dialog, in **Location**, select the same location as your App Service app.
8. In **Resource Group**, select **msdocs-core-sql-tutorial**.
9. In **Name**, type **msdocs-core-sql-XYZVvaultEndpoint**.
10. In **Virtual network**, select **msdocs-core-sql-XYZVnet**.
11. In **Subnet**, **msdocs-core-sql-XYZSubnet**.
12. Select **OK**.
13. Select **Review + create**, then select **Create**. Wait for the key vault deployment to finish. You should see "Your deployment is complete."



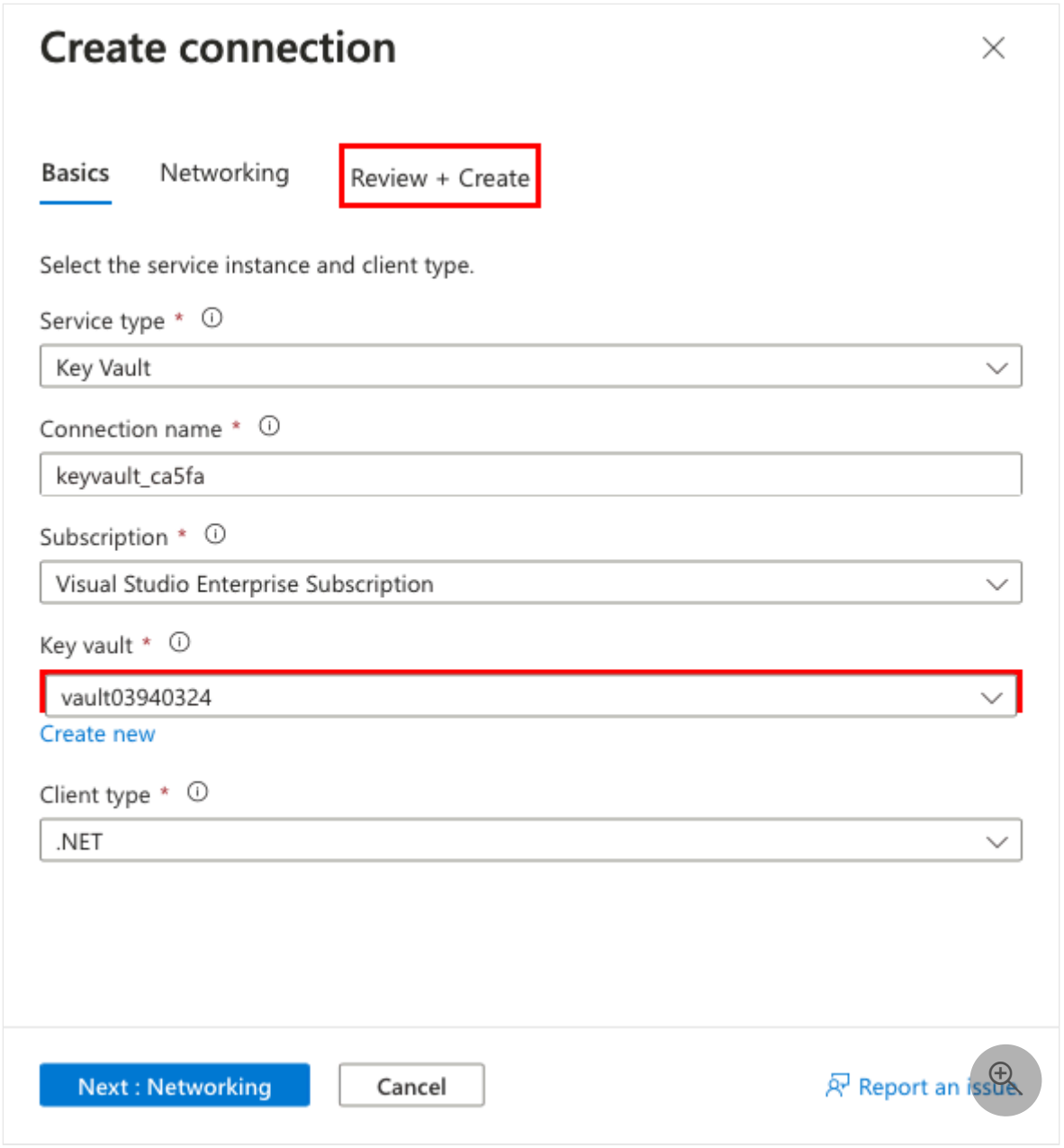
Step 4:

1. In the top search bar, type *msdocs-core-sql*, then the App Service resource called **msdocs-core-sql-XYZ**.
2. In the App Service page, in the left menu, select **Settings** > **Service Connector**. There are already two connectors, which the app creation wizard created for you.
3. Select checkbox next to the SQL Database connector, then select **Edit**.
4. Select the **Authentication** tab.
5. In **Password**, paste the password you copied earlier.
6. Select **Store Secret in Key Vault**.
7. Under **Key Vault Connection**, select **Create new**. A **Create connection** dialog is opened on top of the edit dialog.



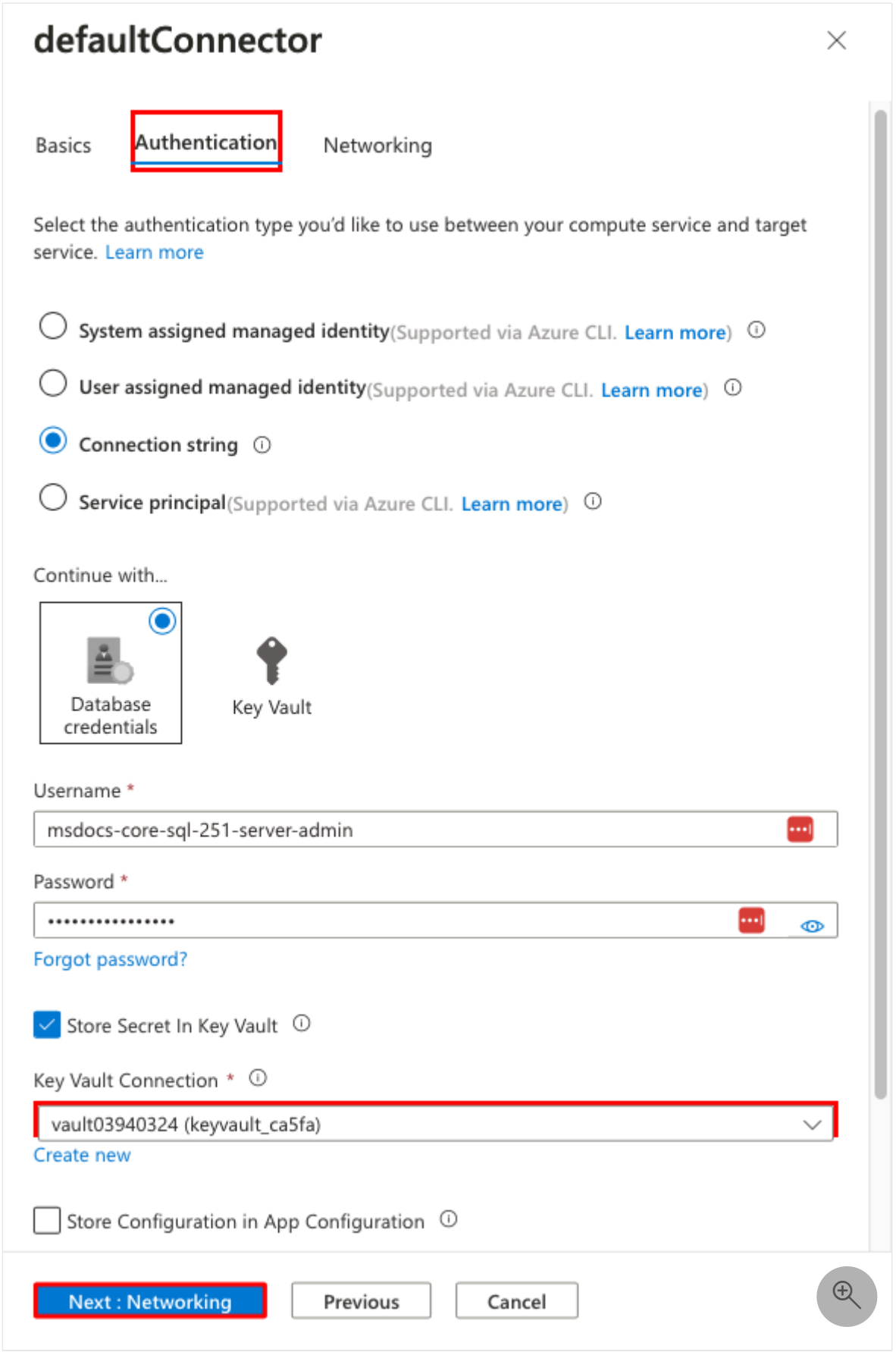
Step 5: In the **Create connection** dialog for the Key Vault connection:

1. In **Key Vault**, select the key vault you created earlier.
2. Select **Review + Create**. You should see that **System assigned managed identity** is set to **Selected**.
3. When validation completes, select **Create**.



Step 6: You're back in the edit dialog for **defaultConnector**.

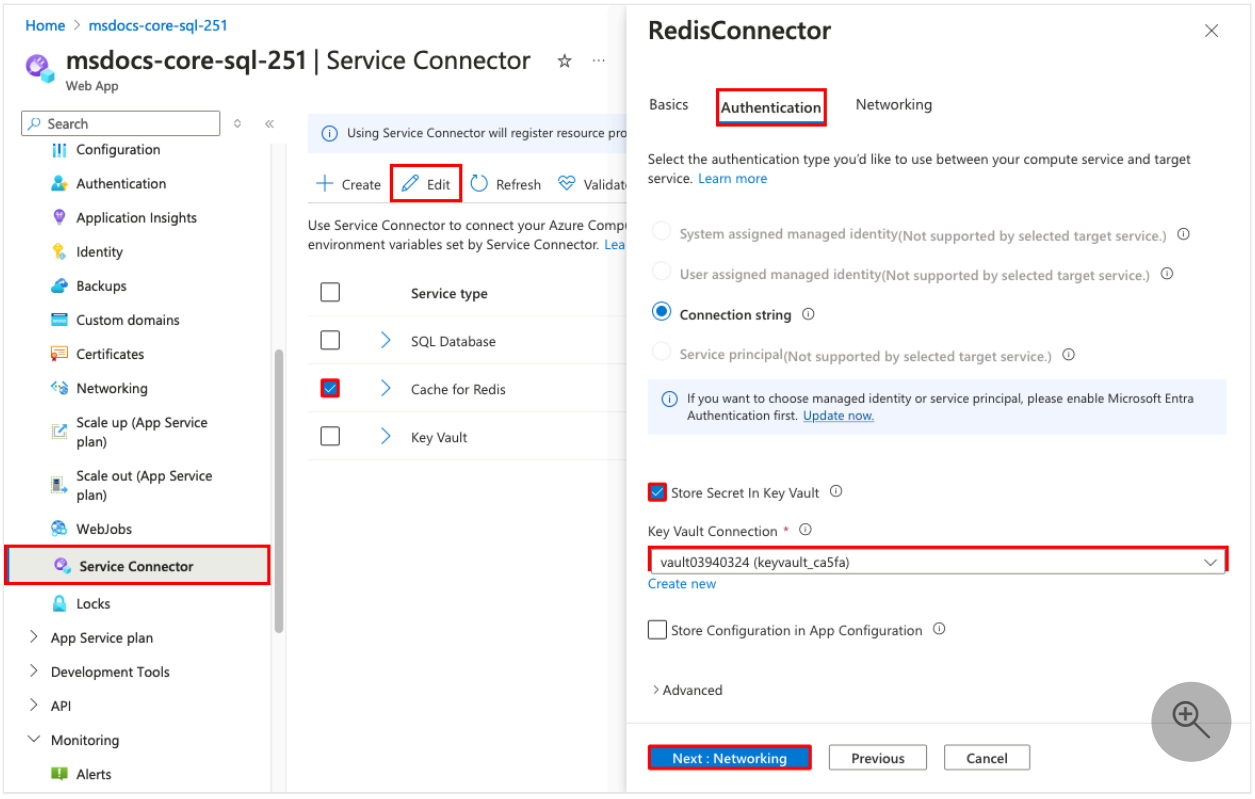
1. In the **Authentication** tab, wait for the key vault connector to be created. When it's finished, the **Key Vault Connection** dropdown automatically selects it.
2. Select **Next: Networking**.
3. Select **Configure firewall rules to enable access to target service**. The app creation wizard already secured the SQL database with a private endpoint.
4. Select **Save**. Wait until the **Update succeeded** notification appears.



Step 7: In the Service Connectors page:

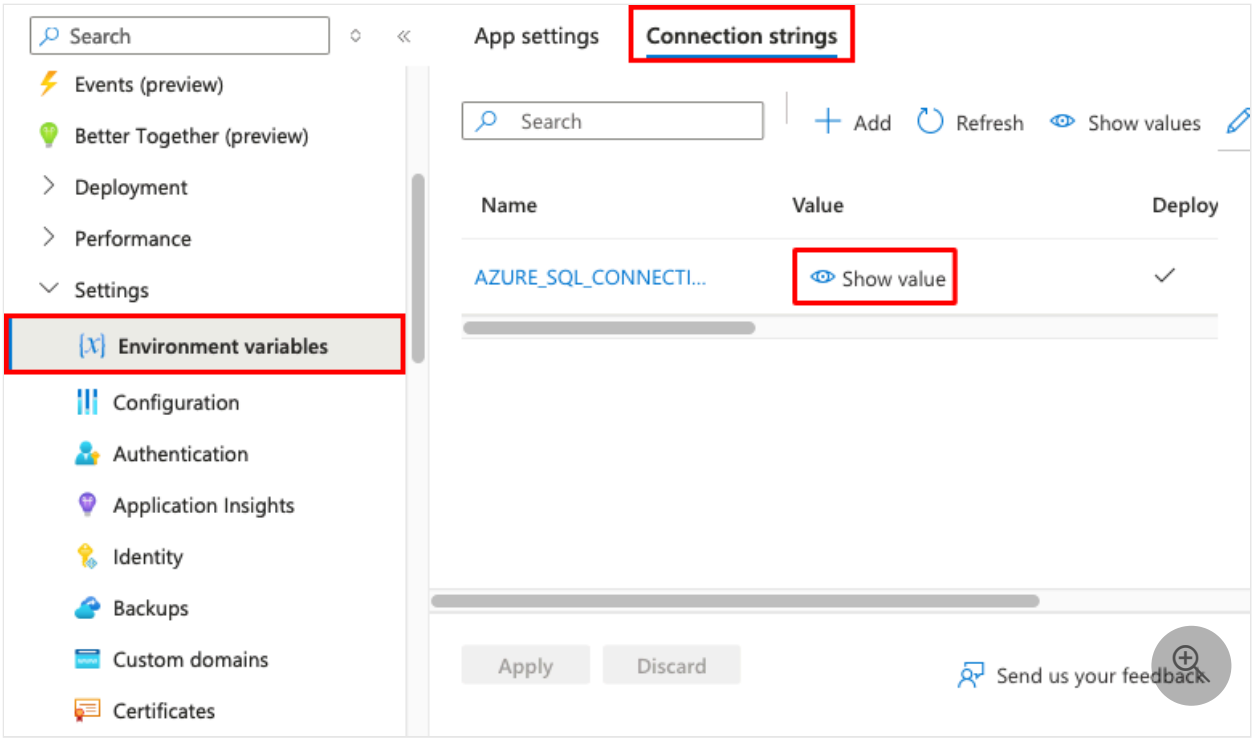
1. Select checkbox next to the Cache for Redis connector, then select **Edit**.
2. Select the **Authentication** tab.
3. Select **Store Secret in Key Vault**.

4. Under **Key Vault Connection**, select the key vault you created.
5. Select **Next: Networking**.
6. Select **Configure firewall rules to enable access to target service**. The app creation wizard already secured the SQL database with a private endpoint.
7. Select **Save**. Wait until the **Update succeeded** notification appears.



Step 8: To verify your changes:

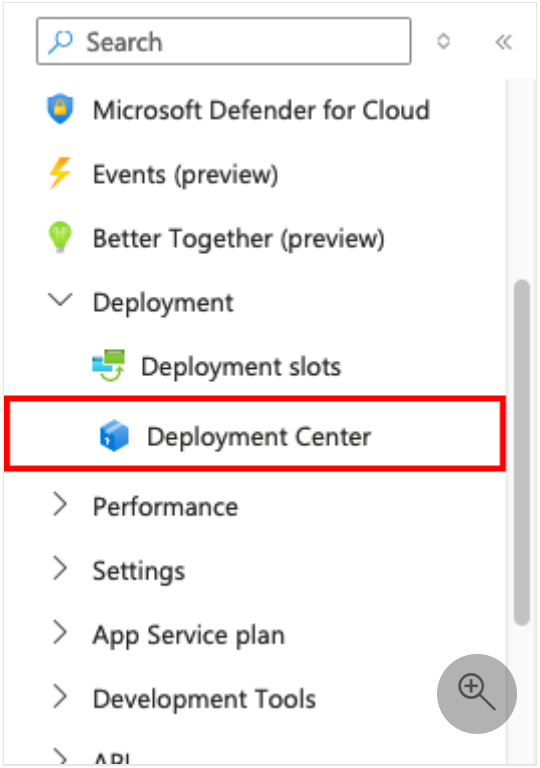
1. From the left menu, select **Environment variables** > **Connection strings** again.
2. Next to **AZURE_SQL_CONNECTIONSTRING**, select **Show value**. The value should be `@Microsoft.KeyVault(...)`, which means that it's a key vault reference because the secret is now managed in the key vault.
3. To verify the Redis connection string, select the **App setting** tab. Next to **AZURE_REDIS_CONNECTIONSTRING**, select **Show value**. The value should be `@Microsoft.KeyVault(...)` too.



4. Deploy sample code

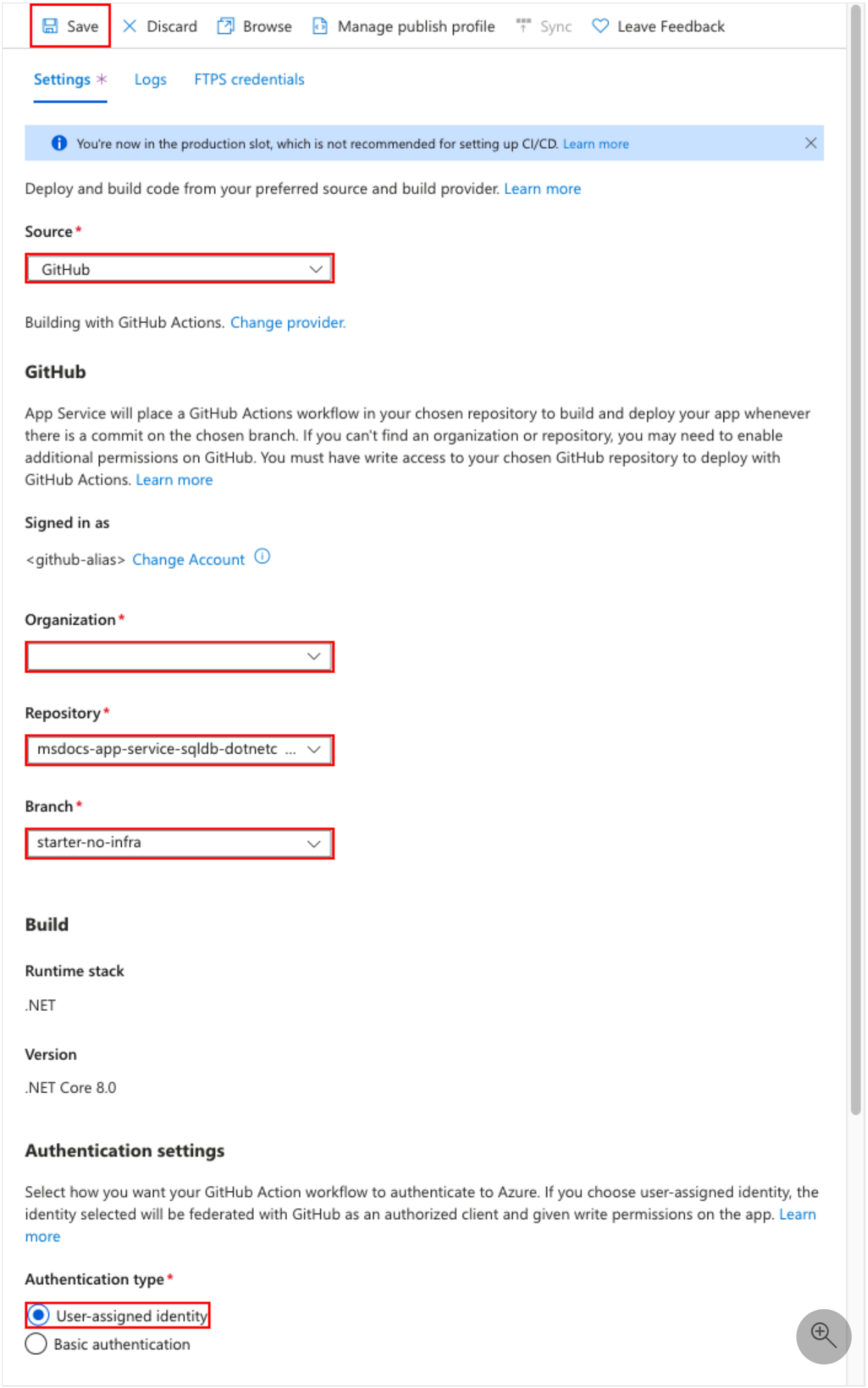
In this step, you configure GitHub deployment using GitHub Actions. It's just one of many ways to deploy to App Service, but also a great way to have continuous integration in your deployment process. By default, every `git push` to your GitHub repository kicks off the build and deploy action.

Step 1: In the left menu, select **Deployment** > **Deployment Center**.

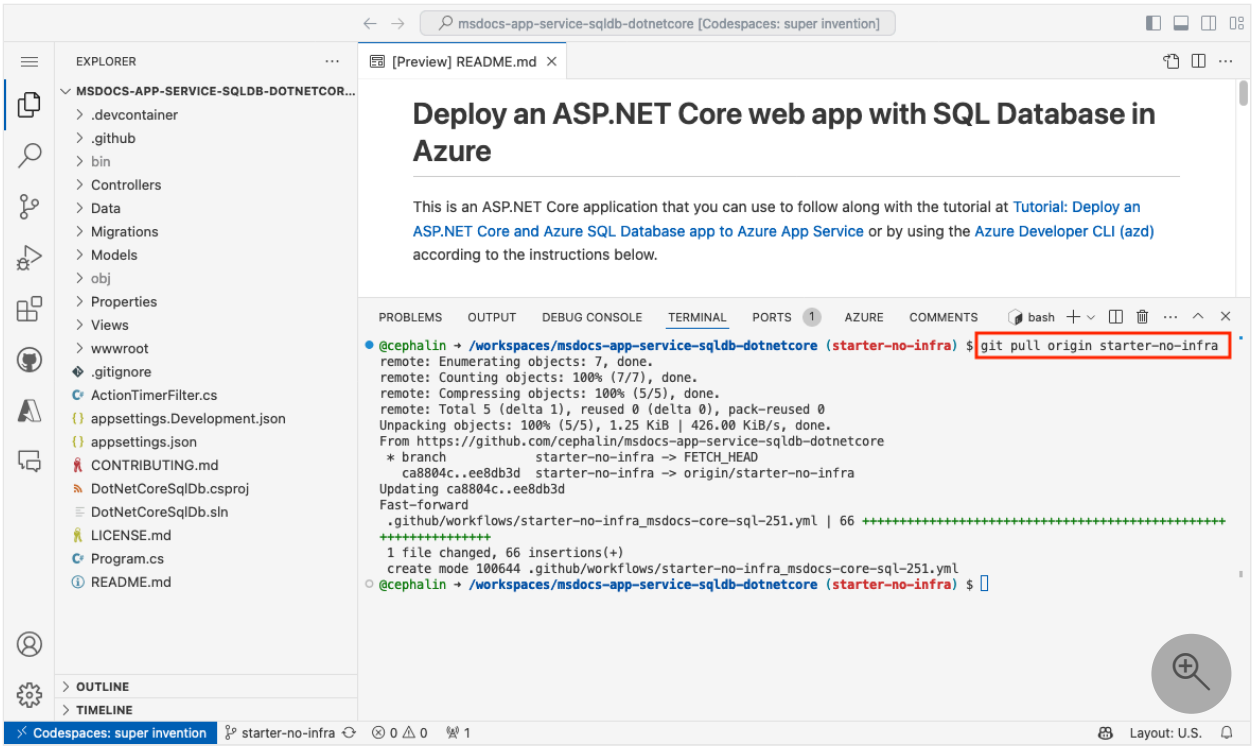


Step 2: In the Deployment Center page:

1. In **Source**, select **GitHub**. By default, **GitHub Actions** is selected as the build provider.
2. Sign in to your GitHub account and follow the prompt to authorize Azure.
3. In **Organization**, select your account.
4. In **Repository**, select **msdocs-app-service-sqldb-dotnetcore**.
5. In **Branch**, select **starter-no-infra**. This is the same branch that you worked in with your sample app, without any Azure-related files or configuration.
6. For **Authentication type**, select **User-assigned identity**.
7. In the top menu, select **Save**. App Service commits a workflow file into the chosen GitHub repository, in the `.github/workflows` directory. By default, the deployment center creates a user-assigned identity for the workflow to authenticate using Microsoft Entra (OIDC authentication). For alternative authentication options, see Deploy to App Service using GitHub Actions.

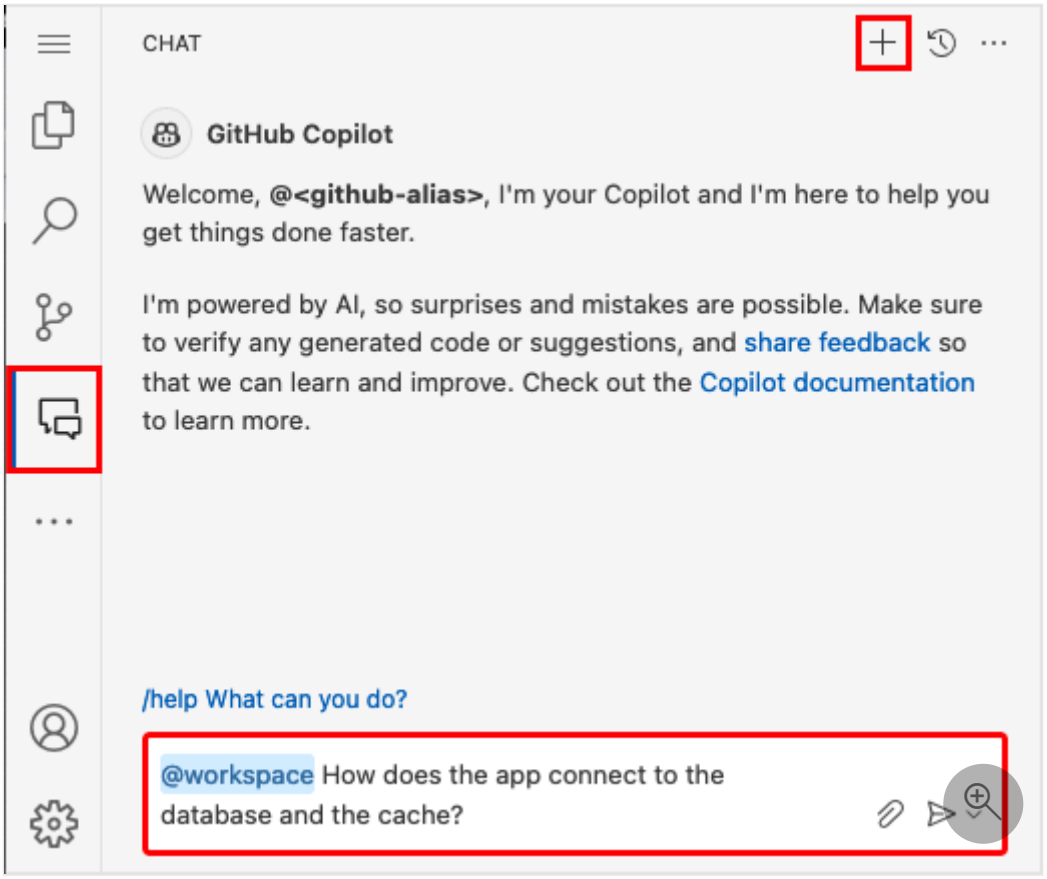

Save
Discard
Browse
Manage publish profile
Sync
Leave Feedback
Settings
Logs
FTPS credentials
You're now in the production slot, which is not recommended for setting up CI/CD. Learn more
Deploy and build code from your preferred source and build provider. Learn more
Source *
GitHub
Building with GitHub Actions. Change provider.
GitHub
App Service will place a GitHub Actions workflow in your chosen repository to build and deploy your app whenever there is a commit on the chosen branch. If you can't find an organization or repository, you may need to enable additional permissions on GitHub. You must have write access to your chosen GitHub repository to deploy with GitHub Actions. Learn more
Signed in as
<github-alias> Change Account
Organization *
Repository *
msdocs-app-service-sqldb-dotnetc ...
Branch *
starter-no-infra
Build
Runtime stack
.NET
Version
.NET Core 8.0
Authentication settings
Select how you want your GitHub Action workflow to authenticate to Azure. If you choose user-assigned identity, the identity selected will be federated with GitHub as an authorized client and given write permissions on the app. Learn more
Authentication type *
User-assigned identity
Basic authentication

Step 3: Back in the GitHub codespace of your sample fork, run `git pull origin starter-no-infra`. This pulls the newly committed workflow file into your codespace.



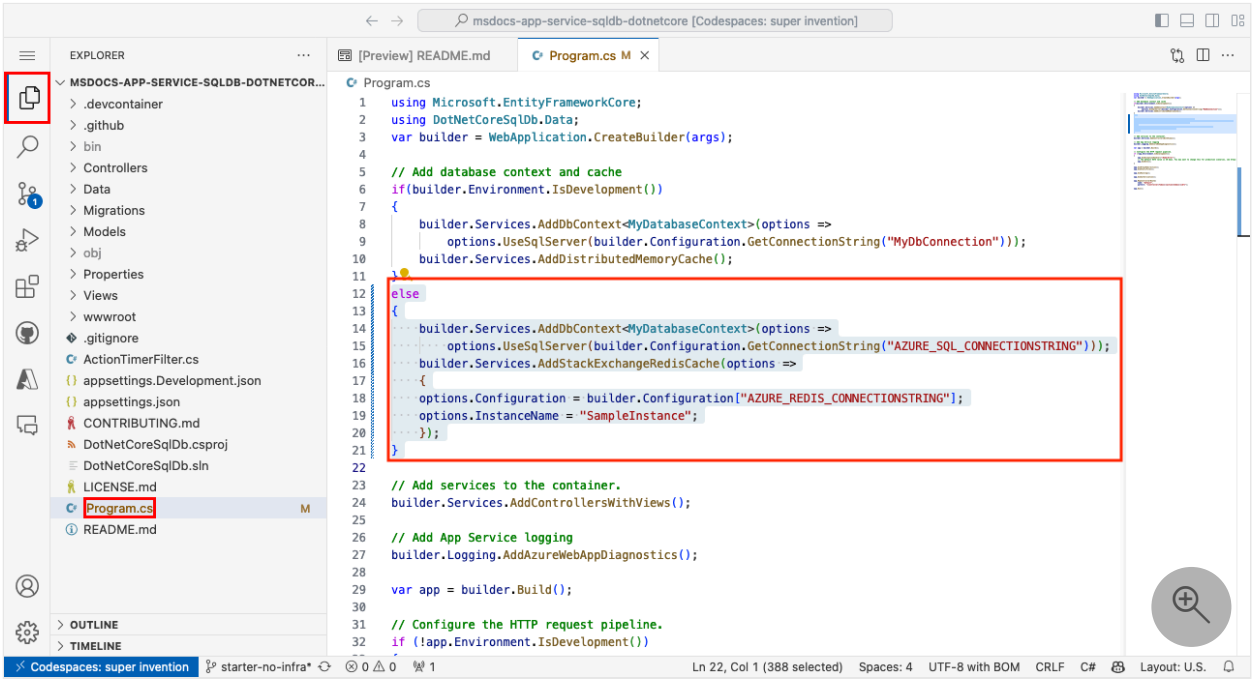
Step 4 (Option 1: with GitHub Copilot):

1. Start a new chat session by selecting the **Chat** view, then selecting **+**.
2. Ask, "*@workspace How does the app connect to the database and the cache?*" Copilot might give you some explanation about the `MyDatabaseContext` class and how it's configured in *Program.cs*.
3. Ask, "In production mode, I want the app to use the connection string called AZURE_SQL_CONNECTIONSTRING for the database and the app setting called AZURE_REDIS_CONNECTIONSTRING*." Copilot might give you a code suggestion similar to the one in the **Option 2: without GitHub Copilot** steps below and even tell you to make the change in the *Program.cs* file.
4. Open *Program.cs* in the explorer and add the code suggestion. GitHub Copilot doesn't give you the same response every time, and it's not always correct. You might need to ask more questions to fine-tune its response. For tips, see What can I do with GitHub Copilot in my codespace?.



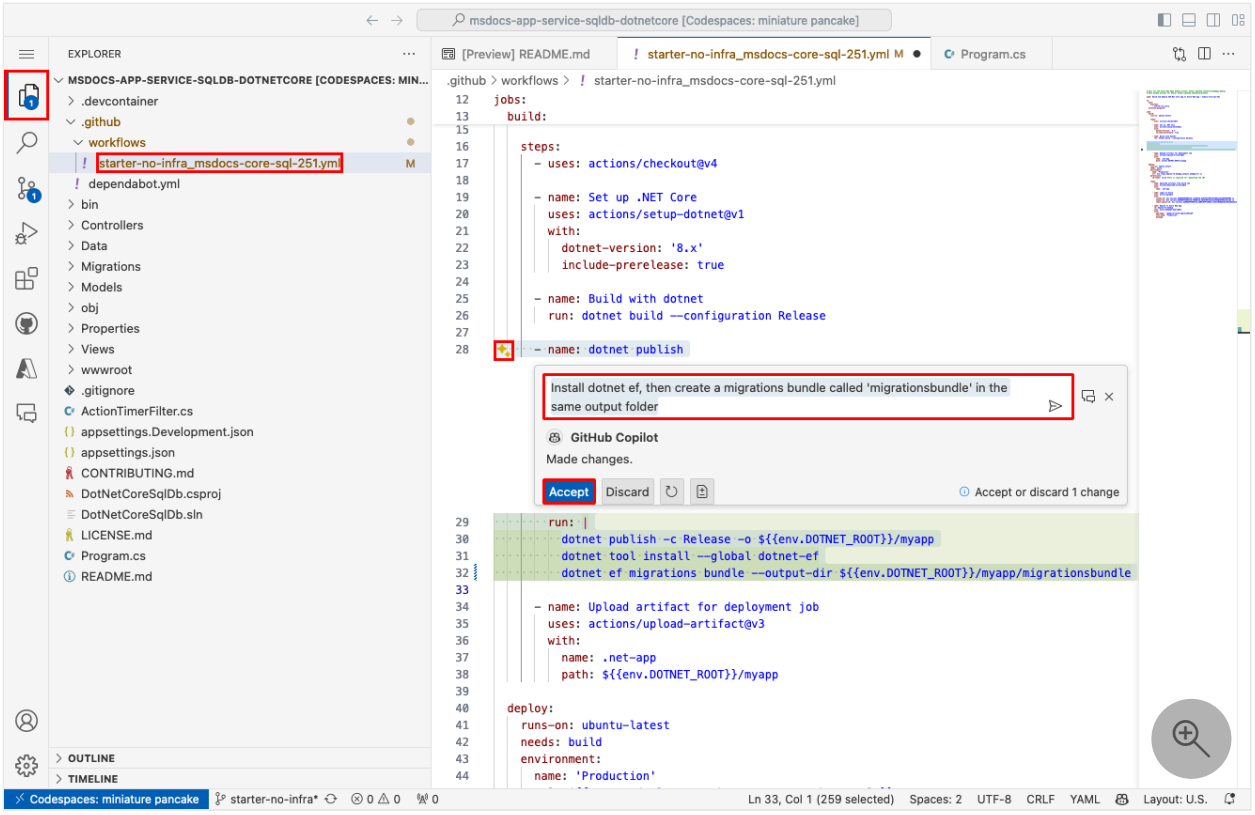
Step 4 (Option 2: without GitHub Copilot):

1. Open *Program.cs* in the explorer.
2. Find the commented code (lines 12-21) and uncomment it. This code connects to the database by using `AZURE_SQL_CONNECTIONSTRING` and connects to the Redis cache by using the app setting `AZURE_REDIS_CONNECTIONSTRING`.



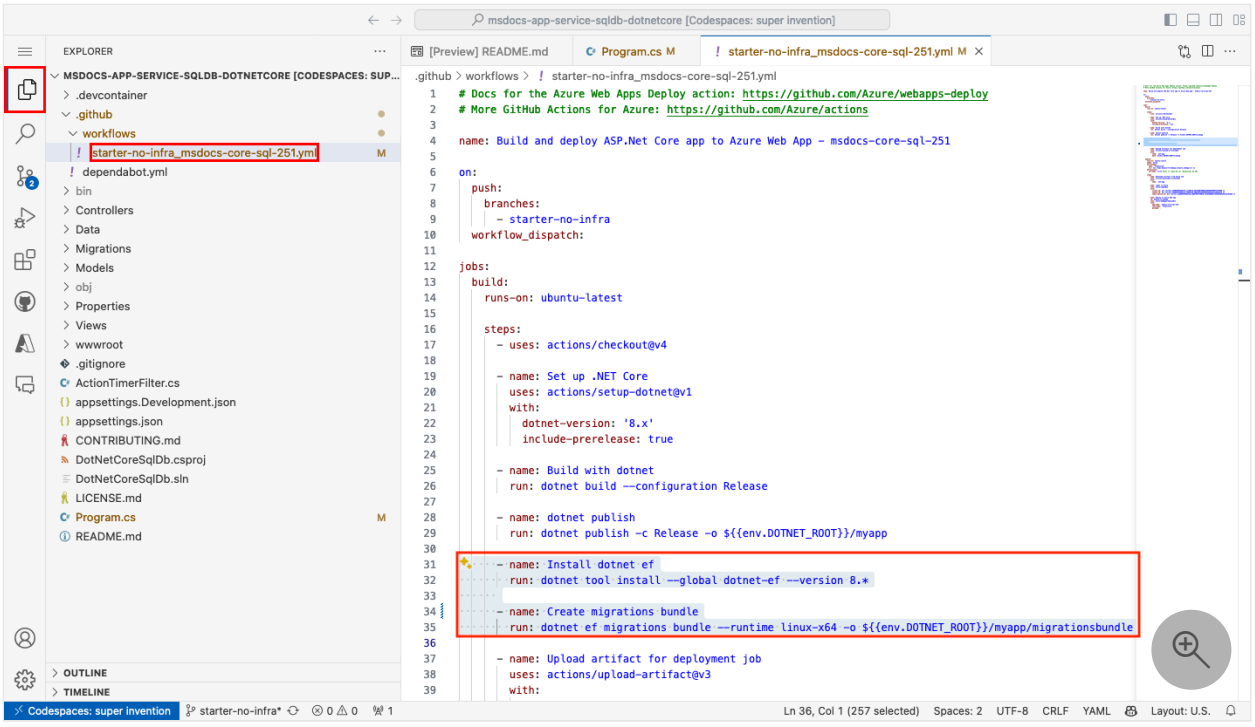
Step 5 (Option 1: with GitHub Copilot):

1. Open *.github/workflows/starter-no-infra_msdocs-core-sql-XYZ* in the explorer. This file was created by the App Service create wizard.
2. Highlight the `dotnet publish` step and select ✨.
3. Ask Copilot, "*Install dotnet ef, then create a migrations bundle in the same output folder.*"
4. If the suggestion is acceptable, select **Accept**. GitHub Copilot doesn't give you the same response every time, and it's not always correct. You might need to ask more questions to fine-tune its response. For tips, see What can I do with GitHub Copilot in my codespace?.



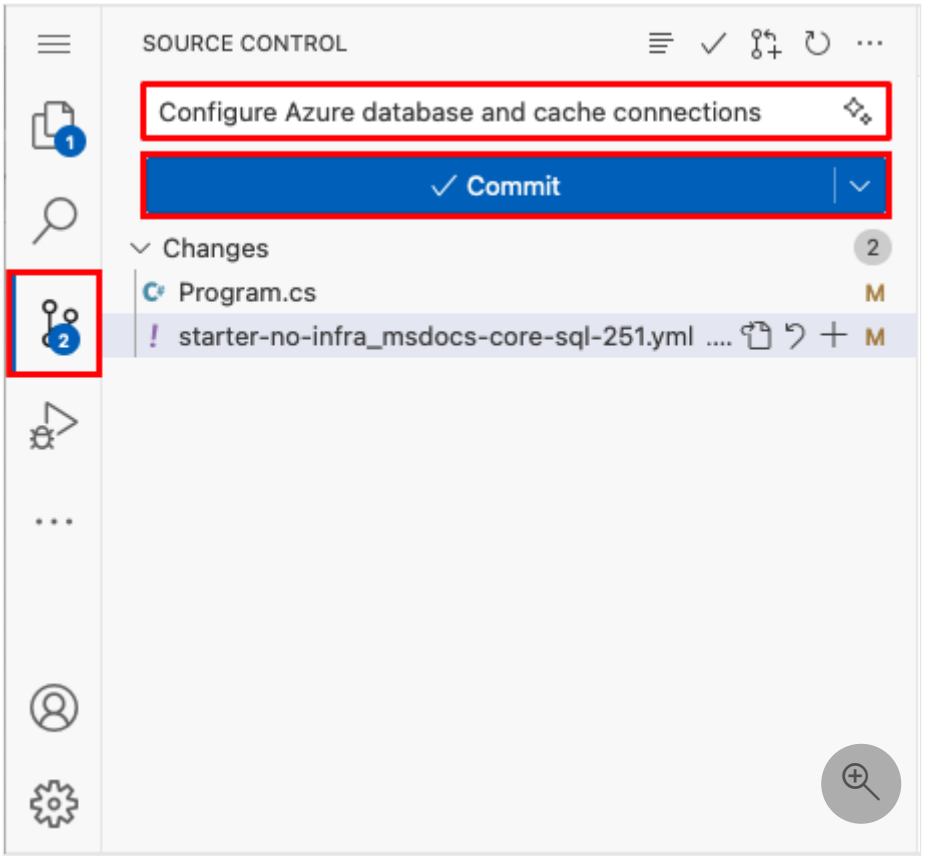
Step 5 (Option 2: without GitHub Copilot):

1. Open *.github/workflows/starter-no-infra_msdocs-core-sql-XYZ* in the explorer. This file was created by the App Service create wizard.
2. Under the `dotnet publish` step, add a step to install the Entity Framework Core tool with the command `dotnet tool install -g dotnet-ef --version 8.*`.
3. Under the new step, add another step to generate a database migration bundle in the deployment package: `dotnet ef migrations bundle --runtime linux-x64 -o ${{env.DOTNET_ROOT}}/myapp/migrationsbundle`. The migration bundle is a self-contained executable that you can run in the production environment without needing the .NET SDK. The App Service linux container only has the .NET runtime and not the .NET SDK.



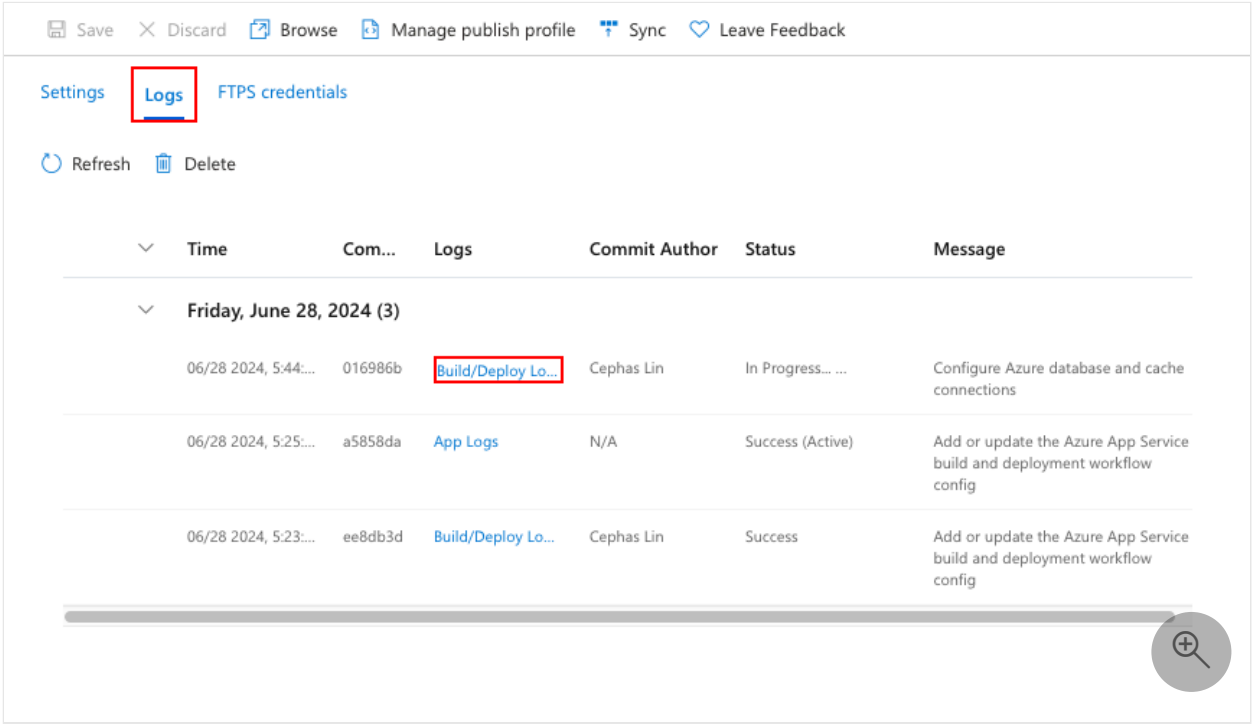
Step 6:

1. Select the **Source Control** extension.
2. In the textbox, type a commit message like `Configure Azure database and cache connections`. Or, select ✨ and let GitHub Copilot generate a commit message for you.
3. Select **Commit**, then confirm with **Yes**.
4. Select **Sync changes 1**, then confirm with **OK**.



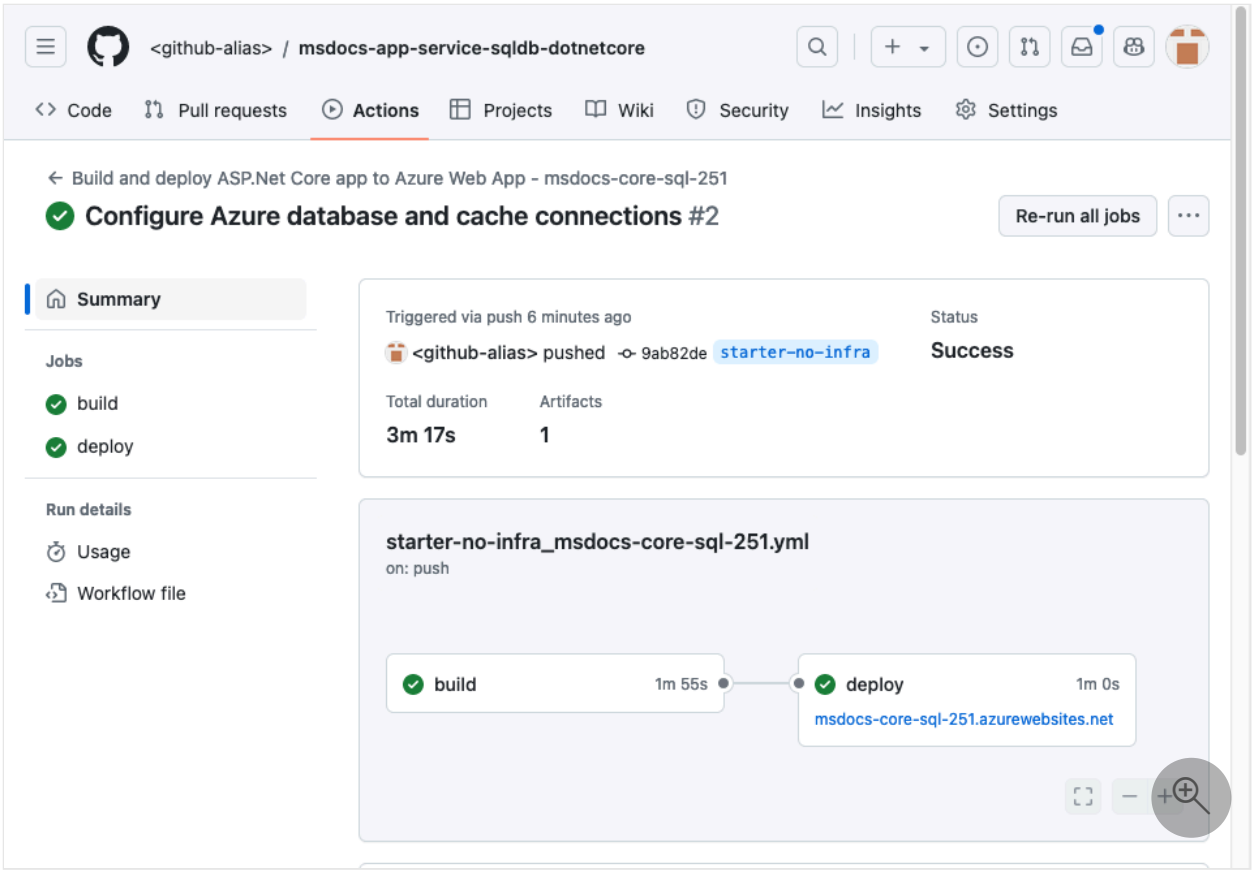
Step 7: Back in the Deployment Center page in the Azure portal:

1. Select the **Logs** tab, then select **Refresh** to see the new deployment run.
2. In the log item for the deployment run, select the **Build/Deploy Logs** entry with the latest timestamp.



Step 8: You're taken to your GitHub repository and see that the GitHub action is running. The workflow file defines two separate stages, build and deploy. Wait for the

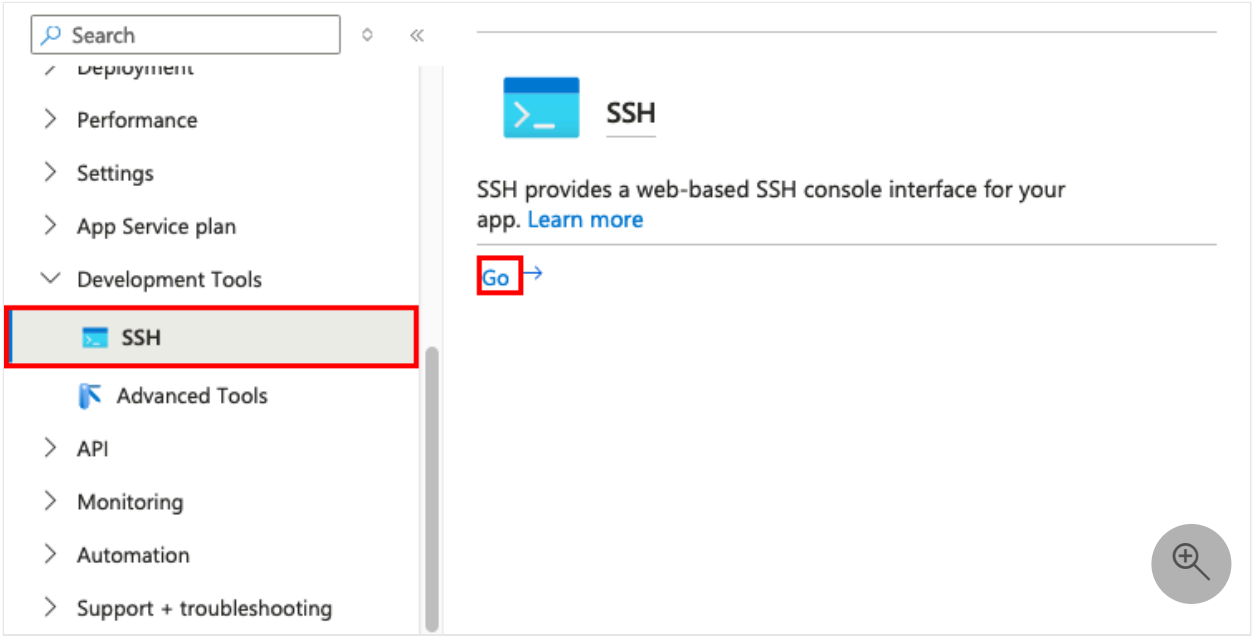
GitHub run to show a status of **Success**. It takes about 5 minutes.



5. Generate database schema

With the SQL Database protected by the virtual network, the easiest way to run dotnet database migrations is in an SSH session with the App Service container.

Step 1: Back in the App Service page, in the left menu, select **Development Tools** > **SSH**, then select **Go**.



Step 2: In the SSH terminal:

1. Run `cd /home/site/wwwroot`. Here are all your deployed files.
2. Run the migration bundle that the GitHub workflow generated, with the command `./migrationsbundle -- --environment Production`. If it succeeds, App Service is connecting successfully to the SQL Database. Remember that `--environment Production` corresponds to the code changes you made in *Program.cs*.



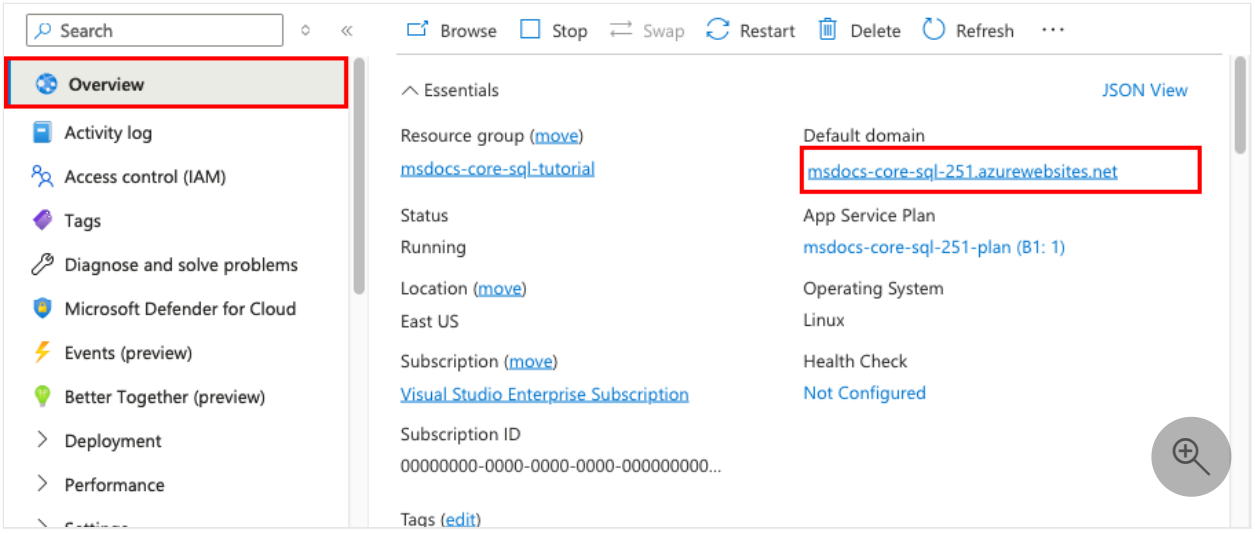
In the SSH session, only changes to files in `/home` can persist beyond app restarts. Changes outside of `/home` aren't persisted.

Having issues? Check the Troubleshooting section.

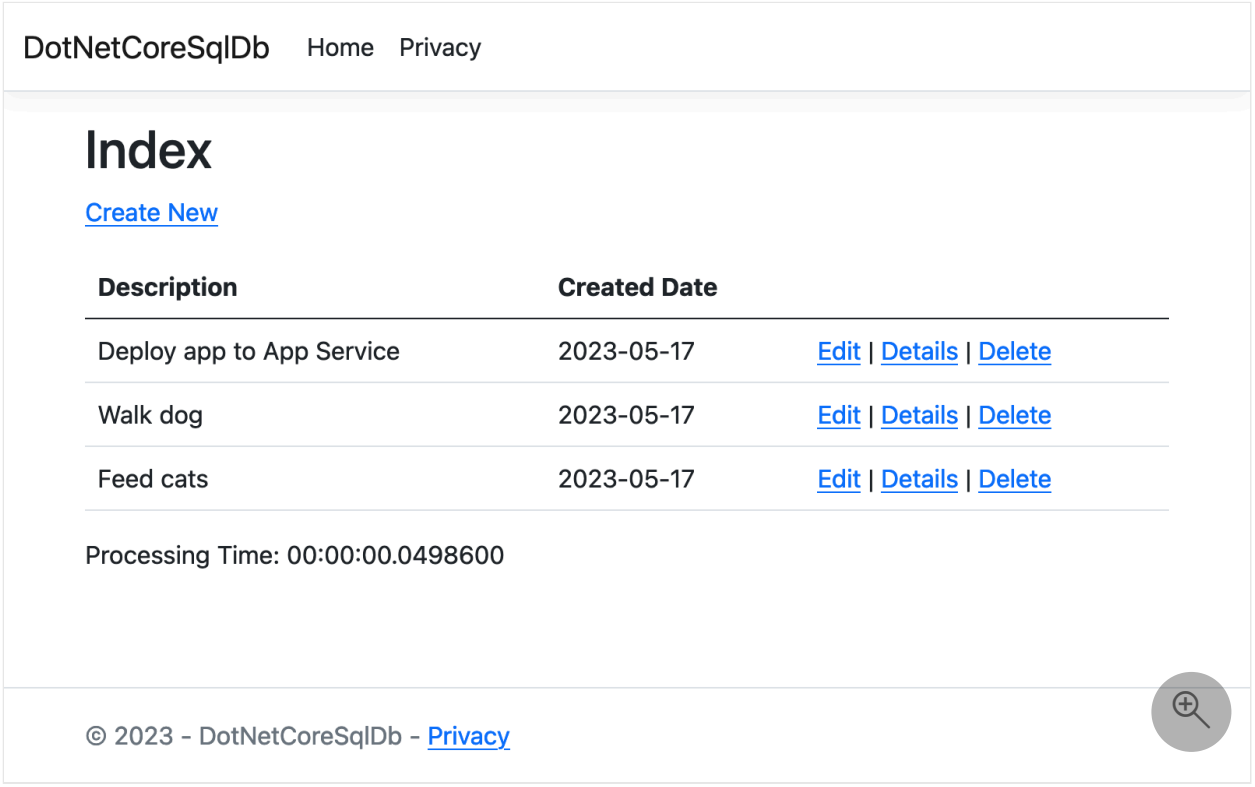
6. Browse to the app

Step 1: In the App Service page:

1. From the left menu, select **Overview**.
2. Select the URL of your app.



Step 2: Add a few tasks to the list. Congratulations, you're running a secure data-driven ASP.NET Core app in Azure App Service.



> **Tip**
>
> The sample application implements the cache-aside pattern. When you visit a data view for the second time, or reload the same page after making data changes,

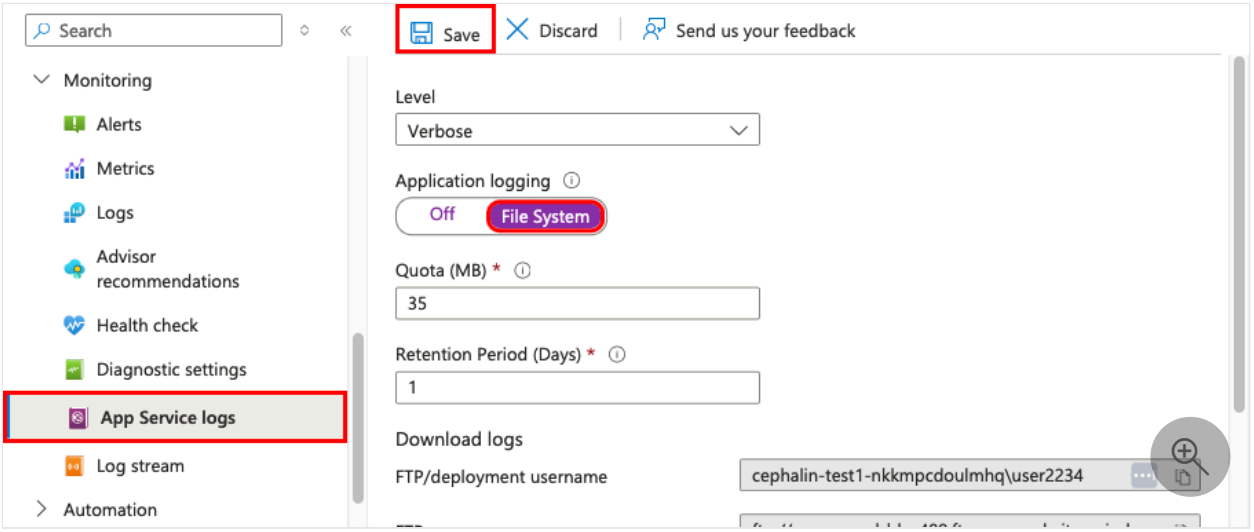
Processing time in the webpage shows a much faster time because it's loading the data from the cache instead of the database.

7. Stream diagnostic logs

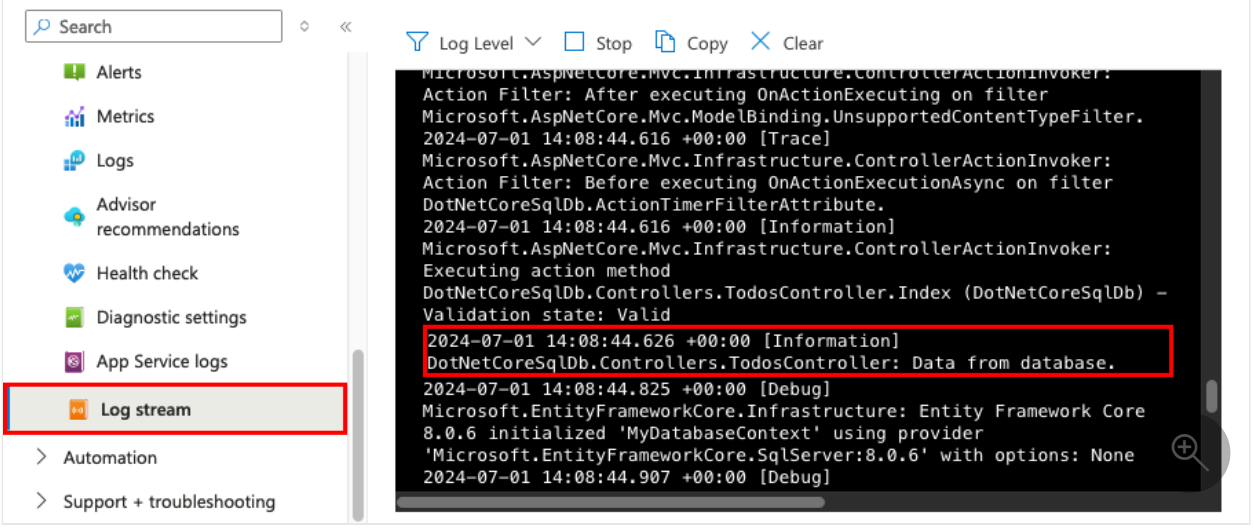
Azure App Service captures all messages logged to the console to assist you in diagnosing issues with your application. The sample app outputs console log messages in each of its endpoints to demonstrate this capability.

Step 1: In the App Service page:

1. From the left menu, select **Monitoring** > **App Service logs**.
2. Under **Application logging**, select **File System**, then select **Save**.



Step 2: From the left menu, select **Log stream**. You see the logs for your app, including platform logs and logs from inside the container.

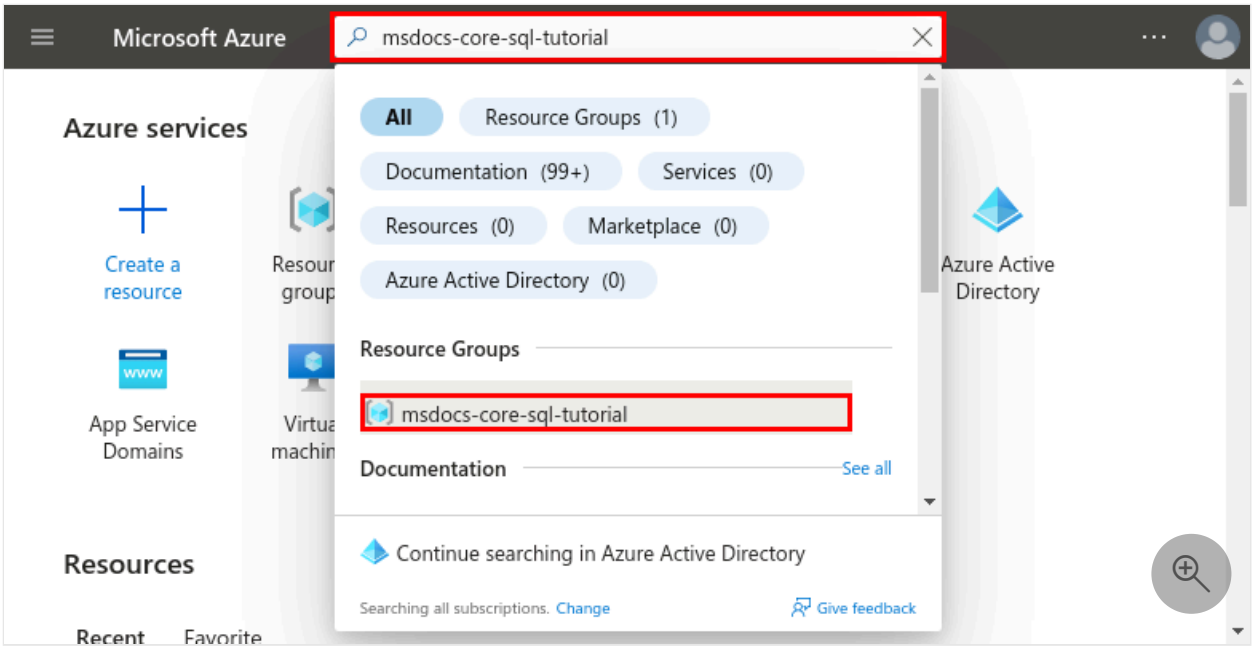


8. Clean up resources

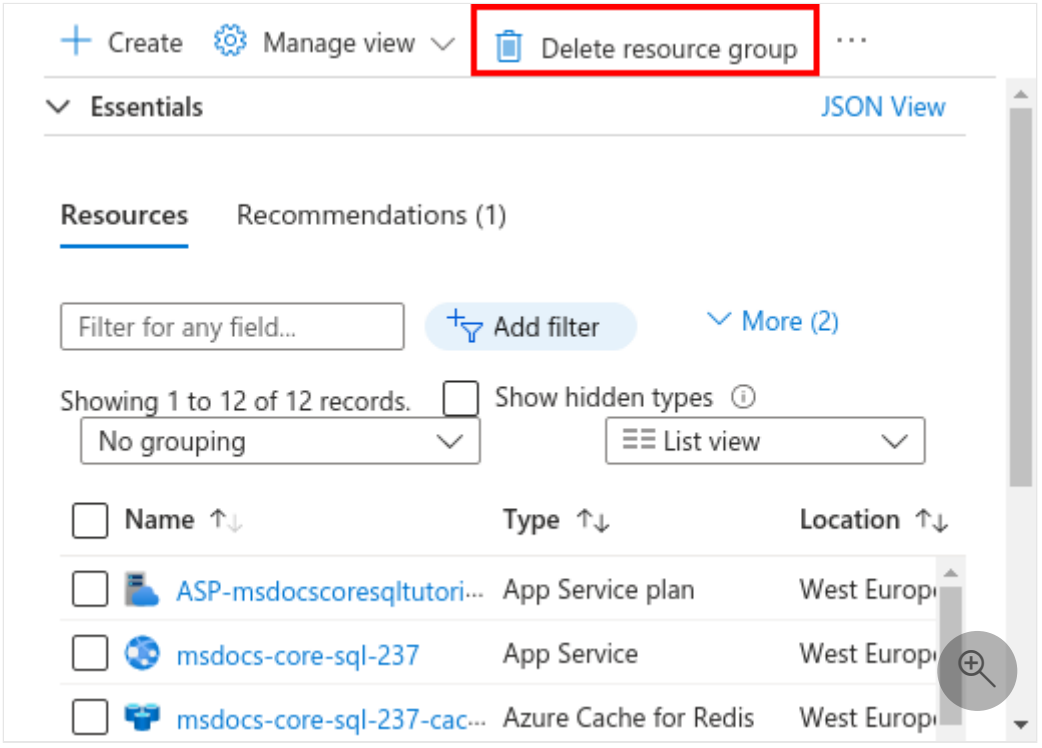
When you're finished, you can delete all of the resources from your Azure subscription by deleting the resource group.

Step 1: In the search bar at the top of the Azure portal:

1. Enter the resource group name.
2. Select the resource group.

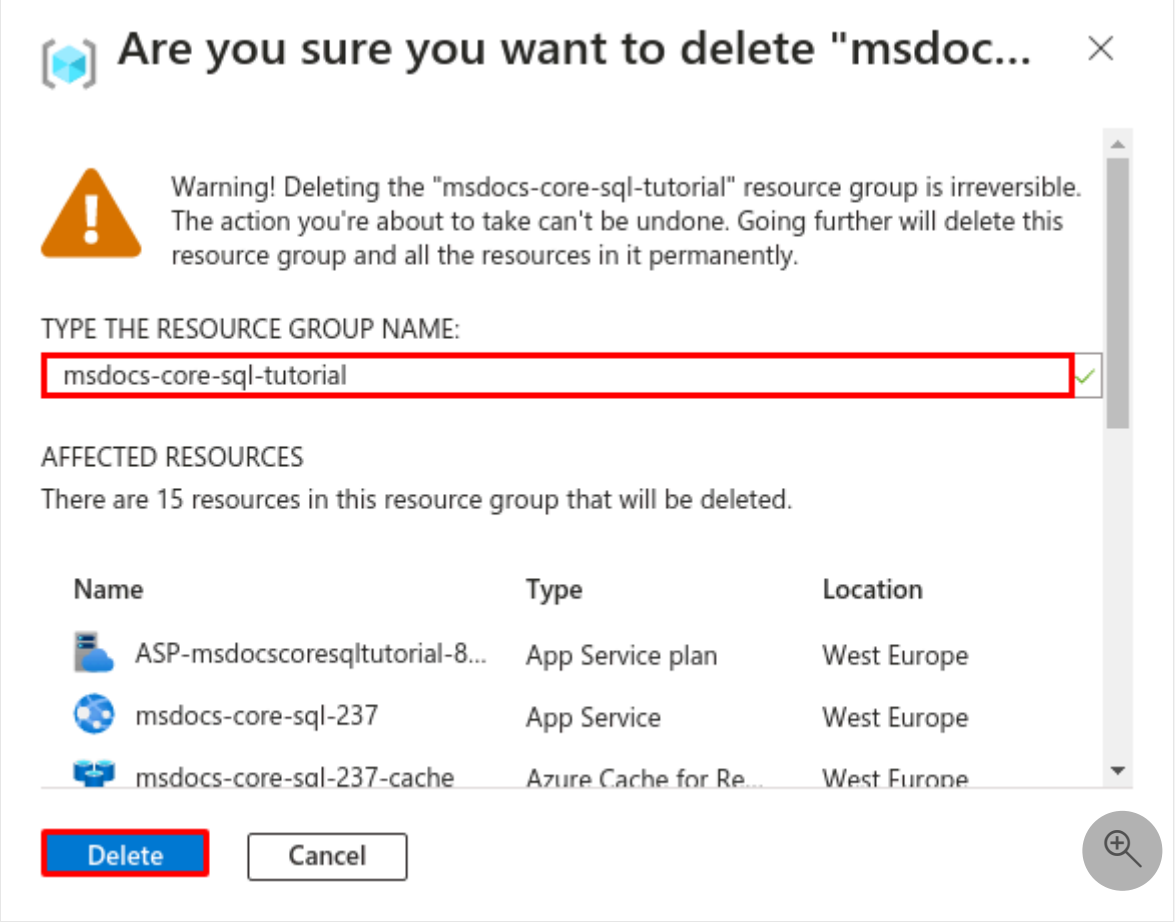


Step 2: In the resource group page, select **Delete resource group**.



Step 3:

1. Enter the resource group name to confirm your deletion.
2. Select **Delete**.



:

Troubleshooting

- The portal deployment view for Azure SQL Database shows a Conflict status
- In the Azure portal, the log stream UI for the web app shows network errors
- The SSH session in the browser shows SSH CONN CLOSED
- The portal log stream page shows Connected! but no logs

The portal deployment view for Azure SQL Database shows a Conflict status

Depending on your subscription and the region you select, you might see the deployment status for Azure SQL Database to be `Conflict`, with the following message in Operation details:

```
Location '<region>' is not accepting creation of new Windows Azure SQL Database servers at this time.
```

This error is most likely caused by a limit on your subscription for the region you select. Try choosing a different region for your deployment.

In the Azure portal, the log stream UI for the web app shows network errors

You might see this error:

```
Unable to open a connection to your app. This may be due to any network
security groups or IP restriction rules that you have placed on your app. To
use log streaming, please make sure you are able to access your app directly
from your current network.
```

This is usually a transient error when the app is first started. Wait a few minutes and check again.

The SSH session in the browser shows `SSH CONN CLOSED`

It takes a few minutes for the Linux container to start up. Wait a few minutes and check again.

The portal log stream page shows `Connected!` but no logs

After you configure diagnostic logs, the app is restarted. You might need to refresh the page for the changes to take effect in the browser.

Frequently asked questions

- How much does this setup cost?
- How do I connect to the Azure SQL Database server that's secured behind the virtual network with other tools?
- How does local app development work with GitHub Actions?
- How do I debug errors during the GitHub Actions deployment?
- How do I change the SQL Database connection to use a managed identity instead?
- I don't have permissions to create a user-assigned identity
- What can I do with GitHub Copilot in my codespace?

How much does this setup cost?

Pricing for the created resources is as follows:

- The App Service plan is created in **Basic** tier and can be scaled up or down. See App Service pricing.
- The Azure SQL Database is created in general-purpose, serverless tier on Standard-series hardware with the minimum cores. There's a small cost and can be distributed to other regions. You can minimize cost even more by reducing its maximum size, or you can scale it up by adjusting the serving tier, compute tier, hardware configuration, number of cores, database size, and zone redundancy. See Azure SQL Database pricing.
- The Azure Cache for Redis is created in **Basic** tier with the minimum cache size. There's a small cost associated with this tier. You can scale it up to higher performance tiers for higher availability, clustering, and other features. See Azure Cache for Redis pricing.
- The virtual network doesn't incur a charge unless you configure extra functionality, such as peering. See Azure Virtual Network pricing.
- The private DNS zone incurs a small charge. See Azure DNS pricing.

How do I connect to the Azure SQL Database server that's secured behind the virtual network with other tools?

- For basic access from a command-line tool, you can run `sqlcmd` from the app's SSH terminal. The app's container doesn't come with `sqlcmd`, so you must install it manually. Remember that the installed client doesn't persist across app restarts.
- To connect from a SQL Server Management Studio client or from Visual Studio, your machine must be within the virtual network. For example, it could be an Azure VM that's connected to one of the subnets, or a machine in an on-premises network that has a site-to-site VPN connection with the Azure virtual network.

How does local app development work with GitHub Actions?

Take the autogenerated workflow file from App Service as an example, each `git push` kicks off a new build and deployment run. From a local clone of the GitHub repository, you make the desired updates push it to GitHub. For example:

terminal

```
git add .
git commit -m "<some-message>"
git push origin main
```

How do I debug errors during the GitHub Actions deployment?

If a step fails in the autogenerated GitHub workflow file, try modifying the failed command to generate more verbose output. For example, you can get more output from any of the `dotnet` commands by adding the `-v` option. Commit and push your changes to trigger another deployment to App Service.

I don't have permissions to create a user-assigned identity

See Set up GitHub Actions deployment from the Deployment Center.

How do I change the SQL Database connection to use a managed identity instead?

The default connection string to the SQL database is managed by Service Connector, with the name *defaultConnector*, and it uses SQL authentication. To replace it with a connection that uses a managed identity, run the following commands in the cloud shell after replacing the placeholders:

Azure CLI

```
az extension add --name serviceconnector-passwordless --upgrade
az sql server update --enable-public-network true
az webapp connection delete sql --connection defaultConnector --resource-
group <group-name> --name <app-name>
az webapp connection create sql --connection defaultConnector --resource-
group <group-name> --name <app-name> --target-resource-group <group-name> --
server <database-server-name> --database <database-name> --client-type
dotnet --system-identity --config-connstr true
az sql server update --enable-public-network false
```

By default, the command `az webapp connection create sql --client-type dotnet --system-identity --config-connstr` does the following:

- Sets your user as the Microsoft Entra ID administrator of the SQL database server.
- Create a system-assigned managed identity and grants it access to the database.
- Generates a passwordless connection string called `AZURE_SQL_CONNECTIONGSTRING`, which your app is already using at the end of the tutorial.

Your app should now have connectivity to the SQL database. For more information, see Tutorial: Connect to Azure databases from App Service without secrets using a managed

identity.

> **Tip**
>
> **Don't want to enable public network connection?** You can skip `az sql server update --enable-public-network true` by running the commands from an **Azure cloud shell that's integrated with your virtual network** if you have the **Owner** role assignment on your subscription.
>
> To grant the identity the required access to the database that's secured by the virtual network, `az webapp connection create sql` needs direct connectivity with Entra ID authentication to the database server. By default, the Azure cloud shell doesn't have this access to the network-secured database.

What can I do with GitHub Copilot in my codespace?

You might have noticed that the GitHub Copilot chat view was already there for you when you created the codespace. For your convenience, we include the GitHub Copilot chat extension in the container definition (see *.devcontainer/devcontainer.json*). However, you need a GitHub Copilot account (30-day free trial available).

A few tips for you when you talk to GitHub Copilot:

- In a single chat session, the questions and answers build on each other and you can adjust your questions to fine-tune the answer you get.
- By default, GitHub Copilot doesn't have access to any file in your repository. To ask questions about a file, open the file in the editor first.
- To let GitHub Copilot have access to all of the files in the repository when preparing its answers, begin your question with `@workspace`. For more information, see Use the @workspace agent.
- In the chat session, GitHub Copilot can suggest changes and (with `@workspace`) even where to make the changes, but it's not allowed to make the changes for you. It's up to you to add the suggested changes and test it.

Here are some other things you can say to fine-tune the answer you get:

- I want this code to run only in production mode.
- I want this code to run only in Azure App Service and not locally.
- The --output-path parameter seems to be unsupported.

Related content

Advance to the next tutorial to learn how to secure your app with a custom domain and certificate.

Secure with custom domain and certificate

Or, check out other resources:

Tutorial: Connect to SQL Database from App Service without secrets using a managed identity

Configure ASP.NET Core app

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Create your first pipeline

Article • 04/03/2024

Azure DevOps Services | Azure DevOps Server 2022 - Azure DevOps Server 2019

This is a step-by-step guide to using Azure Pipelines to build a sample application from a Git repository. This guide uses YAML pipelines configured with the YAML pipeline editor. If you'd like to use Classic pipelines instead, see Define your Classic pipeline. For guidance on using TFVC, see Build TFVC repositories.

Prerequisites - Azure DevOps

Make sure you have the following items:

- A GitHub account where you can create a repository. Create one for free.
- An Azure DevOps organization. Create one for free. If your team already has one, then make sure you're an administrator of the Azure DevOps project that you want to use.
- An ability to run pipelines on Microsoft-hosted agents. To use Microsoft-hosted agents, your Azure DevOps organization must have access to Microsoft-hosted parallel jobs. You can either purchase a parallel job or you can request a free grant.

Create your first pipeline

Java

Get the Java sample code

To get started, fork the following repository into your GitHub account.

```
https://github.com/MicrosoftDocs/pipelines-java
```

Create your first Java pipeline

1. Sign in to your Azure DevOps organization and go to your project.

2. Go to **Pipelines**, and then select **New pipeline** or **Create pipeline** if creating your first pipeline.

3. Do the steps of the wizard by first selecting **GitHub** as the location of your source code.

4. You might be redirected to GitHub to sign in. If so, enter your GitHub credentials.

5. When you see the list of repositories, select your repository.

6. You might be redirected to GitHub to install the Azure Pipelines app. If so, select **Approve & install**.

7. Azure Pipelines will analyze your repository and recommend the **Maven** pipeline template.

8. When your new pipeline appears, take a look at the YAML to see what it does. When you're ready, select **Save and run**.

9. You're prompted to commit a new `azure-pipelines.yml` file to your repository. After you're happy with the message, select **Save and run** again.

   If you want to watch your pipeline in action, select the build job.

   You just created and ran a pipeline that we automatically created for you, because your code appeared to be a good match for the Maven template.

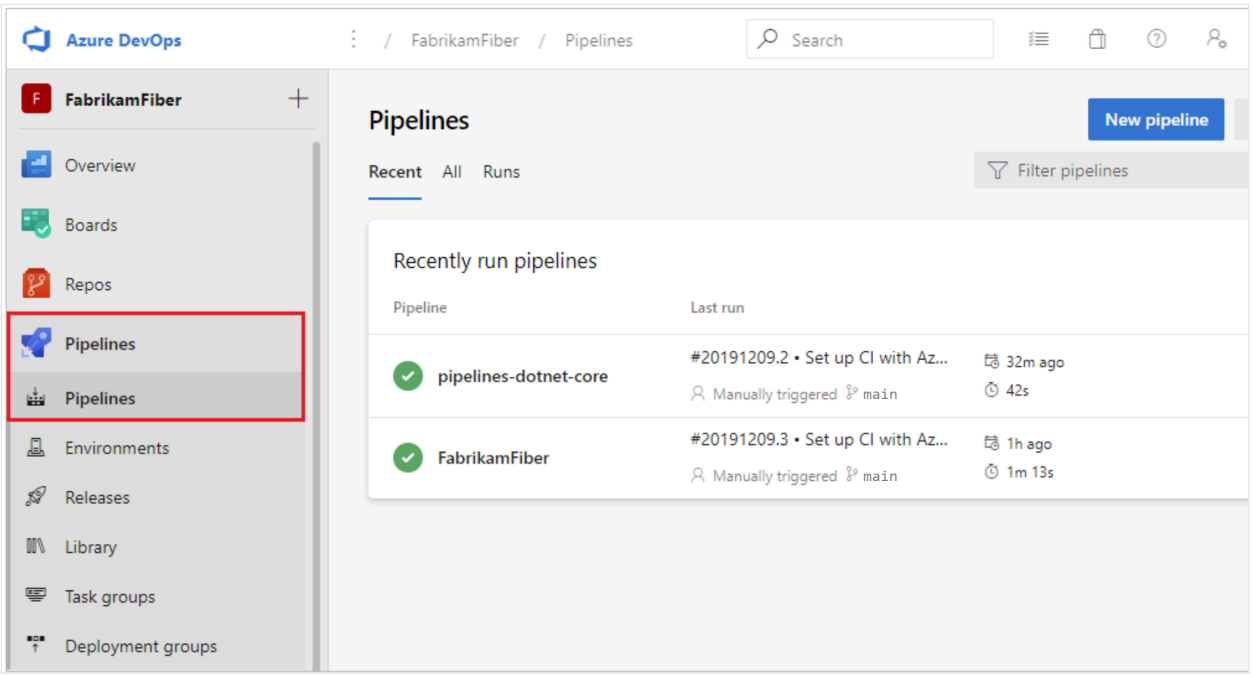
   You now have a working YAML pipeline (`azure-pipelines.yml`) in your repository that's ready for you to customize!

10. When you're ready to make changes to your pipeline, select it in the **Pipelines** page, and then **Edit** the `azure-pipelines.yml` file.

Learn more about working with Java in your pipeline.

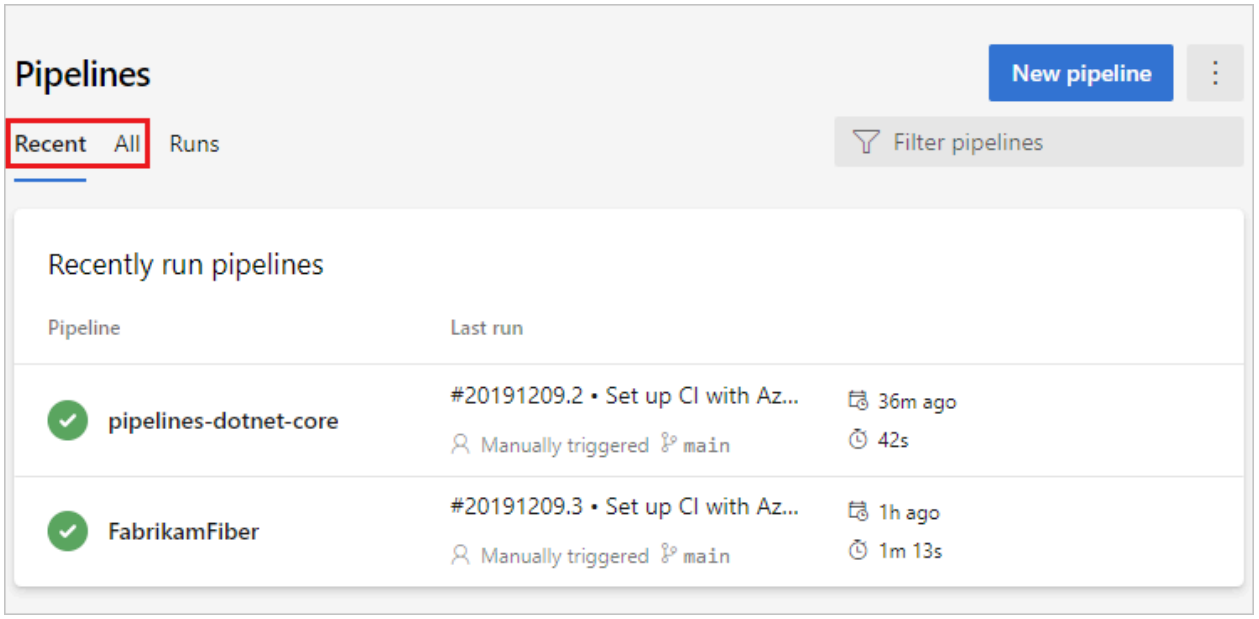
View and manage your pipelines

You can view and manage your pipelines by choosing **Pipelines** from the left-hand menu to go to the pipelines landing page.

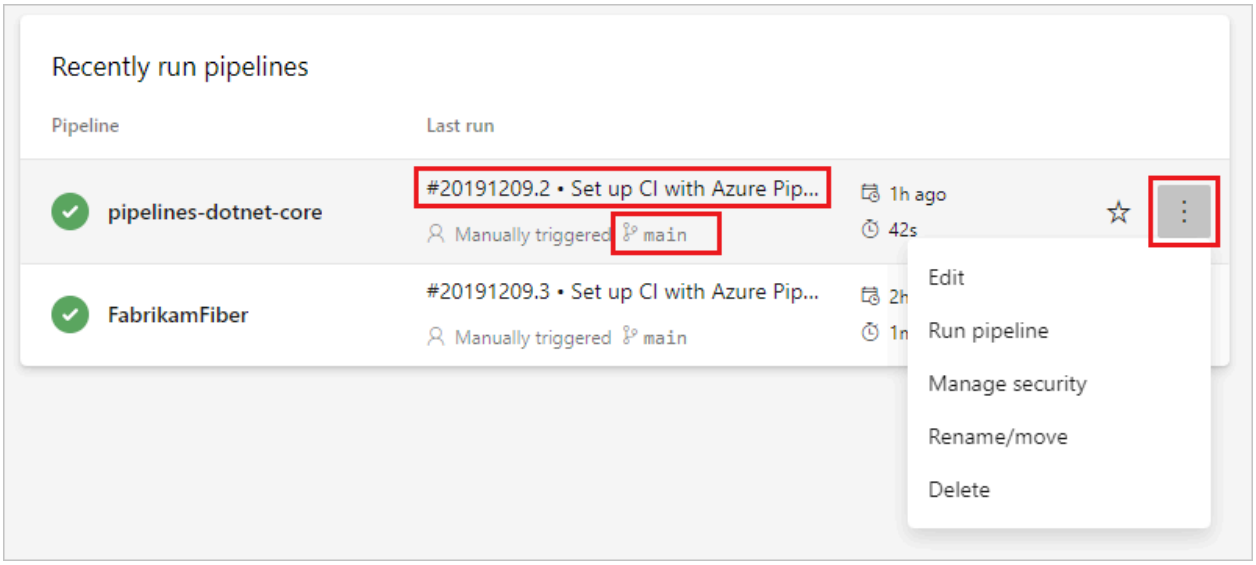


From the pipelines landing page you can view pipelines and pipeline runs, create and import pipelines, manage security, and drill down into pipeline and run details.

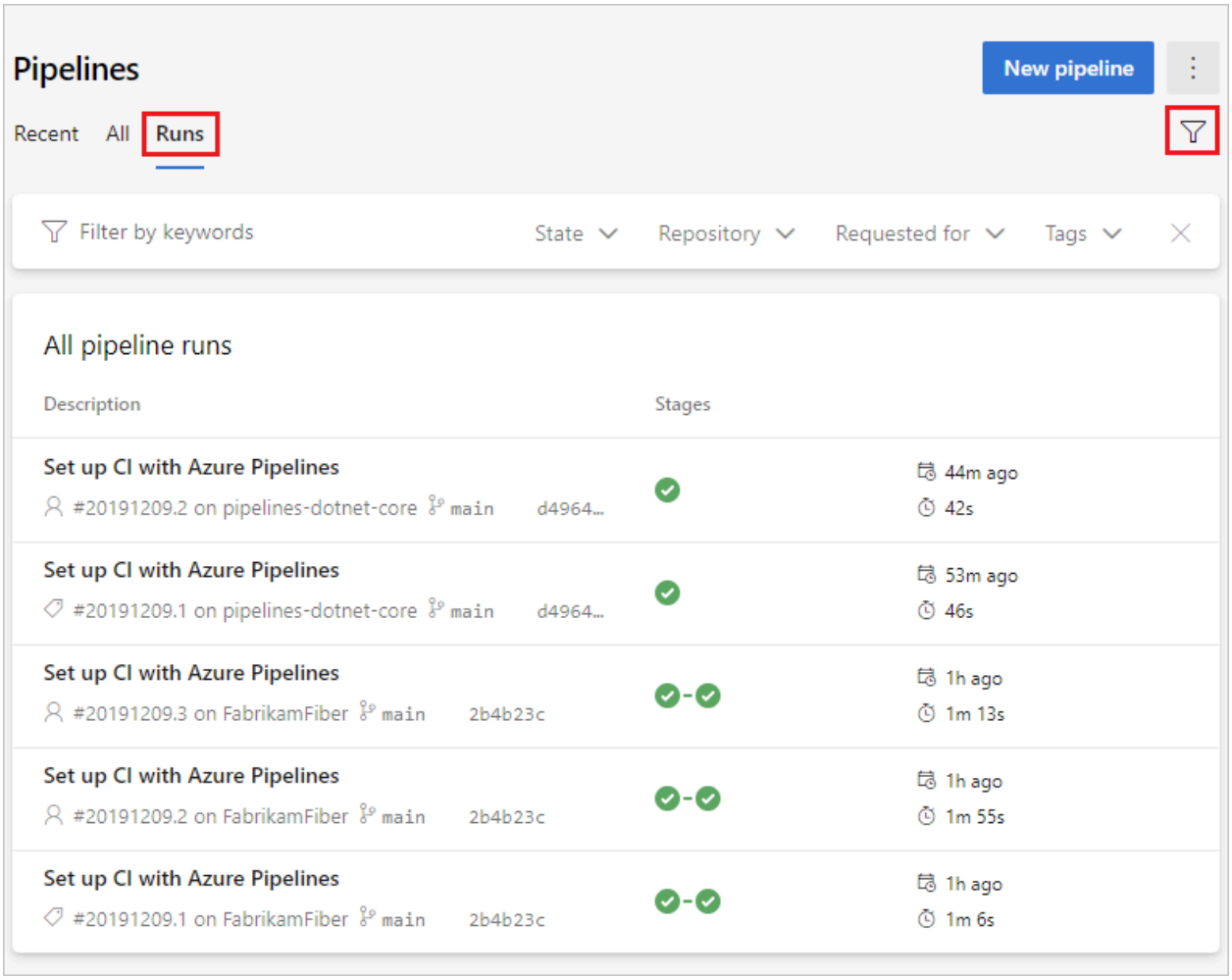
Choose **Recent** to view recently run pipelines (the default view), or choose **All** to view all pipelines.



Select a pipeline to manage that pipeline and view the runs. Select the build number for the last run to view the results of that build, select the branch name to view the branch for that run, or select the context menu to run the pipeline and perform other management actions.

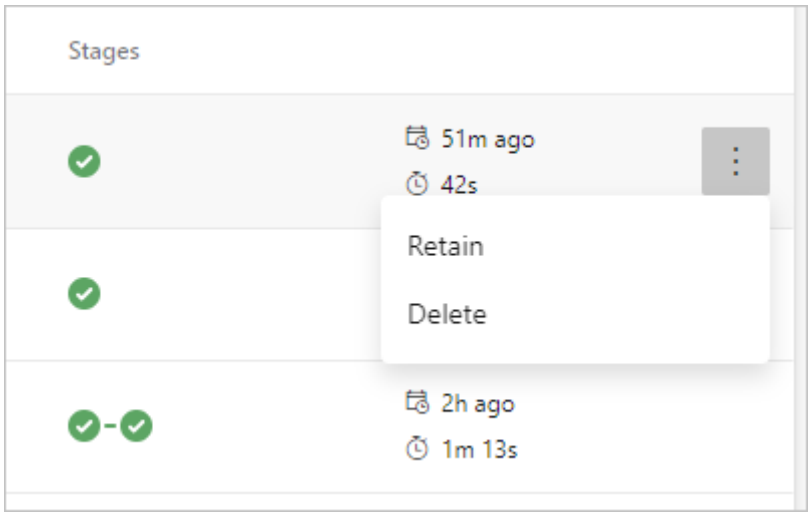


Select **Runs** to view all pipeline runs. You can optionally filter the displayed runs.



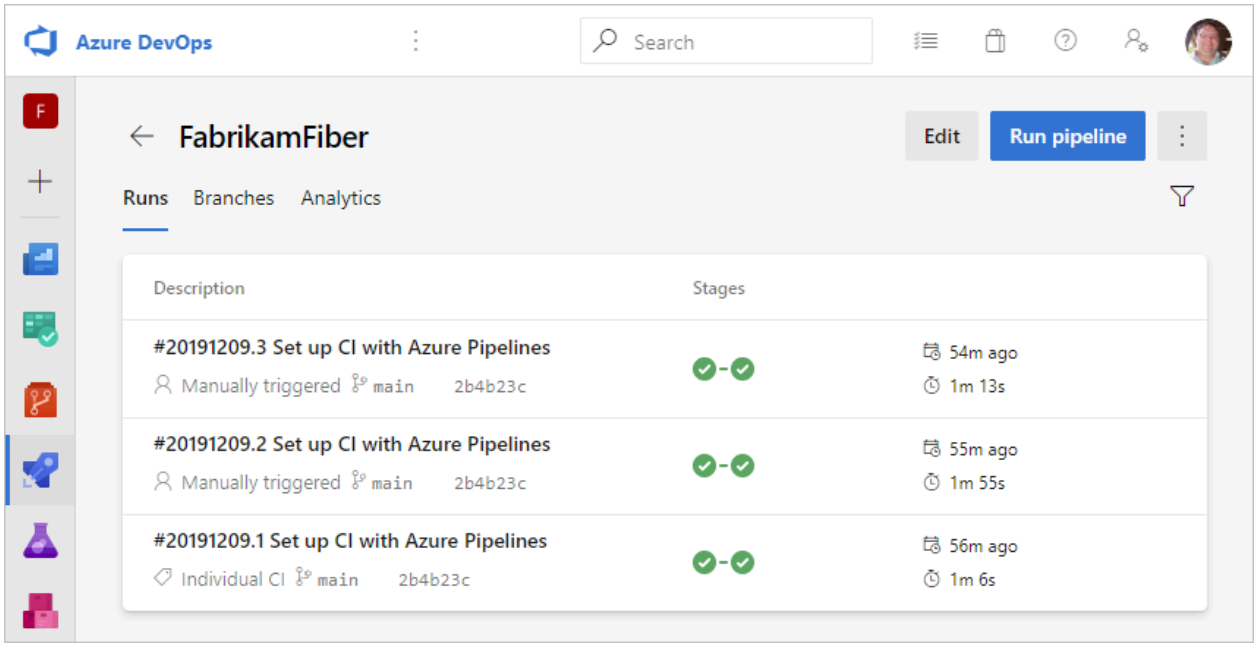
Select a pipeline run to view information about that run.

You can choose to **Retain** or **Delete** a run from the context menu. For more information on run retention, see Build and release retention policies.



View pipeline details

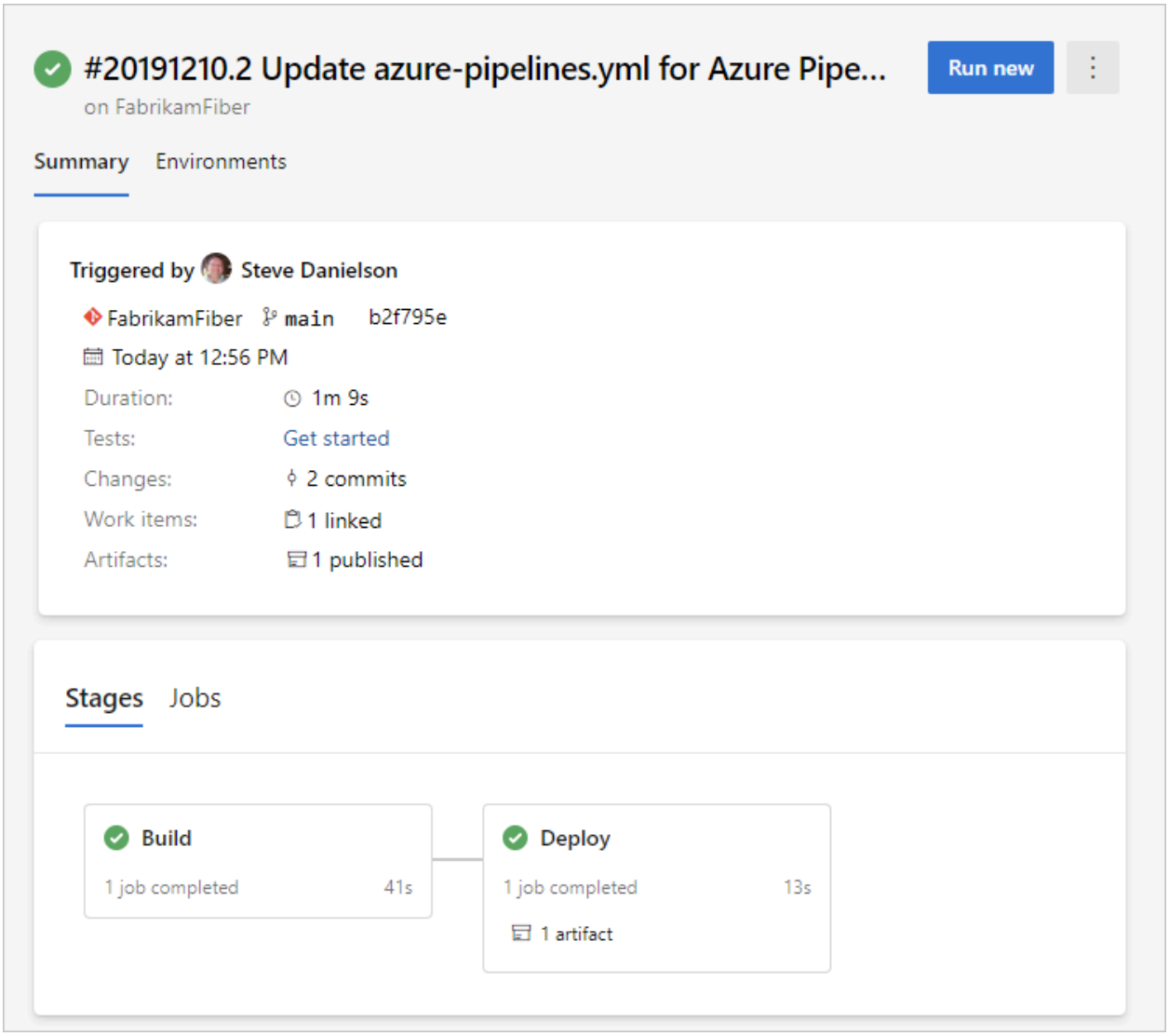
The details page for a pipeline allows you to view and manage that pipeline.



Choose **Edit** to edit your pipeline. For more information, see YAML pipeline editor. You can also edit your pipeline by modifying the **azure-pipelines.yml** file directly in the repository that hosts the pipeline.

View pipeline run details

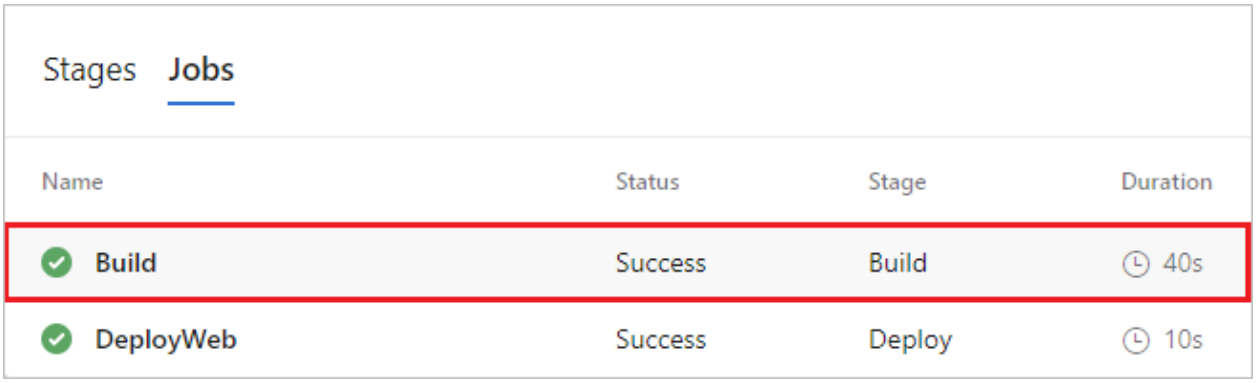
From the pipeline run summary you can view the status of your run, both while it is running and when it is complete.



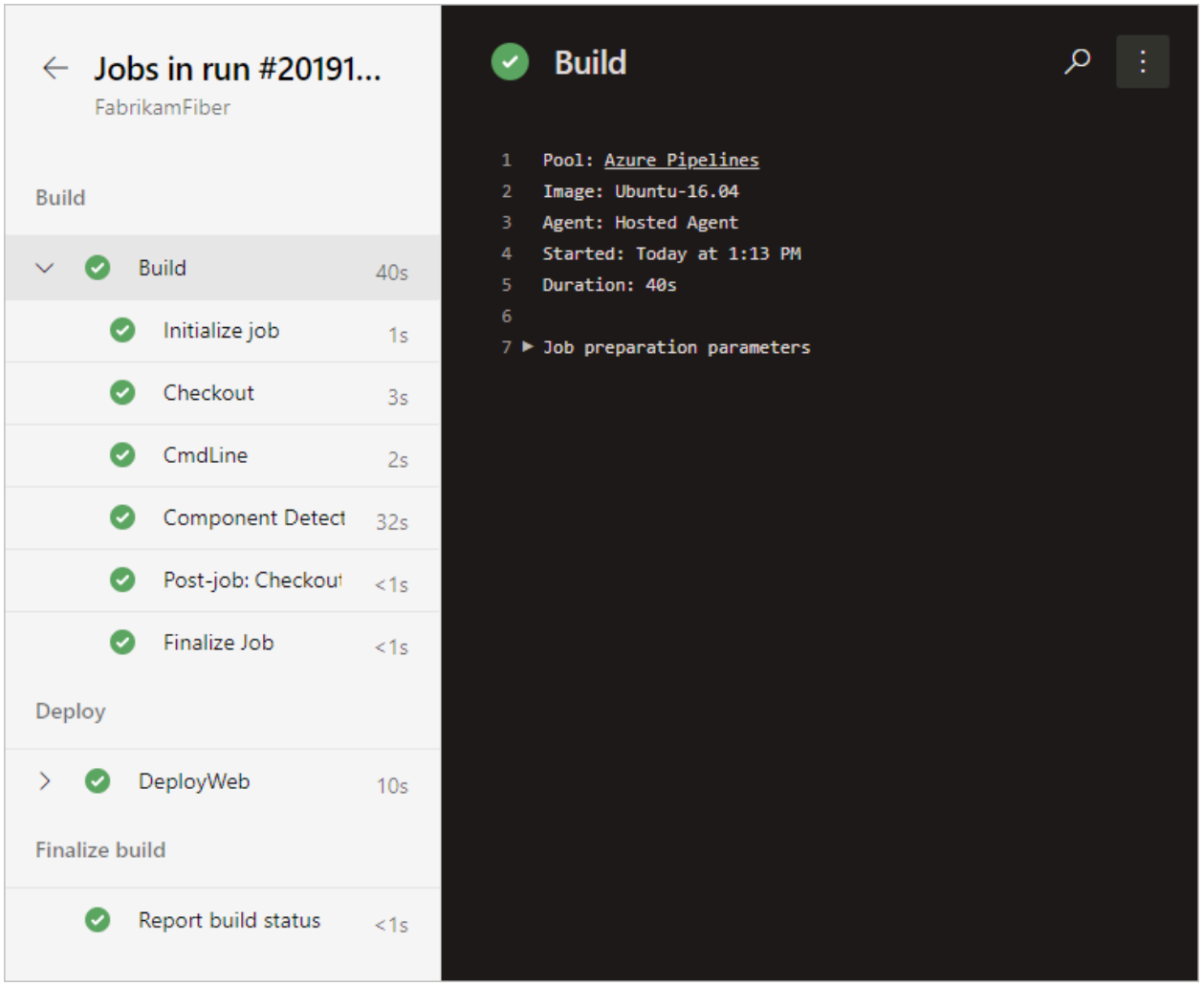
From the summary pane you can view job and stage details, download artifacts, and navigate to linked commits, test results, and work items.

Jobs and stages

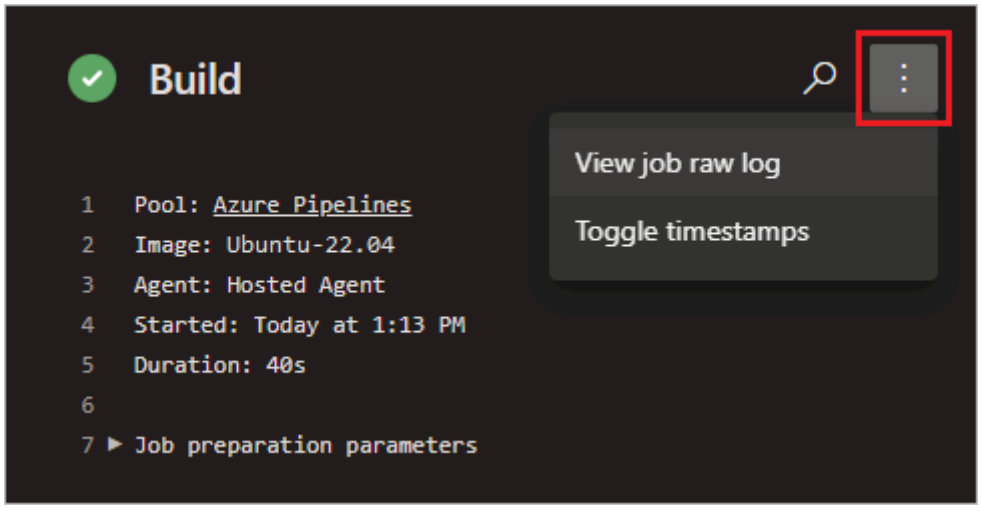
The jobs pane displays an overview of the status of your stages and jobs. This pane may have multiple tabs depending on whether your pipeline has stages and jobs, or just jobs. In this example, the pipeline has two stages named **Build** and **Deploy**. You can drill down into the pipeline steps by choosing the job from either the **Stages** or **Jobs** pane.



Choose a job to see the steps for that job.

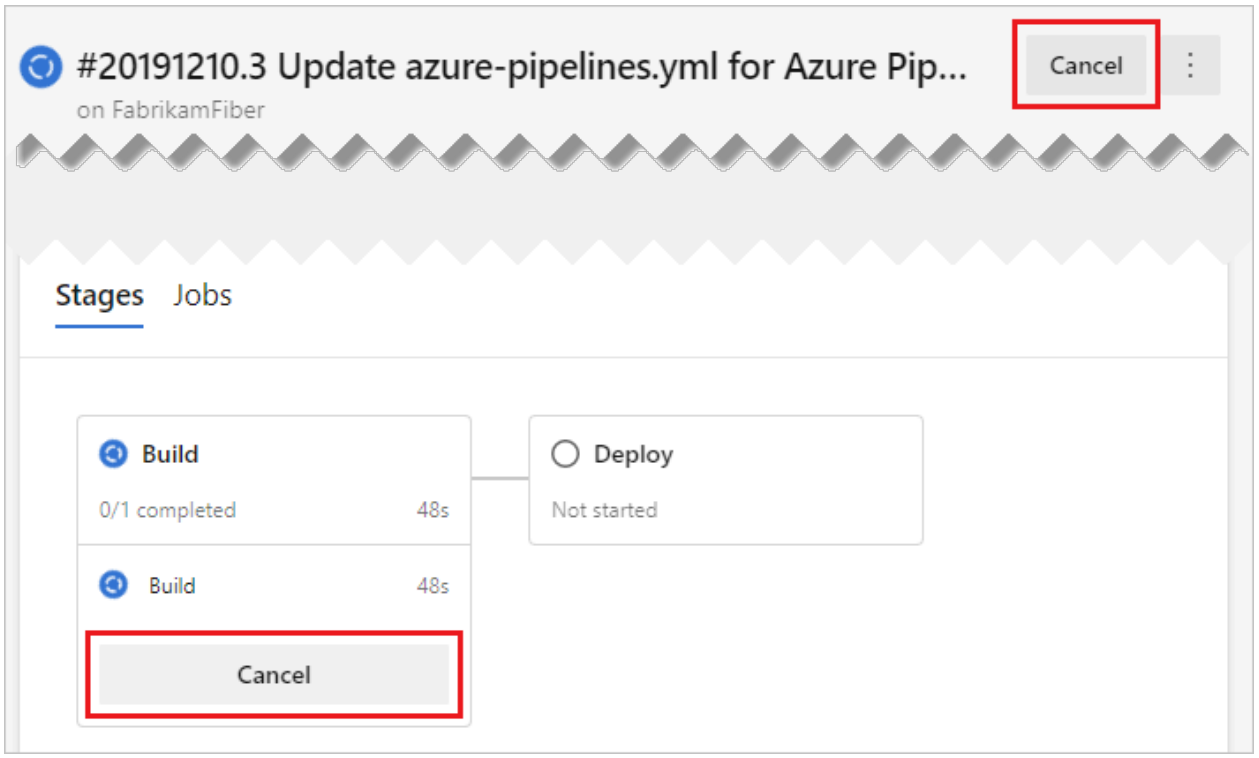


From the steps view, you can review the status and details of each step. From the **More actions** ⋮ you can toggle timestamps or view a raw log of all steps in the pipeline.



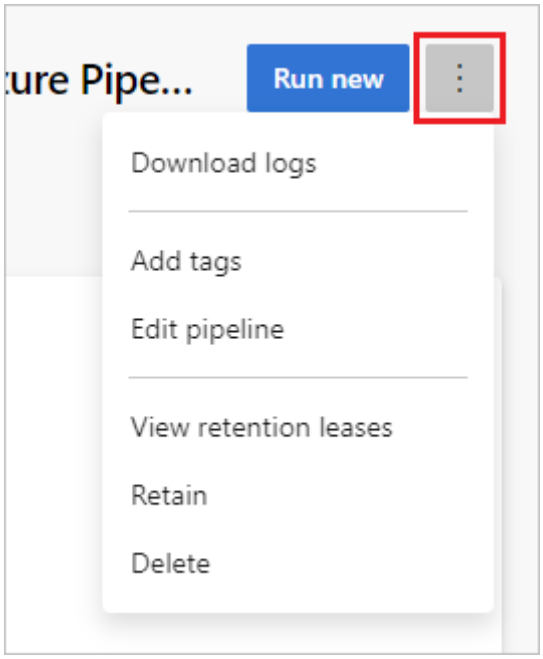
Cancel and re-run a pipeline

If the pipeline is running, you can cancel it by choosing **Cancel**. If the run has completed, you can re-run the pipeline by choosing **Run new**.



Pipeline run more actions menu

From the **More actions** ⋮ menu you can download logs, add tags, edit the pipeline, delete the run, and configure retention for the run.



> ⓘ Note
>
> You can't delete a run if the run is retained. If you don't see **Delete**, choose **Stop retaining run**, and then delete the run. If you see both **Delete** and **View retention releases**, one or more configured retention policies still apply to your run. Choose

View retention releases, delete the policies (only the policies for the selected run are removed), and then delete the run.

Add a status badge to your repository

Many developers like to show that they're keeping their code quality high by displaying a status badge in their repo.



To copy the status badge to your clipboard:

1. In Azure Pipelines, go to the **Pipelines** page to view the list of pipelines. Select the pipeline you created in the previous section.
2. Select ⋮, and then select **Status badge**.
3. Select **Status badge**.
4. Copy the sample Markdown from the Sample markdown section.

Now with the badge Markdown in your clipboard, take the following steps in GitHub:

1. Go to the list of files and select `Readme.md`. Select the pencil icon to edit.
2. Paste the status badge Markdown at the beginning of the file.
3. Commit the change to the `main` branch.
4. Notice that the status badge appears in the description of your repository.

To configure anonymous access to badges for private projects:

1. Navigate to **Project Settings** in the bottom left corner of the page
2. Open the **Settings** tab under **Pipelines**
3. Toggle the **Disable anonymous access to badges** slider under **General**

> ⓘ **Note**
>
> Even in a private project, anonymous badge access is enabled by default. With anonymous badge access enabled, users outside your organization might be able

to query information such as project names, branch names, job names, and build status through the badge status API.

Because you just changed the `Readme.md` file in this repository, Azure Pipelines automatically builds your code, according to the configuration in the `azure-pipelines.yml` file at the root of your repository. Back in Azure Pipelines, observe that a new run appears. Each time you make an edit, Azure Pipelines starts a new run.

Next steps

You learned how to create your first pipeline in Azure. Now, Learn more about configuring pipelines in the language of your choice:

- .NET Core
- Go
- Java
- Node.js
- Python
- Containers

Or, you can proceed to customize the pipeline you created.

To run your pipeline in a container, see Container jobs.

For details about building GitHub repositories, see Build GitHub repositories.

To learn how to publish your Pipeline Artifacts, see Publish Pipeline Artifacts.

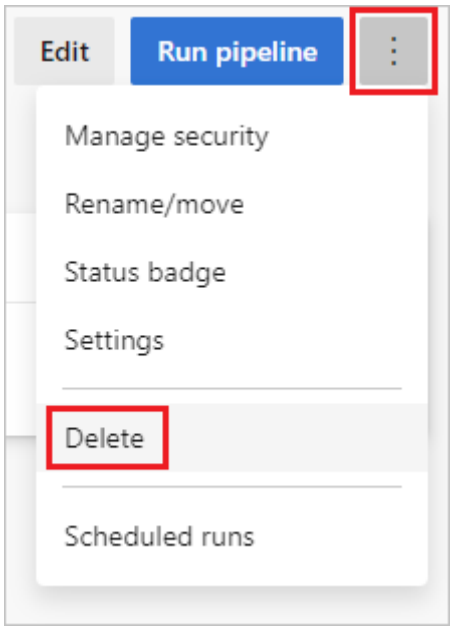
To find out what else you can do in YAML pipelines, see YAML schema reference.

Clean up

If you created any test pipelines, they're easy to delete when you finish with them.

Azure Pipelines UI

To delete a pipeline, navigate to the summary page for that pipeline, and choose **Delete** from the ... menu at the top-right of the page. Type the name of the pipeline to confirm, and choose **Delete**.



FAQ

Where can I read articles about DevOps and CI/CD?

What is Continuous Integration?

What is Continuous Delivery?

What is DevOps?

What version control system can I use?

When you're ready to get going with CI/CD for your app, you can use the version control system of your choice:

- Clients
  - Visual Studio Code for Windows, macOS, and Linux
  - Visual Studio with Git for Windows or Visual Studio for Mac
  - Eclipse
  - Xcode
  - IntelliJ
  - Command line
- Services
  - Azure Pipelines
  - Git service providers such as Azure Repos Git, GitHub, and Bitbucket Cloud
  - Subversion

How can I delete a pipeline?

To delete a pipeline, navigate to the summary page for that pipeline, and choose **Delete** from the **...** menu in the top-right of the page. Type the name of the pipeline to confirm, and choose **Delete**.

What else can I do when I queue a build?

You can queue builds automatically or manually.

When you manually queue a build, you can, for a single run of the build:

- Specify the pool into which the build goes.
- Add and modify some variables.
- Add demands.
- In a Git repository
  - Build a branch or a tag.
  - Build a commit.

Where can I learn more about pipeline settings?

To learn more about pipeline settings, see:

- Getting sources
- Tasks
- Variables
- Triggers
- Retention
- History

How do I programmatically create a build pipeline?

REST API Reference: Create a build pipeline

> **Note**
>
> You can also manage builds and build pipelines from the command line or scripts using the **Azure Pipelines CLI**.

Feedback

Was this page helpful? Yes No

Provide product feedback

ASP.NET Core Module (ANCM) for IIS

Article • 04/27/2023

The ASP.NET Core Module (ANCM) is a native IIS module that plugs into the IIS pipeline, allowing ASP.NET Core applications to work with IIS. Run ASP.NET Core apps with IIS by either:

- Hosting an ASP.NET Core app inside of the IIS worker process (`w3wp.exe`), called the in-process hosting model.
- Forwarding web requests to a backend ASP.NET Core app running the Kestrel server, called the out-of-process hosting model.

There are trade-offs between each of the hosting models. By default, the in-process hosting model is used due to better performance and diagnostics.

For more information and configuration guidance, see the following topics:

- Web server implementations in ASP.NET Core

Install ASP.NET Core Module (ANCM)

The ASP.NET Core Module (ANCM) is installed with the .NET Core Runtime from the .NET Core Hosting Bundle. The ASP.NET Core Module is forward and backward compatible with in-support releases of .NET.

Breaking changes and security advisories are reported on the Announcements repo. Announcements can be limited to a specific version by selecting a **Label** filter.

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

For more information, including installing an earlier version of the module, see Hosting Bundle.

For a tutorial experience on publishing an ASP.NET Core app to an IIS server, see Publish an ASP.NET Core app to IIS.

Additional resources

- Host ASP.NET Core on Windows with IIS
- Deploy ASP.NET Core apps to Azure App Service

- ASP.NET Core Module reference source [default branch (main)]: Use the **Branch** drop down list to select a specific release (for example, `release/3.1`).
- IIS modules with ASP.NET Core

Troubleshoot ASP.NET Core on Azure App Service and IIS

Article • 07/23/2024

This article provides information on common app startup errors and instructions on how to diagnose errors when an app is deployed to Azure App Service or IIS:

App startup errors
Explains common startup HTTP status code scenarios.

Troubleshoot on Azure App Service
Provides troubleshooting advice for apps deployed to Azure App Service.

Troubleshoot on IIS
Provides troubleshooting advice for apps deployed to IIS or running on IIS Express locally. The guidance applies to both Windows Server and Windows desktop deployments.

Clear package caches
Explains what to do when incoherent packages break an app when performing major upgrades or changing package versions.

Additional resources
Lists additional troubleshooting topics.

App startup errors

In Visual Studio, the ASP.NET Core project default server is Kestrel. Visual studio can be configured to use IIS Express. A *502.5 - Process Failure* or a *500.30 - Start Failure* that occurs when debugging locally with IIS Express can be diagnosed using the advice in this topic.

403.14 Forbidden

The app fails to start. The following error is logged:

```
The Web server is configured to not list the contents of this directory.
```

The error is usually caused by a broken deployment on the hosting system, which includes any of the following scenarios:

- The app is deployed to the wrong folder on the hosting system.
- The deployment process failed to move all of the app's files and folders to the deployment folder on the hosting system.
- The *web.config* file is missing from the deployment, or the *web.config* file contents are malformed.

Perform the following steps:

1. Delete all of the files and folders from the deployment folder on the hosting system.
2. Redeploy the contents of the app's *publish* folder to the hosting system using your normal method of deployment, such as Visual Studio, PowerShell, or manual deployment:

    - Confirm that the *web.config* file is present in the deployment and that its contents are correct.
    - When hosting on Azure App Service, confirm that the app is deployed to the `D:\home\site\wwwroot` folder.
    - When the app is hosted by IIS, confirm that the app is deployed to the IIS **Physical path** shown in **IIS Manager**'s **Basic Settings**.

3. Confirm that all of the app's files and folders are deployed by comparing the deployment on the hosting system to the contents of the project's *publish* folder.

For more information on the layout of a published ASP.NET Core app, see ASP.NET Core directory structure. For more information on the *web.config* file, see ASP.NET Core Module (ANCM) for IIS.

500 Internal Server Error

The app starts, but an error prevents the server from fulfilling the request.

This error occurs within the app's code during startup or while creating a response. The response may contain no content, or the response may appear as a *500 Internal Server Error* in the browser. The Application Event Log usually states that the app started normally. From the server's perspective, that's correct. The app did start, but it can't generate a valid response. Run the app at a command prompt on the server or enable the ASP.NET Core Module stdout log to troubleshoot the problem.

This error also may occur when the .NET Core Hosting Bundle isn't installed or is corrupted. Installing or repairing the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express) may fix the problem.

500.0 In-Process Handler Load Failure

The worker process fails. The app doesn't start.

An unknown error occurred loading ASP.NET Core Module components. Take one of the following actions:

- Contact Microsoft Support (select **Developer Tools** then **ASP.NET Core**).
- Ask a question on Stack Overflow.
- File an issue on our GitHub repository.

500.30 In-Process Startup Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core CLR in-process, but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

Common failure conditions:

- The app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine.
- Using Azure Key Vault, lack of permissions to the Key Vault. Check the access policies in the targeted Key Vault to ensure that the correct permissions are granted.

500.31 ANCM Failed to Find Native Dependencies

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the .NET Core runtime in-process, but it fails to start. The most common cause of this startup failure is when the `Microsoft.NETCore.App` or `Microsoft.AspNetCore.App` runtime isn't installed. If the app is deployed to target ASP.NET Core 3.0 and that version doesn't exist on the machine, this error occurs. An example error message follows:

```
```

```
The specified framework 'Microsoft.NETCore.App', version '3.0.0' was not
found.
  - The following frameworks were found:
      2.2.1 at [C:\Program Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview5-27626-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27713-13 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27714-15 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
      3.0.0-preview6-27723-08 at [C:\Program
Files\dotnet\x64\shared\Microsoft.NETCore.App]
```

The error message lists all the installed .NET Core versions and the version requested by the app. To fix this error, either:

- Install the appropriate version of .NET Core on the machine.
- Change the app to target a version of .NET Core that's present on the machine.
- Publish the app as a self-contained deployment.

When running in development (the `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development`), the specific error is written to the HTTP response. The cause of a process startup failure is also found in the Application Event Log.

500.32 ANCM Failed to Load dll

The worker process fails. The app doesn't start.

The most common cause for this error is that the app is published for an incompatible processor architecture. If the worker process is running as a 32-bit app and the app was published to target 64-bit, this error occurs.

To fix this error, either:

- Republish the app for the same processor architecture as the worker process.
- Publish the app as a framework-dependent deployment.

500.33 ANCM Request Handler Load Failure

The worker process fails. The app doesn't start.

The app didn't reference the `Microsoft.AspNetCore.App` framework. Only apps targeting the `Microsoft.AspNetCore.App` framework can be hosted by the ASP.NET Core Module.

To fix this error, confirm that the app is targeting the `Microsoft.AspNetCore.App` framework. Check the `.runtimeconfig.json` to verify the framework targeted by the app.

500.34 ANCM Mixed Hosting Models Not Supported

The worker process can't run both an in-process app and an out-of-process app in the same process.

To fix this error, run apps in separate IIS application pools.

500.35 ANCM Multiple In-Process Applications in same Process

The worker process can't run multiple in-process apps in the same process.

To fix this error, run apps in separate IIS application pools.

500.36 ANCM Out-Of-Process Handler Load Failure

The out-of-process request handler, *aspnetcorev2_outofprocess.dll*, isn't next to the *aspnetcorev2.dll* file. This indicates a corrupted installation of the ASP.NET Core Module.

To fix this error, repair the installation of the .NET Core Hosting Bundle (for IIS) or Visual Studio (for IIS Express).

500.37 ANCM Failed to Start Within Startup Time Limit

ANCM failed to start within the provided startup time limit. By default, the timeout is 120 seconds.

This error can occur when starting a large number of apps on the same machine. Check for CPU/Memory usage spikes on the server during startup. You may need to stagger the startup process of multiple apps.

500.38 ANCM Application DLL Not Found

ANCM failed to locate the application DLL, which should be next to the executable.

This error occurs when hosting an app packaged as a single-file executable using the in-process hosting model. The in-process model requires that the ANCM load the .NET Core app into the existing IIS process. This scenario isn't supported by the single-file

deployment model. Use **one** of the following approaches in the app's project file to fix this error:

1. Disable single-file publishing by setting the `PublishSingleFile` MSBuild property to `false`.
2. Switch to the out-of-process hosting model by setting the `AspNetCoreHostingModel` MSBuild property to `OutOfProcess`.

502.5 Process Failure

The worker process fails. The app doesn't start.

The ASP.NET Core Module attempts to start the worker process but it fails to start. The cause of a process startup failure can usually be determined from entries in the Application Event Log and the ASP.NET Core Module stdout log.

A common failure condition is the app is misconfigured due to targeting a version of the ASP.NET Core shared framework that isn't present. Check which versions of the ASP.NET Core shared framework are installed on the target machine. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and referenced by a metapackage such as `Microsoft.AspNetCore.App`. The metapackage reference can specify a minimum required version. For more information, see The shared framework.

The *502.5 Process Failure* error page is returned when a hosting or app misconfiguration causes the worker process to fail:

Failed to start application (ErrorCode '0x800700c1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/6/ROOT/', ErrorCode '0x800700c1'.
```

The app failed to start because the app's assembly (*.dll*) couldn't be loaded.

This error occurs when there's a bitness mismatch between the published app and the w3wp/iisexpress process.

Confirm that the app pool's 32-bit setting is correct:

1. Select the app pool in IIS Manager's **Application Pools**.

2. Select **Advanced Settings** under **Edit Application Pool** in the **Actions** panel.
3. Set **Enable 32-Bit Applications**:

    - If deploying a 32-bit (x86) app, set the value to `True`.
    - If deploying a 64-bit (x64) app, set the value to `False`.

Confirm that there isn't a conflict between a `<Platform>` MSBuild property in the project file and the published bitness of the app.

Failed to start application (ErrorCode '0x800701b1')

```
EventID: 1010
Source: IIS AspNetCore Module V2
Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x800701b1'.
```

The app failed to start because a Windows Service failed to load.

One common service that needs to be enabled is the "null" service. The following command enables the `null` Windows Service:

Windows Command Prompt

```
sc.exe start null
```

Connection reset

If an error occurs after the headers are sent, it's too late for the server to send a **500 Internal Server Error** when an error occurs. This often happens when an error occurs during the serialization of complex objects for a response. This type of error appears as a *connection reset* error on the client. Application logging can help troubleshoot these types of errors.

Default startup limits

The ASP.NET Core Module is configured with a default *startupTimeLimit* of 120 seconds. When left at the default value, an app may take up to two minutes to start before the module logs a process failure. For information on configuring the module, see Attributes of the aspNetCore element.

Troubleshoot on Azure App Service

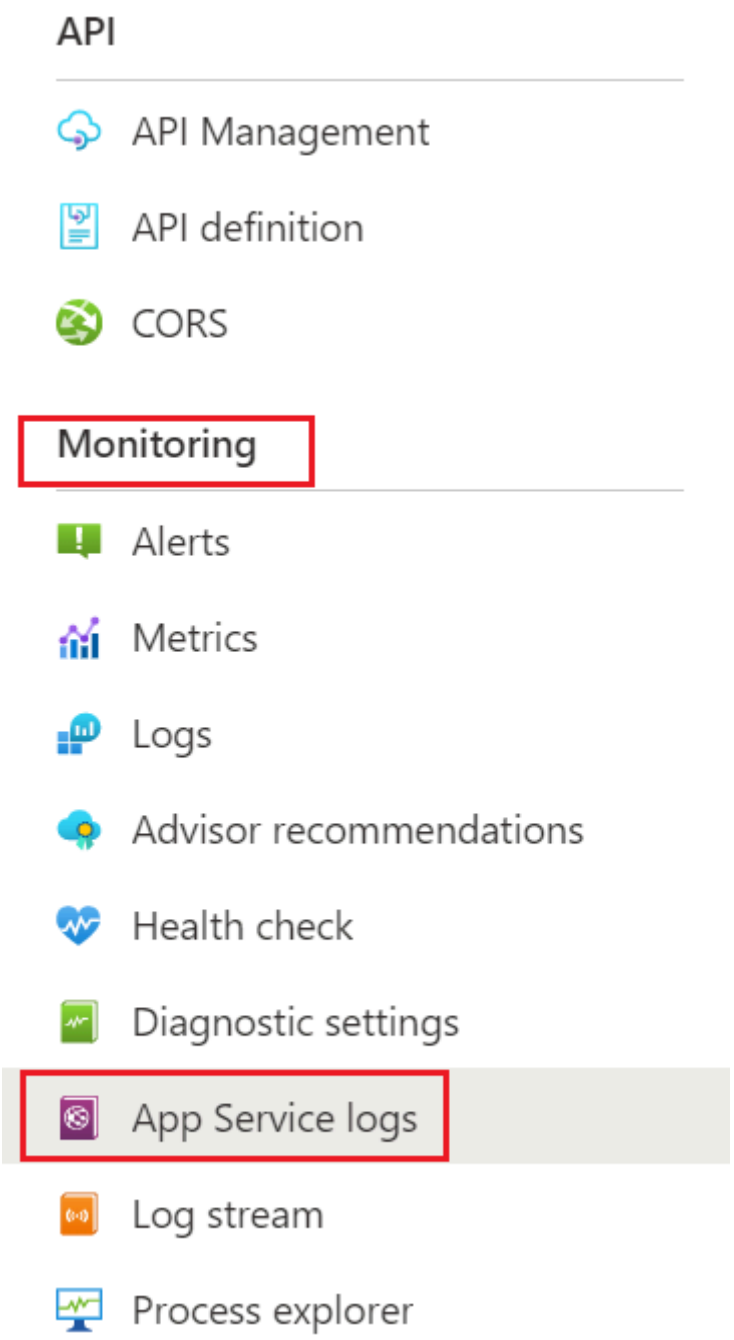
> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**
>
> ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

Azure App Services Log stream

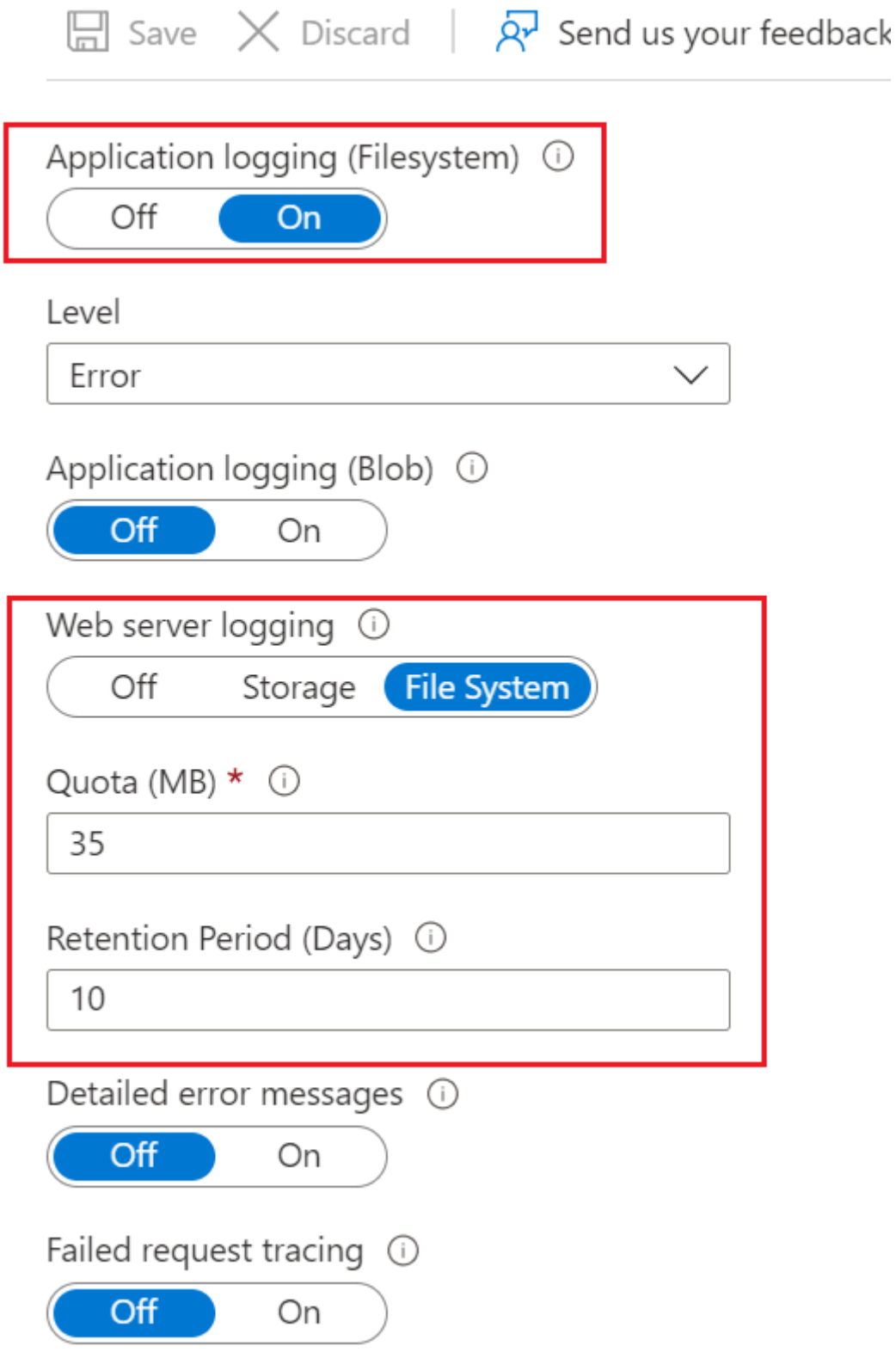
The Azure App Services Log streams logging information as it occurs. To view streaming logs:

1. In the Azure portal, open the app in **App Services**.

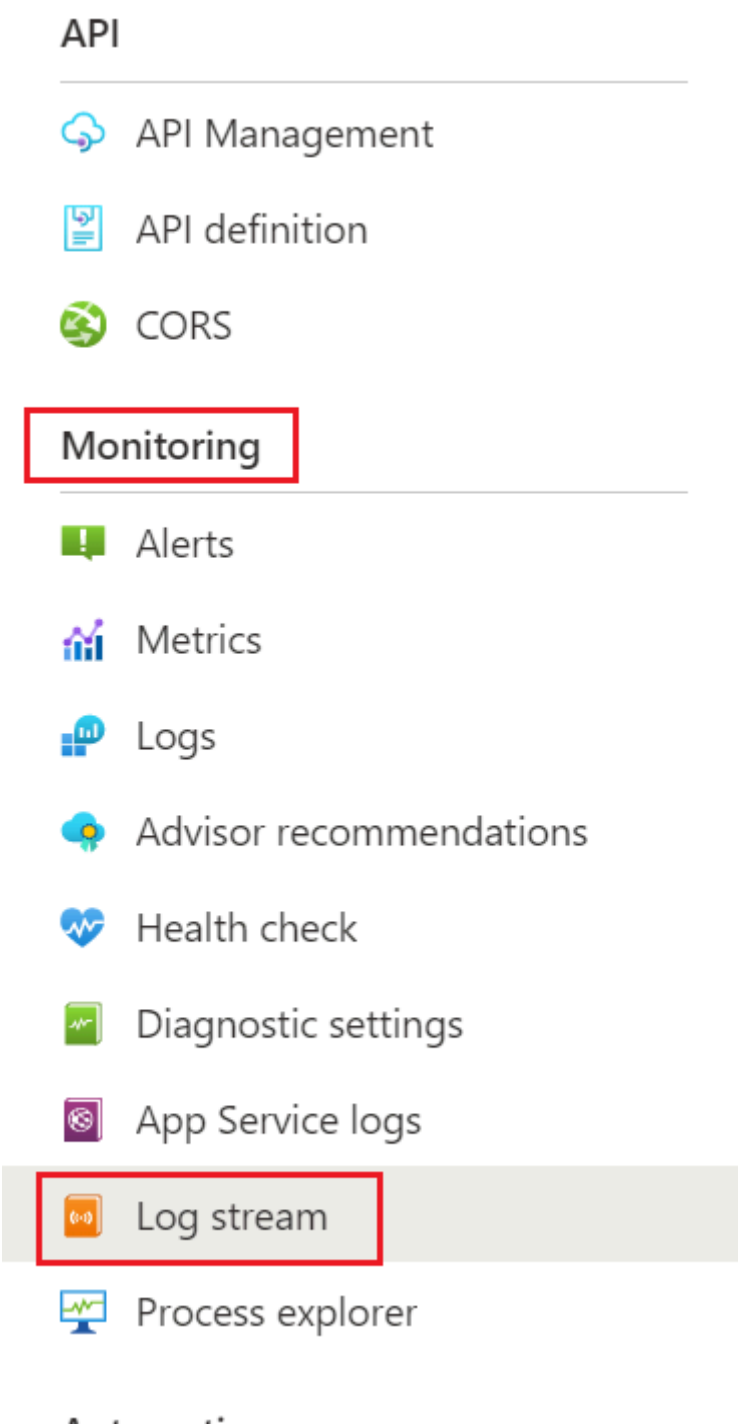
2. In the left pane, navigate to **Monitoring** > **App Service Logs**.



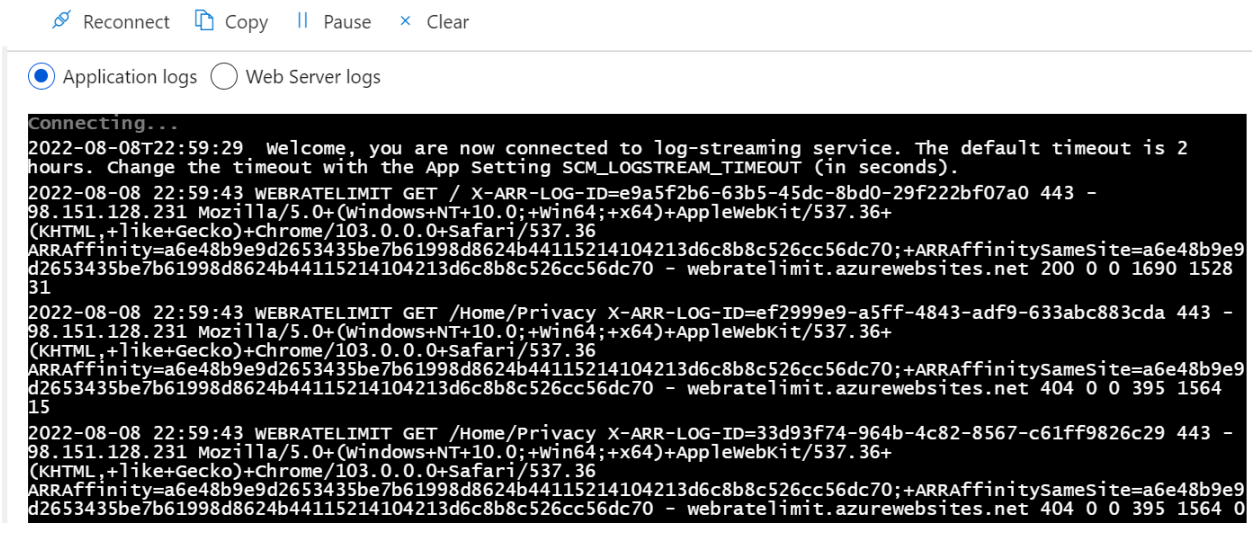
3. Select **File System** for **Web Server Logging**. Optionally enable **Application logging**.



4. In the left pane, navigate to **Monitoring** > **Log stream**, and then select **Application logs** or **Web Server Logs**.



The following images shows the application logs output:



Streaming logs have some latency and might not display immediately.

Application Event Log (Azure App Service)

To access the Application Event Log, use the **Diagnose and solve problems** blade in the Azure portal:

1. In the Azure portal, open the app in **App Services**.
2. Select **Diagnose and solve problems**.
3. Select the **Diagnostic Tools** heading.
4. Under **Support Tools**, select the **Application Events** button.
5. Examine the latest error provided by the *IIS AspNetCoreModule* or *IIS AspNetCoreModule V2* entry in the **Source** column.

An alternative to using the **Diagnose and solve problems** blade is to examine the Application Event Log file directly using Kudu:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the **LogFiles** folder.
4. Select the pencil icon next to the `eventlog.xml` file.
5. Examine the log. Scroll to the bottom of the log to see the most recent events.

Run the app in the Kudu console

Many startup errors don't produce useful information in the Application Event Log. You can run the app in the Kudu Remote Execution Console to discover the error:

1. Open **Advanced Tools** in the **Development Tools** area. Select the **Go** → button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.

Test a 32-bit (x86) app

Current release

1. `cd d:\home\site\wwwroot`
2. Run the app:

- If the app is a framework-dependent deployment:

  .NET CLI

  ```
  dotnet .\{ASSEMBLY NAME}.dll
  ```

- If the app is a self-contained deployment:

  Console

  ```
  {ASSEMBLY NAME}.exe
  ```

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x86) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x32` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

Test a 64-bit (x64) app

Current release

- If the app is a 64-bit (x64) framework-dependent deployment:

  1. `cd D:\Program Files\dotnet`
  2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

- If the app is a self-contained deployment:

  1. `cd D:\home\site\wwwroot`
  2. Run the app: `{ASSEMBLY NAME}.exe`

The console output from the app, showing any errors, is piped to the Kudu console.

Framework-dependent deployment running on a preview release

Requires installing the ASP.NET Core {VERSION} (x64) Runtime site extension.

1. `cd D:\home\SiteExtensions\AspNetCoreRuntime.{X.Y}.x64` (`{X.Y}` is the runtime version)
2. Run the app: `dotnet \home\site\wwwroot\{ASSEMBLY NAME}.dll`

The console output from the app, showing any errors, is piped to the Kudu console.

ASP.NET Core Module stdout log (Azure App Service)

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created. Only use stdout logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

The ASP.NET Core Module stdout log often records useful error messages not found in the Application Event Log. To enable and view stdout logs:

1. In the Azure Portal, navigate to the web app.
2. In the **App Service** blade, enter **kudu** in the search box.
3. Select **Advanced Tools** > **Go**.
4. Select **Debug console** > **CMD**.
5. Navigate to *site/wwwroot*
6. Select the pencil icon to edit the *web.config* file.
7. In the `<aspNetCore />` element, set `stdoutLogEnabled="true"` and select **Save**.

Disable stdout logging when troubleshooting is complete by setting `stdoutLogEnabled="false"`.

For more information, see ASP.NET Core Module (ANCM) for IIS.

ASP.NET Core Module debug log (Azure App Service)

The ASP.NET Core Module debug log provides additional, deeper logging from the ASP.NET Core Module. To enable and view stdout logs:

1. To enable the enhanced diagnostic log, perform either of the following:

- Follow the instructions in Enhanced diagnostic logs to configure the app for an enhanced diagnostic logging. Redeploy the app.
- Add the `<handlerSettings>` shown in Enhanced diagnostic logs to the live app's *web.config* file using the Kudu console:
  a. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
  b. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
  c. Open the folders to the path **site** > **wwwroot**. Edit the *web.config* file by selecting the pencil button. Add the `<handlerSettings>` section as shown in Enhanced diagnostic logs. Select the **Save** button.

2. Open **Advanced Tools** in the **Development Tools** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
3. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
4. Open the folders to the path **site** > **wwwroot**. If you didn't supply a path for the *aspnetcore-debug.log* file, the file appears in the list. If you supplied a path, navigate to the location of the log file.
5. Open the log file with the pencil button next to the file name.

Disable debug logging when troubleshooting is complete:

To disable the enhanced debug log, perform either of the following:

- Remove the `<handlerSettings>` from the *web.config* file locally and redeploy the app.
- Use the Kudu console to edit the *web.config* file and remove the `<handlerSettings>` section. Save the file.

For more information, see Log creation and redirection with the ASP.NET Core Module.

> ⚠ **Warning**
>
> Failure to disable the debug log can lead to app or server failure. There's no limit on log file size. Only use debug logging to troubleshoot app startup problems.
>
> For general logging in an ASP.NET Core app after startup, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Slow or nonresponsive app (Azure App Service)

When an app responds slowly or doesn't respond to a request, see Troubleshoot slow web app performance issues in Azure App Service.

Monitoring blades

Monitoring blades provide an alternative troubleshooting experience to the methods described earlier in the topic. These blades can be used to diagnose 500-series errors.

Confirm that the ASP.NET Core Extensions are installed. If the extensions aren't installed, install them manually:

1. In the **DEVELOPMENT TOOLS** blade section, select the **Extensions** blade.
2. The **ASP.NET Core Extensions** should appear in the list.
3. If the extensions aren't installed, select the **Add** button.
4. Choose the **ASP.NET Core Extensions** from the list.
5. Select **OK** to accept the legal terms.
6. Select **OK** on the **Add extension** blade.
7. An informational pop-up message indicates when the extensions are successfully installed.

If stdout logging isn't enabled, follow these steps:

1. In the Azure portal, select the **Advanced Tools** blade in the **DEVELOPMENT TOOLS** area. Select the **Go**→ button. The Kudu console opens in a new browser tab or window.
2. Using the navigation bar at the top of the page, open **Debug console** and select **CMD**.
3. Open the folders to the path **site** > **wwwroot** and scroll down to reveal the *web.config* file at the bottom of the list.
4. Click the pencil icon next to the *web.config* file.
5. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to: `\\?\%home%\LogFiles\stdout`.
6. Select **Save** to save the updated *web.config* file.

Proceed to activate diagnostic logging:

1. In the Azure portal, select the **Diagnostics logs** blade.
2. Select the **On** switch for **Application Logging (Filesystem)** and **Detailed error messages**. Select the **Save** button at the top of the blade.
3. To include failed request tracing, also known as Failed Request Event Buffering (FREB) logging, select the **On** switch for **Failed request tracing**.
4. Select the **Log stream** blade, which is listed immediately under the **Diagnostics logs** blade in the portal.

5. Make a request to the app.
6. Within the log stream data, the cause of the error is indicated.

Be sure to disable stdout logging when troubleshooting is complete.

To view the failed request tracing logs (FREB logs):

1. Navigate to the **Diagnose and solve problems** blade in the Azure portal.
2. Select **Failed Request Tracing Logs** from the **SUPPORT TOOLS** area of the sidebar.

See Failed request traces section of the Enable diagnostics logging for web apps in Azure App Service topic and the Application performance FAQs for Web Apps in Azure: How do I turn on failed request tracing? for more information.

For more information, see Enable diagnostics logging for web apps in Azure App Service.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

Troubleshoot on IIS

Application Event Log (IIS)

Access the Application Event Log:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app. Errors have a value of *IIS AspNetCore Module* or *IIS Express AspNetCore Module* in the *Source* column.

Run the app at a command prompt

Many startup errors don't produce useful information in the Application Event Log. You can find the cause of some errors by running the app at a command prompt on the hosting system.

Framework-dependent deployment

If the app is a framework-dependent deployment:

1. At a command prompt, navigate to the deployment folder and run the app by executing the app's assembly with *dotnet.exe*. In the following command, substitute the name of the app's assembly for <assembly_name>: `dotnet .\<assembly_name>.dll`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

Self-contained deployment

If the app is a self-contained deployment:

1. At a command prompt, navigate to the deployment folder and run the app's executable. In the following command, substitute the name of the app's assembly for <assembly_name>: `<assembly_name>.exe`.
2. The console output from the app, showing any errors, is written to the console window.
3. If the errors occur when making a request to the app, make a request to the host and port where Kestrel listens. Using the default host and post, make a request to `http://localhost:5000/`. If the app responds normally at the Kestrel endpoint address, the problem is more likely related to the hosting configuration and less likely within the app.

ASP.NET Core Module stdout log (IIS)

To enable and view stdout logs:

1. Navigate to the site's deployment folder on the hosting system.
2. If the *logs* folder isn't present, create the folder. For instructions on how to enable MSBuild to create the *logs* folder in the deployment automatically, see the Directory structure topic.
3. Edit the *web.config* file. Set **stdoutLogEnabled** to `true` and change the **stdoutLogFile** path to point to the *logs* folder (for example, `.\logs\stdout`).

`stdout` in the path is the log file name prefix. A timestamp, process id, and file extension are added automatically when the log is created. Using `stdout` as the file name prefix, a typical log file is named *stdout_20180205184032_5412.log*.

4. Ensure your application pool's identity has write permissions to the *logs* folder.
5. Save the updated *web.config* file.
6. Make a request to the app.
7. Navigate to the *logs* folder. Find and open the most recent stdout log.
8. Study the log for errors.

Disable stdout logging when troubleshooting is complete:

1. Edit the *web.config* file.
2. Set **stdoutLogEnabled** to `false`.
3. Save the file.

For more information, see ASP.NET Core Module (ANCM) for IIS.

> ⚠ **Warning**
>
> Failure to disable the stdout log can lead to app or server failure. There's no limit on log file size or the number of log files created.
>
> For routine logging in an ASP.NET Core app, use a logging library that limits log file size and rotates logs. For more information, see third-party logging providers.

ASP.NET Core Module debug log (IIS)

Add the following handler settings to the app's *web.config* file to enable ASP.NET Core Module debug log:

XML

```
<aspNetCore ...>
  <handlerSettings>
    <handlerSetting name="debugLevel" value="file" />
    <handlerSetting name="debugFile" value="c:\temp\ancm.log" />
  </handlerSettings>
</aspNetCore>
```

Confirm that the path specified for the log exists and that the app pool's identity has write permissions to the location.

For more information, see Log creation and redirection with the ASP.NET Core Module.

Enable the Developer Exception Page

The `ASPNETCORE_ENVIRONMENT` environment variable can be added to web.config to run the app in the Development environment. As long as the environment isn't overridden in app startup by `UseEnvironment` on the host builder, setting the environment variable allows the Developer Exception Page to appear when the app is run.

XML

```
<aspNetCore processPath="dotnet"
      arguments=".\MyApp.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout"
      hostingModel="InProcess">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
/>
  </environmentVariables>
</aspNetCore>
```

Setting the environment variable for `ASPNETCORE_ENVIRONMENT` is only recommended for use on staging and testing servers that aren't exposed to the Internet. Remove the environment variable from the *web.config* file after troubleshooting. For information on setting environment variables in *web.config*, see environmentVariables child element of aspNetCore.

Obtain data from an app

If an app is capable of responding to requests, obtain request, connection, and additional data from the app using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Slow or non-responding app (IIS)

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`. The app pool must have write access to the folder.

2. Run the EnableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\EnableDumps w3wp.exe c:\dumps
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\EnableDumps dotnet.exe c:\dumps
     ```

3. Run the app under the conditions that cause the crash to occur.

4. After the crash has occurred, run the DisableDumps PowerShell script:

   - If the app uses the in-process hosting model, run the script for *w3wp.exe*:

     ```
     Console

     .\DisableDumps w3wp.exe
     ```

   - If the app uses the out-of-process hosting model, run the script for *dotnet.exe*:

     ```
     Console

     .\DisableDumps dotnet.exe
     ```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> ⚠ **Warning**
>
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App doesn't respond, fails during startup, or runs normally

When an app stops responding but doesn't crash, fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.
2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install with the Windows desktop operating system and must be obtained separately from the NuGet website.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Additional resources

- Debug .NET and ASP.NET Core source code with Visual Studio
- Troubleshoot and debug ASP.NET Core projects
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Handle errors in ASP.NET Core
- ASP.NET Core Module (ANCM) for IIS

Azure documentation

- Application Insights for ASP.NET Core
- Remote debugging web apps section of Troubleshoot a web app in Azure App Service using Visual Studio
- Azure App Service diagnostics overview
- How to: Monitor Apps in Azure App Service
- Troubleshoot a web app in Azure App Service using Visual Studio
- Troubleshoot HTTP errors of "502 bad gateway" and "503 service unavailable" in your Azure web apps
- Troubleshoot slow web app performance issues in Azure App Service
- Application performance FAQs for Web Apps in Azure
- Azure Web App sandbox (App Service runtime execution limitations)

Visual Studio documentation

- Remote Debug ASP.NET Core on IIS in Azure in Visual Studio 2017
- Remote Debug ASP.NET Core on a Remote IIS Computer in Visual Studio 2017
- Learn to debug using Visual Studio

Visual Studio Code documentation

- Debugging with Visual Studio Code

Common error troubleshooting for Azure App Service and IIS with ASP.NET Core

Article • 07/26/2024

This topic describes the most common errors and provides troubleshooting advice when hosting ASP.NET Core apps on Azure Apps Service and IIS.

See Troubleshoot ASP.NET Core on Azure App Service and IIS information on common app startup errors and instructions on how to diagnose errors.

Collect the following information:

- Browser behavior such as status code and error message.
- Application Event Log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS

    1. Select **Start** on the **Windows** menu, type *Event Viewer*, and press **Enter**.
    2. After the **Event Viewer** opens, expand **Windows Logs** > **Application** in the sidebar.

- ASP.NET Core Module stdout and debug log entries
  - Azure App Service: See Troubleshoot ASP.NET Core on Azure App Service and IIS.
  - IIS: Follow the instructions in the Log creation and redirection and Enhanced diagnostic logs sections of the ASP.NET Core Module topic.

Compare error information to the following common errors. If a match is found, follow the troubleshooting advice.

The list of errors in this topic isn't exhaustive. If you encounter an error not listed here, open a new issue using the **Content feedback** button at the bottom of this topic with detailed instructions on how to reproduce the error.

> **Important**
>
> **ASP.NET Core preview releases with Azure App Service**

ASP.NET Core preview releases aren't deployed to Azure App Service by default. To host an app that uses an ASP.NET Core preview release, see Deploy ASP.NET Core preview release to Azure App Service.

OS upgrade removed the 32-bit ASP.NET Core Module

Application Log: The Module DLL **C:\WINDOWS\system32\inetsrv\aspnetcore.dll** failed to load. The data is the error.

Troubleshooting:

Non-OS files in the **C:\Windows\SysWOW64\inetsrv** directory aren't preserved during an OS upgrade. If the ASP.NET Core Module is installed prior to an OS upgrade and then any app pool is run in 32-bit mode after an OS upgrade, this issue is encountered. After an OS upgrade, repair the ASP.NET Core Module. See Install the .NET Core Hosting bundle. Select **Repair** when the installer is run.

Missing site extension, 32-bit (x86) and 64-bit (x64) site extensions installed, or wrong process bitness set

Applies to apps hosted by Azure App Services.

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found. Failed to start application '/LM/W3SVC/1416782824/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and

Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff. Could not find inprocess request handler. It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}-preview-*' was not found.

Troubleshooting:

- If running the app on a preview runtime, install either the 32-bit (x86) **or** 64-bit (x64) site extension that matches the bitness of the app and the app's runtime version. **Don't install both extensions or multiple runtime versions of the extension.**
  - ASP.NET Core {RUNTIME VERSION} (x86) Runtime
  - ASP.NET Core {RUNTIME VERSION} (x64) Runtime

  Restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and both the 32-bit (x86) and 64-bit (x64) site extensions are installed, uninstall the site extension that doesn't match the bitness of the app. After removing the site extension, restart the app. Wait several seconds for the app to restart.

- If running the app on a preview runtime and the site extension's bitness matches that of the app, confirm that the preview site extension's *runtime version* matches the app's runtime version.

- Confirm that the app's **Platform** in **Application Settings** matches the bitness of the app.

For more information, see Deploy ASP.NET Core apps to Azure App Service.

An x86 app is deployed but the app pool isn't enabled for 32-bit apps

- **Browser:** HTTP Error 500.30 - ANCM In-Process Start Failure

- **Application Log:** Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' hit unexpected managed exception, exception code = '0xe0434352'. Please check the stderr logs for more information. Application '/LM/W3SVC/5/ROOT' with physical root '{PATH}' failed to load clr and managed application. CLR worker thread exited prematurely

- **ASP.NET Core Module stdout Log:** The log file is created but empty.

- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8007023e

This scenario is trapped by the SDK when publishing a self-contained app. The SDK produces an error if the RID doesn't match the platform target (for example, `win10-x64` RID with `<PlatformTarget>x86</PlatformTarget>` in the project file).

Troubleshooting:

For an x86 framework-dependent deployment (`<PlatformTarget>x86</PlatformTarget>`), enable the IIS app pool for 32-bit apps. In IIS Manager, open the app pool's **Advanced Settings** and set **Enable 32-Bit Applications** to **True**.

Platform conflicts with RID

- **Browser:** HTTP Error 502.5 - Process Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"C:{PATH}{ASSEMBLY}.{exe|dll}" ', ErrorCode = '0x80004005 : ff.
- **ASP.NET Core Module stdout Log:** Unhandled Exception: System.BadImageFormatException: Could not load file or assembly '{ASSEMBLY}.dll'. An attempt was made to load a program with an incorrect format.

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- If this exception occurs for an Azure Apps deployment when upgrading an app and deploying newer assemblies, manually delete all files from the prior deployment. Lingering incompatible assemblies can result in a `System.BadImageFormatException` exception when deploying an upgraded app.

URI endpoint wrong or stopped website

- **Browser:** ERR_CONNECTION_REFUSED **--OR--** Unable to connect
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm the correct URI endpoint for the app is in use. Check the bindings.
- Confirm that the IIS website isn't in the *Stopped* state.

CoreWebEngine or W3SVC server features disabled

OS Exception: The IIS 7.0 CoreWebEngine and W3SVC features must be installed to use the ASP.NET Core Module.

Troubleshooting:

Confirm that the proper role and features are enabled. See IIS Configuration.

Incorrect website physical path or app missing

- **Browser:** 403 Forbidden - Access is denied **--OR--** 403.14 Forbidden - The Web server is configured to not list the contents of this directory.
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Check the IIS website **Basic Settings** and the physical app folder. Confirm that the app is in the folder at the IIS website **Physical path**.

Incorrect role, ASP.NET Core Module not installed, or incorrect permissions

- **Browser:** 500.19 Internal Server Error - The requested page cannot be accessed because the related configuration data for the page is invalid. **--OR--** This page can't be displayed
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.

- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

- Confirm that the proper role is enabled. See IIS Configuration.
- Open **Programs & Features** or **Apps & features** and confirm that **Windows Server Hosting** is installed. If **Windows Server Hosting** isn't present in the list of installed programs, download and install the .NET Core Hosting Bundle.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.
- Make sure that the **Application Pool** > **Process Model** > **Identity** is set to **ApplicationPoolIdentity** or the custom identity has the correct permissions to access the app's deployment folder.
- If you uninstalled the ASP.NET Core Hosting Bundle and installed an earlier version of the hosting bundle, the *applicationHost.config* file doesn't include a section for the ASP.NET Core Module. Open *applicationHost.config* at *%windir%/System32/inetsrv/config* and find the `<configuration><configSections><sectionGroup name="system.webServer">` section group. If the section for the ASP.NET Core Module is missing from the section group, add the section element:

  ```
  XML

  <section name="aspNetCore" overrideModeDefault="Allow" />
  ```

  Alternatively, install the latest version of the ASP.NET Core Hosting Bundle. The latest version is backwards-compatible with supported ASP.NET Core apps.

Incorrect processPath, missing PATH variable, Hosting Bundle not installed, system/IIS not restarted, VC++ Redistributable not installed, or dotnet.exe access violation

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Application 'MACHINE/WEBROOT/APPHOST/{ASSEMBLY}' with physical root 'C:{PATH}' failed to start process with commandline '"{...}" ', ErrorCode = '0x80070002 : 0. Application '{PATH}' wasn't able to start. Executable was not

found at '{PATH}'. Failed to start application '/LM/W3SVC/2/ROOT', ErrorCode '0x8007023e'.

- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** Event Log: 'Application '{PATH}' wasn't able to start. Executable was not found at '{PATH}'. Failed HRESULT returned: 0x8007023e

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Check the *processPath* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's `dotnet` for a framework-dependent deployment (FDD) or `.\{ASSEMBLY}.exe` for a self-contained deployment (SCD).
- For an FDD, *dotnet.exe* might not be accessible via the PATH settings. Confirm that *C:\Program Files\dotnet* exists in the System PATH settings.
- For an FDD, *dotnet.exe* might not be accessible for the user identity of the app pool. Confirm that the app pool user identity has access to the *C:\Program Files\dotnet* directory. Confirm that there are no deny rules configured for the app pool user identity on the *C:\Program Files\dotnet* and app directories.
- An FDD may have been deployed and .NET Core installed without restarting IIS. Either restart the server or restart IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.
- An FDD may have been deployed without installing the .NET Core runtime on the hosting system. If the .NET Core runtime hasn't been installed, run the **.NET Core Hosting Bundle installer** on the system.

  Current .NET Core Hosting Bundle installer (direct download)

  For more information, see Install the .NET Core Hosting Bundle.

  If a specific runtime is required, download the runtime from the .NET Downloads page and install it on the system. Complete the installation by restarting the system or restarting IIS by executing **net stop was /y** followed by **net start w3svc** from a command prompt.

Incorrect arguments of <aspNetCore> element

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure
- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed to start application '/LM/W3SVC/3/ROOT', ErrorCode '0x8000ffff'.
- **ASP.NET Core Module stdout Log:** Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409
- **ASP.NET Core Module Debug Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Failed HRESULT returned: 0x8000ffff Could not find inprocess request handler. Captured output from invoking hostfxr: Did you mean to run dotnet SDK commands? Please install dotnet SDK from: https://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409 Failed HRESULT returned: 0x8000ffff

Troubleshooting:

- Confirm that the app runs locally on Kestrel. A process failure might be the result of a problem within the app. For more information, see Troubleshoot ASP.NET Core on Azure App Service and IIS.
- Examine the *arguments* attribute on the `<aspNetCore>` element in *web.config* to confirm that it's either (a) `.\{ASSEMBLY}.dll` for a framework-dependent deployment (FDD); or (b) not present, an empty string (`arguments=""`), or a list of the app's arguments (`arguments="{ARGUMENT_1}, {ARGUMENT_2}, ... {ARGUMENT_X}"`) for a self-contained deployment (SCD).

Missing .NET Core shared framework

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure

- **Application Log:** Invoking hostfxr to find the inprocess request handler failed without finding any native dependencies. This most likely means the app is misconfigured, please check the versions of Microsoft.NetCore.App and Microsoft.AspNetCore.App that are targeted by the application and are installed on the machine. Could not find inprocess request handler. Captured output from invoking hostfxr: It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.

Failed to start application '/LM/W3SVC/5/ROOT', ErrorCode '0x8000ffff'.

- **ASP.NET Core Module stdout Log:** It was not possible to find any compatible framework version. The specified framework 'Microsoft.AspNetCore.App', version '{VERSION}' was not found.
- **ASP.NET Core Module Debug Log:** Failed HRESULT returned: 0x8000ffff

Troubleshooting:

For a framework-dependent deployment (FDD), confirm that the correct runtime installed on the system.

Stopped Application Pool

- **Browser:** 503 Service Unavailable
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module Debug Log:** The log file isn't created.

Troubleshooting:

Confirm that the Application Pool isn't in the *Stopped* state.

Sub-application includes a <handlers> section

- **Browser:** HTTP Error 500.19 - Internal Server Error
- **Application Log:** No entry
- **ASP.NET Core Module stdout Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

- **ASP.NET Core Module Debug Log:** The root app's log file is created and shows normal operation. The sub-app's log file isn't created.

Troubleshooting:

Confirm that the sub-app's *web.config* file doesn't include a `<handlers>` section or that the sub-app doesn't inherit the parent app's handlers.

The parent app's `<system.webServer>` section of *web.config* is placed inside of a `<location>` element. The InheritInChildApplications property is set to `false` to indicate that the settings specified within the <location> element aren't inherited by apps that reside in a subdirectory of the parent app. For more information, see ASP.NET Core Module (ANCM) for IIS.

stdout log path incorrect

- **Browser:** The app responds normally.
- **Application Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.
- **ASP.NET Core Module stdout Log:** The log file isn't created.
- **ASP.NET Core Module debug Log:** Could not start stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070005 returned at {PATH}\aspnetcoremodulev2\commonlib\fileoutputmanager.cpp:84. Could not stop stdout redirection in C:\Program Files\IIS\Asp.Net Core Module\V2\aspnetcorev2.dll. Exception message: HRESULT 0x80070002 returned at {PATH}. Could not start stdout redirection in {PATH}\aspnetcorev2_inprocess.dll.

Troubleshooting:

- The `stdoutLogFile` path specified in the `<aspNetCore>` element of *web.config* doesn't exist. For more information, see ASP.NET Core Module: Log creation and redirection.
- The app pool user doesn't have write access to the stdout log path.

Application configuration general issue

- **Browser:** HTTP Error 500.0 - ANCM In-Process Handler Load Failure **--OR--** HTTP Error 500.30 - ANCM In-Process Start Failure
- **Application Log:** Variable
- **ASP.NET Core Module stdout Log:** The log file is created but empty or created with normal entries until the point of the app failing.
- **ASP.NET Core Module Debug Log:** Variable

Troubleshooting:

The process failed to start, most likely due to an app configuration or programming issue.

For more information, see the following topics:

- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Troubleshoot and debug ASP.NET Core projects

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Publish an ASP.NET Core app to IIS

Article • 09/17/2023

This tutorial shows how to host an ASP.NET Core app on an IIS server.

This tutorial covers the following subjects:

- ✓ Install the .NET Core Hosting Bundle on Windows Server.
- ✓ Create an IIS site in IIS Manager.
- ✓ Deploy an ASP.NET Core app.

Prerequisites

- .NET Core SDK installed on the development machine.
- Windows Server configured with the **Web Server (IIS)** server role. If your server isn't configured to host websites with IIS, follow the guidance in the *IIS configuration* section of the Host ASP.NET Core on Windows with IIS article and then return to this tutorial.

> ⚠ **Warning**
>
> **IIS configuration and website security involve concepts that aren't covered by this tutorial.** Consult the IIS guidance in the Microsoft IIS documentation and the ASP.NET Core article on hosting with IIS before hosting production apps on IIS.
>
> Important scenarios for IIS hosting not covered by this tutorial include:
>
> - Creation of a registry hive for ASP.NET Core Data Protection
> - Configuration of the app pool's Access Control List (ACL)
> - To focus on IIS deployment concepts, this tutorial deploys an app without HTTPS security configured in IIS. For more information on hosting an app enabled for HTTPS protocol, see the security topics in the Additional resources section of this article. Further guidance for hosting ASP.NET Core apps is provided in the Host ASP.NET Core on Windows with IIS article.

Install the .NET Core Hosting Bundle

Install the *.NET Core Hosting Bundle* on the IIS server. The bundle installs the .NET Core Runtime, .NET Core Library, and the ASP.NET Core Module. The module allows ASP.NET Core apps to run behind IIS.

Download the installer using the following link:

Current .NET Core Hosting Bundle installer (direct download)

1. Run the installer on the IIS server.
2. Restart the server or execute `net stop was /y` followed by `net start w3svc` in a command shell.

Create the IIS site

1. On the IIS server, create a folder to contain the app's published folders and files. In a following step, the folder's path is provided to IIS as the physical path to the app. For more information on an app's deployment folder and file layout, see ASP.NET Core directory structure.
2. In IIS Manager, open the server's node in the **Connections** panel. Right-click the **Sites** folder. Select **Add Website** from the contextual menu.
3. Provide a **Site name** and set the **Physical path** to the app's deployment folder that you created. Provide the **Binding** configuration and create the website by selecting **OK**.

   > ⚠ **Warning**
   >
   > Top-level wildcard bindings (`http://*:80/` and `http://+:80`) should **not** be used. Top-level wildcard bindings can open up your app to security vulnerabilities. This applies to both strong and weak wildcards. Use explicit host names rather than wildcards. Subdomain wildcard binding (for example, `*.mysub.com`) doesn't have this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). See RFC 9110: HTTP Semantics (Section 7.2. Host and :authority) for more information.

4. Confirm the process model identity has the proper permissions.

   If the default identity of the app pool (**Process Model** > **Identity**) is changed from `ApplicationPoolIdentity` to another identity, verify that the new identity has the required permissions to access the app's folder, database, and other required

resources. For example, the app pool requires read and write access to folders where the app reads and writes files.

Create an ASP.NET Core Razor Pages app

Follow the Get started with ASP.NET Core tutorial to create a Razor Pages app.

Publish and deploy the app

Publish an app means to produce a compiled app that can be hosted by a server. *Deploy an app* means to move the published app to a hosting system. The publish step is handled by the .NET Core SDK, while the deployment step can be handled by a variety of approaches. This tutorial adopts the *folder* deployment approach, where:

- The app is published to a folder.
- The folder's contents are moved to the IIS site's folder (the **Physical path** to the site in IIS Manager).

Visual Studio

1. Right-click on the project in **Solution Explorer** and select **Publish**.
2. In the **Pick a publish target** dialog, select the **Folder** publish option.
3. Set the **Folder or File Share** path.

    - If you created a folder for the IIS site that's available on the development machine as a network share, provide the path to the share. The current user must have write access to publish to the share.
    - If you're unable to deploy directly to the IIS site folder on the IIS server, publish to a folder on removable media and physically move the published app to the IIS site folder on the server, which is the site's **Physical path** in IIS Manager. Move the contents of the `bin/Release/{TARGET FRAMEWORK}/publish` folder to the IIS site folder on the server, which is the site's **Physical path** in IIS Manager.

4. Select the **Publish** button.

Browse the website

The app is accessible in a browser after it receives the first request. Make a request to the app at the endpoint binding that you established in IIS Manager for the site.

Next steps

In this tutorial, you learned how to:

- ✓ Install the .NET Core Hosting Bundle on Windows Server.
- ✓ Create an IIS site in IIS Manager.
- ✓ Deploy an ASP.NET Core app.

To learn more about hosting ASP.NET Core apps on IIS, see the IIS Overview article:

Host ASP.NET Core on Windows with IIS

Additional resources

Articles in the ASP.NET Core documentation set

- ASP.NET Core Module (ANCM) for IIS
- ASP.NET Core directory structure
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Enforce HTTPS in ASP.NET Core
- WebSockets on IIS

Articles pertaining to ASP.NET Core app deployment

- Publish an ASP.NET Core app to Azure with Visual Studio
- Publish an ASP.NET Core app to Azure with Visual Studio Code
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

Articles on IIS HTTPS configuration

- Configuring SSL in IIS Manager
- How to Set Up SSL on IIS

Articles on IIS and Windows Server

- The Official Microsoft IIS Site
- Windows Server technical content library

Deployment resources for IIS administrators

- IIS documentation
- Getting Started with the IIS Manager in IIS
- .NET Core application deployment
- ASP.NET Core Module (ANCM) for IIS
- ASP.NET Core directory structure
- IIS modules with ASP.NET Core
- Troubleshoot ASP.NET Core on Azure App Service and IIS
- Common error troubleshooting for Azure App Service and IIS with ASP.NET Core
- Sticky sessions with Application Request Routing

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Host ASP.NET Core in a Windows Service

Article • 09/27/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

An ASP.NET Core app can be hosted on Windows as a Windows Service without using IIS. When hosted as a Windows Service, the app automatically starts after server reboots.

Prerequisites

- ASP.NET Core SDK 7.0 or later
- PowerShell 6.2 or later

Worker Service template

The ASP.NET Core Worker Service template provides a starting point for writing long running service apps. To use the template as a basis for a Windows Service app:

1. Create a Worker Service app from the .NET Core template.
2. Install the NuGet package Microsoft.Extensions.Hosting.WindowsServices.
3. Follow the guidance in the App configuration section to update the Worker Service app so that it can run as a Windows Service.

Visual Studio

1. Create a new project.
2. Select **Worker Service**. Select **Next**.
3. Provide a project name in the **Project name** field or accept the default project name. Select **Create**.
4. In the **Create a new Worker service** dialog, select **Create**.

App configuration

Update Program.cs to call AddWindowsService. When the app is running as a Windows Service, `AddWindowsService`:

- Sets the host lifetime to `WindowsServiceLifetime`.
- Sets the content root to AppContext.BaseDirectory. For more information, see the Current directory and content root section.
- Enables logging to the event log:
  - The application name is used as the default source name.
  - The default log level is *Warning* or higher for an app based on an ASP.NET Core template that calls `CreateDefaultBuilder` to build the host.
  - Override the default log level with the `Logging:EventLog:LogLevel:Default` key in `appsettings.json`/`appsettings.{Environment}.json` or other configuration provider.
  - Only administrators can create new event sources. When an event source can't be created using the application name, a warning is logged to the *Application* source and event logs are disabled.

Consider the following `ServiceA` class:

C#

```
namespace SampleApp.Services;

public class ServiceA : BackgroundService
{
    public ServiceA(ILoggerFactory loggerFactory)
    {
        Logger = loggerFactory.CreateLogger<ServiceA>();
    }

    public ILogger Logger { get; }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        Logger.LogInformation("ServiceA is starting.");

        stoppingToken.Register(() => Logger.LogInformation("ServiceA is
stopping."));

        while (!stoppingToken.IsCancellationRequested)
        {
            Logger.LogInformation("ServiceA is doing background work.");

            await Task.Delay(TimeSpan.FromSeconds(5), stoppingToken);
        }

        Logger.LogInformation("ServiceA has stopped.");
```

```
    }
}
```

The following `Program.cs` calls AddHostedService to register `ServiceA`:

C#

```
using SampleApp.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddWindowsService();
builder.Services.AddHostedService<ServiceA>();

var app = builder.Build();

app.MapRazorPages();

app.Run();
```

The following sample apps accompany this topic:

- Background Worker Service Sample: A non-web app sample based on the Worker Service template that uses hosted services for background tasks.
- Web App Service Sample: A Razor Pages web app sample that runs as a Windows Service with hosted services for background tasks.

For MVC guidance, see the articles under Overview of ASP.NET Core MVC and Migrate from ASP.NET Core 2.2 to 3.0.

Deployment type

For information and advice on deployment scenarios, see .NET Core application deployment.

SDK

For a web app-based service that uses the Razor Pages or MVC frameworks, specify the Web SDK in the project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

If the service only executes background tasks (for example, hosted services), specify the Worker SDK in the project file:

```xml
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

Framework-dependent deployment (FDD)

Framework-dependent deployment (FDD) relies on the presence of a shared system-wide version of .NET Core on the target system. When the FDD scenario is adopted following the guidance in this article, the SDK produces an executable (*.exe*), called a *framework-dependent executable*.

If using the Web SDK, a *web.config* file, which is normally produced when publishing an ASP.NET Core app, is unnecessary for a Windows Services app. To disable the creation of the *web.config* file, add the `<IsTransformWebConfigDisabled>` property set to `true`.

```xml
<PropertyGroup>
  <TargetFramework>net7.0</TargetFramework>
  <IsTransformWebConfigDisabled>true</IsTransformWebConfigDisabled>
</PropertyGroup>
```

Self-contained deployment (SCD)

Self-contained deployment (SCD) doesn't rely on the presence of a shared framework on the host system. The runtime and the app's dependencies are deployed with the app.

A Windows Runtime Identifier (RID) is included in the `<PropertyGroup>` that contains the target framework:

```xml
<RuntimeIdentifier>win-x64</RuntimeIdentifier>
```

To publish for multiple RIDs:

- Provide the RIDs in a semicolon-delimited list.
- Use the property name `<RuntimeIdentifiers>` (plural).

For more information, see .NET Core RID Catalog.

Service user account

To create a user account for a service, use the New-LocalUser cmdlet from an administrative PowerShell 6 command shell.

On Windows 10 October 2018 Update (version 1809/build 10.0.17763) or later:

```
PowerShell
New-LocalUser -Name {SERVICE NAME}
```

On Windows OS earlier than the Windows 10 October 2018 Update (version 1809/build 10.0.17763):

```
Console
powershell -Command "New-LocalUser -Name {SERVICE NAME}"
```

Provide a strong password when prompted.

Unless the `-AccountExpires` parameter is supplied to the New-LocalUser cmdlet with an expiration DateTime, the account doesn't expire.

For more information, see Microsoft.PowerShell.LocalAccounts and Service User Accounts.

An alternative approach to managing users when using Active Directory is to use Managed Service Accounts. For more information, see Group Managed Service Accounts Overview.

Log on as a service rights

To establish *Log on as a service* rights for a service user account:

1. Open the Local Security Policy editor by running *secpol.msc*.
2. Expand the **Local Policies** node and select **User Rights Assignment**.
3. Open the **Log on as a service** policy.
4. Select **Add User or Group**.
5. Provide the object name (user account) using either of the following approaches:
   a. Type the user account (`{DOMAIN OR COMPUTER NAME\USER}`) in the object name field and select **OK** to add the user to the policy.
   b. Select **Advanced**. Select **Find Now**. Select the user account from the list. Select **OK**. Select **OK** again to add the user to the policy.

6. Select **OK** or **Apply** to accept the changes.

Create and manage the Windows Service

Create a service

Use PowerShell commands to register a service. From an administrative PowerShell 6 command shell, execute the following commands:

PowerShell

```
$acl = Get-Acl "{EXE PATH}"
$aclRuleArgs = "{DOMAIN OR COMPUTER NAME\USER}",
"Read,Write,ReadAndExecute", "ContainerInherit,ObjectInherit", "None",
"Allow"
$accessRule = New-Object
System.Security.AccessControl.FileSystemAccessRule($aclRuleArgs)
$acl.SetAccessRule($accessRule)
$acl | Set-Acl "{EXE PATH}"

New-Service -Name {SERVICE NAME} -BinaryPathName "{EXE FILE PATH} --
contentRoot {EXE FOLDER PATH}" -Credential "{DOMAIN OR COMPUTER NAME\USER}"
-Description "{DESCRIPTION}" -DisplayName "{DISPLAY NAME}" -StartupType
Automatic
```

- `{EXE PATH}`: Path of the app's executable on the host (for example, `d:\myservice`). Don't include the app's executable file name in the path. A trailing slash isn't required.
- `{DOMAIN OR COMPUTER NAME\USER}`: Service user account (for example, `Contoso\ServiceUser`).
- `{SERVICE NAME}`: Service name (for example, `MyService`).
- `{EXE FILE PATH}`: The app's full executable path (for example, `d:\myservice\myservice.exe`). Include the executable's file name with extension.
- `{EXE FOLDER PATH}`: The app's full executable folder path (for example `d:\myservice`).
- `{DESCRIPTION}`: Service description (for example, `My sample service`).
- `{DISPLAY NAME}`: Service display name (for example, `My Service`).

Start a service

Start a service with the following PowerShell 6 command:

PowerShell

```
Start-Service -Name {SERVICE NAME}
```

The command takes a few seconds to start the service.

Determine a service's status

To check the status of a service, use the following PowerShell 6 command:

```PowerShell
Get-Service -Name {SERVICE NAME}
```

The status is reported as one of the following values:

- `Starting`
- `Running`
- `Stopping`
- `Stopped`

Stop a service

Stop a service with the following PowerShell 6 command:

```PowerShell
Stop-Service -Name {SERVICE NAME}
```

Remove a service

After a short delay to stop a service, remove a service with the following PowerShell 6 command:

```PowerShell
Remove-Service -Name {SERVICE NAME}
```

Proxy server and load balancer scenarios

Services that interact with requests from the Internet or a corporate network and are behind a proxy or load balancer might require additional configuration. For more

information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Configure endpoints

By default, ASP.NET Core binds to `http://localhost:5000`. Configure the URL and port by setting the `ASPNETCORE_URLS` environment variable.

For additional URL and port configuration approaches, see the relevant server article:

- Configure endpoints for the ASP.NET Core Kestrel web server
- HTTP.sys web server implementation in ASP.NET Core

The preceding guidance covers support for HTTPS endpoints. For example, configure the app for HTTPS when authentication is used with a Windows Service.

> **Note**
>
> Use of the ASP.NET Core HTTPS development certificate to secure a service endpoint isn't supported.

Current directory and content root

The current working directory returned by calling GetCurrentDirectory for a Windows Service is the *C:\WINDOWS\system32* folder. The *system32* folder isn't a suitable location to store a service's files (for example, settings files). Use one of the following approaches to maintain and access a service's assets and settings files.

Use ContentRootPath or ContentRootFileProvider

Use IHostEnvironment.ContentRootPath or ContentRootFileProvider to locate an app's resources.

When the app runs as a service, UseWindowsService sets the ContentRootPath to AppContext.BaseDirectory.

The app's default settings files, `appsettings.json` and `appsettings.{Environment}.json`, are loaded from the app's content root by calling CreateDefaultBuilder during host construction.

For other settings files loaded by developer code in ConfigureAppConfiguration, there's no need to call SetBasePath. In the following example, the `custom_settings.json` file

exists in the app's content root and is loaded without explicitly setting a base path:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseWindowsService()
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("custom_settings.json");
            })
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHostedService<Worker>();
            });
}
```

Don't attempt to use GetCurrentDirectory to obtain a resource path because a Windows Service app returns the *C:\WINDOWS\system32* folder as its current directory.

Store a service's files in a suitable location on disk

Specify an absolute path with SetBasePath when using an IConfigurationBuilder to the folder containing the files.

Troubleshoot

To troubleshoot a Windows Service app, see Troubleshoot and debug ASP.NET Core projects.

Common errors

- An old or pre-release version of PowerShell is in use.
- The registered service doesn't use the app's **published** output from the dotnet publish command. Output of the dotnet build command isn't supported for app deployment. Published assets are found in either of the following folders depending on the deployment type:
  - *bin/Release/{TARGET FRAMEWORK}/publish* (FDD)

    - *bin/Release/{TARGET FRAMEWORK}/{RUNTIME IDENTIFIER}/publish* (SCD)
- The service isn't in the RUNNING state.
- The paths to resources that the app uses (for example, certificates) are incorrect. The base path of a Windows Service is *c:\Windows\System32*.
- The user doesn't have *Log on as a service* rights.
- The user's password is expired or incorrectly passed when executing the `New-Service` PowerShell command.
- The app requires ASP.NET Core authentication but isn't configured for secure connections (HTTPS).
- The request URL port is incorrect or not configured correctly in the app.

System and Application Event Logs

Access the System and Application Event Logs:

1. Open the Start menu, search for *Event Viewer*, and select the **Event Viewer** app.
2. In **Event Viewer**, open the **Windows Logs** node.
3. Select **System** to open the System Event Log. Select **Application** to open the Application Event Log.
4. Search for errors associated with the failing app.

Run the app at a command prompt

Many startup errors don't produce useful information in the event logs. You can find the cause of some errors by running the app at a command prompt on the hosting system. To log additional detail from the app, lower the log level or run the app in the Development environment.

Clear package caches

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Delete the *bin* and *obj* folders.

2. Clear the package caches by executing dotnet nuget locals all --clear from a command shell.

   Clearing package caches can also be accomplished with the nuget.exe tool and executing the command `nuget locals all -clear`. *nuget.exe* isn't a bundled install

with the Windows desktop operating system and must be obtained separately from the NuGet website.

3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

Slow or unresponsive app

A *crash dump* is a snapshot of the system's memory and can help determine the cause of an app crash, startup failure, or slow app.

App crashes or encounters an exception

Obtain and analyze a dump from Windows Error Reporting (WER):

1. Create a folder to hold crash dump files at `c:\dumps`.
2. Run the EnableDumps PowerShell script with the application executable name:

```
PowerShell
.\EnableDumps {APPLICATION EXE} c:\dumps
```

3. Run the app under the conditions that cause the crash to occur.
4. After the crash has occurred, run the DisableDumps PowerShell script:

```
PowerShell
.\DisableDumps {APPLICATION EXE}
```

After an app crashes and dump collection is complete, the app is allowed to terminate normally. The PowerShell script configures WER to collect up to five dumps per app.

> ⚠ **Warning**
>
> Crash dumps might take up a large amount of disk space (up to several gigabytes each).

App is unresponsive, fails during startup, or runs normally

When an app stops responding but doesn't crash, fails during startup, or runs normally, see User-Mode Dump Files: Choosing the Best Tool to select an appropriate tool to produce the dump.

Analyze the dump

A dump can be analyzed using several approaches. For more information, see Analyzing a User-Mode Dump File.

Additional resources

- View or download sample code (how to download)
- Kestrel endpoint configuration (includes HTTPS configuration and SNI support)
- .NET Generic Host in ASP.NET Core
- Troubleshoot and debug ASP.NET Core projects

Host ASP.NET Core on Linux with Nginx

Article • 03/14/2024

By Sourabh Shirhatti

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

This guide explains setting up a production-ready ASP.NET Core environment for Ubuntu, Red Hat Enterprise (RHEL), and SUSE Linux Enterprise Server.

For information on other Linux distributions supported by ASP.NET Core, see Prerequisites for .NET Core on Linux.

This guide:

- Places an existing ASP.NET Core app behind a reverse proxy server.
- Sets up the reverse proxy server to forward requests to the Kestrel web server.
- Ensures the web app runs on startup as a daemon.
- Configures a process management tool to help restart the web app.

Prerequisites

Ubuntu

- Access to Ubuntu 20.04 with a standard user account with sudo privilege.
- The latest stable .NET runtime installed on the server.
- An existing ASP.NET Core app.

At any point in the future after upgrading the shared framework, restart the ASP.NET Core apps hosted by the server.

Publish and copy over the app

Configure the app for a framework-dependent deployment.

If the app is run locally in the Development environment and isn't configured by the server to make secure HTTPS connections, adopt either of the following approaches:

- Configure the app to handle secure local connections. For more information, see the HTTPS configuration section.
- Configure the app to run at the insecure endpoint:
  - Deactivate HTTPS Redirection Middleware in the Development environment (`Program.cs`):

    C#

    ```
    if (!app.Environment.IsDevelopment())
    {
        app.UseHttpsRedirection();
    }
    ```

    For more information, see Use multiple environments in ASP.NET Core.
  - Remove `https://localhost:5001` (if present) from the `applicationUrl` property in the `Properties/launchSettings.json` file.

For more information on configuration by environment, see Use multiple environments in ASP.NET Core.

Run dotnet publish from the development environment to package an app into a directory (for example, `bin/Release/{TARGET FRAMEWORK MONIKER}/publish`, where the `{TARGET FRAMEWORK MONIKER}` placeholder is the Target Framework Moniker (TFM)) that can run on the server:

.NET CLI

```
dotnet publish --configuration Release
```

The app can also be published as a self-contained deployment if you prefer not to maintain the .NET Core runtime on the server.

Copy the ASP.NET Core app to the server using a tool that integrates into the organization's workflow (for example, `SCP`, `SFTP`). It's common to locate web apps under the `var` directory (for example, `var/www/helloapp`).

> **Note**

> Under a production deployment scenario, a continuous integration workflow does the work of publishing the app and copying the assets to the server.

Test the app:

1. From the command line, run the app: `dotnet <app_assembly>.dll`.
2. In a browser, navigate to `http://<serveraddress>:<port>` to verify the app works on Linux locally.

Configure a reverse proxy server

A reverse proxy is a common setup for serving dynamic web apps. A reverse proxy terminates the HTTP request and forwards it to the ASP.NET Core app.

Use a reverse proxy server

Kestrel is great for serving dynamic content from ASP.NET Core. However, the web serving capabilities aren't as feature rich as servers such as IIS, Apache, or Nginx. A reverse proxy server can offload work such as serving static content, caching requests, compressing requests, and HTTPS termination from the HTTP server. A reverse proxy server may reside on a dedicated machine or may be deployed alongside an HTTP server.

For the purposes of this guide, a single instance of Nginx is used. It runs on the same server, alongside the HTTP server. Based on requirements, a different setup may be chosen.

Because requests are forwarded by reverse proxy, use the Forwarded Headers Middleware from the Microsoft.AspNetCore.HttpOverrides package, which is automatically included in ASP.NET Core apps via the shared framework's Microsoft.AspNetCore.App metapackage. The middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header, so that redirect URIs and other security policies work correctly.

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. To run Forwarded Headers Middleware after diagnostics and error handling middleware, see Forwarded Headers Middleware order.

Invoke the UseForwardedHeaders method before calling other middleware. Configure the middleware to forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication();

var app = builder.Build();

app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();

app.MapGet("/", () => "Hello ForwardedHeadersOptions!");

app.Run();
```

If no ForwardedHeadersOptions are specified to the middleware, the default headers to forward are `None`.

Proxies running on loopback addresses (`127.0.0.0/8`, `[::1]`), including the standard localhost address (`127.0.0.1`), are trusted by default. If other trusted proxies or networks within the organization handle requests between the internet and the web server, add them to the list of KnownProxies or KnownNetworks with ForwardedHeadersOptions. The following example adds a trusted proxy server at IP address `10.0.0.100` to the Forwarded Headers Middleware `KnownProxies`:

C#

```
using Microsoft.AspNetCore.HttpOverrides;
using System.Net;

var builder = WebApplication.CreateBuilder(args);

// Configure forwarded headers
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});

builder.Services.AddAuthentication();

var app = builder.Build();

app.UseForwardedHeaders(new ForwardedHeadersOptions
```

```
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto
});

app.UseAuthentication();

app.MapGet("/", () => "10.0.0.100");

app.Run();
```

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Install Nginx

Ubuntu

Use `apt-get` to install Nginx. The installer creates a systemd init script that runs Nginx as daemon on system startup. Follow the installation instructions for Ubuntu at Nginx: Official Debian/Ubuntu packages.

> **Note**
>
> If optional Nginx modules are required, building Nginx from source might be required.

Since Nginx was installed for the first time, explicitly start it by running:

Bash

```
sudo service nginx start
```

Verify a browser displays the default landing page for Nginx. The landing page is reachable at `http://<server_IP_address>/index.nginx-debian.html`.

Configure Nginx

Ubuntu

To configure Nginx as a reverse proxy to forward HTTP requests to the ASP.NET Core app, modify `/etc/nginx/sites-available/default` and recreate the symlink. After creating the `/etc/nginx/sites-available/default` file, use the following command to create the symlink:

Bash

```
sudo ln -s /etc/nginx/sites-available/default /etc/nginx/sites-
enabled/default
```

Open `/etc/nginx/sites-available/default` in a text editor, and replace the contents with the following snippet:

text

```
map $http_connection $connection_upgrade {
  "~*Upgrade" $http_connection;
  default keep-alive;
}

server {
  listen        80;
  server_name   example.com *.example.com;
  location / {
      proxy_pass         http://127.0.0.1:5000/;
      proxy_http_version 1.1;
      proxy_set_header   Upgrade $http_upgrade;
      proxy_set_header   Connection $connection_upgrade;
      proxy_set_header   Host $host;
      proxy_cache_bypass $http_upgrade;
      proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header   X-Forwarded-Proto $scheme;
  }
}
```

If the app is a SignalR or Blazor Server app, see ASP.NET Core SignalR production hosting and scaling and Host and deploy ASP.NET Core server-side Blazor apps respectively for more information.

When no `server_name` matches, Nginx uses the default server. If no default server is defined, the first server in the configuration file is the default server. As a best practice, add a specific default server that returns a status code of 444 in your configuration file. A default server configuration example is:

text

```
server {
    listen   80 default_server;
    # listen [::]:80 default_server deferred;
    return   444;
}
```

With the preceding configuration file and default server, Nginx accepts public traffic on port 80 with host header `example.com` or `*.example.com`. Requests not matching these hosts won't get forwarded to Kestrel. Nginx forwards the matching requests to Kestrel at `http://127.0.0.1:5000/`. For more information, see How nginx processes a request. To change Kestrel's IP/port, see Kestrel: Endpoint configuration.

> **⚠ Warning**
>
> Failure to specify a proper server_name directive exposes your app to security vulnerabilities. Subdomain wildcard binding (for example, `*.example.com`) doesn't pose this security risk if you control the entire parent domain (as opposed to `*.com`, which is vulnerable). For more information, see RFC 9110: HTTP Semantics (Section 7.2: Host and :authority).

Once the Nginx configuration is established, run `sudo nginx -t` to verify the syntax of the configuration files. If the configuration file test is successful, force Nginx to pick up the changes by running `sudo nginx -s reload`.

To directly run the app on the server:

1. Navigate to the app's directory.
2. Run the app: `dotnet <app_assembly.dll>`, where `app_assembly.dll` is the assembly file name of the app.

Ubuntu

If the app runs on the server but fails to respond over the internet, check the server's firewall and confirm port 80 is open. If using an Azure Ubuntu VM, add a Network Security Group (NSG) rule that enables inbound port 80 traffic. There's no need to enable an outbound port 80 rule, as the outbound traffic is automatically granted when the inbound rule is enabled.

When done testing the app, shut down the app with Ctrl+C (Windows) or ⌘+C (macOS) at the command prompt.

Increase keepalive_requests

keepalive_requests can be increased for higher performance, For more information, see this GitHub issue.

Monitor the app

The server is set up to forward requests made to `http://<serveraddress>:80` on to the ASP.NET Core app running on Kestrel at `http://127.0.0.1:5000`. However, Nginx isn't set up to manage the Kestrel process. systemd can be used to create a service file to start and monitor the underlying web app. `systemd` is an init system that provides many powerful features for starting, stopping, and managing processes.

Create the service file

Create the service definition file:

Bash

```
sudo nano /etc/systemd/system/kestrel-helloapp.service
```

The following example is an `.ini` service file for the app:

text

```
[Unit]
Description=Example .NET Web API App running on Linux

[Service]
WorkingDirectory=/var/www/helloapp
ExecStart=/usr/bin/dotnet /var/www/helloapp/helloapp.dll
Restart=always
# Restart service after 10 seconds if the dotnet service crashes:
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_NOLOGO=true

[Install]
WantedBy=multi-user.target
```

In the preceding example, the user that manages the service is specified by the `User` option. The user (`www-data`) must exist and have proper ownership of the app's files.

Use `TimeoutStopSec` to configure the duration of time to wait for the app to shut down after it receives the initial interrupt signal. If the app doesn't shut down in this period, SIGKILL is issued to terminate the app. Provide the value as unitless seconds (for example, `150`), a time span value (for example, `2min 30s`), or `infinity` to disable the timeout. `TimeoutStopSec` defaults to the value of `DefaultTimeoutStopSec` in the manager configuration file (`systemd-system.conf`, `system.conf.d`, `systemd-user.conf`, `user.conf.d`). The default timeout for most distributions is 90 seconds.

text

```
# The default value is 90 seconds for most distributions.
TimeoutStopSec=90
```

Linux has a case-sensitive file system. Setting `ASPNETCORE_ENVIRONMENT` to `Production` results in searching for the configuration file `appsettings.Production.json`, not `appsettings.production.json`.

Some values (for example, SQL connection strings) must be escaped for the configuration providers to read the environment variables. Use the following command to generate a properly escaped value for use in the configuration file:

Console

```
systemd-escape "<value-to-escape>"
```

Colon (`:`) separators aren't supported in environment variable names. Use a double underscore (`__`) in place of a colon. The Environment Variables configuration provider converts double-underscores into colons when environment variables are read into configuration. In the following example, the connection string key `ConnectionStrings:DefaultConnection` is set into the service definition file as `ConnectionStrings__DefaultConnection`:

Console

```
Environment=ConnectionStrings__DefaultConnection={Connection String}
```

Save the file and enable the service.

Bash

```
sudo systemctl enable kestrel-helloapp.service
```

Start the service and verify that it's running.

Bash

```
sudo systemctl start kestrel-helloapp.service
sudo systemctl status kestrel-helloapp.service

● kestrel-helloapp.service - Example .NET Web API App running on Linux
    Loaded: loaded (/etc/systemd/system/kestrel-helloapp.service; enabled)
    Active: active (running) since Thu 2016-10-18 04:09:35 NZDT; 35s ago
Main PID: 9021 (dotnet)
    CGroup: /system.slice/kestrel-helloapp.service
            └─9021 /usr/local/bin/dotnet /var/www/helloapp/helloapp.dll
```

With the reverse proxy configured and Kestrel managed through `systemd`, the web app is fully configured and can be accessed from a browser on the local machine at `http://localhost`. It's also accessible from a remote machine, barring any firewall that might be blocking. Inspecting the response headers, the `Server` header shows the ASP.NET Core app being served by Kestrel.

text

```
HTTP/1.1 200 OK
Date: Tue, 11 Oct 2016 16:22:23 GMT
Server: Kestrel
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Transfer-Encoding: chunked
```

View logs

Since the web app using Kestrel is managed using systemd, all events and processes are logged to a centralized journal. However, this journal includes all entries for all services and processes managed by `systemd`. To view the `kestrel-helloapp.service`-specific items, use the following command:

Bash

```
sudo journalctl -fu kestrel-helloapp.service
```

For further filtering, time options such as `--since today`, `--until 1 hour ago`, or a combination of these can reduce the number of entries returned.

Bash

```
sudo journalctl -fu kestrel-helloapp.service --since "2016-10-18" --until
"2016-10-18 04:00"
```

Data protection

The ASP.NET Core Data Protection stack is used by several ASP.NET Core middlewares, including authentication middleware (for example, cookie middleware) and cross-site request forgery (CSRF) protections. Even if Data Protection APIs aren't called by user code, data protection should be configured to create a persistent cryptographic key store. If data protection isn't configured, the keys are held in memory and discarded when the app restarts.

If the key ring is stored in memory when the app restarts:

- All cookie-based authentication tokens are invalidated.
- Users are required to sign in again on their next request.
- Any data protected with the key ring can no longer be decrypted. This may include CSRF tokens and ASP.NET Core MVC TempData cookies.

To configure data protection to persist and encrypt the key ring, see:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Long request header fields

Proxy server default settings typically limit request header fields to 4 K or 8 K depending on the platform. An app may require fields longer than the default (for example, apps that use Microsoft Entra ID). If longer fields are required, the proxy server's default settings require adjustment. The values to apply depend on the scenario. For more information, see your server's documentation.

- proxy_buffer_size
- proxy_buffers
- proxy_busy_buffers_size
- large_client_header_buffers

> ⚠ **Warning**
>
> Don't increase the default values of proxy buffers unless necessary. Increasing these values increases the risk of buffer overrun (overflow) and Denial of Service (DoS)

attacks by malicious users.

Secure the app

Enable AppArmor

Linux Security Modules (LSM) is a framework that's part of the Linux kernel since Linux 2.6. LSM supports different implementations of security modules. AppArmor is an LSM that implements a Mandatory Access Control system, which allows confining the program to a limited set of resources. Ensure AppArmor is enabled and properly configured.

Configure the firewall

Close off all external ports that aren't in use. Uncomplicated firewall (ufw) provides a front end for `iptables` by providing a CLI for configuring the firewall.

Ubuntu

> ⚠ **Warning**
>
> A firewall prevents access to the whole system if not configured correctly. Failure to specify the correct SSH port effectively locks you out of the system if you are using SSH to connect to it. The default port is 22. For more information, see the introduction to ufw and the manual.

Install `ufw` and configure it to allow traffic on any ports needed.

Bash

```
sudo apt-get install ufw

sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp

sudo ufw enable
```

Secure Nginx

Change the Nginx response name

Edit `src/http/ngx_http_header_filter_module.c`:

```
C

static char ngx_http_server_string[] = "Server: Web Server" CRLF;
static char ngx_http_server_full_string[] = "Server: Web Server" CRLF;
```

Configure options

Configure the server with additional required modules. Consider using a web app firewall, such as ModSecurity, to harden the app.

HTTPS configuration

Configure the app for secure (HTTPS) local connections

The dotnet run command uses the app's `Properties/launchSettings.json` file, which configures the app to listen on the URLs provided by the `applicationUrl` property. For example, `https://localhost:5001;http://localhost:5000`.

Configure the app to use a certificate in development for the `dotnet run` command or development environment (`F5` or `Ctrl`+`F5` in Visual Studio Code) using one of the following approaches:

- Replace the default certificate from configuration (*Recommended*)
- KestrelServerOptions.ConfigureHttpsDefaults

Configure the reverse proxy for secure (HTTPS) client connections

> **⚠ Warning**
>
> The security configuration in this section is a general configuration to be used as a starting point for further customization. We're unable to provide support for third-party tooling, servers, and operating systems. *Use the configuration in this section at your own risk.* For more information, access the following resources:
>
> - Configuring HTTPS servers (Nginx documentation)
> - mozilla.org SSL Configuration Generator

- Configure the server to listen to HTTPS traffic on port 443 by specifying a valid certificate issued by a trusted Certificate Authority (CA).
- Harden the security by employing some of the practices depicted in the following */etc/nginx/nginx.conf* file.
- The following example doesn't configure the server to redirect insecure requests. We recommend using HTTPS Redirection Middleware. For more information, see Enforce HTTPS in ASP.NET Core.

  > **Note**
  >
  > For development environments where the server configuration handles secure redirection instead of HTTPS Redirection Middleware, we recommend using temporary redirects (302) rather than permanent redirects (301). Link caching can cause unstable behavior in development environments.

- Adding a `Strict-Transport-Security` (HSTS) header ensures all subsequent requests made by the client are over HTTPS. For guidance on setting the `Strict-Transport-Security` header, see Enforce HTTPS in ASP.NET Core.
- If HTTPS will be disabled in the future, use one of the following approaches:
  - Don't add the HSTS header.
  - Choose a short `max-age` value.

Add the */etc/nginx/proxy.conf* configuration file:

nginx

```
proxy_redirect          off;
proxy_set_header        Host $host;
proxy_set_header        X-Real-IP $remote_addr;
proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header        X-Forwarded-Proto $scheme;
client_max_body_size    10m;
client_body_buffer_size 128k;
proxy_connect_timeout   90;
proxy_send_timeout      90;
proxy_read_timeout      90;
proxy_buffers           32 4k;
```

Ubuntu

Replace the contents of the */etc/nginx/nginx.conf* configuration file with the following file. The example contains both `http` and `server` sections in one configuration file.

nginx

```
http {
    include        /etc/nginx/proxy.conf;
    limit_req_zone $binary_remote_addr zone=one:10m rate=5r/s;
    server_tokens  off;

    sendfile on;
    # Adjust keepalive_timeout to the lowest possible value that makes sense
    # for your use case.
    keepalive_timeout   29;
    client_body_timeout 10; client_header_timeout 10; send_timeout 10;

    upstream helloapp{
        server 127.0.0.1:5000;
    }

    server {
        listen                  443 ssl http2;
        listen                  [::]:443 ssl http2;
        server_name             example.com *.example.com;
        ssl_certificate         /etc/ssl/certs/testCert.crt;
        ssl_certificate_key     /etc/ssl/certs/testCert.key;
        ssl_session_timeout     1d;
        ssl_protocols           TLSv1.2 TLSv1.3;
        ssl_prefer_server_ciphers off;
        ssl_ciphers             ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-
SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-
AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384;
        ssl_session_cache       shared:SSL:10m;
        ssl_session_tickets     off;
        ssl_stapling            off;

        add_header X-Frame-Options DENY;
        add_header X-Content-Type-Options nosniff;

        #Redirects all traffic
        location / {
            proxy_pass http://helloapp;
            limit_req  zone=one burst=10 nodelay;
        }
    }
}
```

ⓘ Note

Blazor WebAssembly apps require a larger `burst` parameter value to accommodate the larger number of requests made by an app. For more information, see Host and deploy ASP.NET Core Blazor WebAssembly.

> **Note**
>
> The preceding example disables Online Certificate Status Protocol (OCSP) Stapling. If enabled, confirm that the certificate supports the feature. For more information and guidance on enabling OCSP, see the following properties in the Module ngx_http_ssl_module (Nginx documentation) article:
>
> - `ssl_stapling`
> - `ssl_stapling_file`
> - `ssl_stapling_responder`
> - `ssl_stapling_verify`

Secure Nginx from clickjacking

Clickjacking, also known as a *UI redress attack*, is a malicious attack where a website visitor is tricked into clicking a link or button on a different page than they're currently visiting. Use `X-FRAME-OPTIONS` to secure the site.

To mitigate clickjacking attacks:

1. Edit the *nginx.conf* file:

   ```
   Bash

   sudo nano /etc/nginx/nginx.conf
   ```

   Within the `http{}` code block, add the line: `add_header X-Frame-Options "SAMEORIGIN";`

2. Save the file.

3. Restart Nginx.

MIME-type sniffing

This header prevents most browsers from MIME-sniffing a response away from the declared content type, as the header instructs the browser not to override the response

content type. With the `nosniff` option, if the server says the content is `text/html`, the browser renders it as `text/html`.

1. Edit the *nginx.conf* file:

   ```
   Bash

   sudo nano /etc/nginx/nginx.conf
   ```

   Within the `http{}` code block, add the line: `add_header X-Content-Type-Options "nosniff";`

2. Save the file.

3. Restart Nginx.

Additional Nginx suggestions

After upgrading the shared framework on the server, restart the ASP.NET Core apps hosted by the server.

Additional resources

- Prerequisites for .NET Core on Linux
- Nginx: Binary Releases: Official Debian/Ubuntu packages
- Troubleshoot and debug ASP.NET Core projects
- Configure ASP.NET Core to work with proxy servers and load balancers
- NGINX: Using the Forwarded header

Host ASP.NET Core in Docker containers

Article • 07/31/2024

The following articles are available for learning about hosting ASP.NET Core apps in Docker:

Introduction to Containers and Docker

See how containerization is an approach to software development in which an application or service, its dependencies, and its configuration are packaged together as a container image. The image can be tested and then deployed to a host.

What is Docker

Discover how Docker is an open-source project for automating the deployment of apps as portable, self-sufficient containers that can run on the cloud or on-premises.

Docker Terminology

Learn terms and definitions for Docker technology.

Docker containers, images, and registries

Find out how Docker container images are stored in an image registry for consistent deployment across environments.

Run an ASP.NET Core app in Docker containers Learn how to build and dockerize an ASP.NET Core app. Explore Docker images maintained by Microsoft and examine use cases.

.NET Docker samples Samples and guidance that demonstrate how to use .NET and Docker for development, testing and production.

Visual Studio Container Tools

Discover how Visual Studio supports building, debugging, and running ASP.NET Core apps targeting either .NET Framework or .NET Core on Docker for Windows. Both Windows and Linux containers are supported.

Publish to Azure Container Registry

Find out how to use the Visual Studio Container Tools extension to deploy an ASP.NET Core app to a Docker host on Azure using PowerShell.

Configure ASP.NET Core to work with proxy servers and load balancers

Additional configuration might be required for apps hosted behind proxy servers and load balancers. Passing requests through a proxy often obscures information about the original request, such as the scheme and client IP. It might be necessary to forward some information about the request manually to the app.

GC using Docker and small containers Discusses GC selection with small containers.

System.IO.IOException: The configured user limit (128) on the number of inotify instances has been reached

Disabling `reloadOnChange` can significantly reduce the number of opened files. To disable reloading configuration files, set the environment variable `DOTNET_HOSTBUILDER__RELOADCONFIGONCHANGE=false`

For alternative approaches or to leave feedback on this problem, see this GitHub issue.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Run an ASP.NET Core app in Docker containers

Article • 05/15/2024

This article shows how to run an ASP.NET Core app in Docker containers.

Windows Home Edition doesn't support Hyper-V, and Hyper-V is needed for Docker.

See Containerize a .NET app with dotnet publish for information on containerized a .NET app with `dotnet publish`.

ASP.NET Core Docker images

For this tutorial, you download an ASP.NET Core sample app and run it in Docker containers. The sample works with both Linux and Windows containers.

The sample Dockerfile uses the Docker multi-stage build feature to build and run in different containers. The build and run containers are created from images that are provided in Docker Hub by Microsoft:

- `dotnet/sdk`

  The sample uses this image for building the app. The image contains the .NET SDK, which includes the Command Line Tools (CLI). The image is optimized for local development, debugging, and unit testing. The tools installed for development and compilation make the image relatively large.

- `dotnet/aspnet`

  The sample uses this image for running the app. The image contains the ASP.NET Core runtime and libraries and is optimized for running apps in production. Designed for speed of deployment and app startup, the image is relatively small, so network performance from Docker Registry to Docker host is optimized. Only the binaries and content needed to run an app are copied to the container. The contents are ready to run, enabling the fastest time from `docker run` to app startup. Dynamic code compilation isn't needed in the Docker model.

Prerequisites

- .NET SDK 8.0

- Docker client 18.03 or later
  - Linux distributions
    - Debian
    - Fedora
    - Ubuntu
  - macOS
  - Windows
- Git

Download the sample app

- Download the sample by cloning the .NET Docker repository:

```
Console

git clone https://github.com/dotnet/dotnet-docker
```

Run the app locally

- Navigate to the project folder at *dotnet-docker/samples/aspnetapp/aspnetapp*.
- Run the following command to build and run the app locally:

```
.NET CLI

dotnet run
```

- Go to `http://localhost:<port>` in a browser to test the app.
- Press Ctrl+C at the command prompt to stop the app.

Run in a Linux container or Windows container

- To run in a Linux container, right-click the System Tray's Docker client icon and select switch to Linux containers.
- To run in a Windows container, right-click the System Tray's Docker client icon and select switch to Windows containers.
- Navigate to the Dockerfile folder at *dotnet-docker/samples/aspnetapp*.

- Run the following commands to build and run the sample in Docker:

  Console

  ```
  docker build -t aspnetapp .
  docker run -it --rm -p <port>:8080 --name aspnetcore_sample aspnetapp
  ```

  The `build` command arguments:
  - Name the image aspnetapp.
  - Look for the Dockerfile in the current folder (the period at the end).

  The run command arguments:
  - Allocate a pseudo-TTY and keep it open even if not attached. (Same effect as `--interactive --tty`.)
  - Automatically remove the container when it exits.
  - Map `<port>` on the local machine to port 8080 in the container.
  - Name the container aspnetcore_sample.
  - Specify the aspnetapp image.

- Go to `http://localhost:<port>` in a browser to test the app.

Build and deploy manually

In some scenarios, you might want to deploy an app to a container by copying its assets that are needed at run time. This section shows how to deploy manually.

- Navigate to the project folder at *dotnet-docker/samples/aspnetapp/aspnetapp*.

- Run the dotnet publish command:

  .NET CLI

  ```
  dotnet publish -c Release -o published
  ```

  The command arguments:
  - Build the app in release mode (the default is debug mode).
  - Create the assets in the *published* folder.

- Run the app.

  - Windows:

    .NET CLI

```
dotnet published\aspnetapp.dll
```

- Linux:

```
.NET CLI

dotnet published/aspnetapp.dll
```

- Browse to `http://localhost:<port>` to see the home page.

To use the manually published app within a Docker container, create a new *Dockerfile* and use the `docker build .` command to build an image.

```
Dockerfile

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS runtime
WORKDIR /app
COPY published/ ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

To see the new image use the `docker images` command.

The Dockerfile

Here's the *Dockerfile* used by the `docker build` command you ran earlier. It uses `dotnet publish` the same way you did in this section to build and deploy.

```
Dockerfile

# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /source

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore

# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /source/aspnetapp
RUN dotnet publish -c release -o /app --no-restore

# final stage/image
FROM mcr.microsoft.com/dotnet/aspnet:8.0
WORKDIR /app
```

```
COPY --from=build /app ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

In the preceding *Dockerfile*, the `*.csproj` files are copied and restored as distinct *layers*. When the `docker build` command builds an image, it uses a built-in cache. If the `*.csproj` files haven't changed since the `docker build` command last ran, the `dotnet restore` command doesn't need to run again. Instead, the built-in cache for the corresponding `dotnet restore` layer is reused. For more information, see Best practices for writing Dockerfiles.

Additional resources

- Containerize a .NET app with dotnet publish
- Docker build command
- Docker run command
- ASP.NET Core Docker sample (The one used in this tutorial.)
- Configure ASP.NET Core to work with proxy servers and load balancers
- Working with Visual Studio Docker Tools
- Debugging with Visual Studio Code
- GC using Docker and small containers
- System.IO.IOException: The configured user limit (128) on the number of inotify instances has been reached
- Updates to Docker images

Next steps

The Git repository that contains the sample app also includes documentation. For an overview of the resources available in the repository, see the README file. In particular, learn how to implement HTTPS:

Developing ASP.NET Core Applications with Docker over HTTPS

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Visual Studio Container Tools with ASP.NET Core

Article • 07/31/2024

Visual Studio 2017 and later versions support building, debugging, and running containerized ASP.NET Core apps targeting .NET Core. Both Windows and Linux containers are supported.

View or download sample code (how to download)

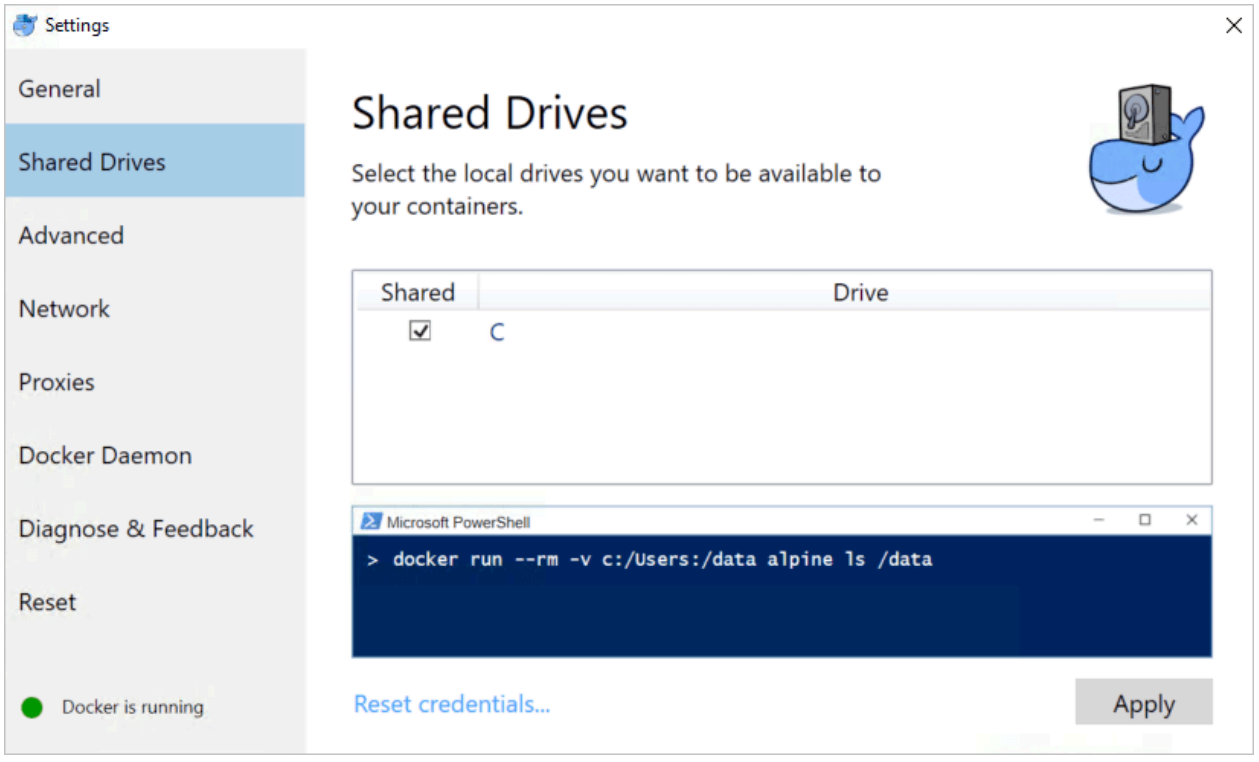
Prerequisites

- Docker for Windows
- Visual Studio 2019 with the **.NET Core cross-platform development** workload

Installation and setup

For Docker installation, first review the information at Docker for Windows: What to know before you install. Next, install Docker For Windows.

Shared Drives in Docker for Windows must be configured to support volume mapping and debugging. Right-click the System Tray's Docker icon, select **Settings**, and select **Shared Drives**. Select the drive where Docker stores files. Click **Apply**.



> **Tip**
>
> Visual Studio 2017 versions 15.6 and later prompt when **Shared Drives** aren't configured.

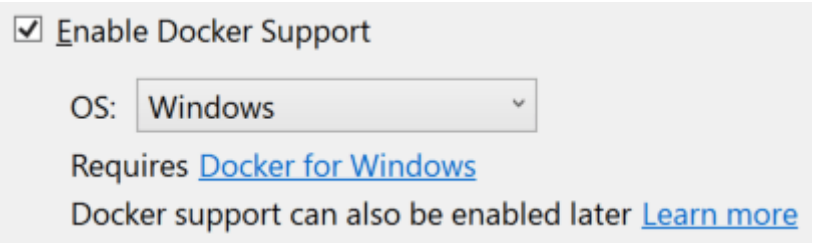
Add a project to a Docker container

To containerize an ASP.NET Core project, the project must target .NET Core. Both Linux and Windows containers are supported.

When adding Docker support to a project, choose either a Windows or a Linux container. The Docker host must be running the same container type. To change the container type in the running Docker instance, right-click the System Tray's Docker icon and choose **Switch to Windows containers...** or **Switch to Linux containers...**.

New app

When creating a new app with the **ASP.NET Core Web Application** project templates, select the **Enable Docker Support** checkbox:



If the target framework is .NET Core, the **OS** drop-down allows for the selection of a container type.

Existing app

For ASP.NET Core projects targeting .NET Core, there are two options for adding Docker support via the tooling. Open the project in Visual Studio, and choose one of the following options:

- Select **Docker Support** from the **Project** menu.
- Right-click the project in **Solution Explorer** and select **Add** > **Docker Support**.

The Visual Studio Container Tools don't support adding Docker to an existing ASP.NET Core project targeting .NET Framework.

Dockerfile overview

A *Dockerfile*, the recipe for creating a final Docker image, is added to the project root. Refer to Dockerfile reference for an understanding of the commands within it. This particular *Dockerfile* uses a multi-stage build with four distinct, named build stages:

Dockerfile

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.1 AS base
WORKDIR /app
EXPOSE 59518
EXPOSE 44364

FROM mcr.microsoft.com/dotnet/core/sdk:2.1 AS build
WORKDIR /src
COPY HelloDockerTools/HelloDockerTools.csproj HelloDockerTools/
RUN dotnet restore HelloDockerTools/HelloDockerTools.csproj
COPY . .
WORKDIR /src/HelloDockerTools
RUN dotnet build HelloDockerTools.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish HelloDockerTools.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
```

The preceding *Dockerfile* image includes the ASP.NET Core runtime and NuGet packages. The packages are just-in-time (JIT) compiled to improve startup performance.

When the new project dialog's **Configure for HTTPS** checkbox is checked, the *Dockerfile* exposes two ports. One port is used for HTTP traffic; the other port is used for HTTPS. If the checkbox isn't checked, a single port (80) is exposed for HTTP traffic.

Add container orchestrator support to an app

Visual Studio 2017 versions 15.7 or earlier support Docker Compose as the sole container orchestration solution. The Docker Compose artifacts are added via **Add** > **Docker Support**.

Visual Studio 2017 versions 15.8 or later add an orchestration solution only when instructed. Right-click the project in **Solution Explorer** and select **Add** > **Container Orchestrator Support**. The following choices are available:

- Docker Compose

- Service Fabric
- Kubernetes/Helm

Docker Compose

The Visual Studio Container Tools add a *docker-compose* project to the solution with the following files:

- *docker-compose.dcproj*: The file representing the project. Includes a `<DockerTargetOS>` element specifying the OS to be used.
- *.dockerignore*: Lists the file and directory patterns to exclude when generating a build context.
- *docker-compose.yml*: The base Docker Compose file used to define the collection of images built and run with `docker-compose build` and `docker-compose run`, respectively.
- *docker-compose.override.yml*: An optional file, read by Docker Compose, with configuration overrides for services. Visual Studio executes `docker-compose -f "docker-compose.yml" -f "docker-compose.override.yml"` to merge these files.

The *docker-compose.yml* file references the name of the image that's created when the project runs:

YAML

```yaml
version: '3.4'

services:
  hellodockertools:
    image: ${DOCKER_REGISTRY}hellodockertools
    build:
      context: .
      dockerfile: HelloDockerTools/Dockerfile
```

In the preceding example, `image: hellodockertools` generates the image `hellodockertools:dev` when the app runs in **Debug** mode. The `hellodockertools:latest` image is generated when the app runs in **Release** mode.

Prefix the image name with the Docker Hub username (for example, `dockerhubusername/hellodockertools`) if the image is pushed to the registry. Alternatively, change the image name to include the private registry URL (for example, `privateregistry.domain.com/hellodockertools`) depending on the configuration.

If you want different behavior based on the build configuration (for example, Debug or Release), add configuration-specific *docker-compose* files. The files should be named

according to the build configuration (for example, *docker-compose.vs.debug.yml* and *docker-compose.vs.release.yml*) and placed in the same location as the *docker-compose-override.yml* file.

Using the configuration-specific override files, you can specify different configuration settings (such as environment variables or entry points) for Debug and Release build configurations.

For Docker Compose to display an option to run in Visual Studio, the docker project must be the startup project.

Service Fabric

In addition to the base Prerequisites, the Service Fabric orchestration solution demands the following prerequisites:

- Microsoft Azure Service Fabric SDK version 2.6 or later
- Visual Studio's **Azure Development** workload

Service Fabric doesn't support running Linux containers in the local development cluster on Windows. If the project is already using a Linux container, Visual Studio prompts to switch to Windows containers.

The Visual Studio Container Tools do the following tasks:

- Adds a <*project_name*>*Application* **Service Fabric Application** project to the solution.
- Adds a *Dockerfile* and a *.dockerignore* file to the ASP.NET Core project. If a *Dockerfile* already exists in the ASP.NET Core project, it's renamed to *Dockerfile.original*. A new *Dockerfile*, similar to the following, is created:

  Dockerfile

  ```
  # See https://aka.ms/containerimagehelp for information on how to use
  Windows Server 1709 containers with Service Fabric.
  # FROM microsoft/aspnetcore:2.0-nanoserver-1709
  FROM microsoft/aspnetcore:2.0-nanoserver-sac2016
  ARG source
  WORKDIR /app
  COPY ${source:-obj/Docker/publish} .
  ENTRYPOINT ["dotnet", "HelloDockerTools.dll"]
  ```

- Adds an `<IsServiceFabricServiceProject>` element to the ASP.NET Core project's `.csproj` file:

XML

```
<IsServiceFabricServiceProject>True</IsServiceFabricServiceProject>
```

- Adds a *PackageRoot* folder to the ASP.NET Core project. The folder includes the service manifest and settings for the new service.

For more information, see Deploy a .NET app in a Windows container to Azure Service Fabric.

Debug

Select **Docker** from the debug drop-down in the toolbar, and start debugging the app. The **Docker** view of the **Output** window shows the following actions taking place:

- The *2.1-aspnetcore-runtime* tag of the *microsoft/dotnet* runtime image is acquired (if not already in the cache). The image installs the ASP.NET Core and .NET Core runtimes and associated libraries. It's optimized for running ASP.NET Core apps in production.
- The `ASPNETCORE_ENVIRONMENT` environment variable is set to `Development` within the container.
- Two dynamically assigned ports are exposed: one for HTTP and one for HTTPS. The port assigned to localhost can be queried with the `docker ps` command.
- The app is copied to the container.
- The default browser is launched with the debugger attached to the container using the dynamically assigned port.

The resulting Docker image of the app is tagged as *dev*. The image is based on the *2.1-aspnetcore-runtime* tag of the *microsoft/dotnet* base image. Run the `docker images` command in the **Package Manager Console** (PMC) window. The images on the machine are displayed:

Console

```
REPOSITORY          TAG                        IMAGE ID       CREATED          SIZE
hellodockertools    dev                        d72ce0f1dfe7   30 seconds ago
255MB
microsoft/dotnet    2.1-aspnetcore-runtime     fcc3887985bb   6 days ago
255MB
```

> **Note**

The *dev* image lacks the app contents, as **Debug** configurations use volume mounting to provide the iterative experience. To push an image, use the **Release** configuration.

Run the `docker ps` command in PMC. Notice the app is running using the container:

Console

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS              PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   21
seconds ago      Up 19 seconds       0.0.0.0:37630->80/tcp
dockercompose4642749010770307127_hellodockertools_1
```

Edit and continue

Changes to static files and Razor views are automatically updated without the need for a compilation step. Make the change, save, and refresh the browser to view the update.

Code file modifications require compilation and a restart of Kestrel within the container. After making the change, use `CTRL+F5` to perform the process and start the app within the container. The Docker container isn't rebuilt or stopped. Run the `docker ps` command in PMC. Notice the original container is still running as of 10 minutes ago:

Console

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED
STATUS              PORTS                   NAMES
baf9a678c88d        hellodockertools:dev   "C:\\remote_debugge..."   10
minutes ago      Up 10 minutes       0.0.0.0:37630->80/tcp
dockercompose4642749010770307127_hellodockertools_1
```

Publish Docker images

Once the develop and debug cycle of the app is completed, the Visual Studio Container Tools assist in creating the production image of the app. Change the configuration drop-down to **Release** and build the app. The tooling acquires the compile/publish image from Docker Hub (if not already in the cache). An image is produced with the *latest* tag, which can be pushed to the private registry or Docker Hub.

Run the `docker images` command in PMC to see the list of images. Output similar to the following is displayed:

Console

```
REPOSITORY          TAG                        IMAGE ID      CREATED
SIZE
hellodockertools    latest                     e3984a64230c  About a minute ago
258MB
hellodockertools    dev                        d72ce0f1dfe7  4 minutes ago
255MB
microsoft/dotnet    2.1-sdk                    9e243db15f91  6 days ago
1.7GB
microsoft/dotnet    2.1-aspnetcore-runtime     fcc3887985bb  6 days ago
255MB
```

> **Note**
>
> The `docker images` command returns intermediary images with repository names and tags identified as *<none>* (not listed above). These unnamed images are produced by the **multi-stage build** *Dockerfile*. They improve the efficiency of building the final image—only the necessary layers are rebuilt when changes occur. When the intermediary images are no longer needed, delete them using the **docker rmi** command.

There may be an expectation for the production or release image to be smaller in size by comparison to the *dev* image. Because of the volume mapping, the debugger and app were running from the local machine and not within the container. The *latest* image has packaged the necessary app code to run the app on a host machine. Therefore, the delta is the size of the app code.

Additional resources

- Container development with Visual Studio
- Azure Service Fabric: Prepare your development environment
- Deploy a .NET app in a Windows container to Azure Service Fabric
- Troubleshoot Visual Studio development with Docker
- Visual Studio Container Tools GitHub repository
- GC using Docker and small containers
- System.IO.IOException: The configured user limit (128) on the number of inotify instances has been reached
- Updates to Docker images

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Deploy an ASP.NET container to a container registry using Visual Studio

Article • 08/22/2024

Docker is a lightweight container engine, similar in some ways to a virtual machine, which you can use to host applications and services. This tutorial walks you through using Visual Studio to publish your containerized application to an Azure Container Registry.

If you don't have an Azure subscription, create a free account before you begin.

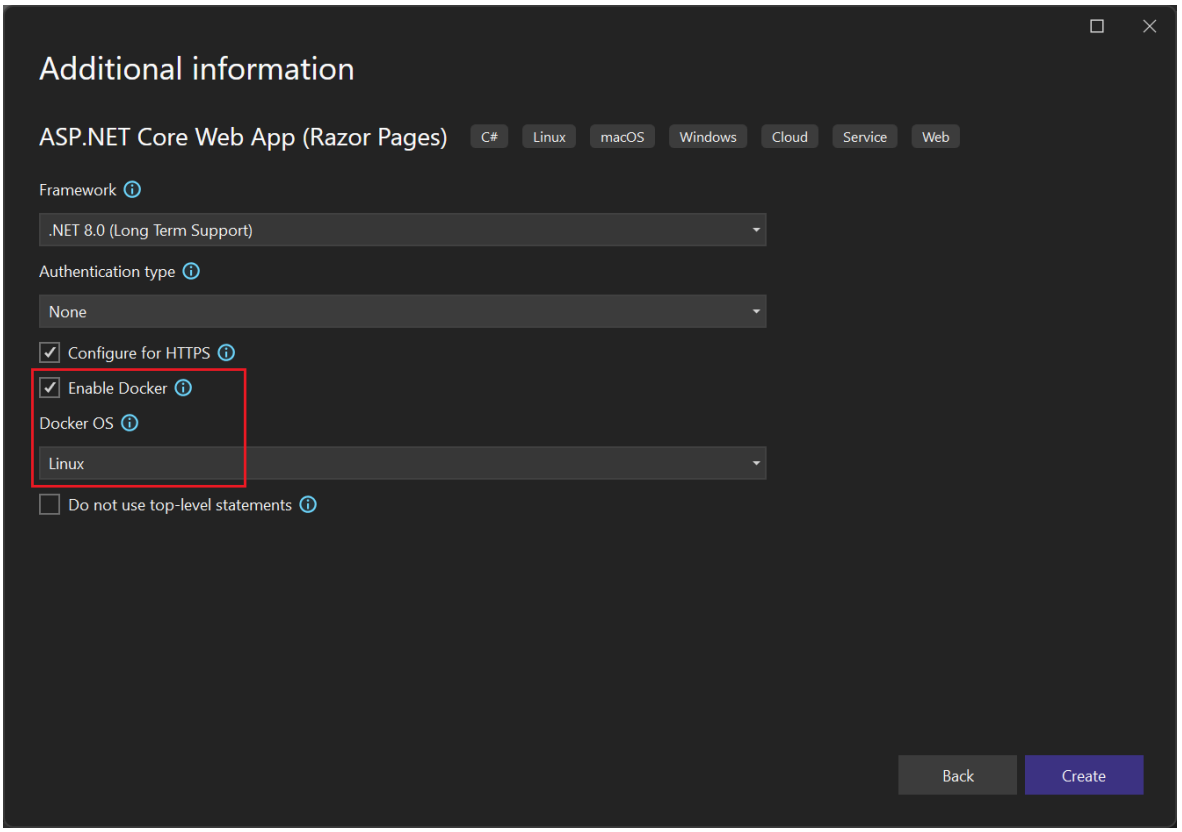
Prerequisites

- Install the latest version of Visual Studio 2022 with the "ASP.NET and web development" workload.
- Install Docker Desktop for Windows.

Create an ASP.NET Core web app

The following steps guide you through creating a basic ASP.NET Core app that you use in this tutorial. If you already have a project, you can skip this section.

1. In the Visual Studio start window, select **Create a new project**.
2. Select **ASP.NET Core Web App**, and then select **Next**.
3. Enter a name for your new application (or use the default name), specify the location on disk, and then select **Next**.
4. Select the .NET version you want to target. If you're not sure, choose the LTS (long-term support) release.



5. Choose whether you want SSL support by selecting or clearing the **Configure for HTTPS** checkbox.

6. Select the **Enable Docker** checkbox.

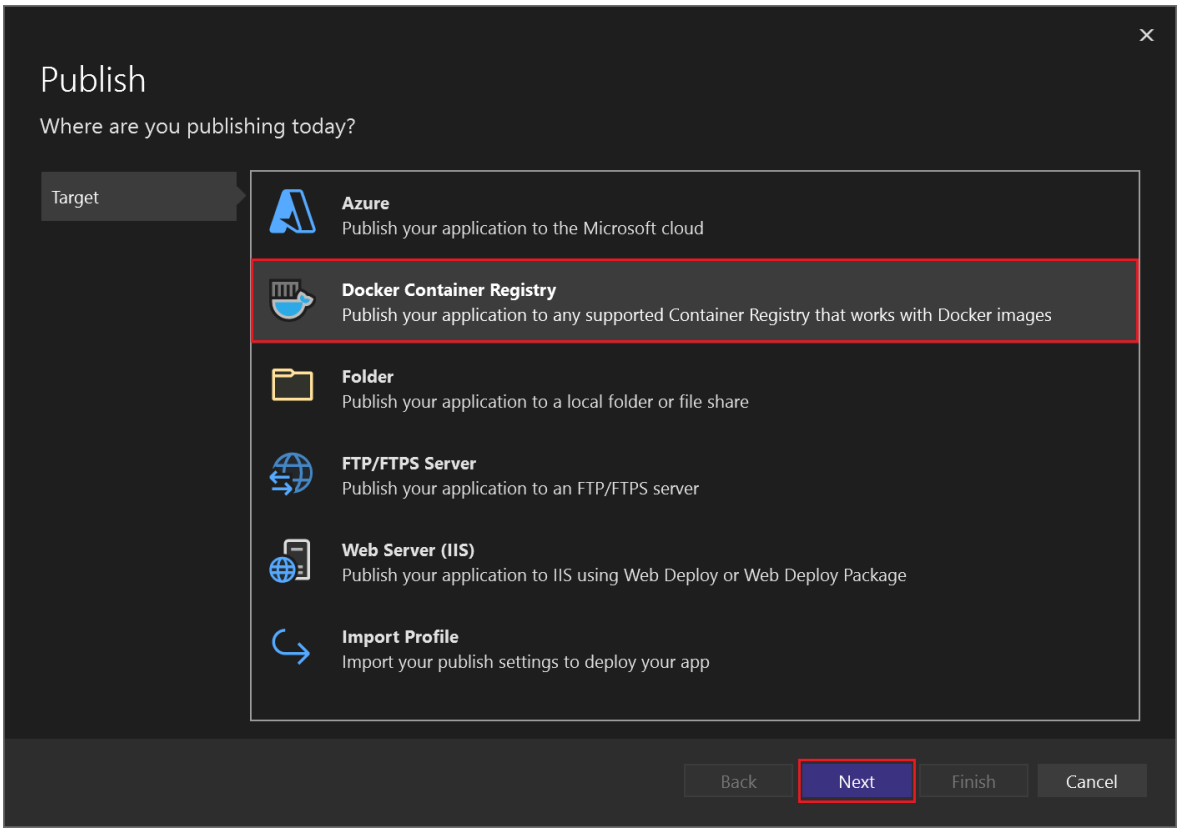
7. Use the **Docker OS** dropdown list to select the type of container you want: **Windows** or **Linux**.

8. Select **Create** to complete the process.

Publish your container to Azure Container Registry

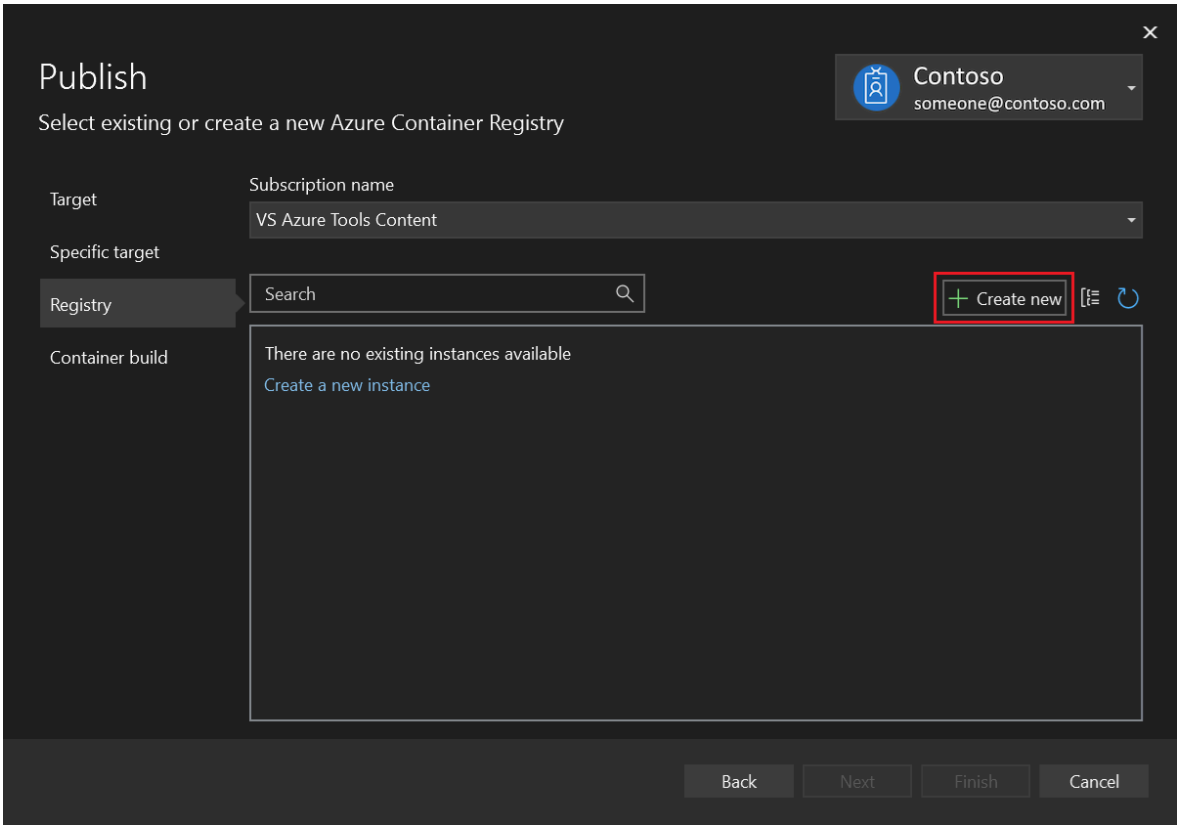
1. Right-click your project in **Solution Explorer** and choose **Publish**. The **Publish** dialog opens.

2. On the **Target** tab, select **Docker Container Registry**, and then select **Next**.



3. On the **Specific target** tab, select **Azure Container Registry**, and then select **Next**.

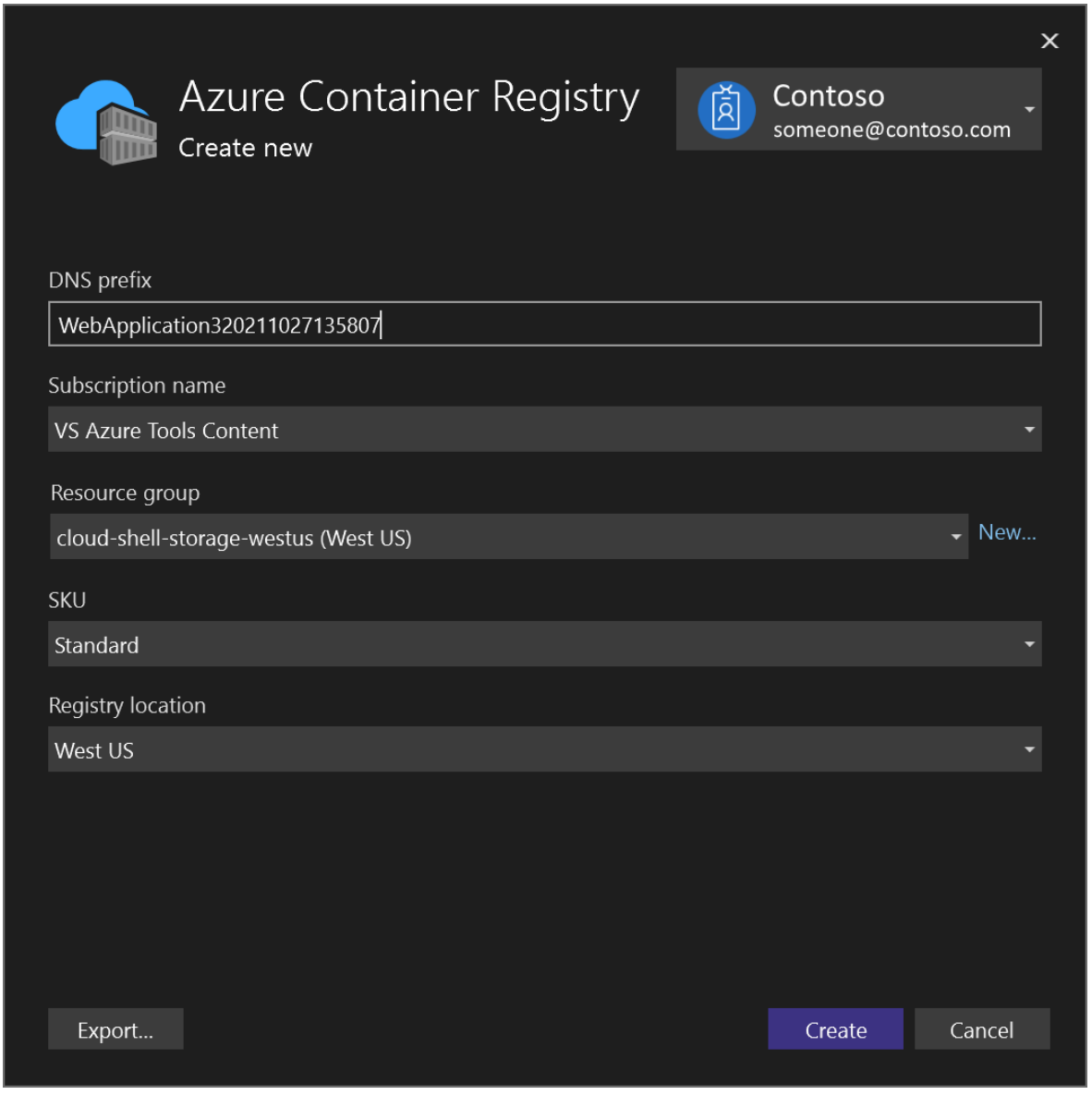
4. On the **Registry** tab, select the **Create new** (+) option at the right:



5. Fill in your desired values in the **Azure Container Registry** screen.

Expand table

Setting	Suggested value	Description
DNS Prefix	Globally unique name	Name that uniquely identifies your container registry.
Subscription	Your subscription	The Azure subscription to use.
Resource Group	Your resource group	Name of the resource group in which to create your container registry. Select **New** to create a new resource group.
SKU	"Standard"	Select the service tier of the container registry.
Registry Location	A nearby location	Choose a location in a region close to you or close to other services that you expect to use the container registry.



6. After you enter the resource values, select **Create**.

Visual Studio validates the property values and creates the new container resource. When the process completes, Visual Studio returns to the **Publish** dialog and selects the new container in the list.

7. Select **Finish** to publish the new container.

You can now pull the container from the registry to any host capable of running Docker images, such as Azure Container Instances.

Related content

- Quickstart: Deploy a container instance in Azure using the Azure CLI

Feedback

Was this page helpful? Yes No

Configure ASP.NET Core to work with proxy servers and load balancers

Article • 07/20/2023

By Chris Ross

In the recommended configuration for ASP.NET Core, the app is hosted using ASP.NET Core Module (ANCM) for IIS, Nginx, or Apache. Proxy servers, load balancers, and other network appliances often obscure information about the request before it reaches the app:

- When HTTPS requests are proxied over HTTP, the original scheme (HTTPS) is lost and must be forwarded in a header.
- Because an app receives a request from the proxy and not its true source on the Internet or corporate network, the originating client IP address must also be forwarded in a header.

This information may be important in request processing, for example in redirects, authentication, link generation, policy evaluation, and client geolocation.

Apps intended to run on web farm should read Host ASP.NET Core in a web farm.

Forwarded headers

By convention, proxies forward information in HTTP headers.

Expand table

Header	Description
X-Forwarded-For (XFF)	Holds information about the client that initiated the request and subsequent proxies in a chain of proxies. This parameter may contain IP addresses and, optionally, port numbers. In a chain of proxy servers, the first parameter indicates the client where the request was first made. Subsequent proxy identifiers follow. The last proxy in the chain isn't in the list of parameters. The last proxy's IP address, and optionally a port number, are available as the remote IP address at the transport layer.
X-Forwarded-Proto (XFP)	The value of the originating scheme, HTTP or HTTPS. The value may also be a list of schemes if the request has traversed multiple proxies.
X-Forwarded-Host (XFH)	The original value of the Host header field. Usually, proxies don't modify the Host header. See Microsoft Security Advisory CVE-2018-0787 for information on an

Header	Description
	elevation-of-privileges vulnerability that affects systems where the proxy doesn't validate or restrict Host headers to known good values.
`X-Forwarded-Prefix`	The original base path requested by the client. This header can be useful for applications to correctly generate URLs, redirects, or links back to the client.

The Forwarded Headers Middleware, ForwardedHeadersMiddleware, reads these headers and fills in the associated fields on HttpContext.

The middleware updates:

- HttpContext.Connection.RemoteIpAddress: Set using the `X-Forwarded-For` header value. Additional settings influence how the middleware sets `RemoteIpAddress`. For details, see the Forwarded Headers Middleware options. The consumed values are removed from `X-Forwarded-For`, and the old value of HttpContext.Connection.RemoteIpAddress is persisted in `X-Original-For`. Note: This process may be repeated several times if there are multiple values in `X-Forwarded-For/Proto/Host/Prefix`, resulting in several values moved to `X-Original-*`, including the original `RemoteIpAddress/Host/Scheme/PathBase`.
- HttpContext.Request.Scheme: Set using the X-Forwarded-Proto header value. The consumed value is removed from `X-Forwarded-Proto`, and the old value of HttpContext.Request.Scheme is persisted in `X-Original-Proto`.
- HttpContext.Request.Host: Set using the `X-Forwarded-Host` header value. The consumed value is removed from `X-Forwarded-Host`, and the old value of HttpContext.Request.Host is persisted in `X-Original-Host`.
- HttpContext.Request.PathBase: Set using the `X-Forwarded-Prefix` header value. The consumed value is removed from `X-Forwarded-Prefix`, and the old value of HttpContext.Request.PathBase is persisted in `X-Original-Prefix`.

For more information on the preceding, see this GitHub issue.

Forwarded Headers Middleware default settings can be configured. For the default settings:

- There is only ***one proxy*** between the app and the source of the requests.
- Only loopback addresses are configured for known proxies and known networks.
- The forwarded headers are named X-Forwarded-For, X-Forwarded-Proto, X-Forwarded-Host and `X-Forwarded-Prefix`.
- The `ForwardedHeaders` value is `ForwardedHeaders.None`, the desired forwarders must be set here to enable the middleware.

Not all network appliances add the `X-Forwarded-For` and `X-Forwarded-Proto` headers without additional configuration. Consult your appliance manufacturer's guidance if proxied requests don't contain these headers when they reach the app. If the appliance uses different header names than `X-Forwarded-For` and `X-Forwarded-Proto`, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the appliance. For more information, see Forwarded Headers Middleware options and Configuration for a proxy that uses different header names.

IIS/IIS Express and ASP.NET Core Module

Forwarded Headers Middleware is enabled by default by IIS Integration Middleware when the app is hosted out-of-process behind IIS and the ASP.NET Core Module (ANCM) for IIS. Forwarded Headers Middleware is activated to run first in the middleware pipeline with a restricted configuration specific to the ASP.NET Core Module. The restricted configuration is due to trust concerns with forwarded headers, for example, IP spoofing. The middleware is configured to forward the X-Forwarded-For and X-Forwarded-Proto headers and is restricted to a single localhost proxy. If additional configuration is required, see the Forwarded Headers Middleware options.

Other proxy server and load balancer scenarios

Outside of using IIS Integration when hosting out-of-process, Forwarded Headers Middleware isn't enabled by default. Forwarded Headers Middleware must be enabled for an app to process forwarded headers with UseForwardedHeaders. After enabling the middleware if no ForwardedHeadersOptions are specified to the middleware, the default ForwardedHeadersOptions.ForwardedHeaders are ForwardedHeaders.None.

Configure the middleware with ForwardedHeadersOptions to forward the X-Forwarded-For and X-Forwarded-Proto headers.

Forwarded Headers Middleware order

Forwarded Headers Middleware should run before other middleware. This ordering ensures that the middleware relying on forwarded headers information can consume the header values for processing. Forwarded Headers Middleware can run after diagnostics and error handling, but it must be run before calling UseHsts:

C#

```
using Microsoft.AspNetCore.HttpOverrides;
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseForwardedHeaders();
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseForwardedHeaders();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Alternatively, call `UseForwardedHeaders` before diagnostics:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
```

```
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

> **Note**
>
> If no ForwardedHeadersOptions are specified or applied directly to the extension method with UseForwardedHeaders, the default headers to forward are ForwardedHeaders.None. The ForwardedHeaders property must be configured with the headers to forward.

Nginx configuration

To forward the `X-Forwarded-For` and `X-Forwarded-Proto` headers, see Host ASP.NET Core on Linux with Nginx. For more information, see NGINX: Using the Forwarded header.

Apache configuration

`X-Forwarded-For` is added automatically. For more information, see Apache Module mod_proxy: Reverse Proxy Request Headers.

Forwarded Headers Middleware options

ForwardedHeadersOptions control the behavior of the Forwarded Headers Middleware. The following example changes the default values:

- Limits the number of entries in the forwarded headers to `2`.
- Adds a known proxy address of `127.0.10.1`.
- Changes the forwarded header name from the default `X-Forwarded-For` to `X-Forwarded-For-My-Custom-Header-Name`.

C#

```
using System.Net;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardLimit = 2;
    options.KnownProxies.Add(IPAddress.Parse("127.0.10.1"));
    options.ForwardedForHeaderName = "X-Forwarded-For-My-Custom-Header-
Name";
});

var app = builder.Build();

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Expand table

Option	Description
AllowedHosts	Restricts hosts by the `X-Forwarded-Host` header to the values provided. • Values are compared using ordinal-ignore-case. • Port numbers must be excluded. • If the list is empty, all hosts are allowed. • A top-level wildcard `*` allows all non-empty hosts. • Subdomain wildcards are permitted but don't match the root domain. For example, `*.contoso.com` matches the subdomain `foo.contoso.com` but not the root domain `contoso.com`. • Unicode host names are allowed but are converted to Punycode for matching. • IPv6 addresses must include bounding brackets and be in conventional form (for example,

Option	Description
	`[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]`). IPv6 addresses aren't special-cased to check for logical equality between different formats, and no canonicalization is performed. • Failure to restrict the allowed hosts may allow a cyberattacker to spoof links generated by the service. The default value is an empty `IList<string>`.
ForwardedForHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedForHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-For` header but uses some other header to forward the information. The default is `X-Forwarded-For`.
ForwardedHeaders	Identifies which forwarders should be processed. See the ForwardedHeaders Enum for the list of fields that apply. Typical values assigned to this property are `ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto`. The default value is ForwardedHeaders.None.
ForwardedHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedHostHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Host` header but uses some other header to forward the information. The default is `X-Forwarded-Host`.
ForwardedProtoHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XForwardedProtoHeaderName. This option is used when the proxy/forwarder doesn't use the `X-Forwarded-Proto` header but uses some other header to forward the information. The default is `X-Forwarded-Proto`.
ForwardLimit	Limits the number of entries in the headers that are processed. Set to `null` to disable the limit, but this should only be done if `KnownProxies` or `KnownNetworks` are configured. Setting a non-`null` value is a precaution (but not a guarantee) to protect

Option	Description
	against misconfigured proxies and malicious requests arriving from side-channels on the network. Forwarded Headers Middleware processes headers in reverse order from right to left. If the default value (`1`) is used, only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased. The default is `1`.
KnownNetworks	Address ranges of known networks to accept forwarded headers from. Provide IP ranges using Classless Interdomain Routing (CIDR) notation. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. The default is an `IList`<IPNetwork> containing a single entry for `new IPNetwork(IPAddress.Loopback, 8)`.
KnownProxies	Addresses of known proxies to accept forwarded headers from. Use `KnownProxies` to specify exact IP address matches. If the server is using dual-mode sockets, IPv4 addresses are supplied in an IPv6 format (for example, `10.0.0.1` in IPv4 represented in IPv6 as `::ffff:10.0.0.1`). See IPAddress.MapToIPv6. Determine if this format is required by looking at the HttpContext.Connection.RemoteIpAddress. The default is an `IList`<IPAddress> containing a single entry for `IPAddress.IPv6Loopback`.
OriginalForHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalForHeaderName. The default is `X-Original-For`.
OriginalHostHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalHostHeaderName. The default is `X-Original-Host`.

Option	Description
OriginalProtoHeaderName	Use the header specified by this property instead of the one specified by ForwardedHeadersDefaults.XOriginalProtoHeaderName. The default is `X-Original-Proto`.
RequireHeaderSymmetry	Require the number of header values to be in sync between the ForwardedHeadersOptions.ForwardedHeaders being processed. The default in ASP.NET Core 1.x is `true`. The default in ASP.NET Core 2.0 or later is `false`.

Scenarios and use cases

When it isn't possible to add forwarded headers and all requests are secure

In some cases, it might not be possible to add forwarded headers to the requests proxied to the app. If the proxy is enforcing that all public external requests are HTTPS, the scheme can be manually set before using any type of middleware:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

app.Use((context, next) =>
{
    context.Request.Scheme = "https";
    return next(context);
});

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
```

```
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

This code can be disabled with an environment variable or other configuration setting in a development or staging environment:

C#

```
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

if (!app.Environment.IsProduction())
{
    app.Use((context, next) =>
    {
        context.Request.Scheme = "https";
        return next(context);
    });
}

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();
```

```
app.MapRazorPages();

app.Run();
```

Work with path base and proxies that change the request path

Some proxies pass the path intact but with an app base path that should be removed so that routing works properly. UsePathBaseExtensions.UsePathBase middleware splits the path into HttpRequest.Path and the app base path into HttpRequest.PathBase.

If `/foo` is the app base path for a proxy path passed as `/foo/api/1`, the middleware sets `Request.PathBase` to `/foo` and `Request.Path` to `/api/1` with the following command:

C#

```
app.UsePathBase("/foo");
// ...
app.UseRouting();
```

> **Note**
>
> When using WebApplication (see Migrate from ASP.NET Core 5.0 to 6.0), app.UseRouting must be called after `UsePathBase` so that the routing middleware can observe the modified path before matching routes. Otherwise, routes are matched before the path is rewritten by `UsePathBase` as described in the Middleware Ordering and Routing articles.

The original path and path base are reapplied when the middleware is called again in reverse. For more information on middleware order processing, see ASP.NET Core Middleware.

If the proxy trims the path (for example, forwarding `/foo/api/1` to `/api/1`), fix redirects and links by setting the request's PathBase property:

C#

```
app.Use((context, next) =>
{
    context.Request.PathBase = new PathString("/foo");
    return next(context);
});
```

If the proxy is adding path data, discard part of the path to fix redirects and links by using StartsWithSegments and assigning to the Path property:

```
C#

app.Use((context, next) =>
{
    if (context.Request.Path.StartsWithSegments("/foo", out var remainder))
    {
        context.Request.Path = remainder;
    }

    return next(context);
});
```

Configuration for a proxy that uses different header names

If the proxy doesn't use headers named `X-Forwarded-For` and `X-Forwarded-Proto` to forward the proxy address/port and originating scheme information, set the ForwardedForHeaderName and ForwardedProtoHeaderName options to match the header names used by the proxy:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedForHeaderName = "HeaderNamUsedByProxy_X-Forwarded-
For_Header";
    options.ForwardedProtoHeaderName = "HeaderNamUsedByProxy_X-Forwarded-
Proto_Header";
});

var app = builder.Build();

app.UseForwardedHeaders();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Forward the scheme for Linux and non-IIS reverse proxies

Apps that call UseHttpsRedirection and UseHsts put a site into an infinite loop if deployed to an Azure Linux App Service, Azure Linux virtual machine (VM), or behind any other reverse proxy besides IIS. TLS is terminated by the reverse proxy, and Kestrel isn't made aware of the correct request scheme. OAuth and OIDC also fail in this configuration because they generate incorrect redirects. UseIISIntegration adds and configures Forwarded Headers Middleware when running behind IIS, but there's no matching automatic configuration for Linux (Apache or Nginx integration).

To forward the scheme from the proxy in non-IIS scenarios, enable the Forwarded Headers Middleware by setting `ASPNETCORE_FORWARDEDHEADERS_ENABLED` to `true`. Warning: This flag uses settings designed for cloud environments and doesn't enable features such as the KnownProxies option to restrict which IPs forwarders are accepted from.

Certificate forwarding

Azure

To configure Azure App Service for certificate forwarding, see Configure TLS mutual authentication for Azure App Service. The following guidance pertains to configuring the ASP.NET Core app.

- Configure Certificate Forwarding Middleware to specify the header name that Azure uses. Add the following code to configure the header from which the middleware builds a certificate.
- Call UseCertificateForwarding before the call to UseAuthentication.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddCertificateForwarding(options =>
    options.CertificateHeader = "X-ARR-ClientCert");
```

```
var app = builder.Build();

app.UseCertificateForwarding();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();
app.UseAuthentication();

app.MapRazorPages();

app.Run();
```

Other web proxies

If a proxy is used that isn't IIS or Azure App Service's Application Request Routing (ARR), configure the proxy to forward the certificate that it received in an HTTP header.

- Configure Certificate Forwarding Middleware to specify the header name. Add the following code to configure the header from which the middleware builds a certificate.
- Call UseCertificateForwarding before the call to UseAuthentication.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddCertificateForwarding(options =>
    options.CertificateHeader = "YOUR_CERTIFICATE_HEADER_NAME");

var app = builder.Build();

app.UseCertificateForwarding();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
```

```
app.UseStaticFiles();

app.UseAuthorization();
app.UseAuthentication();

app.MapRazorPages();

app.Run();
```

If the proxy isn't base64-encoding the certificate, as is the case with Nginx, set the `HeaderConverter` option. Consider the following example:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddCertificateForwarding(options =>
{
    options.CertificateHeader = "YOUR_CUSTOM_HEADER_NAME";
    options.HeaderConverter = (headerValue) =>
    {
        // Conversion logic to create an X509Certificate2.
        var clientCertificate = ConversionLogic.CreateAnX509Certificate2();
        return clientCertificate;
    };
});

var app = builder.Build();

app.UseCertificateForwarding();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();
app.UseAuthentication();

app.MapRazorPages();

app.Run();
```

Troubleshoot

When headers aren't forwarded as expected, enable `debug` level logging and HTTP request logging. UseHttpLogging must be called after UseForwardedHeaders:

C#

```
using Microsoft.AspNetCore.HttpLogging;
using Microsoft.AspNetCore.HttpOverrides;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddHttpLogging(options =>
{
    options.LoggingFields = HttpLoggingFields.RequestPropertiesAndHeaders;
});

builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
});

var app = builder.Build();

app.UseForwardedHeaders();
app.UseHttpLogging();

app.Use(async (context, next) =>
{
    // Connection: RemoteIp
    app.Logger.LogInformation("Request RemoteIp: {RemoteIpAddress}",
        context.Connection.RemoteIpAddress);

    await next(context);
});

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

If there are multiple values in a given header, Forwarded Headers Middleware processes headers in reverse order from right to left. The default `ForwardLimit` is `1` (one), so only the rightmost value from the headers is processed unless the value of `ForwardLimit` is increased.

The request's original remote IP must match an entry in the KnownProxies or KnownNetworks lists before forwarded headers are processed. This limits header spoofing by not accepting forwarders from untrusted proxies. When an unknown proxy is detected, logging indicates the address of the proxy:

Console

```
September 20th 2018, 15:49:44.168 Unknown proxy: 10.0.0.100:54321
```

In the preceding example, 10.0.0.100 is a proxy server. If the server is a trusted proxy, add the server's IP address to `KnownProxies`, or add a trusted network to `KnownNetworks`. For more information, see the Forwarded Headers Middleware options section.

C#

```
using Microsoft.AspNetCore.HttpOverrides;
using System.Net;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
    options.KnownProxies.Add(IPAddress.Parse("10.0.0.100"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseForwardedHeaders();
    app.UseHsts();
}
else
{
    app.UseDeveloperExceptionPage();
    app.UseForwardedHeaders();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

To display the logs, add `"Microsoft.AspNetCore.HttpLogging": "Information"` to the `appsettings.Development.json` file:

JSON

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.HttpLogging": "Information"
    }
  }
}
```

> **Important**
>
> Only allow trusted proxies and networks to forward headers. Otherwise, IP spoofing attacks are possible.

Additional resources

- Host ASP.NET Core in a web farm
- Microsoft Security Advisory CVE-2018-0787: ASP.NET Core Elevation Of Privilege Vulnerability
- YARP: Yet Another Reverse Proxy

Deploying and scaling an ASP.NET Core app on Azure Container Apps

Article • 04/13/2023

Apps deployed to Azure that experience intermittent high demand benefit from scalability to meet demand. Scalable apps can scale out to ensure capacity during workload peaks and then scale down automatically when the peak drops, which can lower costs. Horizontal scaling (scaling out) adds new instances of a resource, such as VMs or database replicas. This article demonstrates how to deploy a horizontally scalable ASP.NET Core app to Azure container apps by completing the following tasks:

1. Set up the sample project
2. Deploy the app to Azure Container Apps
3. Scale and troubleshoot the app
4. Create the Azure Services
5. Connect the Azure services
6. Configure and redeploy the app

This article uses Razor Pages, but most of it applies to other ASP.NET Core apps.

In some cases, basic ASP.NET Core apps are able to scale without special considerations. However, apps that utilize certain framework features or architectural patterns require extra configurations, including the following:

- **Secure form submissions**: Razor Pages, MVC and Web API apps often rely on form submissions. By default these apps use cross site forgery tokens and internal data protection services to secure requests. When deployed to the cloud, these apps must be configured to manage data protection service concerns in a secure, centralized location.

- **SignalR circuits**: Blazor Server apps require the use of a centralized Azure SignalR service in order to securely scale. These services also utilize the data protection services mentioned previously.

- **Centralized caching or state management services**: Scalable apps may use Azure Cache for Redis to provide distributed caching. Azure storage may be needed to store state for frameworks such as Microsoft Orleans, which can help write apps that manage state across many different app instances.

The steps in this article demonstrate how to properly address the preceding concerns by deploying a scalable app to Azure Container Apps. Most of the concepts in this tutorial

also apply when scaling Azure App Service instances.

Set up the sample project

Use the GitHub Explorer sample app to follow along with this tutorial. Clone the app from GitHub using the following command:

.NET CLI

```
git clone "https://github.com/dotnet/AspNetCore.Docs.Samples.git"
```

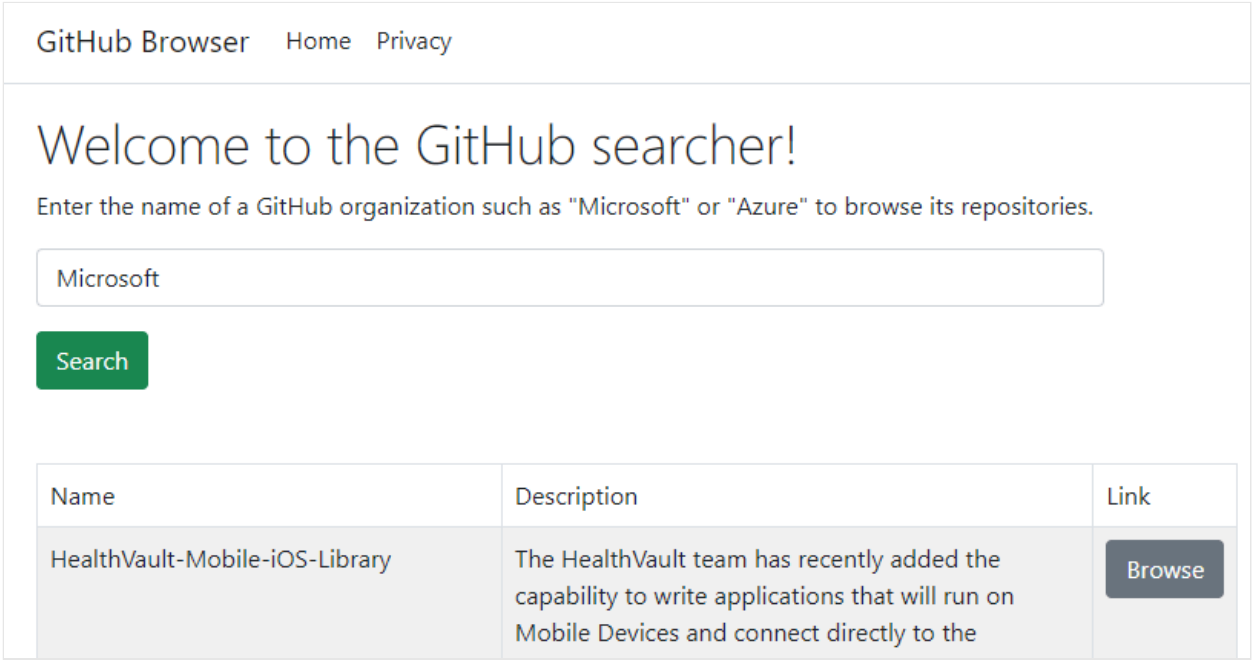
Navigate to the `/tutorials/scalable-razor-apps/start` folder and open the `ScalableRazor.csproj`.

The sample app uses a search form to browse GitHub repositories by name. The form relies on the built-in ASP.NET Core data protection services to handle antiforgery concerns. By default, when the app scales horizontally on Container Apps, the data protection service throws an exception.

Test the app

1. Launch the app in Visual Studio. The project includes a Docker file, which means that the arrow next to the run button can be selected to start the app using either a Docker Desktop setup or the standard ASP.NET Core local web server.

Use the search form to browse for GitHub repositories by name.

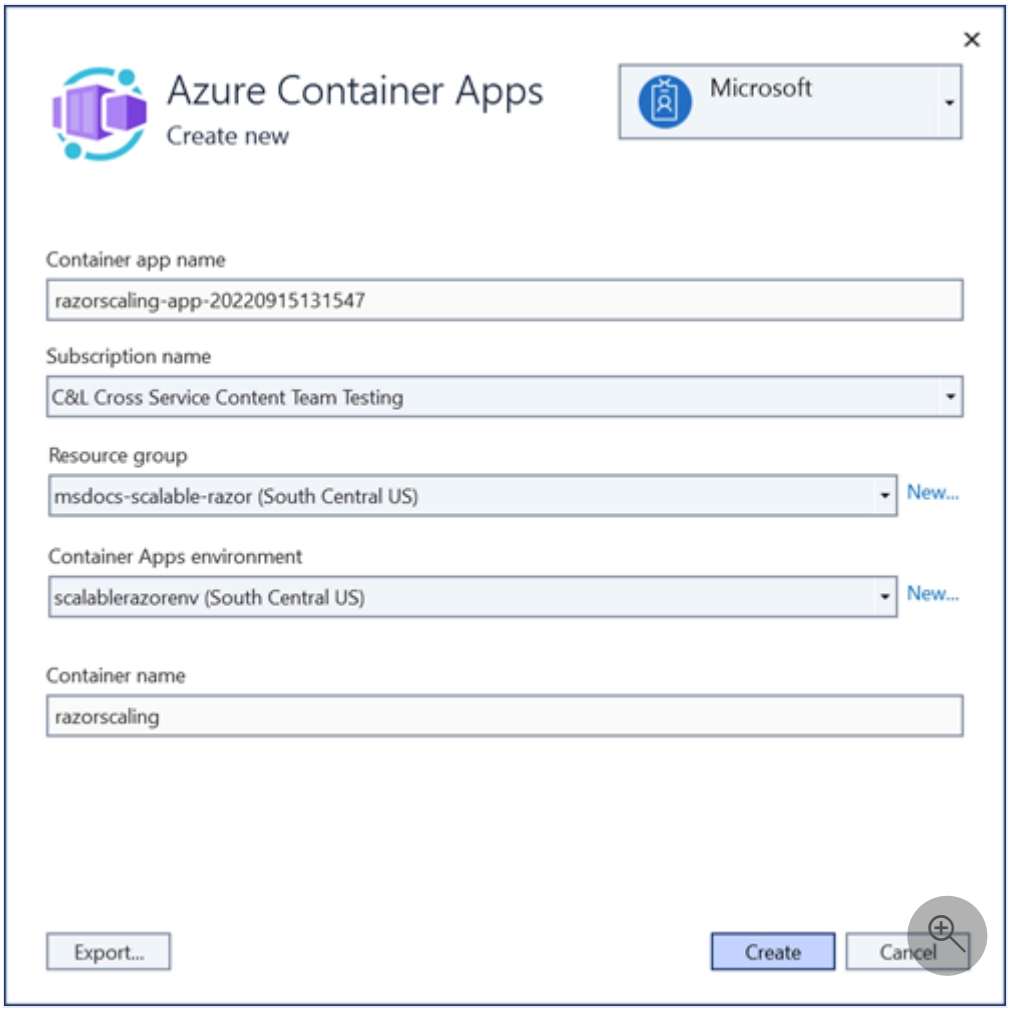


Deploy the app to Azure Container Apps

Visual Studio is used to deploy the app to Azure Container Apps. Container apps provide a managed service designed to simplify hosting containerized apps and microservices.

> **Note**
>
> Many of the resources created for the app require a location. For this app, location isn't important. A real app should select a location closest to the clients. You may want to select a location near you.

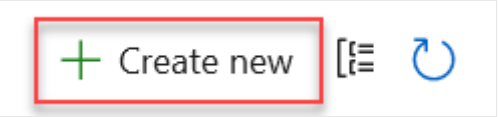
1. In Visual Studio solution explorer, right click on the top level project node and select **Publish**.
2. In the publishing dialog, select **Azure** as the deployment target, and then select **Next**.
3. For the specific target, select **Azure Container Apps (Linux)**, and then select **Next**.
4. Create a new container app to deploy to. Select the green + icon to open a new dialog and enter the following values:



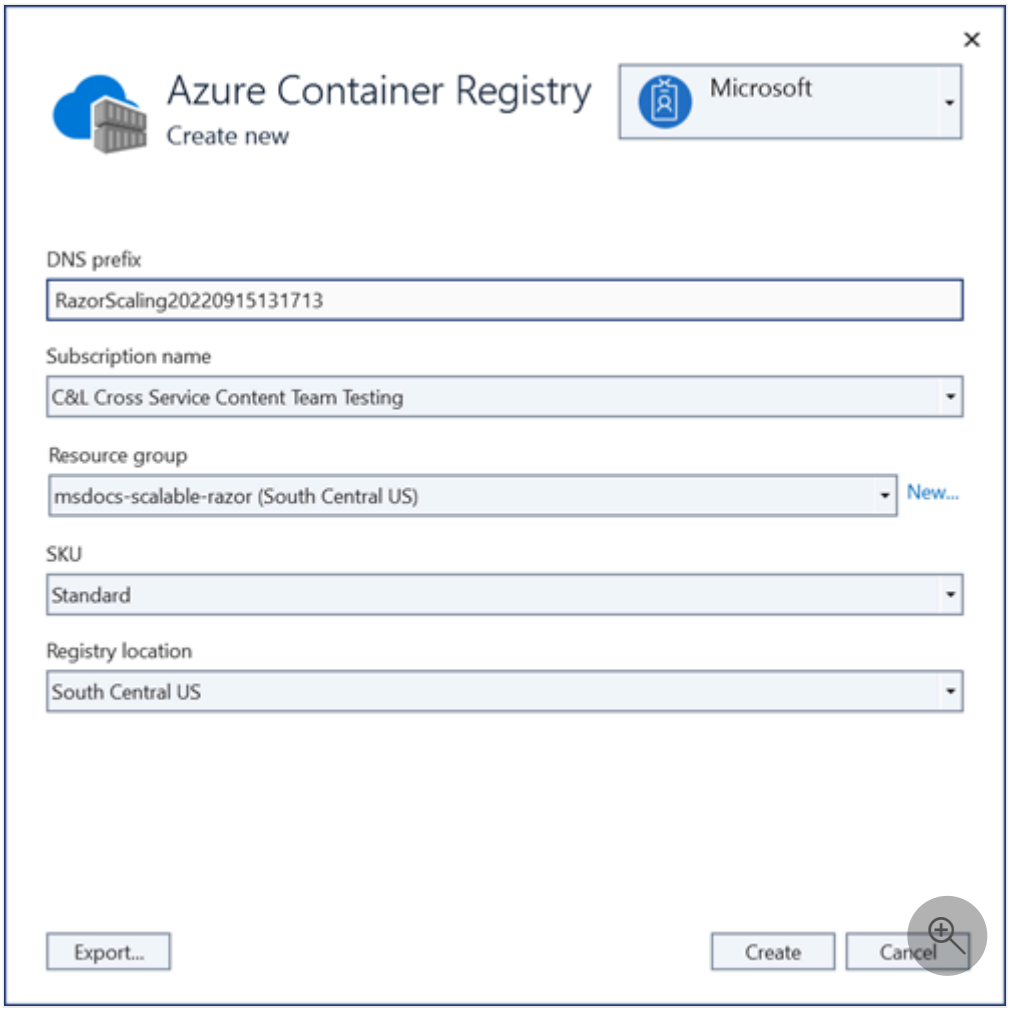
- **Container app name**: Leave the default value or enter a name.
- **Subscription name**: Select the subscription to deploy to.
- **Resource group**: Select **New** and create a new resource group called *msdocs-scalable-razor*.
- **Container apps environment**: Select **New** to open the container apps environment dialog and enter the following values:
  - **Environment name**: Keep the default value.
  - **Location**: Select a location near you.
  - **Azure Log Analytics Workspace**: Select **New** to open the log analytics workspace dialog.
    - **Name**: Leave the default value.
    - **Location**: Select a location near you and then select **Ok** to close the dialog.
  - Select **Ok** to close the container apps environment dialog.
- Select **Create** to close the original container apps dialog. Visual Studio creates the container app resource in Azure.

5. Once the resource is created, make sure it's selected in the list of container apps, and then select **Next**.

6. You'll need to create an Azure Container Registry to store the published image artifact for your app. Select the green + icon on the container registry screen.



7. Leave the default values, and then select **Create**.



8. After the container registry is created, make sure it's selected, and then select finish to close the dialog workflow and display a summary of the publishing profile.

   If Visual Studio prompts you to enable the Admin user to access the published docker container, select **Yes**.

9. Select **Publish** in the upper right of the publishing profile summary to deploy the app to Azure.

When the deployment finishes, Visual Studio launches the browser to display the hosted app. Search for `Microsoft` in the form field, and a list of repositories is displayed.

Scale and troubleshoot the app

The app is currently working without any issues, but we'd like to scale the app across more instances in anticipation of high traffic volumes.

1. In the Azure portal, search for the `razorscaling-app-****` container app in the top level search bar and select it from the results.
2. On the overview page, select **Scale** from the left navigation, and then select **+ Edit and deploy**.
3. On the revisions page, switch to the **Scale** tab.
4. Set both the min and max instances to **4** and then select **Create**. This configuration change guarantees your app is scaled horizontally across four instances.

Navigate back to the app. When the page loads, at first it appears everything is working correctly. However, when a search term is entered and submitted, an error may occur. If an error is not displayed, submit the form several more times.

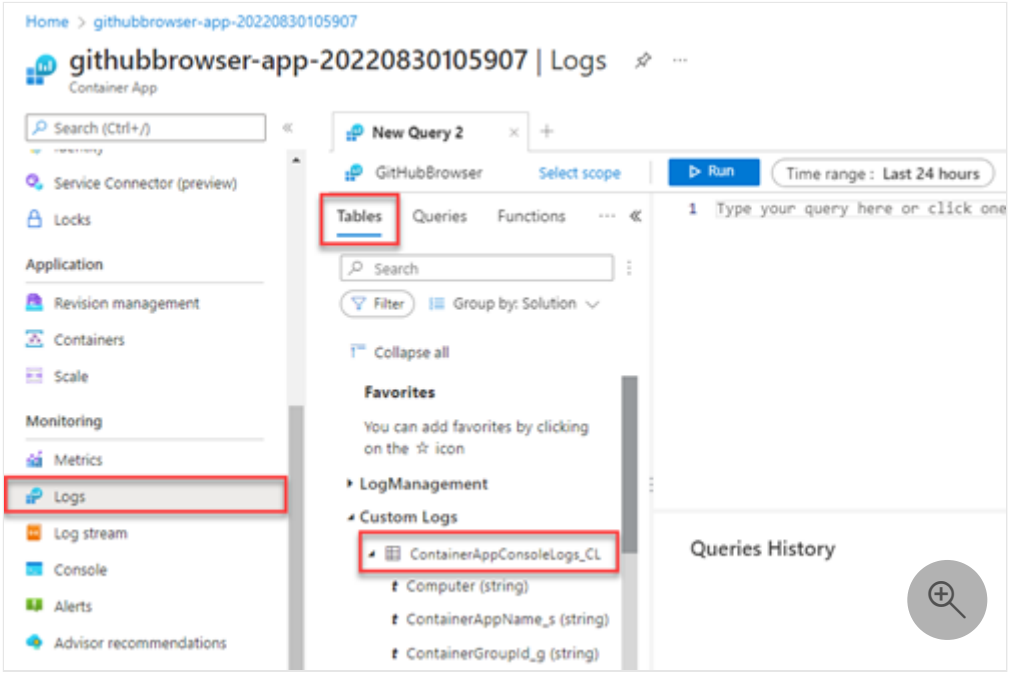
Troubleshooting the error

It's not immediately apparent why the search requests are failing. The browser tools indicate a 400 Bad Request response was sent back. However, you can use the logging features of container apps to diagnose errors occurring in your environment.

1. On the overview page of the container app, select **Logs** from the left navigation.

2. On the **Logs** page, close the pop-up that opens and navigate to the **Tables** tab.

3. Expand the **Custom Logs** item to reveal the **ContainerAppConsoleLogs_CL** node. This table holds various logs for the container app that can be queried to troubleshoot problems.

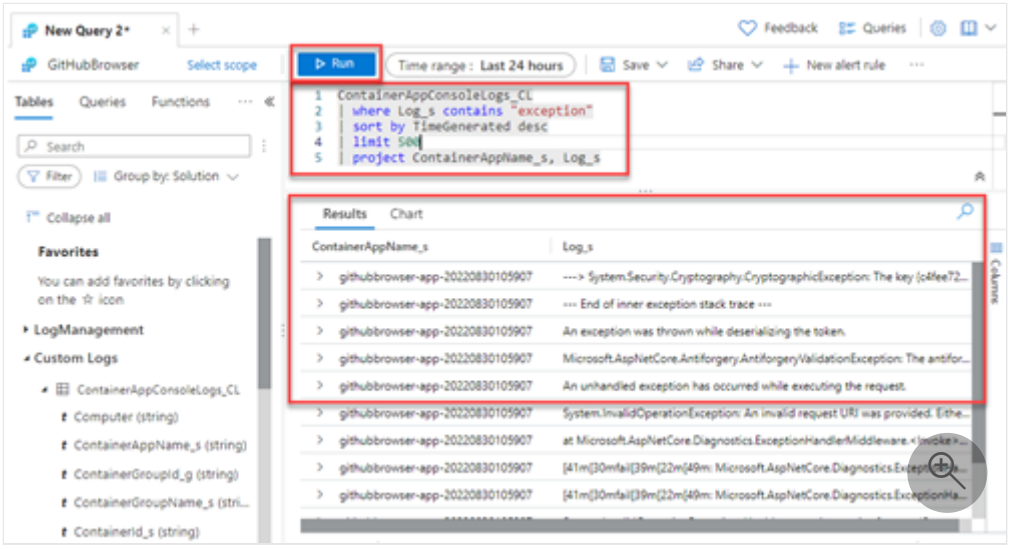


4. In the query editor, compose a basic query to search the **ContainerAppConsoleLogs_CL Logs** table for recent exceptions, such as the following script:

```KQL
ContainerAppConsoleLogs_CL
| where Log_s contains "exception"
| sort by TimeGenerated desc
| limit 500
| project ContainerAppName_s, Log_s
```

   The preceding query searches the **ContainerAppConsoleLogs_CL** table for any rows that contain the word exception. The results are ordered by the time generated, limited to 500 results, and only include the **ContainerAppName_s** and **Log_s** columns to make the results easier to read.

5. Select **Run**, a list of results is displayed. Read through the logs and note that most of them are related to antiforgery tokens and cryptography.



> **ⓘ Important**
>
> The errors in the app are caused by the .NET data protection services. When multiple instances of the app are running, there is no guarantee that the HTTP POST request to submit the form is routed to the same container that initially loaded the page from the HTTP GET request. If the requests are handled by different instances, the antiforgery tokens aren't handled correctly and an exception occurs.

   In the steps ahead, this issue is resolved by centralizing the data protection keys in an Azure storage service and protecting them with Key Vault.

Create the Azure Services

To resolve the preceding errors, the following services are created and connected to the app:

- Azure Storage Account: Handles storing data for the Data Protection Services. Provides a centralized location to store key data as the app scales. Storage accounts can also be used to hold documents, queue data, file shares, and almost any type of blob data.
- Azure KeyVault: This service stores secrets for an app, and is used to help manage encryption concerns for the Data Protection Services.

Create the storage account service

1. In the Azure portal search bar, enter `Storage accounts` and select the matching result.
2. On the storage accounts listing page, select **+ Create**.
3. On the **Basics** tab, enter the following values:

    - **Subscription**: Select the same subscription that you chose for the container app.
    - **Resource Group**: Select the *msdocs-scalable-razor* resource group you created previously.
    - **Storage account name**: Name the account *scalablerazorstorageXXXX* where the X's are random numbers of your choosing. This name must be unique across all of Azure.
    - **Region**: Select the same region you previously selected.

4. Leave the rest of the values at their default and select **Review**. After Azure validates the inputs, select **Create**.

Azure provisions the new storage account. When the task completes, choose **Go to resource** to view the new service.

Create the storage container

Create a Container to store the app's data protection keys.

1. On the overview page for the new storage account, select **Storage browser** on the left navigation.
2. Select **Blob containers**.
3. Select **+ Add container** to open the **New container** flyout menu.

4. Enter a name of *scalablerazorkeys*, leave the rest of the settings at their defaults, and then select **Create**.

The new containers appear on the page list.

Create the key vault service

Create a key vault to hold the keys that protect the data in the blob storage container.

1. In the Azure portal search bar, enter `Key Vault` and select the matching result.
2. On the key vault listing page, select **+ Create**.
3. On the **Basics** tab, enter the following values:

    - **Subscription**: Select the same subscription that was previously selected.
    - **Resource Group**: Select the *msdocs-scalable-razor* resource group previously created.
    - **Key Vault name**: Enter the name *scalablerazorvaultXXXX*.
    - **Region**: Select a region near your location.

4. Leave the rest of the settings at their default, and then select **Review + create**. Wait for Azure to validate your settings, and then select **Create**.

Azure provisions the new key vault. When the task completes, select **Go to resource** to view the new service.

Create the key

Create a secret key to protect the data in the blob storage account.

1. On the main overview page of the key vault, select **Keys** from the left navigation.
2. On the **Create a key** page, select **+ Generate/Import** to open the **Create a key** flyout menu.
3. Enter *razorkey* in the **Name** field. Leave the rest of the settings at their default values and then select **Create**. A new key appears on the key list page.

Connect the Azure Services

The Container App requires a secure connection to the storage account and the key vault services in order to resolve the data protection errors and scale correctly. The new services are connected together using the following steps:

> **ⓘ Important**

> Security role assignments through Service Connector and other tools typically take a minute or two to propagate, and in some rare cases can take up to eight minutes.

Connect the storage account

1. In the Azure portal, navigate to the Container App overview page.
2. On the left navigation, select **Service connector**
3. On the Service Connector page, choose **+ Create** to open the **Creation Connection** flyout panel and enter the following values:

    - **Container**: Select the Container App created previously.
    - **Service type**: Choose **Storage - blob**.
    - **Subscription**: Select the subscription previously used.
    - **Connection name**: Leave the default value.
    - **Storage account**: Select the storage account created previously.
    - **Client type**: Select **.NET**.

4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.
6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector enables a system-assigned managed identity on the container app. It also assigns a role of **Storage Blob Data Contributor** to the identity so it can perform data operations on the storage containers.

Connect the key vault

1. In the Azure portal, navigate to your Container App overview page.
2. On the left navigation, select **Service connector**.
3. On the Service Connector page, choose **+ Create** to open the **Creation Connection** flyout panel and enter the following values:

    - **Container**: Select the container app created previously.
    - **Service type**: Choose **Key Vault**.
    - **Subscription**: Select the subscription previously used.
    - **Connection name**: Leave the default value.
    - **Key vault**: Select the key vault created previously.
    - **Client type**: Select **.NET**.

4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.

6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector assigns a role to the identity so it can perform data operations on the key vault keys.

Configure and redeploy the app

The necessary Azure resources have been created. In this section the app code is configured to use the new resources.

1. Install the following NuGet packages:

    - **Azure.Identity**: Provides classes to work with the Azure identity and access management services.
    - **Microsoft.Extensions.Azure**: Provides helpful extension methods to perform core Azure configurations.
    - **Azure.Extensions.AspNetCore.DataProtection.Blobs**: Allows storing ASP.NET Core DataProtection keys in Azure Blob Storage so that keys can be shared across several instances of a web app.
    - **Azure.Extensions.AspNetCore.DataProtection.Keys**: Enables protecting keys at rest using the Azure Key Vault Key Encryption/Wrapping feature.

    .NET CLI

    ```
    dotnet add package Azure.Identity
    dotnet add package Microsoft.Extensions.Azure
    dotnet add package Azure.Extensions.AspNetCore.DataProtection.Blobs
    dotnet add package Azure.Extensions.AspNetCore.DataProtection.Keys
    ```

2. Update `Program.cs` with the following highlighted code:

    C#

    ```
    using Azure.Identity;
    using Microsoft.AspNetCore.DataProtection;
    using Microsoft.Extensions.Azure;
    var builder = WebApplication.CreateBuilder(args);
    var BlobStorageUri = builder.Configuration["AzureURIs:BlobStorage"];
    var KeyVaultURI = builder.Configuration["AzureURIs:KeyVault"];

    builder.Services.AddRazorPages();
    builder.Services.AddHttpClient();
    builder.Services.AddServerSideBlazor();
    ```

```
builder.Services.AddAzureClientsCore();

builder.Services.AddDataProtection()
                .PersistKeysToAzureBlobStorage(new Uri(BlobStorageUri),
                                               new
DefaultAzureCredential())
                .ProtectKeysWithAzureKeyVault(new Uri(KeyVaultURI),
                                               new
DefaultAzureCredential());
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding changes allow the app to manage data protection using a centralized, scalable architecture. `DefaultAzureCredential` discovers the managed identity configurations enabled earlier when the app is redeployed.

Update the placeholders in `AzureURIs` section of the `appsettings.json` file to include the following:

1. Replace the `<storage-account-name>` placeholder with the name of the `scalablerazorstorageXXXX` storage account.
2. Replace the `<container-name>` placeholder with the name of the `scalablerazorkeys` storage container.
3. Replace the `<key-vault-name>` placeholder with the name of the `scalablerazorvaultXXXX` key vault.
4. Replace the `<key-name>` placeholder in the key vault URI with the `razorkey` name created previously.

```
C#
```

```
{
  "GitHubURL": "https://api.github.com",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "AzureURIs": {
    "BlobStorage": "https://<storage-account-
name>.blob.core.windows.net/<container-name>/keys.xml",
    "KeyVault": "https://<key-vault-name>.vault.azure.net/keys/<key-
name>/"
  }
}
```

Redeploy the app

The app is now configured correctly to use the Azure services created previously. Redeploy the app for the code changes to be applied.

1. Right click on the project node in the solution explorer and select **Publish**.
2. On the publishing profile summary view, select the **Publish** button in the upper right corner.

Visual Studio redeploys the app to the container apps environment created previously. When the processes finished, the browser launches to the app home page.

Test the app again by searching for *Microsoft* in the search field. The page should now reload with the correct results every time you submit.

Configure roles for local development

The existing code and configuration of the app can also work while running locally during development. The `DefaultAzureCredential` class configured previously is able to pick up local environment credentials to authenticate to Azure Services. You'll need to assign the same roles to your own account that were assigned to your app's managed identity in order for the authentication to work. This should be the same account you use to log into Visual Studio or the Azure CLI.

Sign-in to your local development environment

You'll need to be signed in to the Azure CLI, Visual Studio, or Azure PowerShell for your credentials to be picked up by `DefaultAzureCredential`.

Azure CLI

Azure CLI

```
az login
```

Assign roles to your developer account

1. In the Azure portal, navigate to the `scalablerazor****` storage account created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose **+ Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Storage blob data contributor`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select **+ Select members**.
6. In the **Select members** flyout, search for your own *user@domain* account and select it from the results.
7. Choose **Next** and then select **Review + assign**. After Azure validates the settings, select **Review + assign** again.

As previously stated, role assignment permissions might take a minute or two to propagate, or in rare cases up to eight minutes.

Repeat the previous steps to assign a role to your account so that it can access the key vault service and secret.

1. In the Azure portal, navigate to the `razorscalingkeys` key vault created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose **+ Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Key Vault Crypto Service Encryption User`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select **+ Select members**.
6. In the **Select members** flyout, search for your own *user@domain* account and select it from the results.

7. Choose **Next** and then select **Review + assign**. After Azure validates your settings, select **Review + assign** again.

You may need to wait again for this role assignment to propagate.

You can then return to Visual Studio and run the app locally. The code should continue to function as expected. `DefaultAzureCredential` uses your existing credentials from Visual Studio or the Azure CLI.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Host ASP.NET Core in a web farm

Article • 07/31/2024

By Chris Ross

A *web farm* is a group of two or more web servers (or *nodes*) that host multiple instances of an app. When requests from users arrive to a web farm, a *load balancer* distributes the requests to the web farm's nodes. Web farms improve:

- **Reliability/availability**: When one or more nodes fail, the load balancer can route requests to other functioning nodes to continue processing requests.
- **Capacity/performance**: Multiple nodes can process more requests than a single server. The load balancer balances the workload by distributing requests to the nodes.
- **Scalability**: When more or less capacity is required, the number of active nodes can be increased or decreased to match the workload. Web farm platform technologies, such as Azure App Service, can automatically add or remove nodes at the request of the system administrator or automatically without human intervention.
- **Maintainability**: Nodes of a web farm can rely on a set of shared services, which results in easier system management. For example, the nodes of a web farm can rely upon a single database server and a common network location for static resources, such as images and downloadable files.

This topic describes configuration and dependencies for ASP.NET core apps hosted in a web farm that rely upon shared resources.

General configuration

Host and deploy ASP.NET Core
Learn how to set up hosting environments and deploy ASP.NET Core apps. Configure a process manager on each node of the web farm to automate app starts and restarts. Each node requires the ASP.NET Core runtime. For more information, see the topics in the Host and deploy area of the documentation.

Configure ASP.NET Core to work with proxy servers and load balancers
Learn about configuration for apps hosted behind proxy servers and load balancers, which often obscure important request information.

Deploy ASP.NET Core apps to Azure App Service
Azure App Service is a Microsoft cloud computing platform service for hosting web

apps, including ASP.NET Core. App Service is a fully managed platform that provides automatic scaling, load balancing, patching, and continuous deployment.

App data

When an app is scaled to multiple instances, there might be app state that requires sharing across nodes. If the state is transient, consider sharing an IDistributedCache. If the shared state requires persistence, consider storing the shared state in a database.

Required configuration

Data Protection and Caching require configuration for apps deployed to a web farm.

Data Protection

The ASP.NET Core Data Protection system is used by apps to protect data. Data Protection relies upon a set of cryptographic keys stored in a *key ring*. When the Data Protection system is initialized, it applies default settings that store the key ring locally. Under the default configuration, a unique key ring is stored on each node of the web farm. Consequently, each web farm node can't decrypt data that's encrypted by an app on any other node. The default configuration isn't generally appropriate for hosting apps in a web farm. An alternative to implementing a shared key ring is to always route user requests to the same node. For more information on Data Protection system configuration for web farm deployments, see Configure ASP.NET Core Data Protection.

Caching

In a web farm environment, the caching mechanism must share cached items across the web farm's nodes. Caching must either rely upon a common Redis cache, a shared SQL Server database, or a custom caching implementation that shares cached items across the web farm. For more information, see Distributed caching in ASP.NET Core.

Dependent components

The following scenarios don't require additional configuration, but they depend on technologies that require configuration for web farms.

Expand table

Scenario	Depends on ...
Authentication	Data Protection (see Configure ASP.NET Core Data Protection). For more information, see Use cookie authentication without ASP.NET Core Identity and Share authentication cookies among ASP.NET apps.
Identity	Authentication and database configuration. For more information, see Introduction to Identity on ASP.NET Core.
Session	Data Protection (encrypted cookies) (see Configure ASP.NET Core Data Protection) and Caching (see Distributed caching in ASP.NET Core). For more information, see Session and state management: Session state.
TempData	Data Protection (encrypted cookies) (see Configure ASP.NET Core Data Protection) or Session (see Session and state management: Session state). For more information, see Session and state management: TempData.
Antiforgery	Data Protection (see Configure ASP.NET Core Data Protection). For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Troubleshoot

Data Protection and caching

When Data Protection or caching isn't configured for a web farm environment, intermittent errors occur when requests are processed. This occurs because nodes don't share the same resources and user requests aren't always routed back to the same node.

Consider a user who signs into the app using cookie authentication. The user signs into the app on one web farm node. If their next request arrives at the same node where they signed in, the app is able to decrypt the authentication cookie and allows access to the app's resource. If their next request arrives at a different node, the app can't decrypt the authentication cookie from the node where the user signed in, and authorization for the requested resource fails.

When any of the following symptoms occur **intermittently**, the problem is usually traced to improper Data Protection or caching configuration for a web farm environment:

- Authentication breaks: The authentication cookie is misconfigured or can't be decrypted. OAuth (Facebook, Microsoft, Twitter) or OpenIdConnect logins fail with the error "Correlation failed."
- Authorization breaks: Identity is lost.
- Session state loses data.
- Cached items disappear.
- TempData fails.
- POSTs fail: The antiforgery check fails.

For more information on Data Protection configuration for web farm deployments, see Configure ASP.NET Core Data Protection. For more information on caching configuration for web farm deployments, see Distributed caching in ASP.NET Core.

Obtain data from apps

If the web farm apps are capable of responding to requests, obtain request, connection, and additional data from the apps using terminal inline middleware. For more information and sample code, see Troubleshoot and debug ASP.NET Core projects.

Additional resources

- Custom Script Extension for Windows: Downloads and executes scripts on Azure virtual machines, which is useful for post-deployment configuration and software installation.
- Configure ASP.NET Core to work with proxy servers and load balancers

Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

Article • 11/26/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Sayed Ibrahim Hashimi and Rick Anderson

This document focuses on using Visual Studio 2022 or later to create and use publish profiles. The publish profiles created with Visual Studio can be used with MSBuild and Visual Studio. For instructions on publishing to Azure, see Publish an ASP.NET Core app to Azure with Visual Studio.

For the most current and detailed information on:

- Publishing with Visual studio, see Overview of Visual Studio Publish
- MSBuild, see MSBuild
- Publishing with MSBuild, see Microsoft.NET.Sdk.Publish

The `dotnet new mvc` command produces a project file containing the following root-level <Project> element:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">
    <!-- omitted for brevity -->
</Project>
```

The preceding `<Project>` element's `Sdk` attribute imports the MSBuild properties and targets from *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.props* and *$(MSBuildSDKsPath)\Microsoft.NET.Sdk.Web\Sdk\Sdk.targets*, respectively. The default location for `$(MSBuildSDKsPath)` (with Visual Studio 2022) is the *%programfiles%\Microsoft Visual Studio\2022\Preview\MSBuild\Sdks* folder.

`Microsoft.NET.Sdk.Web` (Web SDK) depends on other SDKs, including `Microsoft.NET.Sdk` (.NET Core SDK) and `Microsoft.NET.Sdk.Razor` (Razor SDK). The MSBuild properties and targets associated with each dependent SDK are imported. Publish targets import the appropriate set of targets based on the publish method used.

When MSBuild or Visual Studio loads a project, the following high-level actions occur:

- Build project
- Compute files to publish
- Publish files to destination

Compute project items

When the project is loaded, the MSBuild project items (files) are computed. The item type determines how the file is processed. By default, `.cs` files are included in the `Compile` item list. Files in the `Compile` item list are compiled.

The `Content` item list contains files that are published in addition to the build outputs. By default, files matching the patterns `wwwroot\**`, `**\*.config`, and `**\*.json` are included in the `Content` item list. For example, the `wwwroot\**` globbing pattern matches all files in the *wwwroot* folder and its subfolders.

The Web SDK imports the Razor SDK. As a result, files matching the patterns `**\*.cshtml` and `**\*.razor` are also included in the `Content` item list.

To explicitly add a file to the publish list, add the file directly in the `.csproj` file as shown in the Include Files section.

When selecting the **Publish** button in Visual Studio or when publishing from the command line:

- The properties/items are computed (the files that are needed to build).
- **Visual Studio only**: NuGet packages are restored. (Restore needs to be explicit by the user on the CLI.)
- The project builds.
- The publish items are computed (the files that are needed to publish).
- The project is published (the computed files are copied to the publish destination).

When an ASP.NET Core project references `Microsoft.NET.Sdk.Web` in the project file, an `app_offline.htm` file is placed at the root of the web app directory. When the file is present, the ASP.NET Core Module gracefully shuts down the app and serves the `app_offline.htm` file during the deployment. For more information, see the ASP.NET Core Module configuration reference.

Basic command-line publishing

Command-line publishing works on all .NET Core-supported platforms and doesn't require Visual Studio. In the following examples, the .NET CLI's dotnet publish command is run from the project directory (which contains the `.csproj` file). If the project folder isn't the current working directory, explicitly pass in the project file path. For example:

.NET CLI

```
dotnet publish C:\Webs\Web1
```

Run the following commands to create and publish a web app:

.NET CLI

```
dotnet new mvc
dotnet publish
```

The `dotnet publish` command produces a variation of the following output:

Console

```
C:\Webs\Web1>dotnet publish
Microsoft (R) Build Engine version {VERSION} for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 36.81 ms for C:\Webs\Web1\Web1.csproj.
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\Web1.dll
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\Web1.Views.dll
  Web1 -> C:\Webs\Web1\bin\Debug\{TARGET FRAMEWORK MONIKER}\publish\
```

The default publish folder format is `bin\Debug\{TARGET FRAMEWORK MONIKER}`. For example, `bin\Release\net9.0\`

The following command specifies a `Release` build and the publishing directory:

.NET CLI

```
dotnet publish -c Release -o C:\MyWebs\test
```

The `dotnet publish` command calls MSBuild, which invokes the `Publish` target. Any parameters passed to `dotnet publish` are passed to MSBuild. The `-c` and `-o` parameters map to MSBuild's `Configuration` and `OutputPath` properties, respectively.

MSBuild properties can be passed using either of the following formats:

- `-p:<NAME>=<VALUE>`

- `/p:<NAME>=<VALUE>`

For example, the following command publishes a `Release` build to a network share. The network share is specified with forward slashes (*//r8/*) and works on all .NET Core supported platforms.

.NET CLI

```
dotnet publish -c Release /p:PublishDir=//r8/release/AdminWeb
```

Confirm that the published app for deployment isn't running. Files in the *publish* folder are locked when the app is running. Deployment can't occur because locked files can't be copied.

See the Microsoft.NET.Sdk.Publish readme file for more information.

Publish profiles

This section uses Visual Studio 2022 or later to create a publishing profile. Once the profile is created, publishing from Visual Studio or the command line is available. Publish profiles can simplify the publishing process, and any number of profiles can exist.

Create a publish profile in Visual Studio by choosing one of the following paths:

- Right-click the project in **Solution Explorer** and select **Publish**.
- Select **Publish {PROJECT NAME}** from the **Build** menu.

The **Publish** tab of the app capabilities page is displayed. Several publish targets are available, including:

- Azure
- Docker Container Registry
- Folder
- IIS, FTP, Web Deploy (for any web server)
- Import Profile

To determine the most appropriate publish target, see What publishing options are right for me.

When the **Folder** publish target is selected, specify a folder path to store the published assets. The default folder path is *bin\{PROJECT CONFIGURATION}\{TARGET FRAMEWORK*

MONIKER}\publish\. For example, *bin\Release\netcoreapp2.2\publish*\. Select the **Create Profile** button to finish.

Once a publish profile is created, the **Publish** tab's content changes. The newly created profile appears in a drop-down list. Below the drop-down list, select **Create new profile** to create another new profile.

Visual Studio's publish tool produces a `Properties/PublishProfiles/{PROFILE NAME}.pubxml` MSBuild file describing the publish profile. The *.pubxml* file:

- Contains publish configuration settings and is consumed by the publishing process.
- Can be modified to customize the build and publish process.

When publishing to an Azure target, the *.pubxml* file:

- Contains the Azure subscription identifier.
- Should not be checked into source control because the subscription identifier is sensitive information.

Sensitive information, for example, the publish password, is encrypted on a per user/machine level. The `Properties/PublishProfiles/{PROFILE NAME}.pubxml.user` file contains the information needed by MSBuild to retrieve the user name and password.

For an overview of how to publish an ASP.NET Core web app, see Host and deploy ASP.NET Core. The MSBuild tasks and targets necessary to publish an ASP.NET Core web app are open-source in the dotnet/websdk repository.

`dotnet publish` and `dotnet build`:

- Can use folder, MSDeploy, and Kudu publish profiles. Because MSDeploy lacks cross-platform support, MSDeploy options are supported only on Windows.
- Support Kudu APIs to publish to Azure from any platform. Visual Studio publish supports the Kudu APIs, but it's supported by WebSDK for cross-platform publish to Azure.

Don't pass `DeployOnBuild` to the `dotnet publish` command.

For more information, see Microsoft.NET.Sdk.Publish.

Folder publish example

When publishing with a profile named `FolderProfile`, use any of the following commands:

.NET CLI

```
dotnet publish /p:Configuration=Release /p:PublishProfile=FolderProfile
```

.NET CLI

```
dotnet build /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

Bash

```
msbuild /p:DeployOnBuild=true /p:PublishProfile=FolderProfile
```

The .NET CLI's dotnet build command calls `msbuild` to run the build and publish process. The `dotnet build` and `msbuild` commands are equivalent when passing in a folder profile. When calling `msbuild` directly on Windows, the .NET Framework version of MSBuild is used. Calling `dotnet build` on a non-folder profile:

- Invokes `msbuild`, which uses MSDeploy.
- Results in a failure (even when running on Windows). To publish with a non-folder profile, call `msbuild` directly.

The following folder publish profile was created with Visual Studio and publishes to a network share:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--
https://go.microsoft.com/fwlink/?LinkID=208121.
-->
<Project>
  <PropertyGroup>
    <DeleteExistingFiles>false</DeleteExistingFiles>
    <ExcludeApp_Data>false</ExcludeApp_Data>
    <LaunchSiteAfterPublish>true</LaunchSiteAfterPublish>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <PublishProvider>FileSystem</PublishProvider>
    <PublishUrl>\\r8\Release\AdminWeb</PublishUrl>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <_TargetId>Folder</_TargetId>
  </PropertyGroup>
</Project>
```

In the preceding example:

- The `<ExcludeApp_Data>` property is present to satisfy an XML schema requirement. The `<ExcludeApp_Data>` property has no effect on the publish process, even if there's an *App_Data* folder in the project root. The *App_Data* folder doesn't receive special treatment as it does in ASP.NET 4.x projects.
- The `<LastUsedBuildConfiguration>` property is set to `Release`. When publishing from Visual Studio, the value of `<LastUsedBuildConfiguration>` is set using the value when the publish process is started. `<LastUsedBuildConfiguration>` is special and shouldn't be overridden in an imported MSBuild file. This property can, however, be overridden from the command line using one of the following approaches.
  - Using the .NET CLI:

    .NET CLI

    ```
    dotnet publish /p:Configuration=Release
    /p:PublishProfile=FolderProfile
    ```

    .NET CLI

    ```
    dotnet build -c Release /p:DeployOnBuild=true
    /p:PublishProfile=FolderProfile
    ```

  - Using MSBuild:

    Bash

    ```
    msbuild /p:Configuration=Release /p:DeployOnBuild=true
    /p:PublishProfile=FolderProfile
    ```

Publish to an MSDeploy endpoint from the command line

See Microsoft.NET.Sdk.Publish.

Set the environment

Include the `<EnvironmentName>` property in the publish profile (*.pubxml*) or project file to set the app's environment:

XML

```
<PropertyGroup>
  <EnvironmentName>Development</EnvironmentName>
</PropertyGroup>
```

If you require *web.config* transformations (for example, setting environment variables based on the configuration, profile, or environment), see Transform web.config.

Exclude files

When publishing ASP.NET Core web apps, the following assets are included:

- Build artifacts
- Folders and files matching the following globbing patterns:
  - `**\*.config` (for example, *web.config*)
  - `**\*.json` (for example, `appsettings.json`)
  - `wwwroot\**`

MSBuild supports globbing patterns. For example, the following `<Content>` element suppresses the copying of text (*.txt*) files in the *wwwroot\content* folder and its subfolders:

XML

```
<ItemGroup>
  <Content Update="wwwroot/content/**/*.txt" CopyToPublishDirectory="Never"
/>
</ItemGroup>
```

The preceding markup can be added to a publish profile or the `.csproj` file. When added to the `.csproj` file, the rule is added to all publish profiles in the project.

The following `<MsDeploySkipRules>` element excludes all files from the *wwwroot\content* folder:

XML

```
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFolder">
    <ObjectName>dirPath</ObjectName>
    <AbsolutePath>wwwroot\\content</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

`<MsDeploySkipRules>` won't delete the *skip* targets from the deployment site. `<Content>` targeted files and folders are deleted from the deployment site. For example, suppose a deployed web app had the following files:

- `Views/Home/About1.cshtml`
- `Views/Home/About2.cshtml`
- `Views/Home/About3.cshtml`

If the following `<MsDeploySkipRules>` elements are added, those files wouldn't be deleted on the deployment site.

XML

```xml
<ItemGroup>
  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About1.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About2.cshtml</AbsolutePath>
  </MsDeploySkipRules>

  <MsDeploySkipRules Include="CustomSkipFile">
    <ObjectName>filePath</ObjectName>
    <AbsolutePath>Views\\Home\\About3.cshtml</AbsolutePath>
  </MsDeploySkipRules>
</ItemGroup>
```

The preceding `<MsDeploySkipRules>` elements prevent the *skipped* files from being deployed. It won't delete those files once they're deployed.

The following `<Content>` element deletes the targeted files at the deployment site:

XML

```xml
<ItemGroup>
  <Content Update="Views/Home/About?.cshtml" CopyToPublishDirectory="Never"
/>
</ItemGroup>
```

Using command-line deployment with the preceding `<Content>` element yields a variation of the following output:

Console

```
MSDeployPublish:
  Starting Web deployment task from source:
manifest(C:\Webs\Web1\obj\Release\{TARGET FRAMEWORK
MONIKER}\PubTmp\Web1.SourceManifest.
  xml) to Destination: auto().
  Deleting file (Web11112\Views\Home\About1.cshtml).
  Deleting file (Web11112\Views\Home\About2.cshtml).
  Deleting file (Web11112\Views\Home\About3.cshtml).
  Updating file (Web11112\web.config).
  Updating file (Web11112\Web1.deps.json).
  Updating file (Web11112\Web1.dll).
  Updating file (Web11112\Web1.pdb).
  Updating file (Web11112\Web1.runtimeconfig.json).
  Successfully executed Web deployment task.
  Publish Succeeded.
Done Building Project "C:\Webs\Web1\Web1.csproj" (default targets).
```

Include files

The following sections outline different approaches for file inclusion at publish time. The General file inclusion section uses the `DotNetPublishFiles` item, which is provided by a publish targets file in the Web SDK. The Selective file inclusion section uses the `ResolvedFileToPublish` item, which is provided by a publish targets file in the .NET Core SDK. Because the Web SDK depends on the .NET Core SDK, either item can be used in an ASP.NET Core project.

General file inclusion

The following example's `<ItemGroup>` element demonstrates copying a folder located outside of the project directory to a folder of the published site. Any files added to the following markup's `<ItemGroup>` are included by default.

XML

```
<ItemGroup>
  <_CustomFiles Include="$(MSBuildProjectDirectory)/../images/**/*" />
  <DotNetPublishFiles Include="@(_CustomFiles)">
    <DestinationRelativePath>wwwroot/images/%(RecursiveDir)%(Filename)%
(Extension)</DestinationRelativePath>
  </DotNetPublishFiles>
</ItemGroup>
```

The preceding markup:

- Can be added to the `.csproj` file or the publish profile. If it's added to the `.csproj` file, it's included in each publish profile in the project.
- Declares a `_CustomFiles` item to store files matching the `Include` attribute's globbing pattern. The *images* folder referenced in the pattern is located outside of the project directory. A reserved property, named `$(MSBuildProjectDirectory)`, resolves to the project file's absolute path.
- Provides a list of files to the `DotNetPublishFiles` item. By default, the item's `<DestinationRelativePath>` element is empty. The default value is overridden in the markup and uses well-known item metadata such as `%(RecursiveDir)`. The inner text represents the *wwwroot/images* folder of the published site.

Selective file inclusion

The highlighted markup in the following example demonstrates:

- Copying a file located outside of the project into the published site's *wwwroot* folder. The file name of *ReadMe2.md* is maintained.
- Excluding the *wwwroot\Content* folder.
- Excluding *Views\Home\About2.cshtml*.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <WebPublishMethod>FileSystem</WebPublishMethod>
    <PublishProvider>FileSystem</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>
    <SiteUrlToLaunchAfterPublish />
    <LaunchSiteAfterPublish>True</LaunchSiteAfterPublish>
    <ExcludeApp_Data>False</ExcludeApp_Data>
    <PublishFramework />
    <ProjectGuid>
      <GUID Here=""></GUID>
    </ProjectGuid>
    <publishUrl>bin\Release\PublishOutput</publishUrl>
    <DeleteExistingFiles>False</DeleteExistingFiles>
  </PropertyGroup>
  <ItemGroup>
    <ResolvedFileToPublish Include="..\ReadMe2.md">
      <RelativePath>wwwroot\ReadMe2.md</RelativePath>
    </ResolvedFileToPublish>

    <Content Update="wwwroot\Content\**\*" CopyToPublishDirectory="Never" />
    <Content Update="Views\Home\About2.cshtml"
CopyToPublishDirectory="Never" />
```

```
  </ItemGroup>
</Project>
```

The preceding example uses the `ResolvedFileToPublish` item, whose default behavior is to always copy the files provided in the `Include` attribute to the published site. Override the default behavior by including a `<CopyToPublishDirectory>` child element with inner text of either `Never` or `PreserveNewest`. For example:

XML

```
<ResolvedFileToPublish Include="..\ReadMe2.md">
  <RelativePath>wwwroot\ReadMe2.md</RelativePath>
  <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
</ResolvedFileToPublish>
```

For more deployment samples, see the Web SDK README file.

Run a target before or after publishing

The built-in `BeforePublish` and `AfterPublish` targets execute a target before or after the publish target. Add the following elements to the publish profile to log console messages both before and after publishing:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file is used by the publish/package process of your Web project. You
can customize the behavior of this process
by editing this MSBuild file. In order to learn more about this please visit
https://go.microsoft.com/fwlink/?LinkID=208121.
-->
<Project>
  <PropertyGroup>
    <WebPublishMethod>MSDeploy</WebPublishMethod>

<ResourceId>/subscriptions/SomeGuid/resourcegroups/TP_RG/providers/Microsoft
.Web/sites/TP22</ResourceId>
    <ResourceGroup>TP_RG</ResourceGroup>
    <PublishProvider>AzureWebSite</PublishProvider>
    <LastUsedBuildConfiguration>Release</LastUsedBuildConfiguration>
    <LastUsedPlatform>Any CPU</LastUsedPlatform>

<SiteUrlToLaunchAfterPublish>https://tp22.azurewebsites.net</SiteUrlToLaunch
AfterPublish>
    <LaunchSiteAfterPublish>true</LaunchSiteAfterPublish>
    <ExcludeApp_Data>false</ExcludeApp_Data>
    <ProjectGuid>GuidHere</ProjectGuid>
```

```
<MSDeployServiceURL>something.scm.azurewebsites.net:443</MSDeployServiceURL>
    <DeployIisAppPath>myDeploysIISpath</DeployIisAppPath>
    <RemoteSitePhysicalPath />
    <SkipExtraFilesOnServer>true</SkipExtraFilesOnServer>
    <MSDeployPublishMethod>WMSVC</MSDeployPublishMethod>
    <EnableMSDeployBackup>true</EnableMSDeployBackup>
    <EnableMsDeployAppOffline>true</EnableMsDeployAppOffline>
    <UserName />
    <_SavePWD>false</_SavePWD>
    <_DestinationType>AzureWebSite</_DestinationType>
    <InstallAspNetCoreSiteExtension>false</InstallAspNetCoreSiteExtension>
  </PropertyGroup>
    <Target Name="CustomActionsBeforePublish" BeforeTargets="BeforePublish">
        <Message Text="Inside BeforePublish" Importance="high" />
    </Target>
    <Target Name="CustomActionsAfterPublish" AfterTargets="AfterPublish">
        <Message Text="Inside AfterPublish" Importance="high" />
    </Target>
</Project>
```

Publish to a server using an untrusted certificate

Add the `<AllowUntrustedCertificate>` property with a value of `True` to the publish profile:

XML

```
<PropertyGroup>
  <AllowUntrustedCertificate>True</AllowUntrustedCertificate>
</PropertyGroup>
```

The Kudu service

To view the files in an Azure App Service web app deployment, use the Kudu service. Append the `scm` token to the web app name. For example:

Expand table

URL	Result
`http://mysite.azurewebsites.net/`	Web App
`http://mysite.scm.azurewebsites.net/`	Kudu service

Select the Debug Console menu item to view, edit, delete, or add files.

Additional resources

- Web SDK README file
- Web SDK GitHub repository: File issues and request features for deployment.
- Web Deploy (MSDeploy) simplifies deployment of web apps and websites to IIS servers.
- Transform web.config

ASP.NET Core directory structure

Article • 07/31/2024

The *publish* directory contains the app's deployable assets produced by the dotnet publish command. The directory contains:

- Application files
- Configuration files
- Static assets
- Packages
- A runtime (self-contained deployment only)

Expand table

App Type	Directory Structure
Framework-dependent Executable (FDE)	• publish† ◦ Views† MVC apps; if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}{.EXTENSION}.exe extension on Windows, no extension on macOS or Linux ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json ◦ web.config (IIS deployments) ◦ createdump (Linux createdump utility) ◦ *.so (Linux shared object library) ◦ *.a (macOS archive) ◦ *.dylib (macOS dynamic library)
Self-contained Deployment (SCD)	• publish† ◦ Views† MVC apps, if views aren't precompiled ◦ Pages† MVC or Razor Pages apps, if pages aren't precompiled ◦ wwwroot† ◦ *.dll files ◦ {ASSEMBLY NAME}.deps.json ◦ {ASSEMBLY NAME}.dll ◦ {ASSEMBLY NAME}{.EXTENSION} .exe extension on Windows, no extension on macOS or Linux ◦ {ASSEMBLY NAME}.pdb ◦ {ASSEMBLY NAME}.runtimeconfig.json

App Type	Directory Structure
	○ web.config (IIS deployments)

†Indicates a directory

The *publish* directory represents the *content root path*, also called the *application base path*, of the deployment. Whatever name is given to the *publish* directory of the deployed app on the server, its location serves as the server's physical path to the hosted app.

The *wwwroot* directory, if present, only contains static assets.

Additional resources

- dotnet publish
- .NET Core application deployment
- Target frameworks
- .NET Core RID Catalog

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core security topics

Article • 11/11/2024

ASP.NET Core enables developers to configure and manage security. The following list provides links to security topics:

- Authentication
- Authorization
- Data protection
- HTTPS enforcement
- Safe storage of app secrets in development
- XSRF/CSRF prevention
- Cross Origin Resource Sharing (CORS)
- Cross-Site Scripting (XSS) attacks

These security features allow you to build robust and secure ASP.NET Core apps.

For Blazor security coverage, which adds to or supersedes the guidance in this node, see ASP.NET Core Blazor authentication and authorization and the other articles in Blazor's *Security and Identity* node.

ASP.NET Core security features

ASP.NET Core provides many tools and libraries to secure ASP.NET Core apps such as built-in identity providers and third-party identity services such as Facebook, Twitter, and LinkedIn. ASP.NET Core provides several approaches to store app secrets.

Authentication vs. Authorization

Authentication is a process in which a user provides credentials that are then compared to those stored in an operating system, database, app or resource. If they match, users authenticate successfully, and can then perform actions that they're authorized for, during an authorization process. The authorization refers to the process that determines what a user is allowed to do.

Another way to think of authentication is to consider it as a way to enter a space, such as a server, database, app or resource, while authorization is which actions the user can perform to which objects inside that space (server, database, or app).

Common Vulnerabilities in software

ASP.NET Core and EF contain features that help you secure your apps and prevent security breaches. The following list of links takes you to documentation detailing techniques to avoid the most common security vulnerabilities in web apps:

- Cross-Site Scripting (XSS) attacks
- SQL injection attacks
- Cross-Site Request Forgery (XSRF/CSRF) attacks
- Open redirect attacks

There are more vulnerabilities that you should be aware of. For more information, see the other articles in the **Security and Identity** section of the table of contents.

Secure authentication flows

We recommend using the most secure secure authentication option. For Azure services, the most secure authentication is managed identities.

Avoid Resource Owner Password Credentials Grant because it:

- Exposes the user's password to the client.
- Is a significant security risk.
- Should only be used when other authentication flows are not possible.

Managed identities are a secure way to authenticate to services without needing to store credentials in code, environment variables, or configuration files. Managed identities are available for Azure services, and can be used with Azure SQL, Azure Storage, and other Azure services:

- Managed identities in Microsoft Entra for Azure SQL
- Managed identities for App Service and Azure Functions
- Secure authentication flows

When the app is deployed to a test server, an environment variable can be used to set the connection string to a test database server. For more information, see Configuration. Environment variables are generally stored in plain, unencrypted text. If the machine or process is compromised, environment variables can be accessed by untrusted parties. We recommend against using environment variables to store a production connection string as it's not the most secure approach.

Configuration data guidelines:

- Never store passwords or other sensitive data in configuration provider code or in plain text configuration files. The Secret Manager tool can be used to store secrets in development.

- Don't use production secrets in development or test environments.
- Specify secrets outside of the project so that they can't be accidentally committed to a source code repository.

For more information, see:

- Managed identity best practice recommendations
- Connecting from your application to resources without handling credentials in your code
- Azure services that can use managed identities to access other services
- IETF OAuth 2.0 Security Best Current Practice

For information on other cloud providers, see:

- AWS (Amazon Web Services): AWS Key Management Service (KMS)
- Google Cloud Key Management Service overview

Additional resources

- Introduction to Identity on ASP.NET Core
- Enable QR code generation for TOTP authenticator apps in ASP.NET Core
- Facebook and Google authentication in ASP.NET Core
- Identity management solutions for .NET web apps

Overview of ASP.NET Core authentication

Article • 10/18/2023

By Mike Rousos

Authentication is the process of determining a user's identity. Authorization is the process of determining whether a user has access to a resource. In ASP.NET Core, authentication is handled by the authentication service, IAuthenticationService, which is used by authentication middleware. The authentication service uses registered authentication handlers to complete authentication-related actions. Examples of authentication-related actions include:

- Authenticating a user.
- Responding when an unauthenticated user tries to access a restricted resource.

The registered authentication handlers and their configuration options are called "schemes".

Authentication schemes are specified by registering authentication services in `Program.cs`:

- By calling a scheme-specific extension method after a call to AddAuthentication, such as AddJwtBearer or AddCookie. These extension methods use AuthenticationBuilder.AddScheme to register schemes with appropriate settings.
- Less commonly, by calling `AuthenticationBuilder.AddScheme` directly.

For example, the following code registers authentication services and handlers for cookie and JWT bearer authentication schemes:

C#

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("JwtSettings", options))
    .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme,
        options => builder.Configuration.Bind("CookieSettings", options));
```

The `AddAuthentication` parameter JwtBearerDefaults.AuthenticationScheme is the name of the scheme to use by default when a specific scheme isn't requested.

If multiple schemes are used, authorization policies (or authorization attributes) can specify the authentication scheme (or schemes) they depend on to authenticate the

user. In the example above, the cookie authentication scheme could be used by specifying its name (CookieAuthenticationDefaults.AuthenticationScheme by default, though a different name could be provided when calling `AddCookie`).

In some cases, the call to `AddAuthentication` is automatically made by other extension methods. For example, when using ASP.NET Core Identity, `AddAuthentication` is called internally.

The Authentication middleware is added in `Program.cs` by calling UseAuthentication. Calling `UseAuthentication` registers the middleware that uses the previously registered authentication schemes. Call `UseAuthentication` before any middleware that depends on users being authenticated.

Authentication concepts

Authentication is responsible for providing the ClaimsPrincipal for authorization to make permission decisions against. There are multiple authentication scheme approaches to select which authentication handler is responsible for generating the correct set of claims:

- Authentication scheme
- The default authentication scheme, discussed in the next two sections.
- Directly set HttpContext.User.

When there is only a single authentication scheme registered, it becomes the default scheme. If multiple schemes are registered and the default scheme isn't specified, a scheme must be specified in the authorize attribute, otherwise, the following error is thrown:

> InvalidOperationException: No authenticationScheme was specified, and there was no DefaultAuthenticateScheme found. The default schemes can be set using either AddAuthentication(string defaultScheme) or AddAuthentication(Action<AuthenticationOptions> configureOptions).

`DefaultScheme`

When there is only a single authentication scheme registered, the single authentication scheme:

- Is automatically used as the DefaultScheme.
- Eliminates the need to specify the `DefaultScheme` in AddAuthentication(IServiceCollection) or

AddAuthenticationCore(IServiceCollection).

To disable automatically using the single authentication scheme as the `DefaultScheme`, call `AppContext.SetSwitch("Microsoft.AspNetCore.Authentication.SuppressAutoDefaultScheme")`.

Authentication scheme

The authentication scheme can select which authentication handler is responsible for generating the correct set of claims. For more information, see Authorize with a specific scheme.

An authentication scheme is a name that corresponds to:

- An authentication handler.
- Options for configuring that specific instance of the handler.

Schemes are useful as a mechanism for referring to the authentication, challenge, and forbid behaviors of the associated handler. For example, an authorization policy can use scheme names to specify which authentication scheme (or schemes) should be used to authenticate the user. When configuring authentication, it's common to specify the default authentication scheme. The default scheme is used unless a resource requests a specific scheme. It's also possible to:

- Specify different default schemes to use for authenticate, challenge, and forbid actions.
- Combine multiple schemes into one using policy schemes.

Authentication handler

An authentication handler:

- Is a type that implements the behavior of a scheme.
- Is derived from IAuthenticationHandler or AuthenticationHandler<TOptions>.
- Has the primary responsibility to authenticate users.

Based on the authentication scheme's configuration and the incoming request context, authentication handlers:

- Construct AuthenticationTicket objects representing the user's identity if authentication is successful.
- Return 'no result' or 'failure' if authentication is unsuccessful.

- Have methods for challenge and forbid actions for when users attempt to access resources:
  - They're unauthorized to access (forbid).
  - When they're unauthenticated (challenge).

`RemoteAuthenticationHandler<TOptions>` vs `AuthenticationHandler<TOptions>`

RemoteAuthenticationHandler<TOptions> is the class for authentication that requires a remote authentication step. When the remote authentication step is finished, the handler calls back to the `CallbackPath` set by the handler. The handler finishes the authentication step using the information passed to the HandleRemoteAuthenticateAsync callback path. OAuth 2.0 and OIDC both use this pattern. JWT and cookies don't since they can directly use the bearer header and cookie to authenticate. The remotely hosted provider in this case:

- Is the authentication provider.
- Examples include Facebook, Twitter, Google, Microsoft, and any other OIDC provider that handles authenticating users using the handlers mechanism.

Authenticate

An authentication scheme's authenticate action is responsible for constructing the user's identity based on request context. It returns an AuthenticateResult indicating whether authentication was successful and, if so, the user's identity in an authentication ticket. See AuthenticateAsync. Authenticate examples include:

- A cookie authentication scheme constructing the user's identity from cookies.
- A JWT bearer scheme deserializing and validating a JWT bearer token to construct the user's identity.

Challenge

An authentication challenge is invoked by Authorization when an unauthenticated user requests an endpoint that requires authentication. An authentication challenge is issued, for example, when an anonymous user requests a restricted resource or follows a login link. Authorization invokes a challenge using the specified authentication scheme(s), or the default if none is specified. See ChallengeAsync. Authentication challenge examples include:

- A cookie authentication scheme redirecting the user to a login page.

- A JWT bearer scheme returning a 401 result with a `www-authenticate: bearer` header.

A challenge action should let the user know what authentication mechanism to use to access the requested resource.

Forbid

An authentication scheme's forbid action is called by Authorization when an authenticated user attempts to access a resource they're not permitted to access. See ForbidAsync. Authentication forbid examples include:

- A cookie authentication scheme redirecting the user to a page indicating access was forbidden.
- A JWT bearer scheme returning a 403 result.
- A custom authentication scheme redirecting to a page where the user can request access to the resource.

A forbid action can let the user know:

- They're authenticated.
- They're not permitted to access the requested resource.

See the following links for differences between challenge and forbid:

- Challenge and forbid with an operational resource handler.
- Differences between challenge and forbid.

Authentication providers per tenant

ASP.NET Core doesn't have a built-in solution for multi-tenant authentication. While it's possible for customers to write one using the built-in features, we recommend customers consider Orchard Core, ABP Framework, or Finbuckle.MultiTenant for multi-tenant authentication.

Orchard Core is:

- An open-source, modular, and multi-tenant app framework built with ASP.NET Core.
- A content management system (CMS) built on top of that app framework.

See the Orchard Core source for an example of authentication providers per tenant.

ABP Framework supports various architectural patterns including modularity, microservices, domain driven design, and multi-tenancy. See ABP Framework source on GitHub.

Finbuckle.MultiTenant:

- Open source
- Provides tenant resolution
- Lightweight
- Provides data isolation
- Configure app behavior uniquely for each tenant

Additional resources

- Authorize with a specific scheme in ASP.NET Core
- Policy schemes in ASP.NET Core
- Create an ASP.NET Core app with user data protected by authorization
- Globally require authenticated users
- GitHub issue on using multiple authentication schemes

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Choose an identity management solution

Article • 08/30/2023

Most web apps support authentication to ensure that users are who they claim to be. A *user* might be a person or another app. Management of access ensures users are only able to see and modify the information they're authorized to see and modify. For example, an end user shouldn't have access to the administrative section of a website. Identity management solutions are built to handle the requirements of authentication and authorization-related tasks. To learn more about identity management, see What is identity and access management?. Many identity management solutions for .NET web apps are available, each with different capabilities and requirements to use or install. This article provides guidance on how to choose the right solution.

Basic identity management with ASP.NET Core Identity

ASP.NET Core ships with a built-in authentication provider: ASP.NET Core Identity. The provider includes the APIs, UI, and backend database configuration to support managing user identities, storing user credentials, and granting or revoking permissions. Other features it supports include:

- External logins
- Two-factor authentication (2FA)
- Password management
- Account lockout and reactivation
- Authenticator apps

For most scenarios, this may be the only provider needed.

To learn more:

- Read the Introduction to Identity on ASP.NET Core
- Follow a tutorial to build your own secure .NET web app: Secure a .NET web app with the ASP.NET Core Identity framework.

In other scenarios, a server or service that manages authentication and identity may be beneficial.

Determine if an OIDC server is needed

Web apps require a way to *remember* past actions because the web, by default, is stateless. Otherwise, users would be forced to enter their credentials every time they navigated to a new page. The common solution for remembering state is *cookies*, a browser-based mechanism for storing data. The web server sends the initial cookie, then the browser stores it and sends it back with each request. This is done automatically without the need for the developer to write any code. Cookies are easy to use and built into the browser but are designed for use within a single website or web domain. The default solution that is built into ASP.NET Core uses cookie-based authentication.

Tokens are containers with metadata that are explicitly passed through the headers or body of HTTP requests. The main advantage of tokens over cookies is that they are not tied to a specific app or domain. Instead, tokens are usually *signed* with asymmetric cryptography. For example, OIDC servers issue tokens with information about identity using the JSON Web Token (JWT) format which includes signing. Asymmetric cryptography uses a combination of a private key known only to the signer, and a public key which everyone can know. Tokens may also be encrypted.

The signed token can't be tampered with due to the private key. The public key:

- Makes it possible to validate the token to ensure it hasn't been changed.
- Guarantees that it was generated by the entity holding the private key.

The main disadvantage to using tokens is that they require a service (typically an OIDC server) to both issue and provide validation for tokens. The service must be installed, configured, and maintained.

A common reason an OIDC server is required is for applications that expose web-based APIs that are consumed by other apps. For exposed web-based APIs, client UIs such as Single Page Applications (SPA), mobile clients, and desktop clients are considered to be part of the same app. SPA examples include Angular, React, and Blazor WebAssembly. If apps other than your web app or any client UIs must make a secure API call to your app, you'll likely want to use tokens. If you only have client UIs, ASP.NET Core Identity provides the option to acquire a token during authentication. The authentication token issued by ASP.NET Core Identity:

- Can be used by mobile and desktop clients. Cookies are preferred over tokens for both security and simplicity.
- Isn't suitable for managing access from third-party apps.

Another reason an OIDC server is required is for sharing sign-ins with other apps. Commonly referred to as *single sign on*, this feature enables users to:

- Sign in once with a web app's form.
- Use the resulting credentials to authenticate with other apps without having to sign-in again or choose a different password.

An OIDC server is typically preferred to provide a secure and scalable solution for single sign on.

For apps that don't share logins with other apps, the simplest way to quickly secure an app is to use the built-in ASP.NET Core Identity provider. Otherwise, an OIDC server provided by a third-party identity management solution is needed. OIDC servers are available as:

- Products you install on your server, called *self-host*.
- Containers run in a host like Docker.
- Web-based services you integrate with to manage identity.

Some solutions are free and open source, while others are commercially licensed. See identity management solutions for a list of available options. It's possible that your organization already uses an identity provider. In that case, it may make sense to use the existing provider instead of going with a different solution. All of the major solutions provide documentation for configuring ASP.NET Core to use their product or service.

Disconnected scenarios

Many solutions, such as Microsoft Entra ID, are cloud-based and require an Internet connection to work. If your environment doesn't allow Internet connectivity, you won't be able to use the service.

ASP.NET Core Identity works perfectly well in disconnected scenarios, such as:

- The app can't access the Internet.
- The app must still function on the local network even if the Internet is disconnected.

If you require a full OIDC server for a disconnected scenario, choose one of the following options:

- A solution that allows you to install and run the service on your own machines.
- Run the authentication service locally as a container.

Decide where user data such as sign-ins are stored

Another important factor to consider is where the user sign-in data is stored. Many developers choose external, cloud-based services like Microsoft Entra ID to manage identity. A cloud-based service provider:

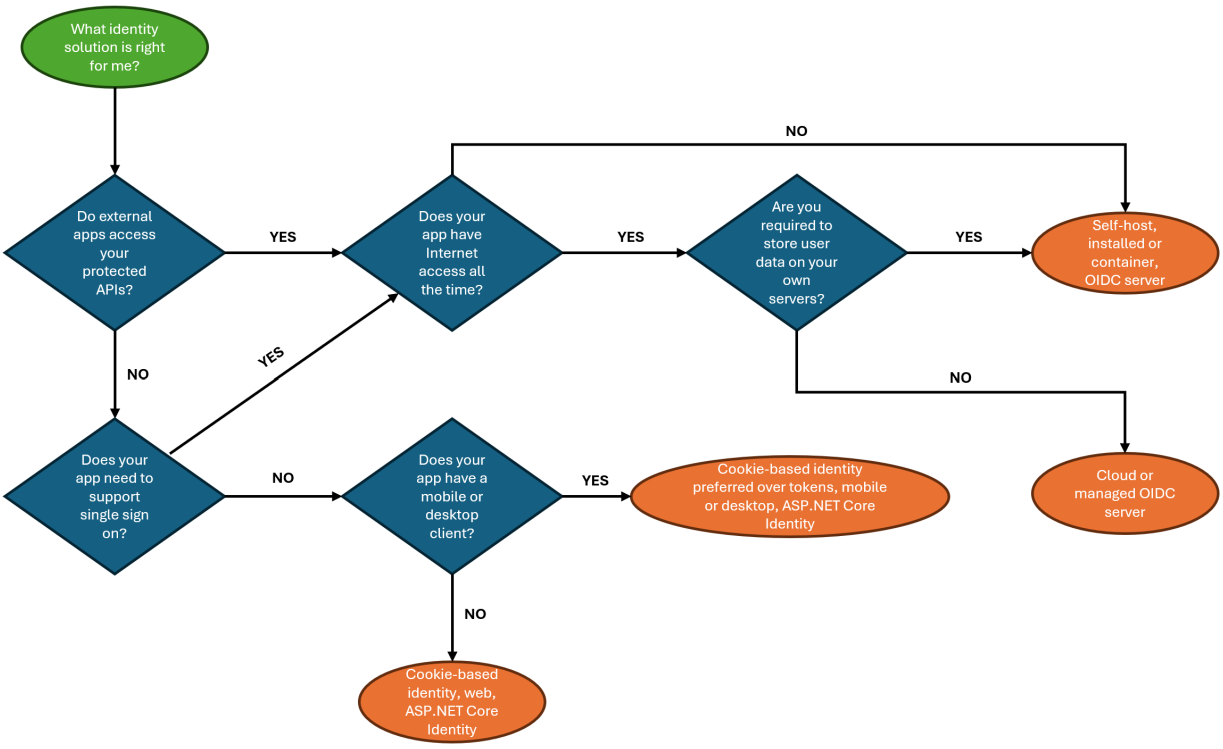
- Takes on the responsibilities of securely storing data.
- keeps the software up to date with the latest security patches and releases.
- Complies with privacy regulations.

Others prefer to store data on their own servers due to regulations, compliance, policy, or other reasons.

If the data is stored on your servers, you'll most likely need to choose an installable or container-based solution.

Identity vs OIDC server

Use the following diagram to help you decide whether to use the ASP.NET Core Identity system or an OIDC server for authentication and authorization:



The following table lists some of the things to consider when choosing your identity management solution.

Feature	Self-host (infrastructure or container)	Cloud
App integration	Local solutions that are implemented as libraries or frameworks can often be	Cloud-based solutions typically integrate at specific points in your

Feature	Self-host (infrastructure or container)	Cloud
	integrated directly in your own app. Container-based solutions require a hand-off to occur between your web app and the container-based service.	sign-in flow and provide configuration to update the UI to match your theme, but the level of customization available is limited.
Configuration	Self host solutions require configuring the software for the environment in addition to setting up how you want to manage identities. Container-based solutions typically provide a web-based UI for configuration.	Cloud-based solutions typically provide a web-based UI for configuration.
Customization	Self-host solutions are usually highly customizable, including code-based changes. Although containerized solutions provide extensibility options, they are often more limited.	Cloud-based services allow customization, but it's typically limited to configuration-based changes.
Maintenance	Installed products require a dedicated resource to ensure all security patches are applied in a timely fashion and to manage upgrades. The upgrade and patch process for containers is usually lower-friction and involves simply installing the provided container image.	The service provider maintains their cloud-based solution, including applying needed patches and handling upgrades.
User credentials storage	You are responsible for data governance and handling breaches.	Managing the risks associated with handling user credentials, and complying with regulations. is delegated to the service provider.

For more information about available options, see Identity management solutions for ASP.NET Core.

Next steps

- Learn about JSON Web Tokens
- Browse sample apps with authentication/authorization and identity for ASP.NET Core.
- Follow a tutorial to secure a .NET web app using built-in ASP.NET Core Identity.
- Learn more about how to protect web APIs.

Introduction to Identity on ASP.NET Core

Article • 04/26/2024

By Rick Anderson

ASP.NET Core Identity:

- Is an API that supports user interface (UI) login functionality.
- Manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.

Users can create an account with the login information stored in Identity or they can use an external login provider. Supported external login providers include Facebook, Google, Microsoft Account, and Twitter.

For information on how to globally require all users to be authenticated, see Require authenticated users.

The Identity source code is available on GitHub. Scaffold Identity and view the generated files to review the template interaction with Identity.

Identity is typically configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.

In this topic, you learn how to use Identity to register, log in, and log out a user. Note: the templates treat username and email as the same for users. For more detailed instructions about creating apps that use Identity, see Next Steps.

ASP.NET Core Identity isn't related to the Microsoft identity platform. Microsoft identity platform is:

- An evolution of the Azure Active Directory (Azure AD) developer platform.
- An alternative identity solution for authentication and authorization in ASP.NET Core apps.

ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps. To secure web APIs and SPAs, use one of the following:

- Microsoft Entra ID
- Azure Active Directory B2C (Azure AD B2C)
- Duende Identity Server

Duende Identity Server is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. Duende Identity Server enables the following security features:

- Authentication as a Service (AaaS)
- Single sign-on/off (SSO) over multiple application types
- Access control for APIs
- Federation Gateway

> **Important**
>
> Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

For more information, see the Duende Identity Server documentation (Duende Software website).

View or download the sample code (how to download).

Create a Web app with authentication

Create an ASP.NET Core Web Application project with Individual User Accounts.

Visual Studio

- Select the **ASP.NET Core Web App** template. Name the project **WebApp1** to have the same namespace as the project download. Click **OK**.
- In the **Authentication type** input, select **Individual User Accounts**.

The generated project provides ASP.NET Core Identity as a Razor class library. The Identity Razor class library exposes endpoints with the `Identity` area. For example:

- /Identity/Account/Login
- /Identity/Account/Logout
- /Identity/Account/Manage

Apply migrations

Apply the migrations to initialize the database.

Visual Studio

Run the following command in the Package Manager Console (PMC):

```
Update-Database
```

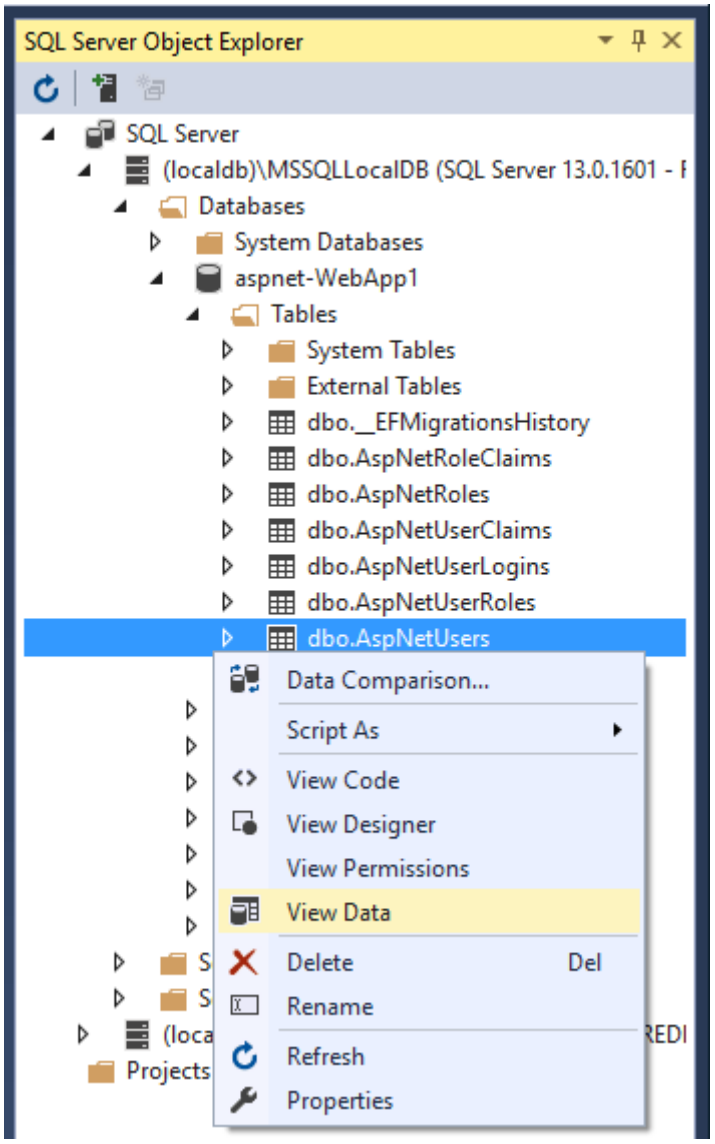
Test Register and Login

Run the app and register a user. Depending on your screen size, you might need to select the navigation toggle button to see the **Register** and **Login** links.

View the Identity database

Visual Studio

- From the **View** menu, select **SQL Server Object Explorer** (SSOX).
- Navigate to **(localdb)MSSQLLocalDB(SQL Server 13)**. Right-click on **dbo.AspNetUsers** > **View Data**:



Configure Identity services

Services are added in `Program.cs`. The typical pattern is to call methods in the following order:

1. `Add{Service}`
2. `builder.Services.Configure{Service}`

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebApp1.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
```

```
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.Configure<IdentityOptions>(options =>
{
    // Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;

    // Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;

    // User settings.
    options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;
});

builder.Services.ConfigureApplicationCookie(options =>
{
    // Cookie settings
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(5);

    options.LoginPath = "/Identity/Account/Login";
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.SlidingExpiration = true;
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

```
app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding code configures Identity with default option values. Services are made available to the app through dependency injection.

Identity is enabled by calling UseAuthentication. `UseAuthentication` adds authentication middleware to the request pipeline.

The template-generated app doesn't use authorization. `app.UseAuthorization` is included to ensure it's added in the correct order should the app add authorization. `UseRouting`, `UseAuthentication`, and `UseAuthorization` must be called in the order shown in the preceding code.

For more information on `IdentityOptions`, see IdentityOptions and Application Startup.

Scaffold Register, Login, LogOut, and RegisterConfirmation

Visual Studio

Add the `Register`, `Login`, `LogOut`, and `RegisterConfirmation` files. Follow the Scaffold identity into a Razor project with authorization instructions to generate the code shown in this section.

Examine Register

When a user clicks the **Register** button on the `Register` page, the `RegisterModel.OnPostAsync` action is invoked. The user is created by CreateAsync(TUser) on the `_userManager` object:

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    ExternalLogins = (await
```

```
_signInManager.GetExternalAuthenticationSchemesAsync())
                                            .ToList();
    if (ModelState.IsValid)
    {
        var user = new IdentityUser { UserName = Input.Email, Email =
Input.Email };
        var result = await _userManager.CreateAsync(user, Input.Password);
        if (result.Succeeded)
        {
            _logger.LogInformation("User created a new account with
password.");

            var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
            code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            var callbackUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = user.Id, code =
code },
                protocol: Request.Scheme);

            await _emailSender.SendEmailAsync(Input.Email, "Confirm your
email",
                $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

            if (_userManager.Options.SignIn.RequireConfirmedAccount)
            {
                return RedirectToPage("RegisterConfirmation",
                                      new { email = Input.Email });
            }
            else
            {
                await _signInManager.SignInAsync(user, isPersistent: false);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

Disable default account verification

With the default templates, the user is redirected to the `Account.RegisterConfirmation` where they can select a link to have the account confirmed. The default `Account.RegisterConfirmation` is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set `DisplayConfirmAccountLink = false` in `/Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs`:

C#

```
[AllowAnonymous]
public class RegisterConfirmationModel : PageModel
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly IEmailSender _sender;

    public RegisterConfirmationModel(UserManager<IdentityUser> userManager,
IEmailSender sender)
    {
        _userManager = userManager;
        _sender = sender;
    }

    public string Email { get; set; }

    public bool DisplayConfirmAccountLink { get; set; }

    public string EmailConfirmationUrl { get; set; }

    public async Task<IActionResult> OnGetAsync(string email, string
returnUrl = null)
    {
        if (email == null)
        {
            return RedirectToPage("/Index");
        }

        var user = await _userManager.FindByEmailAsync(email);
        if (user == null)
        {
            return NotFound($"Unable to load user with email '{email}'.");
        }

        Email = email;
        // Once you add a real email sender, you should remove this code
that lets you confirm the account
        DisplayConfirmAccountLink = false;
        if (DisplayConfirmAccountLink)
        {
            var userId = await _userManager.GetUserIdAsync(user);
            var code = await
```

```
_userManager.GenerateEmailConfirmationTokenAsync(user);
            code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
            EmailConfirmationUrl = Url.Page(
                "/Account/ConfirmEmail",
                pageHandler: null,
                values: new { area = "Identity", userId = userId, code =
code, returnUrl = returnUrl },
                protocol: Request.Scheme);
        }

        return Page();
    }
}
```

Log in

The Login form is displayed when:

- The **Log in** link is selected.
- A user attempts to access a restricted page that they aren't authorized to access **or** when they haven't been authenticated by the system.

When the form on the Login page is submitted, the `OnPostAsync` action is called. `PasswordSignInAsync` is called on the `_signInManager` object.

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
                            Input.Password, Input.RememberMe,
lockoutOnFailure: true);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new
            {
                ReturnUrl = returnUrl,
```

```
                RememberMe = Input.RememberMe
            });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login
attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

For information on how to make authorization decisions, see Introduction to authorization in ASP.NET Core.

Log out

The **Log out** link invokes the `LogoutModel.OnPost` action.

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;
using System.Threading.Tasks;

namespace WebApp1.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class LogoutModel : PageModel
    {
        private readonly SignInManager<IdentityUser> _signInManager;
        private readonly ILogger<LogoutModel> _logger;

        public LogoutModel(SignInManager<IdentityUser> signInManager,
ILogger<LogoutModel> logger)
        {
            _signInManager = signInManager;
            _logger = logger;
        }
```

```
        public void OnGet()
        {
        }

        public async Task<IActionResult> OnPost(string returnUrl = null)
        {
            await _signInManager.SignOutAsync();
            _logger.LogInformation("User logged out.");
            if (returnUrl != null)
            {
                return LocalRedirect(returnUrl);
            }
            else
            {
                return RedirectToPage();
            }
        }
    }
}
```

In the preceding code, the code `return RedirectToPage();` needs to be a redirect so that the browser performs a new request and the identity for the user gets updated.

SignOutAsync clears the user's claims stored in a cookie.

Post is specified in the `Pages/Shared/_LoginPartial.cshtml`:

CSHTML

```
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager

<ul class="navbar-nav">
@if (SignInManager.IsSignedIn(User))
{
    <li class="nav-item">
        <a  class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Manage/Index"
                                                title="Manage">Hello
@User.Identity.Name!</a>
    </li>
    <li class="nav-item">
        <form class="form-inline" asp-area="Identity" asp-
page="/Account/Logout"
                                    asp-route-returnUrl="@Url.Page("/", new {
area = "" })"
                                    method="post" >
            <button  type="submit" class="nav-link btn btn-link text-
dark">Logout</button>
        </form>
    </li>
```

```
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Register">Register</a>
    </li>
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Login">Login</a>
    </li>
}
</ul>
```

Test Identity

The default web project templates allow anonymous access to the home pages. To test Identity, add [Authorize]:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.Extensions.Logging;

namespace WebApp1.Pages
{
    [Authorize]
    public class PrivacyModel : PageModel
    {
        private readonly ILogger<PrivacyModel> _logger;

        public PrivacyModel(ILogger<PrivacyModel> logger)
        {
            _logger = logger;
        }

        public void OnGet()
        {
        }
    }
}
```

If you are signed in, sign out. Run the app and select the **Privacy** link. You are redirected to the login page.

Explore Identity

To explore Identity in more detail:

- Create full identity UI source
- Examine the source of each page and step through the debugger.

Identity Components

All the Identity-dependent NuGet packages are included in the ASP.NET Core shared framework.

The primary package for Identity is Microsoft.AspNetCore.Identity. This package contains the core set of interfaces for ASP.NET Core Identity, and is included by `Microsoft.AspNetCore.Identity.EntityFrameworkCore`.

Migrating to ASP.NET Core Identity

For more information and guidance on migrating your existing Identity store, see Migrate Authentication and Identity.

Setting password strength

See Configuration for a sample that sets the minimum password requirements.

AddDefaultIdentity and AddIdentity

AddDefaultIdentity was introduced in ASP.NET Core 2.1. Calling `AddDefaultIdentity` is similar to calling the following:

- AddIdentity
- AddDefaultUI
- AddDefaultTokenProviders

See AddDefaultIdentity source for more information.

Prevent publish of static Identity assets

To prevent publishing static Identity assets (stylesheets and JavaScript files for Identity UI) to the web root, add the following `ResolveStaticWebAssetsInputsDependsOn` property and `RemoveIdentityAssets` target to the app's project file:

XML

```xml
<PropertyGroup>

<ResolveStaticWebAssetsInputsDependsOn>RemoveIdentityAssets</ResolveStaticWebAssetsInputsDependsOn>
</PropertyGroup>

<Target Name="RemoveIdentityAssets">
  <ItemGroup>
    <StaticWebAsset Remove="@(StaticWebAsset)" Condition="%(SourceId) == 'Microsoft.AspNetCore.Identity.UI'" />
  </ItemGroup>
</Target>
```

Next Steps

- ASP.NET Core Identity source code
- How to work with Roles in ASP.NET Core Identity

- See this GitHub issue for information on configuring Identity using SQLite.
- Configure Identity
- Create an ASP.NET Core app with user data protected by authorization
- Add, download, and delete user data to Identity in an ASP.NET Core project
- Enable QR code generation for TOTP authenticator apps in ASP.NET Core
- Migrate Authentication and Identity to ASP.NET Core
- Account confirmation and password recovery in ASP.NET Core
- Two-factor authentication with SMS in ASP.NET Core
- Host ASP.NET Core in a web farm

How to use Identity to secure a Web API backend for SPAs

Article • 09/10/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

ASP.NET Core Identity provides APIs that handle authentication, authorization, and identity management. The APIs make it possible to secure endpoints of a Web API backend with cookie-based authentication. A token-based option is available for clients that can't use cookies, but in using this you are responsible for ensuring the tokens are kept secure. We recommend using cookies for browser-based applications, because, by default, the browser automatically handles them without exposing them to JavaScript.

This article shows how to use Identity to secure a Web API backend for SPAs such as Angular, React, and Vue apps. The same backend APIs can be used to secure Blazor WebAssembly apps.

Prerequisites

The steps shown in this article add authentication and authorization to an ASP.NET Core Web API app that:

- Isn't already configured for authentication.
- Targets `net8.0` or later.
- Can be either minimal API or controller-based API.

Some of the testing instructions in this article use the Swagger UI that's included with the project template. The Swagger UI isn't required to use Identity with a Web API backend.

Install NuGet packages

Install the following NuGet packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore - Enables Identity to work with Entity Framework Core (EF Core).

- One that enables EF Core to work with a database, such as one of the following packages:
  - Microsoft.EntityFrameworkCore.SqlServer or
  - Microsoft.EntityFrameworkCore.Sqlite or
  - Microsoft.EntityFrameworkCore.InMemory.

For the quickest way to get started, use the in-memory database.

Change the database later to SQLite or SQL Server to save user data between sessions when testing or for production use. That introduces some complexity compared to in-memory, as it requires the database to be created through migrations, as shown in the EF Core getting started tutorial.

Install these packages by using the NuGet package manager in Visual Studio or the dotnet add package CLI command.

Create an `IdentityDbContext`

Add a class named `ApplicationDbContext` that inherits from IdentityDbContext<TUser>:

C#

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext : IdentityDbContext<IdentityUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options) :
        base(options)
    { }
}
```

The code shown provides a special constructor that makes it possible to configure the database for different environments.

Add one or more of the following `using` directives as needed when adding the code shown in these steps.

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
```

Configure the EF Core context

As noted earlier, the simplest way to get started is to use the in-memory database. With in-memory each run starts with a fresh database, and there's no need to use migrations. After the call to `WebApplication.CreateBuilder(args)`, add the following code to configure Identity to use an in-memory database:

```
C#

builder.Services.AddDbContext<ApplicationDbContext>(
    options => options.UseInMemoryDatabase("AppDb"));
```

To save user data between sessions when testing or for production use, change the database later to SQLite or SQL Server.

Add Identity services to the container

After the call to `WebApplication.CreateBuilder(args)`, call AddAuthorization to add services to the dependency injection (DI) container:

```
C#

builder.Services.AddAuthorization();
```

Activate Identity APIs

After the call to `WebApplication.CreateBuilder(args)`, call AddIdentityApiEndpoints<TUser>(IServiceCollection) and AddEntityFrameworkStores<TContext>(IdentityBuilder).

```
C#

builder.Services.AddIdentityApiEndpoints<IdentityUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

By default, both cookies and proprietary tokens are activated. Cookies and tokens are issued at login if the `useCookies` query string parameter in the login endpoint is `true`.

Map Identity routes

After the call to `builder.Build()`, call MapIdentityApi<TUser>(IEndpointRouteBuilder) to map the Identity endpoints:

```
C#

app.MapIdentityApi<IdentityUser>();
```

Secure selected endpoints

To secure an endpoint, use the RequireAuthorization extension method on the `Map{Method}` call that defines the route. For example:

```
C#

app.MapGet("/weatherforecast", (HttpContext httpContext) =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = summaries[Random.Shared.Next(summaries.Length)]
        })
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast")
.WithOpenApi()
.RequireAuthorization();
```

The `RequireAuthorization` method can also be used to:

- Secure Swagger UI endpoints, as shown in the following example:

  ```
  C#

  app.MapSwagger().RequireAuthorization();
  ```

- Secure with a specific claim or permission, as shown in the following example:

  ```
  C#

  .RequireAuthorization("Admin");
  ```

In a controller-based web API project, secure endpoints by applying the [Authorize] attribute to a controller or action.

Test the API

A quick way to test authentication is to use the in-memory database and the Swagger UI that's included with the project template. The following steps show how to test the API with the Swagger UI. Make sure that the Swagger UI endpoints aren't secured.

Attempt to access a secured endpoint

- Run the app and navigate to the Swagger UI.
- Expand a secured endpoint, such as `/weatherforecast` in a project created by the web API template.
- Select **Try it out**.
- Select **Execute**. The response is `401 - not authorized`.

Test registration

- Expand `/register` and select **Try it out**.
- In the **Parameters** section of the UI, a sample request body is shown:

  JSON

  ```
  {
    "email": "string",
    "password": "string"
  }
  ```

- Replace "string" with a valid email address and password, and then select **Execute**.

  To comply with the default password validation rules, the password must be at least six characters long and contain at least one of each of the following characters:
  - Uppercase letter
  - Lowercase letter
  - Numeric digit
  - Nonalphanumeric character

  If you enter an invalid email address or a bad password, the result includes the validation errors. Here's an example of a response body with validation errors:

JSON

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "PasswordTooShort": [
      "Passwords must be at least 6 characters."
    ],
    "PasswordRequiresNonAlphanumeric": [
      "Passwords must have at least one non alphanumeric character."
    ],
    "PasswordRequiresDigit": [
      "Passwords must have at least one digit ('0'-'9')."
    ],
    "PasswordRequiresLower": [
      "Passwords must have at least one lowercase ('a'-'z')."
    ]
  }
}
```

The errors are returned in the ProblemDetails format so the client can parse them and display validation errors as needed.

A successful registration results in a `200 - OK` response.

Test login

- Expand `/login` and select **Try it out**. The example request body shows two additional parameters:

  JSON

  ```
  {
    "email": "string",
    "password": "string",
    "twoFactorCode": "string",
    "twoFactorRecoveryCode": "string"
  }
  ```

  The extra JSON properties aren't needed for this example and can be deleted. Set `useCookies` to `true`.

- Replace "string" with the email address and password that you used to register, and then select **Execute**.

A successful login results in a `200 - OK` response with a cookie in the response header.

Retest the secured endpoint

After a successful login, rerun the secured endpoint. The authentication cookie is automatically sent with the request, and the endpoint is authorized. Cookie-based authentication is securely built-in to the browser and "just works."

Testing with nonbrowser clients

Some web clients might not include cookies in the header by default:

- If you're using a tool for testing APIs, you might need to enable cookies in the settings.
- The JavaScript `fetch` API doesn't include cookies by default. Enable them by setting `credentials` to the value `include` in the options.
- An `HttpClient` running in a Blazor WebAssembly app needs the `HttpRequestMessage` to include credentials, like the following example:

```
C#

request.SetBrowserRequestCredential(BrowserRequestCredentials.Include);
```

Use token-based authentication

We recommend using cookies in browser-based applications, because, by default, the browser automatically handles them without exposing them to JavaScript.

A custom token (one that is proprietary to the ASP.NET Core identity platform) is issued that can be used to authenticate subsequent requests. The token is passed in the `Authorization` header as a bearer token. A refresh token is also provided. This token allows the application to request a new token when the old one expires without forcing the user to log in again.

The tokens aren't standard JSON Web Tokens (JWTs). The use of custom tokens is intentional, as the built-in Identity API is meant primarily for simple scenarios. The token option isn't intended to be a full-featured identity service provider or token server, but instead an alternative to the cookie option for clients that can't use cookies.

To use token-based authentication, set the `useCookies` query string parameter to `false` when calling the `/login` endpoint. Tokens use the *bearer* authentication scheme. Using the token returned from the call to `/login`, subsequent calls to protected endpoints should add the header `Authorization: Bearer <token>` where `<token>` is the access token. For more information, see Use the POST /login endpoint later in this article.

Log out

To provide a way for the user to log out, define a `/logout` endpoint like the following example:

C#

```
app.MapPost("/logout", async (SignInManager<IdentityUser> signInManager,
    [FromBody] object empty) =>
{
    if (empty != null)
    {
        await signInManager.SignOutAsync();
        return Results.Ok();
    }
    return Results.Unauthorized();
})
.WithOpenApi()
.RequireAuthorization();
```

Provide an empty JSON object (`{}`) in the request body when calling this endpoint. The following code is an example of a call to the logout endpoint:

TypeScript

```
public signOut() {
  return this.http.post('/logout', {}, {
    withCredentials: true,
    observe: 'response',
    responseType: 'text'
```

The `MapIdentityApi<TUser>` endpoints

The call to `MapIdentityApi<TUser>` adds the following endpoints to the app:

- POST /register
- POST /login
- POST /refresh

- GET /confirmEmail
- POST /resendConfirmationEmail
- POST /forgotPassword
- POST /resetPassword
- POST /manage/2fa
- GET /manage/info
- POST /manage/info

Use the `POST /register` endpoint

The request body must have Email and Password properties:

JSON

```
{
  "email": "string",
  "password": "string",
}
```

For more information, see:

- Test registration earlier in this article.
- RegisterRequest.

Use the `POST /login` endpoint

In the request body, Email and Password are required. If two-factor authentication (2FA) is enabled, either TwoFactorCode or TwoFactorRecoveryCode is required. If 2FA isn't enabled, omit both `twoFactorCode` and `twoFactorRecoveryCode`. For more information, see Use the POST /manage/2fa endpoint later in this article.

Here's a request body example with 2FA not enabled:

JSON

```
{
  "email": "string",
  "password": "string"
}
```

Here are request body examples with 2FA enabled:

- JSON

```
{
  "email": "string",
  "password": "string",
  "twoFactorCode": "string",
}
```

- JSON

```
{
  "email": "string",
  "password": "string",
  "twoFactorRecoveryCode": "string"
}
```

The endpoint expects a query string parameter:

- `useCookies` - Set to `true` for cookie-based authentication. Set to `false` or omit for token-based authentication.

For more information about cookie-based authentication, see Test login earlier in this article.

Token-based authentication

If `useCookies` is `false` or omitted, token-based authentication is enabled. The response body includes the following properties:

JSON

```
{
  "tokenType": "string",
  "accessToken": "string",
  "expiresIn": 0,
  "refreshToken": "string"
}
```

For more information about these properties, see AccessTokenResponse.

Put the access token in a header to make authenticated requests, as shown in the following example

HTTP

```
Authorization: Bearer {access token}
```

When the access token is about to expire, call the /refresh endpoint.

Use the `POST /refresh` endpoint

For use only with token-based authentication. Gets a new access token without forcing the user to log in again. Call this endpoint when the access token is about to expire.

The request body contains only the RefreshToken. Here's a request body example:

JSON

```
{
  "refreshToken": "string"
}
```

If the call is successful, the response body is a new AccessTokenResponse, as shown in the following example:

JSON

```
{
  "tokenType": "string",
  "accessToken": "string",
  "expiresIn": 0,
  "refreshToken": "string"
}
```

Use the `GET /confirmEmail` endpoint

If Identity is set up for email confirmation, a successful call to the `/register` endpoint sends an email that contains a link to the `/confirmEmail` endpoint. The link contains the following query string parameters:

- `userId`
- `code`
- `changedEmail` - Included only if the user changed the email address during registration.

Identity provides default text for the confirmation email. By default, the email subject is "Confirm your email" and the email body looks like the following example:

HTTP

```
 Please confirm your account by <a href='https://contoso.com/confirmEmail?
userId={user ID}&code={generated code}&changedEmail={new email
address}'>clicking here</a>.
```

If the RequireConfirmedEmail property is set to `true`, the user can't log in until the email address is confirmed by clicking the link in the email. The `/confirmEmail` endpoint:

- Confirms the email address and enables the user to log in.
- Returns the text "Thank you for confirming your email." in the response body.

To set up Identity for email confirmation, add code in `Program.cs` to set `RequireConfirmedEmail` to `true` and add a class that implements IEmailSender to the DI container. For example:

C#

```
builder.Services.Configure<IdentityOptions>(options =>
{
    options.SignIn.RequireConfirmedEmail = true;
});

builder.Services.AddTransient<IEmailSender, EmailSender>();
```

For more information, see Account confirmation and password recovery in ASP.NET Core.

Identity provides default text for the other emails that need to be sent as well, such as for 2FA and password reset. To customize these emails, provide a custom implementation of the `IEmailSender` interface. In the preceding example, `EmailSender` is a class that implements `IEmailSender`. For more information, including an example of a class that implements `IEmailSender`, see Account confirmation and password recovery in ASP.NET Core.

Use the `POST /resendConfirmationEmail` endpoint

Sends an email only if the address is valid for a registered user.

The request body contains only the Email. Here's a request body example:

JSON

```
{
  "email": "string"
}
```

For more information, see Use the GET /confirmEmail endpoint earlier in this article.

Use the `POST /forgotPassword` endpoint

Generates an email that contains a password reset code. Send that code to /resetPassword with a new password.

The request body contains only the Email. Here's an example:

JSON

```
{
  "email": "string"
}
```

For information about how to enable Identity to send emails, see Use the GET /confirmEmail endpoint.

Use the `POST /resetPassword` endpoint

Call this endpoint after getting a reset code by calling the `/forgotPassword` endpoint.

The request body requires Email, ResetCode, and NewPassword. Here's an example:

JSON

```
{
  "email": "string",
  "resetCode": "string",
  "newPassword": "string"
}
```

Use the `POST /manage/2fa` endpoint

Configures two-factor authentication (2FA) for the user. When 2FA is enabled, successful login requires a code produced by an authenticator app in addition to the email address and password.

Enable 2FA

To enable 2FA for the currently authenticated user:

- Call the `/manage/2fa` endpoint, sending an empty JSON object (`{}`) in the request body.
- The response body provides the SharedKey along with some other properties that aren't needed at this point. The shared key is used to set up the authenticator app. Response body example:

  JSON

  ```
  {
    "sharedKey": "string",
    "recoveryCodesLeft": 0,
    "recoveryCodes": null,
    "isTwoFactorEnabled": false,
    "isMachineRemembered": false
  }
  ```

- Use the shared key to get a Time-based one-time password (TOTP). For more information, see Enable QR code generation for TOTP authenticator apps in ASP.NET Core.
- Call the `/manage/2fa` endpoint, sending the TOTP and `"enable": true` in the request body. For example:

  JSON

  ```
  {
    "enable": true,
    "twoFactorCode": "string"
  }
  ```

- The response body confirms that IsTwoFactorEnabled is true and provides the RecoveryCodes. The recovery codes are used to log in when the authenticator app isn't available. Response body example after successfully enabling 2FA:

  JSON

  ```
  {
    "sharedKey": "string",
    "recoveryCodesLeft": 10,
    "recoveryCodes": [
      "string",
      "string",
  ```

```
      "string",
      "string",
      "string",
      "string",
      "string",
      "string",
      "string",
      "string"
    ],
    "isTwoFactorEnabled": true,
    "isMachineRemembered": false
  }
```

Log in with 2FA

Call the `/login` endpoint, sending the email address, password, and TOTP in the request body. For example:

JSON

```
{
  "email": "string",
  "password": "string",
  "twoFactorCode": "string"
}
```

If the user doesn't have access to the authenticator app, log in by calling the `/login` endpoint with one of the recovery codes that was provided when 2FA was enabled. The request body looks like the following example:

JSON

```
{
  "email": "string",
  "password": "string",
  "twoFactorRecoveryCode": "string"
}
```

Reset the recovery codes

To get a new collection of recovery codes, call this endpoint with ResetRecoveryCodes set to `true`. Here's a request body example:

JSON

```
{
  "resetRecoveryCodes": true
}
```

Reset the shared key

To get a new random shared key, call this endpoint with ResetSharedKey set to `true`. Here's a request body example:

JSON

```
{
  "resetSharedKey": true
}
```

Resetting the key automatically disables the two-factor login requirement for the authenticated user until it's re-enabled by a later request.

Forget the machine

To clear the cookie "remember me flag" if present, call this endpoint with ForgetMachine set to true. Here's a request body example:

JSON

```
{
  "forgetMachine": true
}
```

This endpoint has no impact on token-based authentication.

Use the `GET /manage/info` endpoint

Gets email address and email confirmation status of the logged-in user. Claims were omitted from this endpoint for security reasons. If claims are needed, use the server-side APIs to set up an endpoint for claims. Or instead of sharing all of the users' claims, provide a validation endpoint that accepts a claim and responds whether the user has it.

The request doesn't require any parameters. The response body includes the Email and IsEmailConfirmed properties, as in the following example:

JSON

```
{
  "email": "string",
  "isEmailConfirmed": true
}
```

Use the `POST /manage/info` endpoint

Updates the email address and password of the logged-in user. Send NewEmail, NewPassword, and OldPassword in the request body, as shown in the following example:

JSON

```
{
  "newEmail": "string",
  "newPassword": "string",
  "oldPassword": "string"
}
```

Here's an example of the response body:

JSON

```
{
  "email": "string",
  "isEmailConfirmed": false
}
```

See also

For more information, see the following resources:

- Choose an identity management solution
- Identity management solutions for .NET web apps
- Simple authorization in ASP.NET Core
- Add, download, and delete user data to Identity in an ASP.NET Core project
- Create an ASP.NET Core app with user data protected by authorization
- Account confirmation and password recovery in ASP.NET Core
- Enable QR code generation for TOTP authenticator apps in ASP.NET Core
- Sample Web API backend for SPAs The .http file shows token-based authentication. For example:
  - Doesn't set `useCookies`

    - Uses the Authorization header to pass the token
    - Shows refresh to extend session without forcing the user to login again
- Sample Angular app that uses Identity to secure a Web API backend

Scaffold Identity in ASP.NET Core projects

Article • 07/23/2024

By Rick Anderson

Blazor Identity scaffolding

ASP.NET Core Identity scaffolding adds ASP.NET Core Identity to Blazor Web Apps and Blazor Server apps. After the scaffolder adds the Identity Razor components to the app, you can customize the components to suit your app's requirements.

Although the scaffolder generates the necessary C# code to scaffold Identity into the app, you must update the project's database with an Entity Framework (EF) Core database migration to complete the process. This article explains the steps required to migrate a database.

Inspect the changes after running the Identity scaffolder. We recommend using GitHub or another source control system that shows file changes with a revert changes feature.

Services are required when using two-factor authentication (2FA), account confirmation and password recovery, and other security features with Identity. Services or service stubs aren't generated when scaffolding Identity. Services to enable these features must be added manually.

Razor Pages and MVC Identity scaffolding

ASP.NET Core provides ASP.NET Core Identity as a Razor class library (RCL). Applications that include Identity can apply the scaffolder to selectively add the source code contained in the Identity RCL. You might want to generate source code so you can modify the code and change the behavior. For example, you could instruct the scaffolder to generate the code used in registration. Customized Identity code overrides the default implementation provided by the Identity RCL. To gain full control of the UI and not use the default RCL, see the Create full Identity UI source section.

Applications that do **not** include authentication can apply the scaffolder to add the RCL Identity package. You have the option of selecting Identity code to be generated.

Although the scaffolder generates most of the necessary code, you need to update your project to complete the process. This document explains the steps needed to complete

an Identity scaffolding update.

We recommend using a source control system that shows file differences and allows you to back out of changes. Inspect the changes after running the Identity scaffolder.

Services are required when using Two Factor Authentication, Account confirmation and password recovery, and other security features with Identity. Services or service stubs aren't generated when scaffolding Identity. Services to enable these features must be added manually. For example, see Require Email Confirmation.

Typically, apps created with individual accounts should ***not*** create a new data context.

Scaffold Identity into a Blazor project

This section applies to Blazor Web Apps and Blazor Server apps.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Blazor Identity** in the center pane. Select the **Add** button.
- In the **Add Blazor Identity** dialog:
  - Select or add with the plus (+) button the database context class (**DbContext class**).
  - Select the database provider (**Database provider**), which defaults to SQL Server.
  - Select or add with the plus (+) button the user class (**User class**).
  - Select the **Add** button.

The generated Identity database code requires EF Core Migrations. The following steps explain how to create and apply a migration to the database.

Visual Studio

Visual Studio Connected Services are used to add an EF Core migration and update the database.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name**, such as `CreateIdentitySchema`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration.

Select the **Close** button after the operation finishes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_{MIGRATION NAME}.cs` file, which creates the Identity tables, constraints, and indexes. The `{TIME STAMP}` placeholder is a time stamp, and the `{MIGRATION NAME}` placeholder is the migration name.

Client-side Blazor apps (Standalone Blazor WebAssembly)

Client-side Blazor apps (Standalone Blazor WebAssembly) use their own Identity UI approaches and can't use ASP.NET Core Identity scaffolding.

For more information, see the Blazor Security and Identity articles.

Scaffold Identity into a Razor project without existing authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

> **Note**

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the + button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the + button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Migrations, UseAuthentication, and layout

The generated Identity database code requires Entity Framework (EF) Core Migrations. If a migration to generate the Identity schema hasn't been created and applied to the database, create a migration and update the database.

Visual Studio

Visual Studio Connected Services are used to add an EF Core migration and update the database.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name**, such as `CreateIdentitySchema`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration.

Select the **Close** button after the operation finishes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_{MIGRATION NAME}.cs` file, which creates the Identity tables, constraints, and indexes. The `{TIME STAMP}` placeholder is a time stamp, and the `{MIGRATION NAME}` placeholder is the migration name.

If the Identity schema has already been created but not applied to the database, only the command to update the database must be executed:

Visual Studio

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

You can confirm the application of an Identity schema with the following command. The output of the command includes an "`applied`" column to show which migrations are applied to the database.

Visual Studio

In the Visual Studio **Package Manager Console**, execute Get-Migration:

PowerShell

```
Get-Migration
```

If more than one database context exists, specify the context with the `-Context` parameter.

Layout changes

Optional: Add the login partial (`_LoginPartial`) to the layout file:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebRPnoAuth2Auth</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/WebRPnoAuth2Auth.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">WebRPnoAuth2Auth</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
```

```
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Privacy">Privacy</a>
                        </li>
                    </ul>
                    <partial name="_LoginPartial" />
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - WebRPnoAuth2Auth - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

Scaffold Identity into a Razor project with authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation).

Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the + button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the + button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Scaffold Identity into an MVC project without existing authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the + button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the + button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Optional: Add the login partial (`_LoginPartial`) to the `Views/Shared/_Layout.cshtml` file:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - WebRPnoAuth2Auth</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/WebRPnoAuth2Auth.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">WebRPnoAuth2Auth</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
page="/Privacy">Privacy</a>
                        </li>
                    </ul>
                    <partial name="_LoginPartial" />
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2021 - WebRPnoAuth2Auth - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
```

```
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

The generated Identity database code requires Entity Framework (EF) Core Migrations. If a migration to generate the Identity schema hasn't been created and applied to the database, create a migration and update the database.

Visual Studio

Visual Studio Connected Services are used to add an EF Core migration and update the database.

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by **Add migration**.

Give the migration a **Migration name**, such as `CreateIdentitySchema`, which is a name that describes the migration. Wait for the database context to load in the **DbContext class names** field, which may take a few seconds. Select **Finish** to create the migration.

Select the **Close** button after the operation finishes.

Select the ellipsis (`...`) again followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

The update database command executes the `Up` method migrations that haven't been applied in a migration code file created by the scaffolder. In this case, the command executes the `Up` method in the `Migrations/{TIME STAMP}_{MIGRATION NAME}.cs` file, which creates the Identity tables, constraints, and indexes. The `{TIME STAMP}` placeholder is a time stamp, and the `{MIGRATION NAME}` placeholder is the migration name.

If the Identity schema has already been created but not applied to the database, only the command to update the database must be executed:

Visual Studio

In **Solution Explorer**, double-click **Connected Services**. In the **SQL Server Express LocalDB** area of **Service Dependencies**, select the ellipsis (`...`) followed by the **Update database** command.

The **Update database with the latest migration** dialog opens. Wait for the **DbContext class names** field to update and for prior migrations to load, which may take a few seconds. Select the **Finish** button.

Select the **Close** button after the operation finishes.

You can confirm the application of an Identity schema with the following command. The output of the command includes an "`applied`" column to show which migrations are applied to the database.

Visual Studio

In the Visual Studio **Package Manager Console**, execute Get-Migration:

PowerShell

```
Get-Migration
```

If more than one database context exists, specify the context with the `-Context` parameter.

Add `MapRazorPages` to `Program.cs` as shown in the following highlighted code:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebMVCauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
```

```
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddControllersWithViews();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
app.MapRazorPages();

app.Run();
```

Scaffold Identity into an MVC project with authorization

Install the Microsoft.VisualStudio.Web.CodeGeneration.Design NuGet package.

> **Note**
>
> For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at Package consumption workflow (NuGet documentation). Confirm correct package versions at NuGet.org.

Run the Identity scaffolder:

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add New Scaffolded Item** dialog, select **Identity**. Select **Identity** in the center pane. Select the **Add** button.
- In the **Add Identity** dialog, select the options you want.
  - If you have an existing, customized layout page for Identity (`_Layout.cshtml`), select your existing layout page to avoid overwriting your layout with incorrect markup by the scaffolder. For example, select either:
    - `Pages/Shared/_Layout.cshtml` for Razor Pages or Blazor Server projects with existing Razor Pages infrastructure.
    - `Views/Shared/_Layout.cshtml` for MVC projects or Blazor Server projects with existing MVC infrastructure.
  - For the data context (**DbContext class**):
    - Select your data context class. You must select at least one file to add your data context.
    - To create a data context and possibly create a new user class for Identity, select the + button. Accept the default value or specify a class (for example, `Contoso.Data.ApplicationDbContext` for a company named "Contoso"). To create a new user class, select the + button for **User class** and specify the class (for example, `ContosoUser` for a company named "Contoso").
  - Select the **Add** button to run the scaffolder.

Create full Identity UI source

To maintain full control of the Identity UI, run the Identity scaffolder and select **Override all files**.

Password configuration

If PasswordOptions are configured in `Startup.ConfigureServices`, [StringLength] attribute configuration might be required for the `Password` property in scaffolded Identity pages. `InputModel` `Password` properties are found in the following files:

- `Areas/Identity/Pages/Account/Register.cshtml.cs`
- `Areas/Identity/Pages/Account/ResetPassword.cshtml.cs`

Disable a page

This section shows how to disable the register page but the approach can be used to disable any page.

To disable user registration:

- Scaffold Identity. Include Account.Register, Account.Login, and Account.RegisterConfirmation. For example:

  .NET CLI

  ```
  dotnet aspnet-codegenerator identity -dc
  RPauth.Data.ApplicationDbContext --files
  "Account.Register;Account.Login;Account.RegisterConfirmation"
  ```

- Update `Areas/Identity/Pages/Account/Register.cshtml.cs` so users can't register from this endpoint:

  C#

  ```
  public class RegisterModel : PageModel
  {
      public IActionResult OnGet()
      {
          return RedirectToPage("Login");
      }

      public IActionResult OnPost()
      {
          return RedirectToPage("Login");
      }
  }
  ```

- Update `Areas/Identity/Pages/Account/Register.cshtml` to be consistent with the preceding changes:

  CSHTML

  ```
  @page
  @model RegisterModel
  @{
      ViewData["Title"] = "Go to Login";
  }

  <h1>@ViewData["Title"]</h1>

  <li class="nav-item">
  ```

```
    <a class="nav-link text-dark" asp-area="Identity" asp-
page="/Account/Login">Login</a>
</li>
```

- Comment out or remove the registration link from `Areas/Identity/Pages/Account/Login.cshtml`

CSHTML

```
@*
<p>
    <a asp-page="./Register" asp-route-
returnUrl="@Model.ReturnUrl">Register as a new user</a>
</p>
*@
```

- Update the `Areas/Identity/Pages/Account/RegisterConfirmation` page.
  - Remove the code and links from the cshtml file.
  - Remove the confirmation code from the `PageModel`:

C#

```
[AllowAnonymous]
  public class RegisterConfirmationModel : PageModel
  {
      public IActionResult OnGet()
      {
          return Page();
      }
  }
```

Use another app to add users

Provide a mechanism to add users outside the web app. Options to add users include:

- A dedicated admin web app.
- A console app.

The following code outlines one approach to adding users:

- A list of users is read into memory.
- A strong unique password is generated for each user.
- The user is added to the Identity database.
- The user is notified and told to change the password.

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();

        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;

            try
            {
                var context = services.GetRequiredService<AppDbCntx>();
                context.Database.Migrate();

                var config =
host.Services.GetRequiredService<IConfiguration>();
                var userList =
config.GetSection("userList").Get<List<string>>();

                SeedData.Initialize(services, userList).Wait();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Program>>
();
                logger.LogError(ex, "An error occurred adding users.");
            }
        }

        host.Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code outlines adding a user:

C#

```
public static async Task Initialize(IServiceProvider serviceProvider,
                                    List<string> userList)
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>
();
```

```
    foreach (var userName in userList)
    {
        var userPassword = GenerateSecurePassword();
        var userId = await EnsureUser(userManager, userName, userPassword);

        NotifyUser(userName, userPassword);
    }
}

private static async Task<string> EnsureUser(UserManager<IdentityUser>
userManager,
                                             string userName, string
userPassword)
{
    var user = await userManager.FindByNameAsync(userName);

    if (user == null)
    {
        user = new IdentityUser(userName)
        {
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, userPassword);
    }

    return user.Id;
}
```

A similar approach can be followed for production scenarios.

Prevent publish of static Identity assets

To prevent publishing static Identity assets to the web root, see Introduction to Identity on ASP.NET Core.

Add, download, and delete custom user data to Identity in an ASP.NET Core project

Article • 07/28/2023

By Rick Anderson

This article shows how to:

- Add custom user data to an ASP.NET Core web app.
- Mark the custom user data model with the PersonalDataAttribute attribute so it's automatically available for download and deletion. Making the data able to be downloaded and deleted helps meet GDPR requirements.

The project sample is created from a Razor Pages web app, but the instructions are similar for an ASP.NET Core MVC web app.

View or download sample code (how to download)

Prerequisites

.NET 6.0 SDK

Create a Razor web app

Visual Studio

- From the Visual Studio **File** menu, select **New** > **Project**. Name the project **WebApp1** if you want to it match the namespace of the download sample code.
- Select **ASP.NET Core Web Application** > **OK**
- Select **Web Application** > **OK**
- Build and run the project.

Run the Identity scaffolder

Visual Studio

- From **Solution Explorer**, right-click on the project > **Add** > **New Scaffolded Item**.
- From the left pane of the **Add Scaffold** dialog, select **Identity** > **Add**.
- In the **Add Identity** dialog, the following options:
  - Select the existing layout file `~/Pages/Shared/_Layout.cshtml`
  - Select the following files to override:
    - **Account/Register**
    - **Account/Manage/Index**
  - Select the + button to create a new **Data context class**. Accept the type (**WebApp1.Models.WebApp1Context** if the project is named **WebApp1**).
  - Select the + button to create a new **User class**. Accept the type (**WebApp1User** if the project is named **WebApp1**) > **Add**.
- Select **Add**.

Follow the instruction in Migrations, UseAuthentication, and layout to perform the following steps:

- Create a migration and update the database.
- Add `UseAuthentication` to Program.cs
- Add `<partial name="_LoginPartial" />` to the layout file.
- Test the app:
  - Register a user
  - Select the new user name (next to the **Logout** link). You might need to expand the window or select the navigation bar icon to show the user name and other links.
  - Select the **Personal Data** tab.
  - Select the **Download** button and examined the `PersonalData.json` file.
  - Test the **Delete** button, which deletes the logged on user.

Add custom user data to the Identity DB

Update the `IdentityUser` derived class with custom properties. If you named the project WebApp1, the file is named `Areas/Identity/Data/WebApp1User.cs`. Update the file with the following code:

C#

```
using Microsoft.AspNetCore.Identity;

namespace WebApp1.Areas.Identity.Data;

public class WebApp1User : IdentityUser
{
    [PersonalData]
    public string? Name { get; set; }
    [PersonalData]
    public DateTime DOB { get; set; }
}
```

Properties with the PersonalData attribute are:

- Deleted when the `Areas/Identity/Pages/Account/Manage/DeletePersonalData.cshtml` Razor Page calls `UserManager.Delete`.
- Included in the downloaded data by the `Areas/Identity/Pages/Account/Manage/DownloadPersonalData.cshtml` Razor Page.

Update the `Account/Manage/Index.cshtml` page

Update the `InputModel` in `Areas/Identity/Pages/Account/Manage/Index.cshtml.cs` with the following highlighted code:

C#

```
public class IndexModel : PageModel
{
    private readonly UserManager<WebApp1User> _userManager;
    private readonly SignInManager<WebApp1User> _signInManager;

    public IndexModel(
        UserManager<WebApp1User> userManager,
        SignInManager<WebApp1User> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }

    /// <summary>
    ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
    ///     directly from your code. This API may change or be removed in
future releases.
    /// </summary>
    public string Username { get; set; }
```

```
    // Remaining API warnings ommited.

    [TempData]
    public string StatusMessage { get; set; }

    [BindProperty]
    public InputModel Input { get; set; }

    public class InputModel
    {
        [Required]
        [DataType(DataType.Text)]
        [Display(Name = "Full name")]
        public string Name { get; set; }

        [Required]
        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Phone]
        [Display(Name = "Phone number")]
        public string PhoneNumber { get; set; }
    }

    private async Task LoadAsync(WebApp1User user)
    {
        var userName = await _userManager.GetUserNameAsync(user);
        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);

        Username = userName;

        Input = new InputModel
        {
            Name = user.Name,
            DOB = user.DOB,
            PhoneNumber = phoneNumber
        };
    }

    public async Task<IActionResult> OnGetAsync()
    {
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID
'{_userManager.GetUserId(User)}'.");
        }

        await LoadAsync(user);
        return Page();
    }

    public async Task<IActionResult> OnPostAsync()
    {
```

```
        var user = await _userManager.GetUserAsync(User);
        if (user == null)
        {
            return NotFound($"Unable to load user with ID
'{_userManager.GetUserId(User)}'.");
        }

        if (!ModelState.IsValid)
        {
            await LoadAsync(user);
            return Page();
        }

        var phoneNumber = await _userManager.GetPhoneNumberAsync(user);
        if (Input.PhoneNumber != phoneNumber)
        {
            var setPhoneResult = await
_userManager.SetPhoneNumberAsync(user, Input.PhoneNumber);
            if (!setPhoneResult.Succeeded)
            {
                StatusMessage = "Unexpected error when trying to set phone
number.";
                return RedirectToPage();
            }
        }

        if (Input.Name != user.Name)
        {
            user.Name = Input.Name;
        }

        if (Input.DOB != user.DOB)
        {
            user.DOB = Input.DOB;
        }

        await _userManager.UpdateAsync(user);
        await _signInManager.RefreshSignInAsync(user);
        StatusMessage = "Your profile has been updated";
        return RedirectToPage();
    }
}
```

Update the `Areas/Identity/Pages/Account/Manage/Index.cshtml` with the following highlighted markup:

CSHTML

```
@page
@model IndexModel
@{
    ViewData["Title"] = "Profile";
    ViewData["ActivePage"] = ManageNavPages.Index;
```

```
}

<h3>@ViewData["Title"]</h3>
<partial name="_StatusMessage" for="StatusMessage" />
<div class="row">
    <div class="col-md-6">
        <form id="profile-form" method="post">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-floating">
                <input asp-for="Username" class="form-control" disabled />
                <label asp-for="Username" class="form-label"></label>
            </div>
            <div class="form-floating">
                <input asp-for="Input.Name" class="form-control" />
                <label asp-for="Input.Name" class="form-label"></label>
            </div>
            <div class="form-floating">
                <input asp-for="Input.DOB" class="form-control" />
                <label asp-for="Input.DOB" class="form-label"></label>
            </div>
            <div class="form-floating">
                <input asp-for="Input.PhoneNumber" class="form-control" />
                <label asp-for="Input.PhoneNumber" class="form-label">
</label>
                <span asp-validation-for="Input.PhoneNumber" class="text-
danger"></span>
            </div>
            <button id="update-profile-button" type="submit" class="w-100
btn btn-lg btn-primary">Save</button>
        </form>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Update the `Account/Register.cshtml` page

Update the `InputModel` in `Areas/Identity/Pages/Account/Register.cshtml.cs` with the following highlighted code:

C#

```
    public class RegisterModel : PageModel
    {
        private readonly SignInManager<WebApp1User> _signInManager;
        private readonly UserManager<WebApp1User> _userManager;
        private readonly IUserStore<WebApp1User> _userStore;
        private readonly IUserEmailStore<WebApp1User> _emailStore;
```

```
        private readonly ILogger<RegisterModel> _logger;
        private readonly IEmailSender _emailSender;

        public RegisterModel(
            UserManager<WebApp1User> userManager,
            IUserStore<WebApp1User> userStore,
            SignInManager<WebApp1User> signInManager,
            ILogger<RegisterModel> logger,
            IEmailSender emailSender)
        {
            _userManager = userManager;
            _userStore = userStore;
            _emailStore = GetEmailStore();
            _signInManager = signInManager;
            _logger = logger;
            _emailSender = emailSender;
        }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        [BindProperty]
        public InputModel Input { get; set; }

        // Remaining API warnings ommited.
        public string ReturnUrl { get; set; }

        public IList<AuthenticationScheme> ExternalLogins { get; set; }

        public class InputModel
        {
            [Required]
            [DataType(DataType.Text)]
            [Display(Name = "Full name")]
            public string Name { get; set; }

            [Required]
            [Display(Name = "Birth Date")]
            [DataType(DataType.Date)]
            public DateTime DOB { get; set; }

            [Required]
            [EmailAddress]
            [Display(Name = "Email")]
            public string Email { get; set; }

            [Required]
            [StringLength(100, ErrorMessage = "The {0} must be at least {2}
and at max {1} characters long.", MinimumLength = 6)]
            [DataType(DataType.Password)]
            [Display(Name = "Password")]
            public string Password { get; set; }
```

```
            [DataType(DataType.Password)]
            [Display(Name = "Confirm password")]
            [Compare("Password", ErrorMessage = "The password and
confirmation password do not match.")]
            public string ConfirmPassword { get; set; }
        }


        public async Task OnGetAsync(string returnUrl = null)
        {
            ReturnUrl = returnUrl;
            ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
        }

        public async Task<IActionResult> OnPostAsync(string returnUrl =
null)
        {
            returnUrl ??= Url.Content("~/");
            ExternalLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
            if (ModelState.IsValid)
            {
                var user = CreateUser();

                user.Name = Input.Name;
                user.DOB = Input.DOB;

                await _userStore.SetUserNameAsync(user, Input.Email,
CancellationToken.None);
                await _emailStore.SetEmailAsync(user, Input.Email,
CancellationToken.None);
                var result = await _userManager.CreateAsync(user,
Input.Password);

                if (result.Succeeded)
                {
                    _logger.LogInformation("User created a new account with
password.");

                    var userId = await _userManager.GetUserIdAsync(user);
                    var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                    code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                    var callbackUrl = Url.Page(
                        "/Account/ConfirmEmail",
                        pageHandler: null,
                        values: new { area = "Identity", userId = userId,
code = code, returnUrl = returnUrl },
                        protocol: Request.Scheme);

                    await _emailSender.SendEmailAsync(Input.Email, "Confirm
your email",
```

```
                        $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                    if (_userManager.Options.SignIn.RequireConfirmedAccount)
                    {
                        return RedirectToPage("RegisterConfirmation", new {
email = Input.Email, returnUrl = returnUrl });
                    }
                    else
                    {
                        await _signInManager.SignInAsync(user, isPersistent:
false);
                        return LocalRedirect(returnUrl);
                    }
                }
                foreach (var error in result.Errors)
                {
                    ModelState.AddModelError(string.Empty,
error.Description);
                }
            }

            // If we got this far, something failed, redisplay form
            return Page();
        }

        private WebApp1User CreateUser()
        {
            try
            {
                return Activator.CreateInstance<WebApp1User>();
            }
            catch
            {
                throw new InvalidOperationException($"Can't create an
instance of '{nameof(WebApp1User)}'. " +
                    $"Ensure that '{nameof(WebApp1User)}' is not an abstract
class and has a parameterless constructor, or alternatively " +
                    $"override the register page in
/Areas/Identity/Pages/Account/Register.cshtml");
            }
        }

        private IUserEmailStore<WebApp1User> GetEmailStore()
        {
            if (!_userManager.SupportsUserEmail)
            {
                throw new NotSupportedException("The default UI requires a
user store with email support.");
            }
            return (IUserEmailStore<WebApp1User>)_userStore;
        }
    }
}
```

Update the `Areas/Identity/Pages/Account/Register.cshtml` with the following highlighted markup:

CSHTML

```
@page
@model RegisterModel
@{
    ViewData["Title"] = "Register";
}

<h1>@ViewData["Title"]</h1>

<div class="row">
    <div class="col-md-4">
        <form id="registerForm" asp-route-returnUrl="@Model.ReturnUrl"
method="post">
            <h2>Create a new account.</h2>
            <hr />
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>

            <div class="form-floating">
                <label asp-for="Input.Name"></label>
                <input asp-for="Input.Name" class="form-control" />
                <span asp-validation-for="Input.Name" class="text-danger">
</span>
            </div>
            <div class="form-floating">
                <label asp-for="Input.DOB"></label>
                <input asp-for="Input.DOB" class="form-control" />
                <span asp-validation-for="Input.DOB" class="text-danger">
</span>
            </div>

            <div class="form-floating">
                <input asp-for="Input.Email" class="form-control"
autocomplete="username" aria-required="true" />
                <label asp-for="Input.Email"></label>
                <span asp-validation-for="Input.Email" class="text-danger">
</span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.Password" class="form-control"
autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.Password"></label>
                <span asp-validation-for="Input.Password" class="text-
danger"></span>
            </div>
            <div class="form-floating">
                <input asp-for="Input.ConfirmPassword" class="form-control"
autocomplete="new-password" aria-required="true" />
                <label asp-for="Input.ConfirmPassword"></label>
                <span asp-validation-for="Input.ConfirmPassword"
```

```
class="text-danger"></span>
            </div>
            <button id="registerSubmit" type="submit" class="w-100 btn btn-
lg btn-primary">Register</button>
        </form>
    </div>
    <div class="col-md-6 col-md-offset-2">
        <section>
            <h3>Use another service to register.</h3>
            <hr />
            @{
                if ((Model.ExternalLogins?.Count ?? 0) == 0)
                {
                    <div>
                        <p>
                            There are no external authentication services
configured. See this <a href="https://go.microsoft.com/fwlink/?
LinkID=532715">article
                            about setting up this ASP.NET application to
support logging in via external services</a>.
                        </p>
                    </div>
                }
                else
                {
                    <form id="external-account" asp-page="./ExternalLogin"
asp-route-returnUrl="@Model.ReturnUrl" method="post" class="form-
horizontal">
                        <div>
                            <p>
                                @foreach (var provider in
Model.ExternalLogins!)
                                {
                                    <button type="submit" class="btn btn-
primary" name="provider" value="@provider.Name" title="Log in using your
@provider.DisplayName account">@provider.DisplayName</button>
                                }
                            </p>
                        </div>
                    </form>
                }
            }
        </section>
    </div>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}
```

Build the project.

Update the layout

See Layout changes for instructions to add sign-in and sign-out links to every page.

Add a migration for the custom user data

Visual Studio

In the Visual Studio **Package Manager Console**:

PowerShell

```
Add-Migration CustomUserData
Update-Database
```

Test create, view, download, delete custom user data

Test the app:

- Register a new user.
- View the custom user data on the `/Identity/Account/Manage` page.
- Download and view the users personal data from the `/Identity/Account/Manage/PersonalData` page.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Identity model customization in ASP.NET Core

Article • 10/30/2024

By Arthur Vickers

ASP.NET Core Identity provides a framework for managing and storing user accounts in ASP.NET Core apps. Identity is added to your project when **Individual User Accounts** is selected as the authentication mechanism. By default, Identity makes use of an Entity Framework (EF) Core data model. This article describes how to customize the Identity model.

Identity and EF Core Migrations

Before examining the model, it's useful to understand how Identity works with EF Core Migrations to create and update a database. At the top level, the process is:

1. Define or update a data model in code.
2. Add a Migration to translate this model into changes that can be applied to the database.
3. Check that the Migration correctly represents your intentions.
4. Apply the Migration to update the database to be in sync with the model.
5. Repeat steps 1 through 4 to further refine the model and keep the database in sync.

Use one of the following approaches to add and apply Migrations:

- The **Package Manager Console** (PMC) window if using Visual Studio. For more information, see EF Core PMC tools.
- The .NET CLI if using the command line. For more information, see EF Core .NET command line tools.
- Clicking the **Apply Migrations** button on the error page when the app is run.

ASP.NET Core has a development-time error page handler. The handler can apply migrations when the app is run. Production apps typically generate SQL scripts from the migrations and deploy database changes as part of a controlled app and database deployment.

When a new app using Identity is created, steps 1 and 2 above have already been completed. That is, the initial data model already exists, and the initial migration has

been added to the project. The initial migration still needs to be applied to the database. The initial migration can be applied via one of the following approaches:

- Run `Update-Database` in PMC.
- Run `dotnet ef database update` in a command shell.
- Click the **Apply Migrations** button on the error page when the app is run.

Repeat the preceding steps as changes are made to the model.

The Identity model

Entity types

The Identity model consists of the following entity types.

Expand table

Entity type	Description
`User`	Represents the user.
`Role`	Represents a role.
`UserClaim`	Represents a claim that a user possesses.
`UserToken`	Represents an authentication token for a user.
`UserLogin`	Associates a user with a login.
`RoleClaim`	Represents a claim that's granted to all users within a role.
`UserRole`	A join entity that associates users and roles.

Entity type relationships

The entity types are related to each other in the following ways:

- Each `User` can have many `UserClaims`.
- Each `User` can have many `UserLogins`.
- Each `User` can have many `UserTokens`.
- Each `Role` can have many associated `RoleClaims`.
- Each `User` can have many associated `Roles`, and each `Role` can be associated with many `Users`. This is a many-to-many relationship that requires a join table in the

database. The join table is represented by the `UserRole` entity.

Default model configuration

Identity defines many *context classes* that inherit from DbContext to configure and use the model. This configuration is done using the EF Core Code First Fluent API in the OnModelCreating method of the context class. The default configuration is:

C#

```
builder.Entity<TUser>(b =>
{
    // Primary key
    b.HasKey(u => u.Id);

    // Indexes for "normalized" username and email, to allow efficient
lookups
    b.HasIndex(u =>
u.NormalizedUserName).HasName("UserNameIndex").IsUnique();
    b.HasIndex(u => u.NormalizedEmail).HasName("EmailIndex");

    // Maps to the AspNetUsers table
    b.ToTable("AspNetUsers");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(u => u.ConcurrencyStamp).IsConcurrencyToken();

    // Limit the size of columns to use efficient database types
    b.Property(u => u.UserName).HasMaxLength(256);
    b.Property(u => u.NormalizedUserName).HasMaxLength(256);
    b.Property(u => u.Email).HasMaxLength(256);
    b.Property(u => u.NormalizedEmail).HasMaxLength(256);

    // The relationships between User and other entity types
    // Note that these relationships are configured with no navigation
properties

    // Each User can have many UserClaims
    b.HasMany<TUserClaim>().WithOne().HasForeignKey(uc =>
uc.UserId).IsRequired();

    // Each User can have many UserLogins
    b.HasMany<TUserLogin>().WithOne().HasForeignKey(ul =>
ul.UserId).IsRequired();

    // Each User can have many UserTokens
    b.HasMany<TUserToken>().WithOne().HasForeignKey(ut =>
ut.UserId).IsRequired();

    // Each User can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur =>
ur.UserId).IsRequired();
```

```
});

builder.Entity<TUserClaim>(b =>
{
    // Primary key
    b.HasKey(uc => uc.Id);

    // Maps to the AspNetUserClaims table
    b.ToTable("AspNetUserClaims");
});

builder.Entity<TUserLogin>(b =>
{
    // Composite primary key consisting of the LoginProvider and the key to
use
    // with that provider
    b.HasKey(l => new { l.LoginProvider, l.ProviderKey });

    // Limit the size of the composite key columns due to common DB
restrictions
    b.Property(l => l.LoginProvider).HasMaxLength(128);
    b.Property(l => l.ProviderKey).HasMaxLength(128);

    // Maps to the AspNetUserLogins table
    b.ToTable("AspNetUserLogins");
});

builder.Entity<TUserToken>(b =>
{
    // Composite primary key consisting of the UserId, LoginProvider and
Name
    b.HasKey(t => new { t.UserId, t.LoginProvider, t.Name });

    // Limit the size of the composite key columns due to common DB
restrictions
    b.Property(t => t.LoginProvider).HasMaxLength(maxKeyLength);
    b.Property(t => t.Name).HasMaxLength(maxKeyLength);

    // Maps to the AspNetUserTokens table
    b.ToTable("AspNetUserTokens");
});

builder.Entity<TRole>(b =>
{
    // Primary key
    b.HasKey(r => r.Id);

    // Index for "normalized" role name to allow efficient lookups
    b.HasIndex(r => r.NormalizedName).HasName("RoleNameIndex").IsUnique();

    // Maps to the AspNetRoles table
    b.ToTable("AspNetRoles");

    // A concurrency token for use with the optimistic concurrency checking
    b.Property(r => r.ConcurrencyStamp).IsConcurrencyToken();
```

```
    // Limit the size of columns to use efficient database types
    b.Property(u => u.Name).HasMaxLength(256);
    b.Property(u => u.NormalizedName).HasMaxLength(256);

    // The relationships between Role and other entity types
    // Note that these relationships are configured with no navigation
properties

    // Each Role can have many entries in the UserRole join table
    b.HasMany<TUserRole>().WithOne().HasForeignKey(ur =>
ur.RoleId).IsRequired();

    // Each Role can have many associated RoleClaims
    b.HasMany<TRoleClaim>().WithOne().HasForeignKey(rc =>
rc.RoleId).IsRequired();
});

builder.Entity<TRoleClaim>(b =>
{
    // Primary key
    b.HasKey(rc => rc.Id);

    // Maps to the AspNetRoleClaims table
    b.ToTable("AspNetRoleClaims");
});

builder.Entity<TUserRole>(b =>
{
    // Primary key
    b.HasKey(r => new { r.UserId, r.RoleId });

    // Maps to the AspNetUserRoles table
    b.ToTable("AspNetUserRoles");
});
```

Model generic types

Identity defines default Common Language Runtime (CLR) types for each of the entity types listed above. These types are all prefixed with *Identity*:

- `IdentityUser`
- `IdentityRole`
- `IdentityUserClaim`
- `IdentityUserToken`
- `IdentityUserLogin`
- `IdentityRoleClaim`
- `IdentityUserRole`

Rather than using these types directly, the types can be used as base classes for the app's own types. The `DbContext` classes defined by Identity are generic, such that different CLR types can be used for one or more of the entity types in the model. These generic types also allow the `User` primary key (PK) data type to be changed.

When using Identity with support for roles, an IdentityDbContext class should be used. For example:

C#

```
// Uses all the built-in Identity types
// Uses `string` as the key type
public class IdentityDbContext
    : IdentityDbContext<IdentityUser, IdentityRole, string>
{
}

// Uses the built-in Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityDbContext<TUser>
    : IdentityDbContext<TUser, IdentityRole, string>
        where TUser : IdentityUser
{
}

// Uses the built-in Identity types except with custom User and Role types
// The key type is defined by TKey
public class IdentityDbContext<TUser, TRole, TKey> : IdentityDbContext<
    TUser, TRole, TKey, IdentityUserClaim<TKey>, IdentityUserRole<TKey>,
    IdentityUserLogin<TKey>, IdentityRoleClaim<TKey>,
IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TRole : IdentityRole<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic
arguments
// The key type is defined by TKey
public abstract class IdentityDbContext<
    TUser, TRole, TKey, TUserClaim, TUserRole, TUserLogin, TRoleClaim,
TUserToken>
    : IdentityUserContext<TUser, TKey, TUserClaim, TUserLogin, TUserToken>
         where TUser : IdentityUser<TKey>
         where TRole : IdentityRole<TKey>
         where TKey : IEquatable<TKey>
         where TUserClaim : IdentityUserClaim<TKey>
         where TUserRole : IdentityUserRole<TKey>
         where TUserLogin : IdentityUserLogin<TKey>
```

```
        where TRoleClaim : IdentityRoleClaim<TKey>
        where TUserToken : IdentityUserToken<TKey>
```

It's also possible to use Identity without roles (only claims), in which case an IdentityUserContext<TUser> class should be used:

C#

```
// Uses the built-in non-role Identity types except with a custom User type
// Uses `string` as the key type
public class IdentityUserContext<TUser>
    : IdentityUserContext<TUser, string>
        where TUser : IdentityUser
{
}

// Uses the built-in non-role Identity types except with a custom User type
// The key type is defined by TKey
public class IdentityUserContext<TUser, TKey> : IdentityUserContext<
    TUser, TKey, IdentityUserClaim<TKey>, IdentityUserLogin<TKey>,
    IdentityUserToken<TKey>>
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
{
}

// No built-in Identity types are used; all are specified by generic
arguments, with no roles
// The key type is defined by TKey
public abstract class IdentityUserContext<
    TUser, TKey, TUserClaim, TUserLogin, TUserToken> : DbContext
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
        where TUserClaim : IdentityUserClaim<TKey>
        where TUserLogin : IdentityUserLogin<TKey>
        where TUserToken : IdentityUserToken<TKey>
{
}
```

Customize the model

The starting point for model customization is to derive from the appropriate context type. See the Model generic types section. This context type is customarily called `ApplicationDbContext` and is created by the ASP.NET Core templates.

The context is used to configure the model in two ways:

- Supplying entity and key types for the generic type parameters.

- Overriding `OnModelCreating` to modify the mapping of these types.

When overriding `OnModelCreating`, `base.OnModelCreating` should be called first; the overriding configuration should be called next. EF Core generally has a last-one-wins policy for configuration. For example, if the `ToTable` method for an entity type is called first with one table name and then again later with a different table name, the table name in the second call is used.

NOTE: If the `DbContext` doesn't derive from `IdentityDbContext`, `AddEntityFrameworkStores` may not infer the correct POCO types for `TUserClaim`, `TUserLogin`, and `TUserToken`. If `AddEntityFrameworkStores` doesn't infer the correct POCO types, a workaround is to directly add the correct types via `services.AddScoped<IUser/RoleStore<TUser>` and `UserStore<...>>`.

Custom user data

Custom user data is supported by inheriting from `IdentityUser`. It's customary to name this type `ApplicationUser`:

C#

```
public class ApplicationUser : IdentityUser
{
    public string CustomTag { get; set; }
}
```

Use the `ApplicationUser` type as a generic argument for the context:

C#

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
    }
}
```

There's no need to override `OnModelCreating` in the `ApplicationDbContext` class. EF Core maps the `CustomTag` property by convention. However, the database needs to be updated to create a new `CustomTag` column. To create the column, add a migration, and then update the database as described in Identity and EF Core Migrations.

Update `Pages/Shared/_LoginPartial.cshtml` and replace `IdentityUser` with `ApplicationUser`:

CSHTML

```
@using Microsoft.AspNetCore.Identity
@using WebApp1.Areas.Identity.Data
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager
```

Update `Areas/Identity/IdentityHostingStartup.cs` or `Startup.ConfigureServices` and replace `IdentityUser` with `ApplicationUser`.

C#

```
services.AddDefaultIdentity<ApplicationUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

Calling AddDefaultIdentity is equivalent to the following code:

C#

```
services.AddAuthentication(o =>
{
    o.DefaultScheme = IdentityConstants.ApplicationScheme;
    o.DefaultSignInScheme = IdentityConstants.ExternalScheme;
})
.AddIdentityCookies(o => { });

services.AddIdentityCore<TUser>(o =>
{
    o.Stores.MaxLengthForKeys = 128;
    o.SignIn.RequireConfirmedAccount = true;
})
.AddDefaultUI()
.AddDefaultTokenProviders();
```

Identity is provided as a Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to

AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`. For more information, see:

- Scaffold Identity
- Add, download, and delete custom user data to Identity

Change the primary key type

A change to the PK column's data type after the database has been created is problematic on many database systems. Changing the PK typically involves dropping and re-creating the table. Therefore, key types should be specified in the initial migration when the database is created.

Follow these steps to change the PK type:

1. If the database was created before the PK change, run `Drop-Database` (PMC) or `dotnet ef database drop` (.NET CLI) to delete it.

2. After confirming deletion of the database, remove the initial migration with `Remove-Migration` (PMC) or `dotnet ef migrations remove` (.NET CLI).

3. Update the `ApplicationDbContext` class to derive from IdentityDbContext<TUser,TRole,TKey>. Specify the new key type for `TKey`. For example, to use a `Guid` key type:

   C#

   ```
   public class ApplicationDbContext
       : IdentityDbContext<IdentityUser<Guid>, IdentityRole<Guid>, Guid>
   {
       public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
   options)
           : base(options)
       {
       }
   }
   ```

   In the preceding code, the generic classes IdentityUser<TKey> and IdentityRole<TKey> must be specified to use the new key type.

   `Startup.ConfigureServices` must be updated to use the generic user:

   C#

   ```
   services.AddDefaultIdentity<IdentityUser<Guid>>(options =>
   options.SignIn.RequireConfirmedAccount = true)
   ```

```
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

4. If a custom `ApplicationUser` class is being used, update the class to inherit from `IdentityUser`. For example:

```
C#

using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationUser : IdentityUser<Guid>
{
    public string CustomTag { get; set; }
}
```

Update `ApplicationDbContext` to reference the custom `ApplicationUser` class:

```
C#

public class ApplicationDbContext
    : IdentityDbContext<ApplicationUser, IdentityRole<Guid>, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

```
C#

services.AddDefaultIdentity<ApplicationUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
```

The primary key's data type is inferred by analyzing the DbContext object.

Identity is provided as a Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`.

5. If a custom `ApplicationRole` class is being used, update the class to inherit from `IdentityRole<TKey>`. For example:

```
C#

using System;
using Microsoft.AspNetCore.Identity;

public class ApplicationRole : IdentityRole<Guid>
{
    public string Description { get; set; }
}
```

Update `ApplicationDbContext` to reference the custom `ApplicationRole` class. For example, the following class references a custom `ApplicationUser` and a custom `ApplicationRole`:

```
C#

using System;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;

public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, ApplicationRole, Guid>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }
}
```

Register the custom database context class when adding the Identity service in `Startup.ConfigureServices`:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
```

```
    services.AddIdentity<ApplicationUser, ApplicationRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultUI()
            .AddDefaultTokenProviders();

    services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
}
```

The primary key's data type is inferred by analyzing the DbContext object.

Identity is provided as a Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects. Consequently, the preceding code requires a call to AddDefaultUI. If the Identity scaffolder was used to add Identity files to the project, remove the call to `AddDefaultUI`.

Add navigation properties

Changing the model configuration for relationships can be more difficult than making other changes. Care must be taken to replace the existing relationships rather than create new, additional relationships. In particular, the changed relationship must specify the same foreign key (FK) property as the existing relationship. For example, the relationship between `Users` and `UserClaims` is, by default, specified as follows:

C#

```
builder.Entity<TUser>(b =>
{
    // Each User can have many UserClaims
    b.HasMany<TUserClaim>()
     .WithOne()
     .HasForeignKey(uc => uc.UserId)
     .IsRequired();
});
```

The FK for this relationship is specified as the `UserClaim.UserId` property. `HasMany` and `WithOne` are called without arguments to create the relationship without navigation properties.

Add a navigation property to `ApplicationUser` that allows associated `UserClaims` to be referenced from the user:

C#

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set;
}
}
```

The `TKey` for `IdentityUserClaim<TKey>` is the type specified for the PK of users. In this case, `TKey` is `string` because the defaults are being used. It's **not** the PK type for the `UserClaim` entity type.

Now that the navigation property exists, it must be configured in `OnModelCreating`:

C#

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();
        });
    }
}
```

Notice that relationship is configured exactly as it was before, only with a navigation property specified in the call to `HasMany`.

The navigation properties only exist in the EF model, not the database. Because the FK for the relationship hasn't changed, this kind of model change doesn't require the database to be updated. This can be checked by adding a migration after making the change. The `Up` and `Down` methods are empty.

Add all User navigation properties

Using the section above as guidance, the following example configures unidirectional navigation properties for all relationships on User:

C#

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set;
}
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set;
}
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set;
}
    public virtual ICollection<IdentityUserRole<string>> UserRoles { get;
set; }
}
```

C#

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
```

```
            b.HasMany(e => e.UserRoles)
                .WithOne()
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });
    }
}
```

Add User and Role navigation properties

Using the section above as guidance, the following example configures navigation properties for all relationships on User and Role:

C#

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<IdentityUserClaim<string>> Claims { get; set;
}
    public virtual ICollection<IdentityUserLogin<string>> Logins { get; set;
}
    public virtual ICollection<IdentityUserToken<string>> Tokens { get; set;
}
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}
```

C#

```
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        IdentityUserClaim<string>, ApplicationUserRole,
IdentityUserLogin<string>,
        IdentityRoleClaim<string>, IdentityUserToken<string>>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
```

```
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne()
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne()
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne()
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();
        });

    }
}
```

Notes:

- This example also includes the `UserRole` join entity, which is needed to navigate the many-to-many relationship from Users to Roles.
- Remember to change the types of the navigation properties to reflect that `Application{...}` types are now being used instead of `Identity{...}` types.

- Remember to use the `Application{...}` in the generic `ApplicationContext` definition.

Add all navigation properties

Using the section above as guidance, the following example configures navigation properties for all relationships on all entity types:

C#

```
public class ApplicationUser : IdentityUser
{
    public virtual ICollection<ApplicationUserClaim> Claims { get; set; }
    public virtual ICollection<ApplicationUserLogin> Logins { get; set; }
    public virtual ICollection<ApplicationUserToken> Tokens { get; set; }
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
}

public class ApplicationRole : IdentityRole
{
    public virtual ICollection<ApplicationUserRole> UserRoles { get; set; }
    public virtual ICollection<ApplicationRoleClaim> RoleClaims { get; set;
}
}

public class ApplicationUserRole : IdentityUserRole<string>
{
    public virtual ApplicationUser User { get; set; }
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserClaim : IdentityUserClaim<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationUserLogin : IdentityUserLogin<string>
{
    public virtual ApplicationUser User { get; set; }
}

public class ApplicationRoleClaim : IdentityRoleClaim<string>
{
    public virtual ApplicationRole Role { get; set; }
}

public class ApplicationUserToken : IdentityUserToken<string>
{
    public virtual ApplicationUser User { get; set; }
}
```

C#

```
public class ApplicationDbContext
    : IdentityDbContext<
        ApplicationUser, ApplicationRole, string,
        ApplicationUserClaim, ApplicationUserRole, ApplicationUserLogin,
        ApplicationRoleClaim, ApplicationUserToken>
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<ApplicationUser>(b =>
        {
            // Each User can have many UserClaims
            b.HasMany(e => e.Claims)
                .WithOne(e => e.User)
                .HasForeignKey(uc => uc.UserId)
                .IsRequired();

            // Each User can have many UserLogins
            b.HasMany(e => e.Logins)
                .WithOne(e => e.User)
                .HasForeignKey(ul => ul.UserId)
                .IsRequired();

            // Each User can have many UserTokens
            b.HasMany(e => e.Tokens)
                .WithOne(e => e.User)
                .HasForeignKey(ut => ut.UserId)
                .IsRequired();

            // Each User can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.User)
                .HasForeignKey(ur => ur.UserId)
                .IsRequired();
        });

        modelBuilder.Entity<ApplicationRole>(b =>
        {
            // Each Role can have many entries in the UserRole join table
            b.HasMany(e => e.UserRoles)
                .WithOne(e => e.Role)
                .HasForeignKey(ur => ur.RoleId)
                .IsRequired();

            // Each Role can have many associated RoleClaims
```

```
            b.HasMany(e => e.RoleClaims)
                .WithOne(e => e.Role)
                .HasForeignKey(rc => rc.RoleId)
                .IsRequired();
        });
    }
}
```

Use composite keys

The preceding sections demonstrated changing the type of key used in the Identity model. Changing the Identity key model to use composite keys isn't supported or recommended. Using a composite key with Identity involves changing how the Identity manager code interacts with the model. This customization is beyond the scope of this document.

Change table/column names and facets

To change the names of tables and columns, call `base.OnModelCreating`. Then, add configuration to override any of the defaults. For example, to change the name of all the Identity tables:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.ToTable("MyUsers");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.ToTable("MyUserClaims");
    });

    modelBuilder.Entity<IdentityUserLogin<string>>(b =>
    {
        b.ToTable("MyUserLogins");
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.ToTable("MyUserTokens");
    });
```

```
    modelBuilder.Entity<IdentityRole>(b =>
    {
        b.ToTable("MyRoles");
    });

    modelBuilder.Entity<IdentityRoleClaim<string>>(b =>
    {
        b.ToTable("MyRoleClaims");
    });

    modelBuilder.Entity<IdentityUserRole<string>>(b =>
    {
        b.ToTable("MyUserRoles");
    });
}
```

These examples use the default Identity types. If using an app type such as `ApplicationUser`, configure that type instead of the default type.

The following example changes some column names:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
    {
        b.Property(e => e.Email).HasColumnName("EMail");
    });

    modelBuilder.Entity<IdentityUserClaim<string>>(b =>
    {
        b.Property(e => e.ClaimType).HasColumnName("CType");
        b.Property(e => e.ClaimValue).HasColumnName("CValue");
    });
}
```

Some types of database columns can be configured with certain *facets* (for example, the maximum `string` length allowed). The following example sets column maximum lengths for several `string` properties in the model:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<IdentityUser>(b =>
```

```
    {
        b.Property(u => u.UserName).HasMaxLength(128);
        b.Property(u => u.NormalizedUserName).HasMaxLength(128);
        b.Property(u => u.Email).HasMaxLength(128);
        b.Property(u => u.NormalizedEmail).HasMaxLength(128);
    });

    modelBuilder.Entity<IdentityUserToken<string>>(b =>
    {
        b.Property(t => t.LoginProvider).HasMaxLength(128);
        b.Property(t => t.Name).HasMaxLength(128);
    });
}
```

Map to a different schema

Schemas can behave differently across database providers. For SQL Server, the default is to create all tables in the *dbo* schema. The tables can be created in a different schema. For example:

C#

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.HasDefaultSchema("notdbo");
}
```

Lazy loading

In this section, support for lazy-loading proxies in the Identity model is added. Lazy-loading is useful since it allows navigation properties to be used without first ensuring they're loaded.

Entity types can be made suitable for lazy-loading in several ways, as described in the EF Core documentation. For simplicity, use lazy-loading proxies, which requires:

- Installation of the Microsoft.EntityFrameworkCore.Proxies package.
- A call to UseLazyLoadingProxies inside AddDbContext.
- Public entity types with `public virtual` navigation properties.

The following example demonstrates calling `UseLazyLoadingProxies` in `Startup.ConfigureServices`:

C#

```
services
    .AddDbContext<ApplicationDbContext>(
        b => b.UseSqlServer(connectionString)
              .UseLazyLoadingProxies())
    .AddDefaultIdentity<ApplicationUser>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Refer to the preceding examples for guidance on adding navigation properties to the entity types.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

Additional resources

- Scaffold Identity in ASP.NET Core projects

Configure ASP.NET Core Identity

Article • 03/09/2024

ASP.NET Core Identity uses default values for settings such as password policy, lockout, and cookie configuration. These settings can be overridden at application startup.

Identity options

The IdentityOptions class represents the options that can be used to configure the Identity system. IdentityOptions must be set **after** calling AddIdentity or AddDefaultIdentity.

Claims Identity

IdentityOptions.ClaimsIdentity specifies the ClaimsIdentityOptions with the properties shown in the following table.

Expand table

Property	Description	Default
RoleClaimType	Gets or sets the claim type used for a role claim.	ClaimTypes.Role
SecurityStampClaimType	Gets or sets the claim type used for the security stamp claim.	`AspNet.Identity.SecurityStamp`
UserIdClaimType	Gets or sets the claim type used for the user identifier claim.	ClaimTypes.NameIdentifier
UserNameClaimType	Gets or sets the claim type used for the user name claim.	ClaimTypes.Name

Lockout

Lockout is set in the PasswordSignInAsync method:

C#

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl ??= Url.Content("~/");

    ExternalLogins = (await
```

```
_signInManager.GetExternalAuthenticationSchemesAsync()).ToList();

    if (ModelState.IsValid)
    {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set
lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
             Input.Password, Input.RememberMe,
             lockoutOnFailure: false);
        if (result.Succeeded)
        {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor)
        {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl =
returnUrl, RememberMe = Input.RememberMe });
        }
        if (result.IsLockedOut)
        {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else
        {
            ModelState.AddModelError(string.Empty, "Invalid login
attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
```

The preceding code is based on the Login Identity template.

Lockout options are set in `Program.cs`:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
```

```
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>

options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.Configure<IdentityOptions>(options =>
{
    // Default Lockout settings.
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding code sets the IdentityOptions LockoutOptions with default values.

A successful authentication resets the failed access attempts count and resets the clock.

IdentityOptions.Lockout specifies the LockoutOptions with the properties shown in the table.

Expand table

Property	Description	Default
AllowedForNewUsers	Determines if a new user can be locked out.	`true`

Property	Description	Default
DefaultLockoutTimeSpan	The amount of time a user is locked out when a lockout occurs.	5 minutes
MaxFailedAccessAttempts	The number of failed access attempts until a user is locked out, if lockout is enabled.	5

Password

By default, Identity requires that passwords contain an uppercase character, lowercase character, a digit, and a non-alphanumeric character. Passwords must be at least six characters long.

Passwords are configured with:

- PasswordOptions in `Program.cs`.
- [StringLength] attributes of `Password` properties if Identity is scaffolded into the app. `InputModel` `Password` properties are found in the following files:
  - `Areas/Identity/Pages/Account/Register.cshtml.cs`
  - `Areas/Identity/Pages/Account/ResetPassword.cshtml.cs`

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using RPauth.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
                                options.SignIn.RequireConfirmedAccount =
true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
```

```
builder.Services.Configure<IdentityOptions>(options =>
{
    // Default Password settings.
    options.Password.RequireDigit = true;
    options.Password.RequireLowercase = true;
    options.Password.RequireNonAlphanumeric = true;
    options.Password.RequireUppercase = true;
    options.Password.RequiredLength = 6;
    options.Password.RequiredUniqueChars = 1;
});

var app = builder.Build();

// Remaining code removed for brevity.
```

IdentityOptions.Password specifies the PasswordOptions with the properties shown in the table.

Expand table

Property	Description	Default
RequireDigit	Requires a number between 0-9 in the password.	`true`
RequiredLength	The minimum length of the password.	6
RequireLowercase	Requires a lowercase character in the password.	`true`
RequireNonAlphanumeric	Requires a non-alphanumeric character in the password.	`true`
RequiredUniqueChars	Only applies to ASP.NET Core 2.0 or later. Requires the number of distinct characters in the password.	1
RequireUppercase	Requires an uppercase character in the password.	`true`

Sign-in

The following code sets `SignIn` settings (to default values):

C#

```
builder.Services.Configure<IdentityOptions>(options =>
{
    // Default SignIn settings.
    options.SignIn.RequireConfirmedEmail = false;
    options.SignIn.RequireConfirmedPhoneNumber = false;
});
```

IdentityOptions.SignIn specifies the SignInOptions with the properties shown in the table.

Expand table

Property	Description	Default
RequireConfirmedEmail	Requires a confirmed email to sign in.	`false`
RequireConfirmedPhoneNumber	Requires a confirmed phone number to sign in.	`false`

Tokens

IdentityOptions.Tokens specifies the TokenOptions with the properties shown in the table.

Expand table

Property	Description
AuthenticatorTokenProvider	Gets or sets the `AuthenticatorTokenProvider` used to validate two-factor sign-ins with an authenticator.
ChangeEmailTokenProvider	Gets or sets the `ChangeEmailTokenProvider` used to generate tokens used in email change confirmation emails.
ChangePhoneNumberTokenProvider	Gets or sets the `ChangePhoneNumberTokenProvider` used to generate tokens used when changing phone numbers.
EmailConfirmationTokenProvider	Gets or sets the token provider used to generate tokens used in account confirmation emails.
PasswordResetTokenProvider	Gets or sets the IUserTwoFactorTokenProvider<TUser> used to generate tokens used in password reset emails.
ProviderMap	Used to construct a User Token Provider with the key used as the provider's name.

User

C#

```
builder.Services.Configure<IdentityOptions>(options =>
{
    // Default User settings.
    options.User.AllowedUserNameCharacters =
```

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
    options.User.RequireUniqueEmail = false;

});
```

IdentityOptions.User specifies the UserOptions with the properties shown in the table.

Expand table

Property	Description	Default
AllowedUserNameCharacters	Allowed characters in the username.	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 -._@+
RequireUniqueEmail	Requires each user to have a unique email.	`false`

Cookie settings

Configure the app's cookie in `Program.cs`. ConfigureApplicationCookie must be called **after** calling `AddIdentity` or `AddDefaultIdentity`.

C#

```
builder.Services.ConfigureApplicationCookie(options =>
{
    options.AccessDeniedPath = "/Identity/Account/AccessDenied";
    options.Cookie.Name = "YourAppCookieName";
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(60);
    options.LoginPath = "/Identity/Account/Login";
    // ReturnUrlParameter requires
    //using Microsoft.AspNetCore.Authentication.Cookies;
    options.ReturnUrlParameter =
CookieAuthenticationDefaults.ReturnUrlParameter;
    options.SlidingExpiration = true;
});
```

For more information, see CookieAuthenticationOptions.

Password Hasher options

PasswordHasherOptions gets and sets options for password hashing.

Expand table

Option	Description
CompatibilityMode	The compatibility mode used when hashing new passwords. Defaults to IdentityV3. The first byte of a hashed password, called a *format marker*, specifies the version of the hashing algorithm used to hash the password. When verifying a password against a hash, the VerifyHashedPassword method selects the correct algorithm based on the first byte. A client is able to authenticate regardless of which version of the algorithm was used to hash the password. Setting the compatibility mode affects the hashing of *new passwords*.
IterationCount	The number of iterations used when hashing passwords using PBKDF2. This value is only used when the CompatibilityMode is set to IdentityV3. The value must be a positive integer and defaults to `100000`.

In the following example, the IterationCount is set to `12000` in `Program.cs`:

C#

```
// using Microsoft.AspNetCore.Identity;

builder.Services.Configure<PasswordHasherOptions>(option =>
{
    option.IterationCount = 12000;
});
```

Globally require all users to be authenticated

For information on how to globally require all users to be authenticated, see Require authenticated users.

ISecurityStampValidator and SignOut everywhere

Apps need to react to events involving security sensitive actions by regenerating the users ClaimsPrincipal. For example, the `ClaimsPrincipal` should be regenerated when joining a role, changing the password, or other security sensitive events. Identity uses the ISecurityStampValidator interface to regenerate the `ClaimsPrincipal`. The default implementation of Identity registers a SecurityStampValidator with the main application cookie and the two-factor cookie. The validator hooks into the OnValidatePrincipal event of each cookie to call into Identity to verify that the user's security stamp claim is

unchanged from what's stored in the cookie. The validator calls in at regular intervals. The call interval is a tradeoff between hitting the datastore too frequently and not often enough. Checking with a long interval results in stale claims. Call `userManager.UpdateSecurityStampAsync(user)` to force existing cookies to be invalided the next time they are checked. Most of the Identity UI account and manage pages call `userManager.UpdateSecurityStampAsync(user)` after changing the password or adding a login. Apps can call `userManager.UpdateSecurityStampAsync(user)` to implement a sign out everywhere action.

Changing the validation interval is shown in the following highlighted code:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebClaimsPrincipal.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new InvalidOperationException("'DefaultConnection' not
found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

// Force Identity's security stamp to be validated every minute.
builder.Services.Configure<SecurityStampValidatorOptions>(o =>
                    o.ValidationInterval = TimeSpan.FromMinutes(1));

builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Custom storage providers for ASP.NET Core Identity

Article • 10/29/2024

By Steve Smith

ASP.NET Core Identity is an extensible system which enables you to create a custom storage provider and connect it to your app. This topic describes how to create a customized storage provider for ASP.NET Core Identity. It covers the important concepts for creating your own storage provider, but isn't a step-by-step walk through. See Identity model customization to customize an Identity model.

Introduction

By default, the ASP.NET Core Identity system stores user information in a SQL Server database using Entity Framework Core. For many apps, this approach works well. However, you may prefer to use a different persistence mechanism or data schema. For example:

- You use Azure Table Storage or another data store.
- Your database tables have a different structure.
- You may wish to use a different data access approach, such as Dapper.

In each of these cases, you can write a customized provider for your storage mechanism and plug that provider into your app.

ASP.NET Core Identity is included in project templates in Visual Studio with the "Individual User Accounts" option.

When using the .NET CLI, add `-au Individual`:

.NET CLI

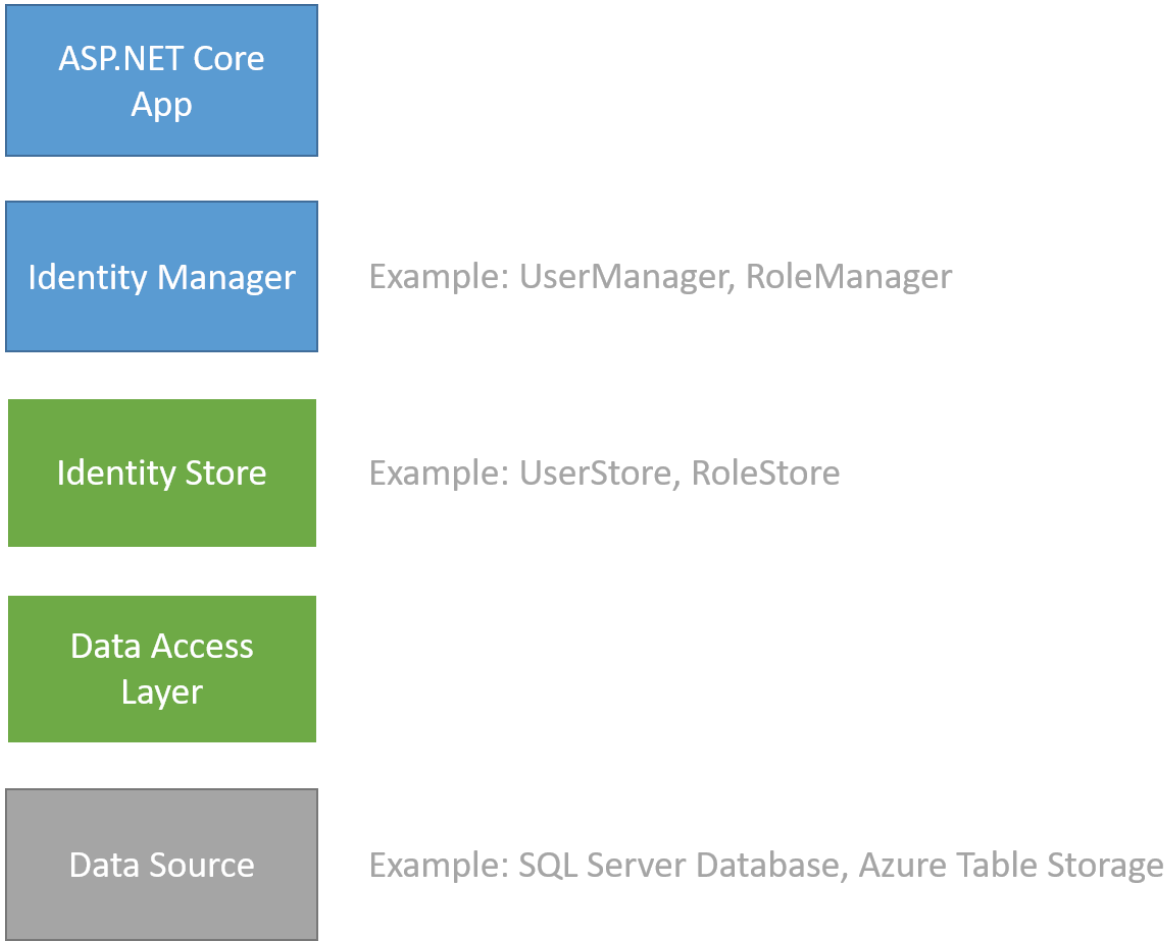
```
dotnet new mvc -au Individual
```

The ASP.NET Core Identity architecture

ASP.NET Core Identity consists of classes called managers and stores. *Managers* are high-level classes which an app developer uses to perform operations, such as creating an Identity user. *Stores* are lower-level classes that specify how entities, such as users

and roles, are persisted. Stores follow the repository pattern and are closely coupled with the persistence mechanism. Managers are decoupled from stores, which means you can replace the persistence mechanism without changing your application code (except for configuration).

The following diagram shows how a web app interacts with the managers, while stores interact with the data access layer.



To create a custom storage provider, create the data source, the data access layer, and the store classes that interact with this data access layer (the green and grey boxes in the diagram above). You don't need to customize the managers or your app code that interacts with them (the blue boxes above).

When creating a new instance of `UserManager` or `RoleManager` you provide the type of the user class and pass an instance of the store class as an argument. This approach enables you to plug your customized classes into ASP.NET Core.

Reconfigure app to use new storage provider shows how to instantiate `UserManager` and `RoleManager` with a customized store.

ASP.NET Core Identity stores data types

ASP.NET Core Identity data types are detailed in the following sections:

Users

Registered users of your web site. The IdentityUser type may be extended or used as an example for your own custom type. You don't need to inherit from a particular type to implement your own custom identity storage solution.

User Claims

A set of statements (or Claims) about the user that represent the user's identity. Can enable greater expression of the user's identity than can be achieved through roles.

User Logins

Information about the external authentication provider (like Facebook or a Microsoft account) to use when logging in a user. Example

Roles

Authorization groups for your site. Includes the role Id and role name (like "Admin" or "Employee"). Example

The data access layer

This topic assumes you are familiar with the persistence mechanism that you are going to use and how to create entities for that mechanism. This topic doesn't provide details about how to create the repositories or data access classes; it provides some suggestions about design decisions when working with ASP.NET Core Identity.

You have a lot of freedom when designing the data access layer for a customized store provider. You only need to create persistence mechanisms for features that you intend to use in your app. For example, if you are not using roles in your app, you don't need to create storage for roles or user role associations. Your technology and existing infrastructure may require a structure that's very different from the default implementation of ASP.NET Core Identity. In your data access layer, you provide the logic to work with the structure of your storage implementation.

The data access layer provides the logic to save the data from ASP.NET Core Identity to a data source. The data access layer for your customized storage provider might include the following classes to store user and role information.

Context class

Encapsulates the information to connect to your persistence mechanism and execute queries. Several data classes require an instance of this class, typically provided through dependency injection. Example.

User Storage

Stores and retrieves user information (such as user name and password hash). Example

Role Storage

Stores and retrieves role information (such as the role name). Example

UserClaims Storage

Stores and retrieves user claim information (such as the claim type and value). Example

UserLogins Storage

Stores and retrieves user login information (such as an external authentication provider). Example

UserRole Storage

Stores and retrieves which roles are assigned to which users. Example

TIP: Only implement the classes you intend to use in your app.

In the data access classes, provide code to perform data operations for your persistence mechanism. For example, within a custom provider, you might have the following code to create a new user in the *store* class:

C#

```
public async Task<IdentityResult> CreateAsync(ApplicationUser user,
    CancellationToken cancellationToken = default(CancellationToken))
{
    cancellationToken.ThrowIfCancellationRequested();
    if (user == null) throw new ArgumentNullException(nameof(user));

    return await _usersTable.CreateAsync(user);
}
```

The implementation logic for creating the user is in the `_usersTable.CreateAsync` method, shown below.

Customize the user class

When implementing a storage provider, create a user class which is equivalent to the IdentityUser class.

At a minimum, your user class must include an `Id` and a `UserName` property.

The `IdentityUser` class defines the properties that the `UserManager` calls when performing requested operations. The default type of the `Id` property is a string, but you can inherit from `IdentityUser<TKey, TUserClaim, TUserRole, TUserLogin, TUserToken>` and specify a different type. The framework expects the storage implementation to handle data type conversions.

Customize the user store

Create a `UserStore` class that provides the methods for all data operations on the user. This class is equivalent to the UserStore<TUser> class. In your `UserStore` class, implement `IUserStore<TUser>` and the optional interfaces required. You select which optional interfaces to implement based on the functionality provided in your app.

Optional interfaces

- IUserRoleStore
- IUserClaimStore
- IUserPasswordStore
- IUserSecurityStampStore
- IUserEmailStore
- IUserPhoneNumberStore
- IQueryableUserStore
- IUserLoginStore
- IUserTwoFactorStore
- IUserLockoutStore

The optional interfaces inherit from `IUserStore<TUser>`. You can see a partially implemented sample user store in the sample app.

Within the `UserStore` class, you use the data access classes that you created to perform operations. These are passed in using dependency injection. For example, in the SQL Server with Dapper implementation, the `UserStore` class has the `CreateAsync` method which uses an instance of `DapperUsersTable` to insert a new record:

```
C#

public async Task<IdentityResult> CreateAsync(ApplicationUser user)
{
    string sql = "INSERT INTO dbo.CustomUser " +
        "VALUES (@id, @Email, @EmailConfirmed, @PasswordHash, @UserName)";

    int rows = await _connection.ExecuteAsync(sql, new { user.Id,
user.Email, user.EmailConfirmed, user.PasswordHash, user.UserName });

    if(rows > 0)
    {
        return IdentityResult.Success;
    }
    return IdentityResult.Failed(new IdentityError { Description = $"Could
not insert user {user.Email}." });
}
```

Interfaces to implement when customizing user store

- **IUserStore**
  The IUserStore<TUser> interface is the only interface you must implement in the user store. It defines methods for creating, updating, deleting, and retrieving users.
- **IUserClaimStore**
  The IUserClaimStore<TUser> interface defines the methods you implement to enable user claims. It contains methods for adding, removing and retrieving user claims.
- **IUserLoginStore**
  The IUserLoginStore<TUser> defines the methods you implement to enable external authentication providers. It contains methods for adding, removing and retrieving user logins, and a method for retrieving a user based on the login information.
- **IUserRoleStore**
  The IUserRoleStore<TUser> interface defines the methods you implement to map a user to a role. It contains methods to add, remove, and retrieve a user's roles, and a method to check if a user is assigned to a role.
- **IUserPasswordStore**
  The IUserPasswordStore<TUser> interface defines the methods you implement to

persist hashed passwords. It contains methods for getting and setting the hashed password, and a method that indicates whether the user has set a password.

- **IUserSecurityStampStore**
  The IUserSecurityStampStore<TUser> interface defines the methods you implement to use a security stamp for indicating whether the user's account information has changed. This stamp is updated when a user changes the password, or adds or removes logins. It contains methods for getting and setting the security stamp.
- **IUserTwoFactorStore**
  The IUserTwoFactorStore<TUser> interface defines the methods you implement to support two factor authentication. It contains methods for getting and setting whether two factor authentication is enabled for a user.
- **IUserPhoneNumberStore**
  The IUserPhoneNumberStore<TUser> interface defines the methods you implement to store user phone numbers. It contains methods for getting and setting the phone number and whether the phone number is confirmed.
- **IUserEmailStore**
  The IUserEmailStore<TUser> interface defines the methods you implement to store user email addresses. It contains methods for getting and setting the email address and whether the email is confirmed.
- **IUserLockoutStore**
  The IUserLockoutStore<TUser> interface defines the methods you implement to store information about locking an account. It contains methods for tracking failed access attempts and lockouts.
- **IQueryableUserStore**
  The IQueryableUserStore<TUser> interface defines the members you implement to provide a queryable user store.

You implement only the interfaces that are needed in your app. For example:

C#

```
public class UserStore : IUserStore<IdentityUser>,
                         IUserClaimStore<IdentityUser>,
                         IUserLoginStore<IdentityUser>,
                         IUserRoleStore<IdentityUser>,
                         IUserPasswordStore<IdentityUser>,
                         IUserSecurityStampStore<IdentityUser>
{
    // interface implementations not shown
}
```

IdentityUserClaim, IdentityUserLogin, and IdentityUserRole

The `Microsoft.AspNet.Identity.EntityFramework` namespace contains implementations of the IdentityUserClaim, IdentityUserLogin, and IdentityUserRole classes. If you are using these features, you may want to create your own versions of these classes and define the properties for your app. However, sometimes it's more efficient to not load these entities into memory when performing basic operations (such as adding or removing a user's claim). Instead, the backend store classes can execute these operations directly on the data source. For example, the `UserStore.GetClaimsAsync` method can call the `userClaimTable.FindByUserId(user.Id)` method to execute a query on that table directly and return a list of claims.

Customize the role class

When implementing a role storage provider, you can create a custom role type. It need not implement a particular interface, but it must have an `Id` and typically it will have a `Name` property.

The following is an example role class:

C#

```
using System;

namespace CustomIdentityProviderSample.CustomProvider
{
    public class ApplicationRole
    {
        public Guid Id { get; set; } = Guid.NewGuid();
        public string Name { get; set; }
    }
}
```

Customize the role store

You can create a `RoleStore` class that provides the methods for all data operations on roles. This class is equivalent to the RoleStore<TRole> class. In the `RoleStore` class, you implement the `IRoleStore<TRole>` and optionally the `IQueryableRoleStore<TRole>` interface.

- **IRoleStore<TRole>**
  The IRoleStore<TRole> interface defines the methods to implement in the role store class. It contains methods for creating, updating, deleting, and retrieving roles.
- **RoleStore<TRole>**
  To customize `RoleStore`, create a class that implements the `IRoleStore<TRole>` interface.

Reconfigure app to use a new storage provider

Once you have implemented a storage provider, you configure your app to use it. If your app used the default provider, replace it with your custom provider.

1. Remove the `Microsoft.AspNetCore.EntityFramework.Identity` NuGet package.
2. If the storage provider resides in a separate project or package, add a reference to it.
3. Replace all references to `Microsoft.AspNetCore.EntityFramework.Identity` with a using statement for the namespace of your storage provider.
4. Change the `AddIdentity` method to use the custom types. You can create your own extension methods for this purpose. See IdentityServiceCollectionExtensions for an example.
5. If you are using Roles, update the `RoleManager` to use your `RoleStore` class.
6. Update the connection string and credentials to your app's configuration.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

Example:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add identity types
```

```
builder.Services.AddIdentity<ApplicationUser, ApplicationRole>()
    .AddDefaultTokenProviders();

// Identity Services
builder.Services.AddTransient<IUserStore<ApplicationUser>, CustomUserStore>
();
builder.Services.AddTransient<IRoleStore<ApplicationRole>, CustomRoleStore>
();
var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddTransient<SqlConnection>(e => new
SqlConnection(connectionString));
builder.Services.AddTransient<DapperUsersTable>();

// additional configuration

builder.Services.AddRazorPages();

var app = builder.Build();
```

References

- Identity model customization
- Custom Storage Providers for ASP.NET 4.x Identity
- ASP.NET Core Identity : This repository includes links to community maintained store providers.
- View or download sample from GitHub .

Account confirmation and password recovery in ASP.NET Core

Article • 09/18/2024

By Rick Anderson, Ponant, and Joe Audette

This tutorial shows how to build an ASP.NET Core app with email confirmation and password reset. This tutorial is **not** a beginning topic. You should be familiar with:

- ASP.NET Core
- Authentication
- Entity Framework Core

For Blazor guidance, which adds to or supersedes the guidance in this article, see the following resources:

- Account confirmation and password recovery in ASP.NET Core Blazor
- Account confirmation and password recovery in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

Prerequisites

- .NET Core 6.0 SDK or later
- Successfully send email from a C# console app.

Create and test a web app with authentication

Run the following commands to create a web app with authentication.

.NET CLI

```
dotnet new webapp -au Individual -o WebPWrecover
cd WebPWrecover
dotnet run
```

Register user with simulated email confirmation

Run the app, select the **Register** link, and register a user. Once registered, you are redirected to the to `/Identity/Account/RegisterConfirmation` page which contains a link to simulate email confirmation:

- Select the `Click here to confirm your account` link.
- Select the **Login** link and sign-in with the same credentials.
- Select the `Hello YourEmail@provider.com!` link, which redirects to the `/Identity/Account/Manage/PersonalData` page.
- Select the **Personal data** tab on the left, and then select **Delete**.

The `Click here to confirm your account` link is displayed because an IEmailSender has not been implemented and registered with the dependency injection container. See the RegisterConfirmation source.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Configure an email provider

In this tutorial, SendGrid is used to send email. A SendGrid account and key is needed to send email. We recommend using SendGrid or another email service to send email rather than SMTP. SMTP is difficult to secure and set up correctly.

The SendGrid account may require adding a Sender.

Create a class to fetch the secure email key. For this sample, create `Services/AuthMessageSenderOptions.cs`:

C#

```
namespace WebPWrecover.Services;

public class AuthMessageSenderOptions
{
    public string? SendGridKey { get; set; }
}
```

Configure SendGrid user secrets

Set the `SendGridKey` with the secret-manager tool. For example:

.NET CLI

```
dotnet user-secrets set SendGridKey <key>

Successfully saved SendGridKey to the secret store.
```

On Windows, Secret Manager stores keys/value pairs in a `secrets.json` file in the `%APPDATA%/Microsoft/UserSecrets/<WebAppName-userSecretsId>` directory.

The contents of the `secrets.json` file aren't encrypted. The following markup shows the `secrets.json` file. The `SendGridKey` value has been removed.

JSON

```
{
  "SendGridKey": "<key removed>"
}
```

For more information, see the Options pattern and configuration.

Install SendGrid

This tutorial shows how to add email notifications through SendGrid, but other email providers can be used.

Install the `SendGrid` NuGet package:

Visual Studio

From the Package Manager Console, enter the following command:

PowerShell

```
Install-Package SendGrid
```

See Get Started with SendGrid for Free to register for a free SendGrid account.

Implement IEmailSender

To Implement `IEmailSender`, create `Services/EmailSender.cs` with code similar to the following:

C#

```
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.Extensions.Options;
using SendGrid;
using SendGrid.Helpers.Mail;

namespace WebPWrecover.Services;

public class EmailSender : IEmailSender
{
    private readonly ILogger _logger;

    public EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor,
                       ILogger<EmailSender> logger)
    {
        Options = optionsAccessor.Value;
        _logger = logger;
    }

    public AuthMessageSenderOptions Options { get; } //Set with Secret
Manager.

    public async Task SendEmailAsync(string toEmail, string subject, string
message)
    {
        if (string.IsNullOrEmpty(Options.SendGridKey))
        {
            throw new Exception("Null SendGridKey");
        }
        await Execute(Options.SendGridKey, subject, message, toEmail);
    }

    public async Task Execute(string apiKey, string subject, string message,
string toEmail)
    {
        var client = new SendGridClient(apiKey);
        var msg = new SendGridMessage()
        {
            From = new EmailAddress("Joe@contoso.com", "Password Recovery"),
            Subject = subject,
            PlainTextContent = message,
            HtmlContent = message
        };
        msg.AddTo(new EmailAddress(toEmail));

        // Disable click tracking.
        // See https://sendgrid.com/docs/User_Guide/Settings/tracking.html
        msg.SetClickTracking(false, false);
        var response = await client.SendEmailAsync(msg);
        _logger.LogInformation(response.IsSuccessStatusCode
                               ? $"Email to {toEmail} queued successfully!"
                               : $"Failure Email to {toEmail}");
```

```
    }
}
```

Configure app to support email

Add the following code to the `Program.cs` file:

- Add `EmailSender` as a transient service.
- Register the `AuthMessageSenderOptions` configuration instance.

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
```

```
app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Disable default account verification when Account.RegisterConfirmation has been scaffolded

This section only applies when `Account.RegisterConfirmation` is scaffolded. Skip this section if you have not scaffolded `Account.RegisterConfirmation`.

The user is redirected to the `Account.RegisterConfirmation` where they can select a link to have the account confirmed. The default `Account.RegisterConfirmation` is used ***only*** for testing, automatic account verification should be disabled in a production app.

To require a confirmed account and prevent immediate login at registration, set `DisplayConfirmAccountLink = false` in the scaffolded `/Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs` file:

C#

```
// Licensed to the .NET Foundation under one or more agreements.
// The .NET Foundation licenses this file to you under the MIT license.
#nullable disable

using System;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.WebUtilities;

namespace WebPWrecover.Areas.Identity.Pages.Account
{
    [AllowAnonymous]
    public class RegisterConfirmationModel : PageModel
    {
        private readonly UserManager<IdentityUser> _userManager;
        private readonly IEmailSender _sender;

        public RegisterConfirmationModel(UserManager<IdentityUser>
userManager, IEmailSender sender)
        {
            _userManager = userManager;
```

```
            _sender = sender;
        }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        public string Email { get; set; }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        public bool DisplayConfirmAccountLink { get; set; }

        /// <summary>
        ///     This API supports the ASP.NET Core Identity default UI
infrastructure and is not intended to be used
        ///     directly from your code. This API may change or be removed
in future releases.
        /// </summary>
        public string EmailConfirmationUrl { get; set; }

        public async Task<IActionResult> OnGetAsync(string email, string
returnUrl = null)
        {
            if (email == null)
            {
                return RedirectToPage("/Index");
            }
            returnUrl = returnUrl ?? Url.Content("~/");

            var user = await _userManager.FindByEmailAsync(email);
            if (user == null)
            {
                return NotFound($"Unable to load user with email
'{email}'.");
            }

            Email = email;
            // Once you add a real email sender, you should remove this code
that lets you confirm the account
            DisplayConfirmAccountLink = false;
            if (DisplayConfirmAccountLink)
            {
                var userId = await _userManager.GetUserIdAsync(user);
                var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                EmailConfirmationUrl = Url.Page(
```

```
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { area = "Identity", userId = userId, code =
code, returnUrl = returnUrl },
                    protocol: Request.Scheme);
            }

            return Page();
        }
    }
}
```

This step is only necessary when `Account.RegisterConfirmation` is scaffolded. The non-scaffolded RegisterConfirmation automatically detects when an IEmailSender has been implemented and registered with the dependency injection container.

Register, confirm email, and reset password

Run the web app, and test the account confirmation and password recovery flow.

- Run the app and register a new user
- Check your email for the account confirmation link. See Debug email if you don't get the email.
- Click the link to confirm your email.
- Sign in with your email and password.
- Sign out.

Test password reset

- If you're signed in, select **Logout**.
- Select the **Log in** link and select the **Forgot your password?** link.
- Enter the email you used to register the account.
- An email with a link to reset your password is sent. Check your email and click the link to reset your password. After your password has been successfully reset, you can sign in with your email and new password.

Resend email confirmation

Select the **Resend email confirmation** link on the **Login** page.

Change email and activity timeout

The default inactivity timeout is 14 days. The following code sets the inactivity timeout to 5 days:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

builder.Services.ConfigureApplicationCookie(o => {
    o.ExpireTimeSpan = TimeSpan.FromDays(5);
    o.SlidingExpiration = true;
});

var app = builder.Build();

// Code removed for brevity
```

Change all data protection token lifespans

The following code changes all data protection tokens timeout period to 3 hours:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
```

```
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

builder.Services.Configure<DataProtectionTokenProviderOptions>(o =>
       o.TokenLifespan = TimeSpan.FromHours(3));

var app = builder.Build();

// Code removed for brevity.
```

The built in Identity user tokens (see AspNetCore/src/Identity/Extensions.Core/src/TokenOptions.cs)have a one day timeout.

Change the email token lifespan

The default token lifespan of the Identity user tokens is one day. This section shows how to change the email token lifespan.

Add a custom DataProtectorTokenProvider<TUser> and DataProtectionTokenProviderOptions:

C#

```
public class CustomEmailConfirmationTokenProvider<TUser>
                                : DataProtectorTokenProvider<TUser> where
TUser : class
{
    public CustomEmailConfirmationTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<EmailConfirmationTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
                                    : base(dataProtectionProvider,
options, logger)
    {

    }
}
public class EmailConfirmationTokenProviderOptions :
DataProtectionTokenProviderOptions
```

```
{
    public EmailConfirmationTokenProviderOptions()
    {
        Name = "EmailDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(4);
    }
}
```

Add the custom provider to the service container:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.EntityFrameworkCore;
using WebPWrecover.Data;
using WebPWrecover.Services;
using WebPWrecover.TokenProviders;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(config =>
{
    config.SignIn.RequireConfirmedEmail = true;
    config.Tokens.ProviderMap.Add("CustomEmailConfirmation",
        new TokenProviderDescriptor(
            typeof(CustomEmailConfirmationTokenProvider<IdentityUser>)));
    config.Tokens.EmailConfirmationTokenProvider =
"CustomEmailConfirmation";
}).AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddTransient<CustomEmailConfirmationTokenProvider<IdentityU
ser>>();

builder.Services.AddRazorPages();

builder.Services.AddTransient<IEmailSender, EmailSender>();
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);

var app = builder.Build();

// Code removed for brevity.
```

Debug email

If you can't get email working:

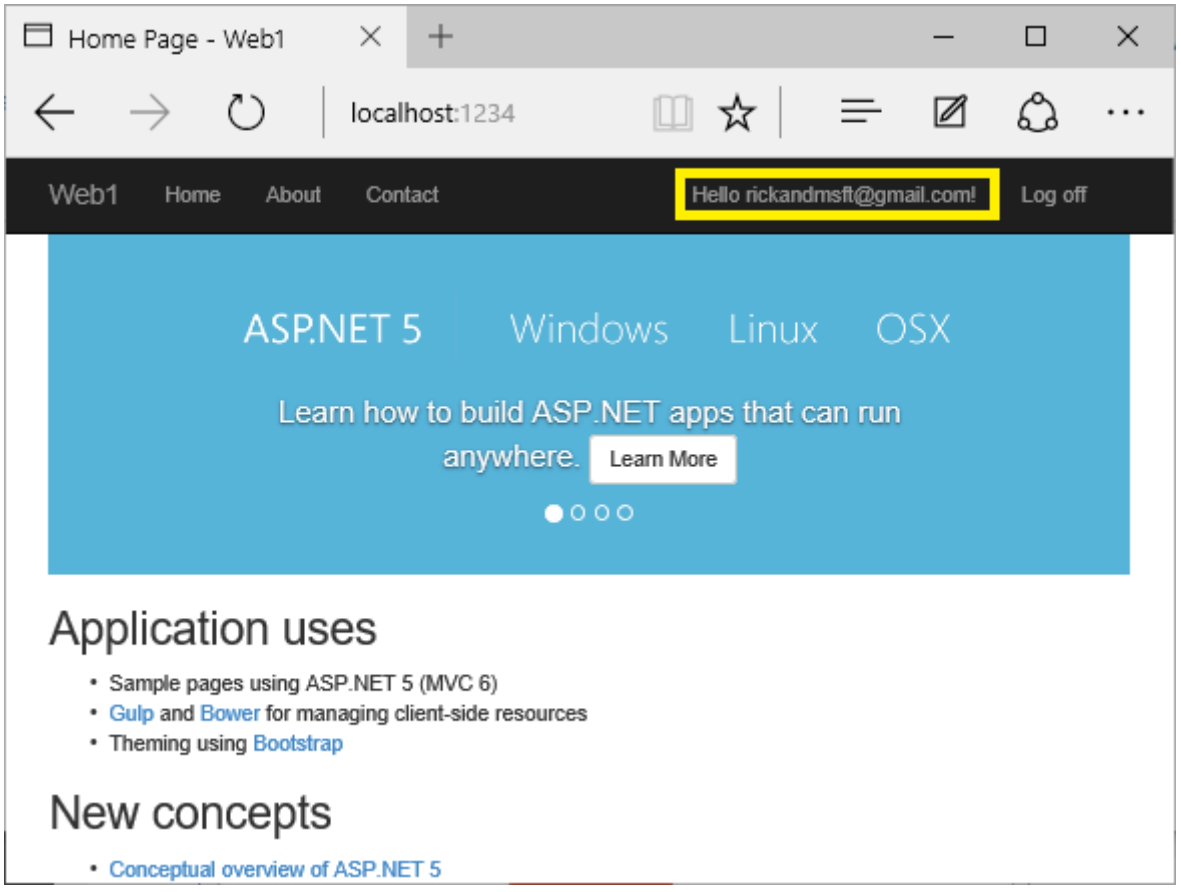
- Set a breakpoint in `EmailSender.Execute` to verify `SendGridClient.SendEmailAsync` is called.
- Create a console app to send email using similar code to `EmailSender.Execute`.
- Review the Email Activity page.
- Check your spam folder.
- Try another email alias on a different email provider (Microsoft, Yahoo, Gmail, etc.)
- Try sending to different email accounts.

A security best practice is to **not** use production secrets in test and development. If you publish the app to Azure, set the SendGrid secrets as application settings in the Azure Web App portal. The configuration system is set up to read keys from environment variables.

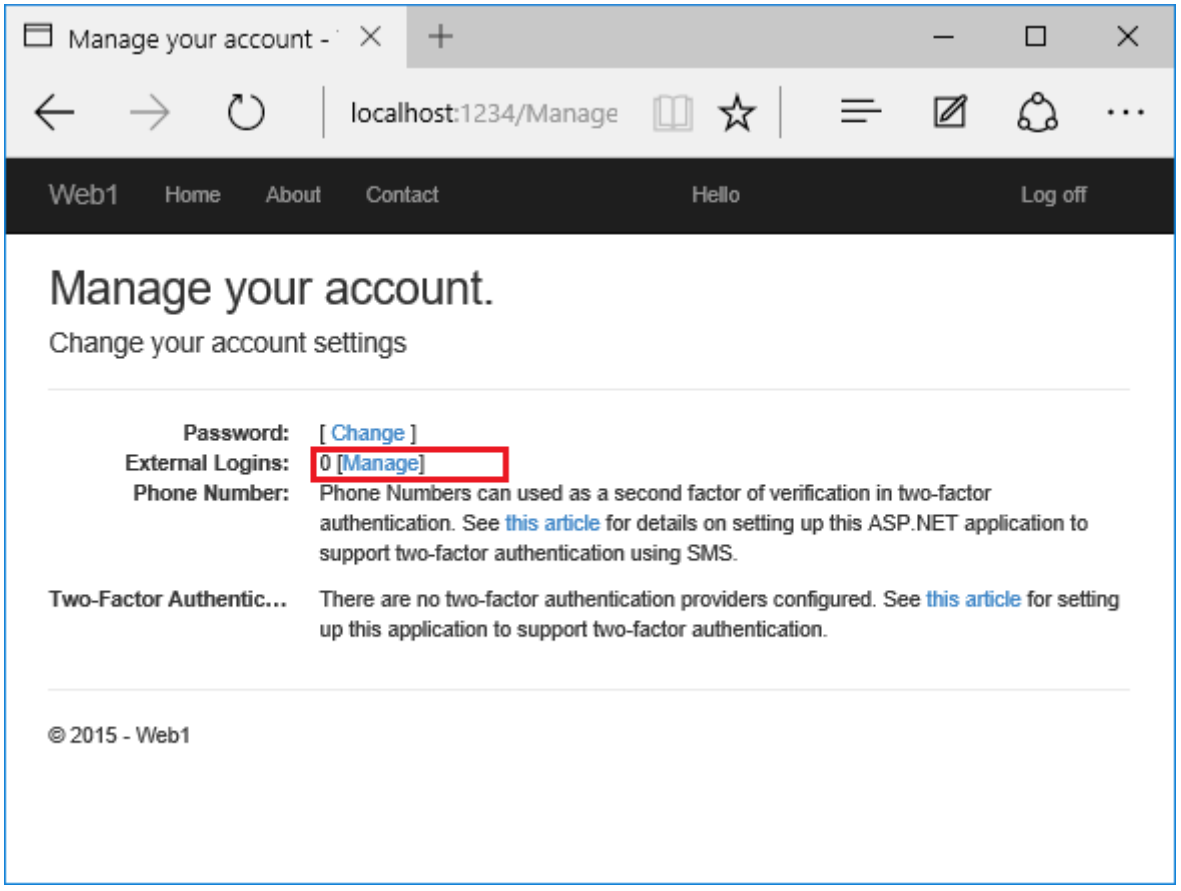
Combine social and local login accounts

To complete this section, you must first enable an external authentication provider. See Facebook, Google, and external provider authentication.

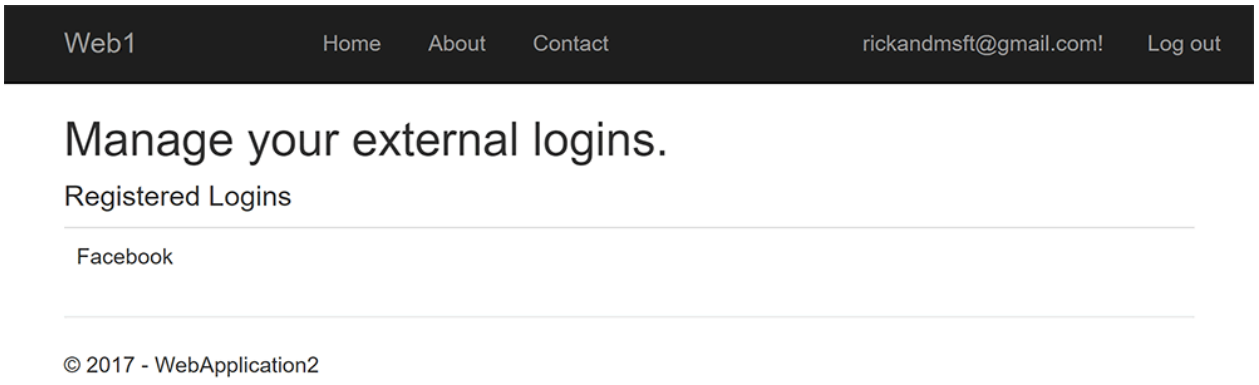
You can combine local and social accounts by clicking on your email link. In the following sequence, "RickAndMSFT@gmail.com" is first created as a local login; however, you can create the account as a social login first, then add a local login.



Click on the **Manage** link. Note the 0 external (social logins) associated with this account.



Click the link to another login service and accept the app requests. In the following image, Facebook is the external authentication provider:



The two accounts have been combined. You are able to sign in with either account. You might want your users to add local accounts in case their social login authentication service is down, or more likely they've lost access to their social account.

Enable account confirmation after a site has users

Enabling account confirmation on a site with users locks out all the existing users. Existing users are locked out because their accounts aren't confirmed. To work around existing user lockout, use one of the following approaches:

- Update the database to mark all existing users as being confirmed.
- Confirm existing users. For example, batch-send emails with confirmation links.

Enable QR code generation for TOTP authenticator apps in ASP.NET Core

Article • 12/11/2024

ASP.NET Core ships with support for authenticator applications for individual authentication. Two factor authentication (2FA) authenticator apps, using a Time-based One-time Password Algorithm (TOTP), are the industry recommended approach for 2FA. 2FA using TOTP is preferred to SMS 2FA. An authenticator app provides a 6 to 8 digit code which users must enter after confirming their username and password. Typically an authenticator app is installed on a smartphone.

> ⚠ **Warning**
>
> An ASP.NET Core TOTP code should be kept secret because it can be used to authenticate successfully multiple times before it expires.

The ASP.NET Core web app templates support authenticators but don't provide support for QR code generation. QR code generators ease the setup of 2FA. This document provides guidance for Razor Pages and MVC apps on how to add QR code generation to the 2FA configuration page. For guidance that applies to Blazor Web Apps, see Enable QR code generation for TOTP authenticator apps in an ASP.NET Core Blazor Web App. For guidance that applies to Blazor WebAssembly apps, see Enable QR code generation for TOTP authenticator apps in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

Two-factor authentication does not happen using an external authentication provider, such as Google or Facebook. External logins are protected by whatever mechanism the external login provider provides. Consider, for example, the Microsoft authentication provider requires a hardware key or another 2FA approach. If the default templates required 2FA for both the web app and the external authentication provider, then users would be required to satisfy two 2FA approaches. Requiring two 2FA approaches deviates from established security practices, which typically rely on a single, strong 2FA method for authentication.

Adding QR codes to the 2FA configuration page

These instructions use `qrcode.js` from the https://davidshimjs.github.io/qrcodejs/ repo.

- Download the qrcode.js JavaScript library to the `wwwroot\lib` folder in your project.
- Follow the instructions in Scaffold Identity to generate `/Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml`.
- In `/Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml`, locate the `Scripts` section at the end of the file:

CSHTML

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")
}
```

- Create a new JavaScript file called `qr.js` in `wwwroot/js` and add the following code to generate the QR Code:

JavaScript

```
window.addEventListener("load", () => {
  const uri = document.getElementById("qrCodeData").getAttribute('data-
url');
  new QRCode(document.getElementById("qrCode"),
    {
      text: uri,
      width: 150,
      height: 150
    });
});
```

- Update the `Scripts` section to add a reference to the `qrcode.js` library previously downloaded.
- Add the `qr.js` file with the call to generate the QR code:

CSHTML

```
@section Scripts {
    @await Html.PartialAsync("_ValidationScriptsPartial")

    <script type="text/javascript" src="~/lib/qrcode.js"></script>
    <script type="text/javascript" src="~/js/qr.js"></script>
}
```

- Delete the paragraph which links you to these instructions.

Run your app and ensure that you can scan the QR code and validate the code the authenticator proves.

Change the site name in the QR code

The site name in the QR code is taken from the project name you choose when initially creating your project. You can change it by looking for the `GenerateQrCodeUri(string email, string unformattedKey)` method in the `/Areas/Identity/Pages/Account/Manage/EnableAuthenticator.cshtml.cs`.

The default code from the template looks as follows:

```
C#

private string GenerateQrCodeUri(string email, string unformattedKey)
{
    return string.Format(
        AuthenticatorUriFormat,
        _urlEncoder.Encode("Razor Pages"),
        _urlEncoder.Encode(email),
        unformattedKey);
}
```

The second parameter in the call to `string.Format` is your site name, taken from your solution name. It can be changed to any value, but it must always be URL encoded.

Using a different QR Code library

You can replace the QR Code library with your preferred library. The HTML contains a `qrCode` element into which you can place a QR Code by whatever mechanism your library provides.

The correctly formatted URL for the QR Code is available in the:

- `AuthenticatorUri` property of the model.
- `data-url` property in the `qrCodeData` element.

TOTP client and server time skew

TOTP (Time-based One-Time Password) authentication depends on both the server and authenticator device having an accurate time. Tokens only last for 30 seconds. If TOTP

2FA logins are failing, check that the server time is accurate, and preferably synchronized to an accurate NTP service.

Facebook, Google, and external provider authentication in ASP.NET Core

Article • 09/19/2022

By Valeriy Novytskyy and Rick Anderson

This tutorial demonstrates how to build an ASP.NET Core app that enables users to sign in using OAuth 2.0 with credentials from external authentication providers.

Facebook, Twitter, Google, and Microsoft providers are covered in the following sections and use the starter project created in this article. Other providers are available in third-party packages such as OpenIddict, AspNet.Security.OAuth.Providers and AspNet.Security.OpenId.Providers.

Enabling users to sign in with their existing credentials:

- Is convenient for the users.
- Shifts many of the complexities of managing the sign-in process onto a third party.

Create a New ASP.NET Core Project

Visual Studio

- Select the **ASP.NET Core Web App** template. Select **OK**.
- In the **Authentication type** input, select **Individual Accounts**.

Apply migrations

- Run the app and select the **Register** link.
- Enter the email and password for the new account, and then select **Register**.
- Follow the instructions to apply migrations.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information

usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Use SecretManager to store tokens assigned by login providers

Social login providers assign **Application Id** and **Application Secret** tokens during the registration process. The exact token names vary by provider. These tokens represent the credentials your app uses to access their API. The tokens constitute the "user secrets" that can be linked to your app configuration with the help of Secret Manager. User secrets are a more secure alternative to storing the tokens in a configuration file, such as `appsettings.json`.

> **Important**
>
> Secret Manager is for development purposes only. You can store and protect Azure test and production secrets with the **Azure Key Vault configuration provider**.

Follow the steps in Safe storage of app secrets in development in ASP.NET Core topic to store tokens assigned by each login provider below.

Setup login providers required by your application

Use the following topics to configure your application to use the respective providers:

- Facebook instructions
- Twitter instructions
- Google instructions

- Microsoft instructions
- Other provider instructions

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods from AddAuthentication:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebApplication16.Data;

var builder = WebApplication.CreateBuilder(args);
var config = builder.Configuration;

var connectionString = config.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
                                    options.SignIn.RequireConfirmedAccount =
true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();
```

```
builder.Services.AddAuthentication()
    .AddGoogle(options =>
    {
        IConfigurationSection googleAuthNSection =
        config.GetSection("Authentication:Google");
        options.ClientId = googleAuthNSection["ClientId"];
        options.ClientSecret = googleAuthNSection["ClientSecret"];
    })
    .AddFacebook(options =>
    {
        IConfigurationSection FBAuthNSection =
        config.GetSection("Authentication:FB");
        options.ClientId = FBAuthNSection["ClientId"];
        options.ClientSecret = FBAuthNSection["ClientSecret"];
    })
    .AddMicrosoftAccount(microsoftOptions =>
    {
        microsoftOptions.ClientId =
config["Authentication:Microsoft:ClientId"];
        microsoftOptions.ClientSecret =
config["Authentication:Microsoft:ClientSecret"];
    })
    .AddTwitter(twitterOptions =>
    {
        twitterOptions.ConsumerKey =
config["Authentication:Twitter:ConsumerAPIKey"];
        twitterOptions.ConsumerSecret =
config["Authentication:Twitter:ConsumerSecret"];
        twitterOptions.RetrieveUserDetails = true;
    });
```

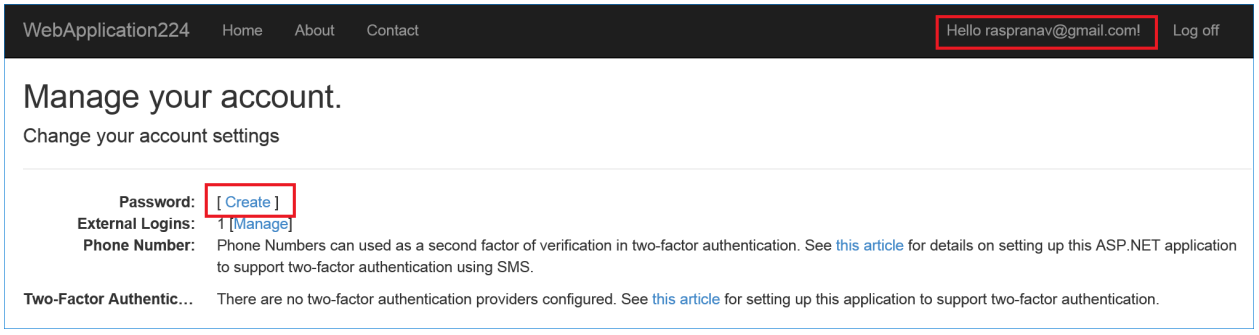
```
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Optionally set password

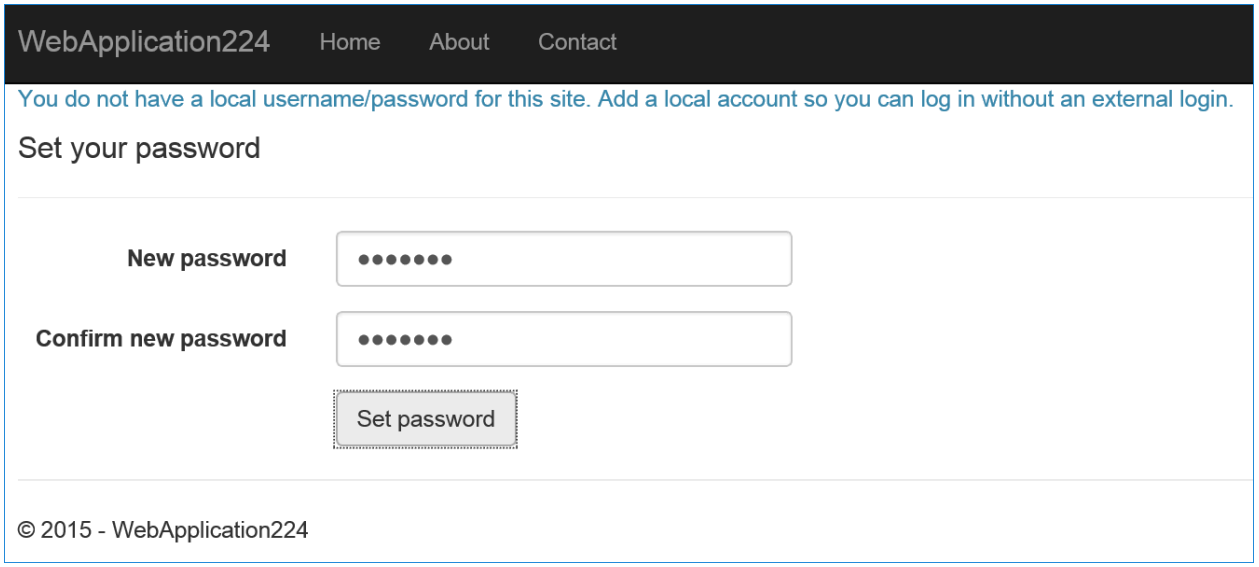
When you register with an external login provider, you don't have a password registered with the app. This alleviates you from creating and remembering a password for the site, but it also makes you dependent on the external login provider. If the external login provider is unavailable, you won't be able to sign in to the web site.

To create a password and sign in using your email that you set during the sign in process with external providers:

- Select the **Hello <email alias>** link at the top-right corner to navigate to the **Manage** view.



- Select **Create**



- Set a valid password and you can use this to sign in with your email.

Additional information

- Sign in with Apple Example Integration
- See this GitHub issue for information on how to customize the login buttons.

- Persist additional data about the user and their access and refresh tokens. For more information, see Persist additional claims and tokens from external providers in ASP.NET Core.

Google external login setup in ASP.NET Core

Article • 02/10/2023

By Valeriy Novytskyy and Rick Anderson

This tutorial shows you how to enable users to sign in with their Google account using the ASP.NET Core project created on the previous page.

Create the Google OAuth 2.0 Client ID and secret

- Follow the guidance in Integrating Google Sign-In into your web app (Google documentation).
- Go to Google API & Services.
- A **Project** must exist first, you may have to create one. Once a project is selected, enter the **Dashboard**.
- In the **Oauth consent screen** of the **Dashboard**:
  - Select **User Type - External** and **CREATE**.
  - In the **App information** dialog, Provide an **app name** for the app, **user support email**, and **developer contact information**.
  - Step through the **Scopes** step.
  - Step through the **Test users** step.
  - Review the **OAuth consent screen** and go back to the app **Dashboard**.
- In the **Credentials** tab of the application Dashboard, select **CREATE CREDENTIALS** > **OAuth client ID**.
- Select **Application type** > **Web application**, choose a **name**.
- In the **Authorized redirect URIs** section, select **ADD URI** to set the redirect URI. Example redirect URI: `https://localhost:{PORT}/signin-google`, where the `{PORT}` placeholder is the app's port.
- Select the **CREATE** button.
- Save the **Client ID** and **Client Secret** for use in the app's configuration.
- When deploying the site, either:

- Update the app's redirect URI in the **Google Console** to the app's deployed redirect URI.
- Create a new Google API registration in the **Google Console** for the production app with its production redirect URI.

Store the Google client ID and secret

Store sensitive settings such as the Google client ID and secret values with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret`:

.NET CLI

```
dotnet user-secrets set "Authentication:Google:ClientId" "<client-id>"
dotnet user-secrets set "Authentication:Google:ClientSecret" "<client-
secret>"
```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

You can manage your API credentials and usage in the API Console.

Configure Google authentication

Add the Microsoft.AspNetCore.Authentication.Google NuGet package to the app.

Add the Authentication service to the `Program`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddGoogle(googleOptions =>
    {
```

```
        googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
        googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];
    });
```

The call to AddIdentity configures the default scheme settings. The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Sign in with Google

- Run the app and select **Log in**. An option to sign in with Google appears.
- Select the **Google** button, which redirects to Google for authentication.
- After entering your Google credentials, you are redirected back to the web site.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

```
C#

services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

For more information on configuration options supported by Google authentication, see the GoogleOptions API reference . This can be used to request different information about the user.

Change the default callback URI

The URI segment `/signin-google` is set as the default callback of the Google authentication provider. You can change the default callback URI while configuring the Google authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the GoogleOptions class.

Troubleshooting

- If the sign-in doesn't work and you aren't getting any errors, switch to development mode to make the issue easier to debug.
- If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate results in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures Identity is configured.
- If the site database has not been created by applying the initial migration, you get *A database operation failed while processing the request* error. Select **Apply Migrations** to create the database, and refresh the page to continue past the error.
- HTTP 500 error after successfully authenticating the request by the OAuth 2.0 provider such as Google: See this GitHub issue .
- How to implement external authentication with Google for React and other SPA apps: See this GitHub issue .

Next steps

- This article showed how you can authenticate with Google. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish the app to Azure, reset the `ClientSecret` in the Google API Console.
- Set the `Authentication:Google:ClientId` and `Authentication:Google:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Facebook external login setup in ASP.NET Core

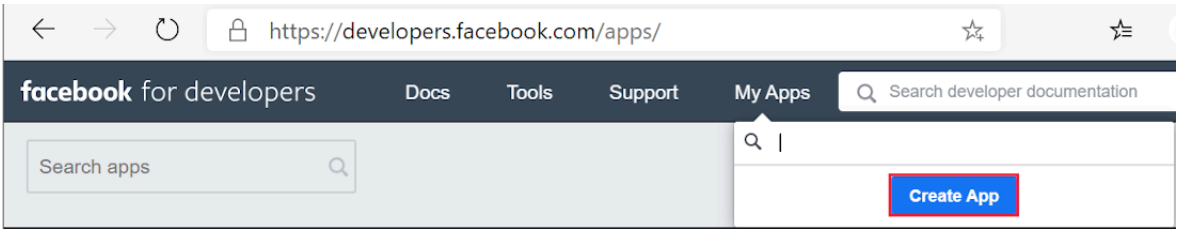
Article • 06/03/2022

By Valeriy Novytskyy and Rick Anderson

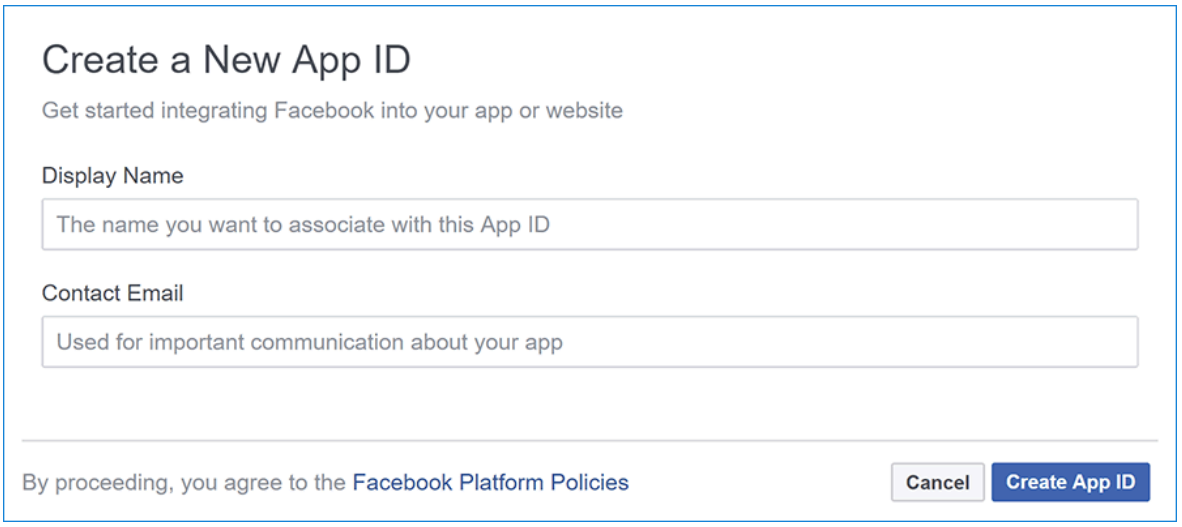
This tutorial with code examples shows how to enable your users to sign in with their Facebook account using a sample ASP.NET Core project created on the previous page. We start by creating a Facebook App ID by following the official steps.

Create the app in Facebook

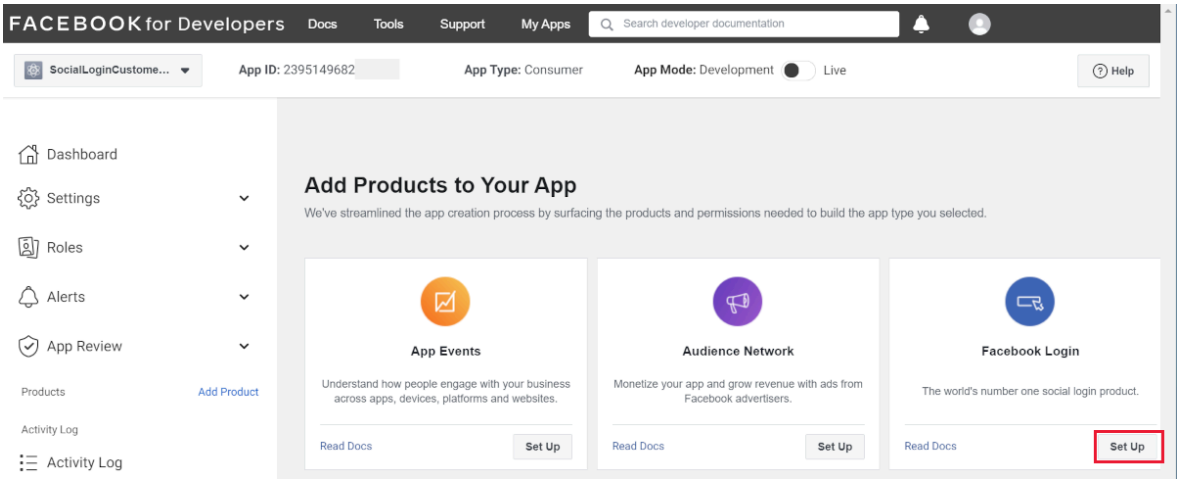
- Add the Microsoft.AspNetCore.Authentication.Facebook NuGet package to the project.
- Navigate to the Facebook Developers app page and sign in. If you don't already have a Facebook account, use the **Sign up for Facebook** link on the login page to create one. Once you have a Facebook account, follow the instructions to register as a Facebook Developer.
- From the **My Apps** menu select **Create App**. The **Create an app** form appears.



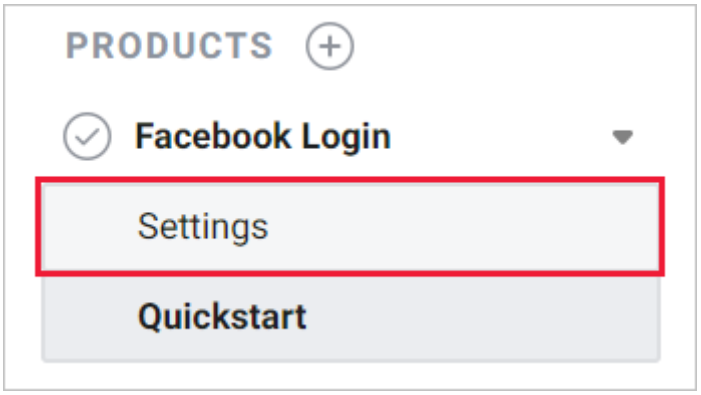
- Select an app type that best fits your project. For this project, select **Consumer**, and then **Next**. A new App ID is created.
- Fill out the form and tap the **Create App** button.



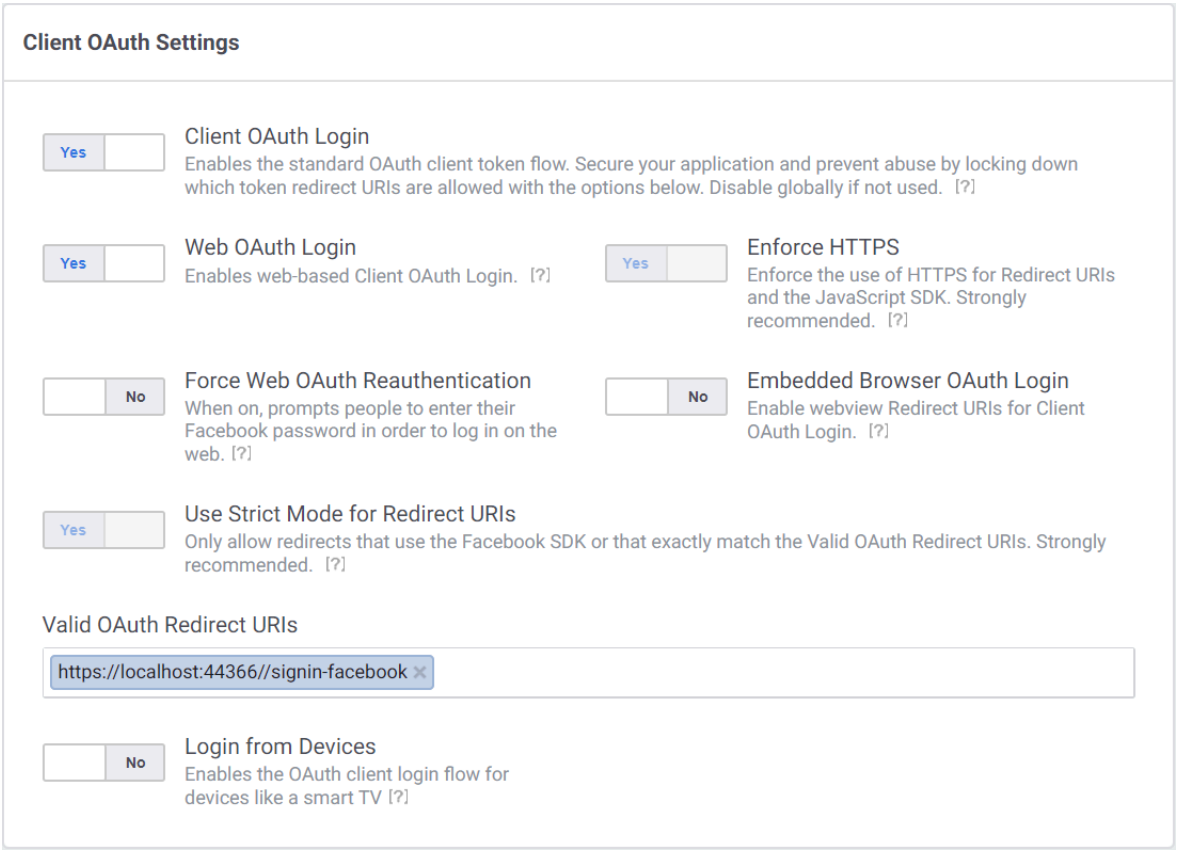
- On the **Add Products to Your App** page, select **Set Up** on the **Facebook Login** card.



- The **Quickstart** wizard launches with **Choose a Platform** as the first page. Bypass the wizard for now by clicking the **FaceBook Login Settings** link in the menu on the lower left:



- The **Client OAuth Settings** page is presented:



- Enter your development URI with */signin-facebook* appended into the **Valid OAuth Redirect URIs** field (for example: `https://localhost:44320/signin-facebook`). The Facebook authentication configured later in this tutorial will automatically handle requests at */signin-facebook* route to implement the OAuth flow.

> ⓘ **Note**
>
> The URI */signin-facebook* is set as the default callback of the Facebook authentication provider. You can change the default callback URI while configuring the Facebook authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the FacebookOptions class.

- Select **Save Changes**.
- Select **Settings** > **Basic** link in the left navigation.
- Make a note of your `App ID` and your `App Secret`. You will add both into your ASP.NET Core application in the next section:
- When deploying the site you need to revisit the **Facebook Login** setup page, and register a new public URI.

Store the Facebook app ID and secret

Store sensitive settings such as the Facebook app ID and secret values with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret`:

   .NET CLI

   ```
   dotnet user-secrets set "Authentication:Facebook:AppId" "<app-id>"
   dotnet user-secrets set "Authentication:Facebook:AppSecret" "<app-
   secret>"
   ```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

Configure Facebook Authentication

Add the Authentication service to the `Program`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddFacebook(facebookOptions =>
    {
        facebookOptions.AppId =
configuration["Authentication:Facebook:AppId"];
        facebookOptions.AppSecret =
configuration["Authentication:Facebook:AppSecret"];
    });
```

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default

authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Sign in with Facebook

- Run the app and select **Log in**.
- Under **Use another service to log in.**, select Facebook.
- You are redirected to **Facebook** for authentication.
- Enter your Facebook credentials.
- You are redirected back to your site where you can set your email.

You are now logged in using your Facebook credentials:

React to cancel authorize external sign-in

AccessDeniedPath can provide a redirect path to the user agent when the user doesn't approve the requested authorization demand.

The following code sets the `AccessDeniedPath` to `"/AccessDeniedPathInfo"`:

C#

```
services.AddAuthentication().AddFacebook(options =>
{
    options.AppId = Configuration["Authentication:Facebook:AppId"];
    options.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
    options.AccessDeniedPath = "/AccessDeniedPathInfo";
});
```

We recommend the `AccessDeniedPath` page contains the following information:

- Remote authentication was canceled.
- This app requires authentication.
- To try sign-in again, select the Login link.

Test AccessDeniedPath

- Navigate to facebook.com
- If you are signed in, you must sign out.

- Run the app and select Facebook sign-in.
- Select **Not now**. You are redirected to the specified `AccessDeniedPath` page.

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

C#

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

For more information on configuration options supported by Facebook authentication, see the FacebookOptions API reference. Configuration options can be used to:

- Request different information about the user.
- Add query string arguments to customize the login experience.

Troubleshooting

- **ASP.NET Core 2.x only:** If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this tutorial ensures that this is done.
- If the site database has not been created by applying the initial migration, you get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Facebook. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish your web site to Azure web app, you should reset the `AppSecret` in the Facebook developer portal.
- Set the `Authentication:Facebook:AppId` and `Authentication:Facebook:AppSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Microsoft Account external login setup with ASP.NET Core

Article • 06/03/2022

By Valeriy Novytskyy and Rick Anderson

This sample shows you how to enable users to sign in with their work, school, or personal Microsoft account using the ASP.NET Core 6.0 project created on the previous page.

Create the app in Microsoft Developer Portal

- Add the Microsoft.AspNetCore.Authentication.MicrosoftAccount NuGet package to the project.
- Navigate to the Azure portal - App registrations page and create or sign into a Microsoft account:

If you don't have a Microsoft account, select **Create one**. After signing in, you are redirected to the **App registrations** page:

- Select **New registration**
- Enter a **Name**.
- Select an option for **Supported account types**.
  - The `MicrosoftAccount` package supports App Registrations created using "Accounts in any organizational directory" or "Accounts in any organizational directory and Microsoft accounts" options by default.
  - To use other options, set `AuthorizationEndpoint` and `TokenEndpoint` members of `MicrosoftAccountOptions` used to initialize the Microsoft Account authentication to the URLs displayed on **Endpoints** page of the App Registration after it is created (available by clicking Endpoints on the **Overview** page).
- Under **Redirect URI**, enter your development URL with `/signin-microsoft` appended. For example, `https://localhost:5001/signin-microsoft`. The Microsoft authentication scheme configured later in this sample will automatically handle requests at `/signin-microsoft` route to implement the OAuth flow.
- Select **Register**

Create client secret

- In the left pane, select **Certificates & secrets**.
- Under **Client secrets**, select **New client secret**
  - Add a description for the client secret.
  - Select the **Add** button.
- Under **Client secrets**, copy the value of the client secret.

The URI segment `/signin-microsoft` is set as the default callback of the Microsoft authentication provider. You can change the default callback URI while configuring the Microsoft authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the MicrosoftAccountOptions class.

Store the Microsoft client ID and secret

Store sensitive settings such as the Microsoft **Application (client) ID** found on the **Overview** page of the App Registration and **Client Secret** you created on the **Certificates & secrets page** with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secret keys `Authentication:Microsoft:ClientId` and `Authentication:Microsoft:ClientSecret`:

   .NET CLI

   ```
   dotnet user-secrets set "Authentication:Microsoft:ClientId" "<client-id>"
   dotnet user-secrets set "Authentication:Microsoft:ClientSecret" "<client-secret>"
   ```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. `__`, the double underscore, is:

- Supported by all platforms. For example, the `:` separator is not supported by Bash, but `__` is.
- Automatically replaced by a `:`

Configure Microsoft Account Authentication

Add the Authentication service to the `Program`:

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddMicrosoftAccount(microsoftOptions =>
    {
        microsoftOptions.ClientId =
configuration["Authentication:Microsoft:ClientId"];
        microsoftOptions.ClientSecret =
configuration["Authentication:Microsoft:ClientSecret"];
    });
```

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

For more information about configuration options supported by Microsoft Account authentication, see the MicrosoftAccountOptions API reference. This can be used to request different information about the user.

Sign in with Microsoft Account

- Run the app and select **Log in**. An option to sign in with Microsoft appears.
- Select to sign in with Microsoft. You are redirected to Microsoft for authentication. After signing in with your Microsoft Account, you will be prompted to let the app access your info:
- Select **Yes**. You are redirected back to the web site where you can set your email.

You are now logged in using your Microsoft credentials.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

C#

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Troubleshooting

- If the Microsoft Account provider redirects you to a sign in error page, note the error title and description query string parameters directly following the `#` (hashtag) in the Uri.

  Although the error message seems to indicate a problem with Microsoft authentication, the most common cause is your application Uri not matching any of the **Redirect URIs** specified for the **Web** platform.

- If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this sample ensures that this is done.

- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Microsoft. You can follow a similar approach to authenticate with other providers listed on the previous page.
- Once you publish your web site to Azure web app, create a new client secrets in the Microsoft Developer Portal.
- Set the `Authentication:Microsoft:ClientId` and `Authentication:Microsoft:ClientSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

Twitter external sign-in setup with ASP.NET Core

Article • 06/03/2022

By Valeriy Novytskyy and Rick Anderson

This sample shows how to enable users to sign in with their Twitter account using a sample ASP.NET Core project created on the previous page.

> **Note**
>
> The Microsoft.AspNetCore.Authentication.Twitter package described below uses the OAuth 1.0 APIs provided by Twitter. Twitter has since added OAuth 2.0 APIs with a different set of functionality. The OpenIddict and AspNet.Security.OAuth.Twitter packages are community implementations that use the new OAuth 2.0 APIs.

Create the app in Twitter

- Add the Microsoft.AspNetCore.Authentication.Twitter NuGet package to the project.
- Navigate to twitter developer portal Dashboard and sign in. If you don't already have a Twitter account, use the **Sign up now** link to create one.
- If you don't have a project, create one.
- Select + **Add app**. Fill out the **App name** then record the generated API Key, API Key Secret and Bearer Token. These will be needed later.
- In the **App Settings** page, select **Edit** in the **Authentication settings** section, then:
  - Enable 3-legged OAuth
  - Request email address from users
  - Fill out the required fields and select **Save**

> **Note**

Microsoft.AspNetCore.Identity requires users to have an email address by default. For Callback URLs during development, use `https://localhost:{PORT}/signin-twitter`, where the `{PORT}` placeholder is the app's port.

> **Note**
>
> The URI segment `/signin-twitter` is set as the default callback of the Twitter authentication provider. You can change the default callback URI while configuring the Twitter authentication middleware via the inherited RemoteAuthenticationOptions.CallbackPath property of the TwitterOptions class.

Store the Twitter consumer API key and secret

Store sensitive settings such as the Twitter consumer API key and secret with Secret Manager. For this sample, use the following steps:

1. Initialize the project for secret storage per the instructions at Enable secret storage.
2. Store the sensitive settings in the local secret store with the secrets keys `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret`:

.NET CLI

```
dotnet user-secrets set "Authentication:Twitter:ConsumerAPIKey" "
<consumer-api-key>"
dotnet user-secrets set "Authentication:Twitter:ConsumerSecret" "
<consumer-secret>"
```

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

These tokens can be found on the **Keys and Access Tokens** tab after creating a new Twitter application:

Configure Twitter Authentication

C#

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;

services.AddAuthentication().AddTwitter(twitterOptions =>
    {
        twitterOptions.ConsumerKey =
configuration["Authentication:Twitter:ConsumerAPIKey"];
        twitterOptions.ConsumerSecret =
configuration["Authentication:Twitter:ConsumerSecret"];
    });
```

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

C#

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

For more information on configuration options supported by Twitter authentication, see the TwitterOptions API reference. This can be used to request different information about the user.

Sign in with Twitter

Run the app and select **Log in**. An option to sign in with Twitter appears:

Selecting **Twitter** redirects to Twitter for authentication:

After entering your Twitter credentials, you are redirected back to the web site where you can set your email.

You are now logged in using your Twitter credentials:

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Troubleshooting

- **ASP.NET Core 2.x only:** If Identity isn't configured by calling `services.AddIdentity` in `ConfigureServices`, attempting to authenticate will result in *ArgumentException: The 'SignInScheme' option must be provided*. The project template used in this sample ensures Identity is configured.
- If the site database has not been created by applying the initial migration, you will get *A database operation failed while processing the request* error. Tap **Apply Migrations** to create the database and refresh to continue past the error.

Next steps

- This article showed how you can authenticate with Twitter. You can follow a similar approach to authenticate with other providers listed on the previous page.

- Once you publish your web site to Azure web app, you should reset the `ConsumerSecret` in the Twitter developer portal.

- Set the `Authentication:Twitter:ConsumerKey` and `Authentication:Twitter:ConsumerSecret` as application settings in the Azure portal. The configuration system is set up to read keys from environment variables.

External OAuth authentication providers

Article • 05/22/2024

By Rick Anderson, Pranav Rastogi, and Valeriy Novytskyy

The following list includes common external OAuth authentication providers that work with ASP.NET Core apps. Third-party NuGet packages, such as the ones maintained by OpenIddict or aspnet-contrib, can be used to complement the authentication providers implemented by the ASP.NET Core team.

- LinkedIn
- Instagram
- Reddit (Instructions)
- Github (Instructions)
- Yahoo (Instructions)
- Tumblr (Instructions)
- Pinterest (Instructions)
- Pocket (Instructions)
- Flickr (Instructions)
- Dribbble (Instructions)
- Vimeo (Instructions)
- SoundCloud (Instructions)
- VK (Instructions)

Multiple authentication providers

When the app requires multiple providers, chain the provider extension methods behind AddAuthentication:

C#

```
services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions => { ... })
    .AddGoogle(googleOptions => { ... })
```

```
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

Persist additional claims and tokens from external providers in ASP.NET Core

Article • 09/07/2023

An ASP.NET Core app can establish additional claims and tokens from external authentication providers, such as Facebook, Google, Microsoft, and Twitter. Each provider reveals different information about users on its platform, but the pattern for receiving and transforming user data into additional claims is the same.

Prerequisites

Decide which external authentication providers to support in the app. For each provider, register the app and obtain a client ID and client secret. For more information, see Facebook and Google authentication in ASP.NET Core. The sample app uses the Google authentication provider.

Set the client ID and client secret

The OAuth authentication provider establishes a trust relationship with an app using a client ID and client secret. Client ID and client secret values are created for the app by the external authentication provider when the app is registered with the provider. Each external provider that the app uses must be configured independently with the provider's client ID and client secret. For more information, see the external authentication provider topics that apply:

- Facebook authentication
- Google authentication
- Microsoft authentication
- Twitter authentication
- Other authentication providers
- OpenIdConnect

Optional claims sent in the ID or access token from the authentication provider are usually configured in the provider's online portal. For example, Microsoft Entra ID permits assigning optional claims to the app's ID token in the app registration's **Token configuration** blade. For more information, see How to: Provide optional claims to your app (Azure documentation). For other providers, consult their external documentation sets.

The sample app configures the Google authentication provider with a client ID and client secret provided by Google:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebGoogOauth.Data;

var builder = WebApplication.CreateBuilder(args);
var configuration = builder.Configuration;

builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;

    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
                                    options.SignIn.RequireConfirmedAccount =
true)
```

```
.AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

// Remaining code removed for brevity.
```

Establish the authentication scope

Specify the list of permissions to retrieve from the provider by specifying the Scope. Authentication scopes for common external providers appear in the following table.

Expand table

Provider	Scope
Facebook	`https://www.facebook.com/dialog/oauth`
Google	`profile`, `email`, `openid`
Microsoft	`https://login.microsoftonline.com/common/oauth2/v2.0/authorize`
Twitter	`https://api.twitter.com/oauth/authenticate`

In the sample app, Google's `profile`, `email`, and `openid` scopes are automatically added by the framework when AddGoogle is called on the AuthenticationBuilder. If the app requires additional scopes, add them to the options. In the following example, the Google `https://www.googleapis.com/auth/user.birthday.read` scope is added to retrieve a user's birthday:

C#

```
options.Scope.Add("https://www.googleapis.com/auth/user.birthday.read");
```

Map user data keys and create claims

In the provider's options, specify a MapJsonKey or MapJsonSubKey for each key or subkey in the external provider's JSON user data for the app identity to read on sign in. For more information on claim types, see ClaimTypes.

The sample app creates locale (`urn:google:locale`) and picture (`urn:google:picture`) claims from the `locale` and `picture` keys in Google user data:

```
```

C#

```
builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;

    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

In `Microsoft.AspNetCore.Identity.UI.Pages.Account.Internal.ExternalLoginModel.OnPostConfirmationAsync`, an IdentityUser (`ApplicationUser`) is signed into the app with SignInAsync. During the sign in process, the UserManager<TUser> can store an `ApplicationUser` claims for user data available from the Principal.

In the sample app, `OnPostConfirmationAsync` (`Account/ExternalLogin.cshtml.cs`) establishes the locale (`urn:google:locale`) and picture (`urn:google:picture`) claims for the signed in `ApplicationUser`, including a claim for GivenName:

C#

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl =
null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
```

```
    var info = await _signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ErrorMessage = "Error loading external login information during
confirmation.";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.Email,
CancellationToken.None);
        await _emailStore.SetEmailAsync(user, Input.Email,
CancellationToken.None);

        var result = await _userManager.CreateAsync(user);
        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);
            if (result.Succeeded)
            {
                _logger.LogInformation("User created an account using {Name}
provider.", info.LoginProvider);

                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type ==
ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                // using Microsoft.AspNetCore.Authentication;
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
```

```
                props.IsPersistent = false;

                var userId = await _userManager.GetUserIdAsync(user);
                var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                var callbackUrl = Url.Page(
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { area = "Identity", userId = userId, code =
code },
                    protocol: Request.Scheme);

                await _emailSender.SendEmailAsync(Input.Email, "Confirm your
email",
                    $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                // If account confirmation is required, we need to show the
link if we don't have a real email sender
                if (_userManager.Options.SignIn.RequireConfirmedAccount)
                {
                    return RedirectToPage("./RegisterConfirmation", new {
Email = Input.Email });
                }

                await _signInManager.SignInAsync(user, props,
info.LoginProvider);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }

    ProviderDisplayName = info.ProviderDisplayName;
    ReturnUrl = returnUrl;
    return Page();
}
```

By default, a user's claims are stored in the authentication cookie. If the authentication cookie is too large, it can cause the app to fail because:

- The browser detects that the cookie header is too long.
- The overall size of the request is too large.

If a large amount of user data is required for processing user requests:

- Limit the number and size of user claims for request processing to only what the app requires.
- Use a custom ITicketStore for the Cookie Authentication Middleware's SessionStore to store identity across requests. Preserve large quantities of identity information on the server while only sending a small session identifier key to the client.

Save the access token

SaveTokens defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. `SaveTokens` is set to `false` by default to reduce the size of the final authentication cookie.

The sample app sets the value of `SaveTokens` to `true` in GoogleOptions:

C#

```
builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;

    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

When `OnPostConfirmationAsync` executes, store the access token (ExternalLoginInfo.AuthenticationTokens) from the external provider in the `ApplicationUser`'s `AuthenticationProperties`.

The sample app saves the access token in `OnPostConfirmationAsync` (new user registration) and `OnGetCallbackAsync` (previously registered user) in `Account/ExternalLogin.cshtml.cs`:

C#

```
public async Task<IActionResult> OnPostConfirmationAsync(string returnUrl =
null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    // Get the information about the user from the external login provider
    var info = await _signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ErrorMessage = "Error loading external login information during
confirmation.";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    if (ModelState.IsValid)
    {
        var user = CreateUser();

        await _userStore.SetUserNameAsync(user, Input.Email,
CancellationToken.None);
        await _emailStore.SetEmailAsync(user, Input.Email,
CancellationToken.None);

        var result = await _userManager.CreateAsync(user);
        if (result.Succeeded)
        {
            result = await _userManager.AddLoginAsync(user, info);
            if (result.Succeeded)
            {
                _logger.LogInformation("User created an account using {Name}
provider.", info.LoginProvider);

                // If they exist, add claims to the user for:
                //    Given (first) name
                //    Locale
                //    Picture
                if (info.Principal.HasClaim(c => c.Type ==
ClaimTypes.GivenName))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst(ClaimTypes.GivenName));
                }
```

```
                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:locale"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:locale"));
                }

                if (info.Principal.HasClaim(c => c.Type ==
"urn:google:picture"))
                {
                    await _userManager.AddClaimAsync(user,
                        info.Principal.FindFirst("urn:google:picture"));
                }

                // Include the access token in the properties
                // using Microsoft.AspNetCore.Authentication;
                var props = new AuthenticationProperties();
                props.StoreTokens(info.AuthenticationTokens);
                props.IsPersistent = false;

                var userId = await _userManager.GetUserIdAsync(user);
                var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
                code =
WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
                var callbackUrl = Url.Page(
                    "/Account/ConfirmEmail",
                    pageHandler: null,
                    values: new { area = "Identity", userId = userId, code =
code },
                    protocol: Request.Scheme);

                await _emailSender.SendEmailAsync(Input.Email, "Confirm your
email",
                    $"Please confirm your account by <a
href='{HtmlEncoder.Default.Encode(callbackUrl)}'>clicking here</a>.");

                // If account confirmation is required, we need to show the
link if we don't have a real email sender
                if (_userManager.Options.SignIn.RequireConfirmedAccount)
                {
                    return RedirectToPage("./RegisterConfirmation", new {
Email = Input.Email });
                }

                await _signInManager.SignInAsync(user, props,
info.LoginProvider);
                return LocalRedirect(returnUrl);
            }
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(string.Empty, error.Description);
        }
    }
```

```
    ProviderDisplayName = info.ProviderDisplayName;
    ReturnUrl = returnUrl;
    return Page();
}
```

> **Note**
>
> For information on passing tokens to the Razor components of a server-side Blazor app, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

How to add additional custom tokens

To demonstrate how to add a custom token, which is stored as part of `SaveTokens`, the sample app adds an AuthenticationToken with the current DateTime for an AuthenticationToken.Name of `TicketCreated`:

C#

```
builder.Services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId =
configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];

    googleOptions.ClaimActions.MapJsonKey("urn:google:picture", "picture",
"url");
    googleOptions.ClaimActions.MapJsonKey("urn:google:locale", "locale",
"string");

    googleOptions.SaveTokens = true;
```

```
    googleOptions.Events.OnCreatingTicket = ctx =>
    {
        List<AuthenticationToken> tokens =
ctx.Properties.GetTokens().ToList();

        tokens.Add(new AuthenticationToken()
        {
            Name = "TicketCreated",
            Value = DateTime.UtcNow.ToString()
        });

        ctx.Properties.StoreTokens(tokens);

        return Task.CompletedTask;
    };
});
```

Create and add claims

The framework provides common actions and extension methods for creating and adding claims to the collection. For more information, see the ClaimActionCollectionMapExtensions and ClaimActionCollectionUniqueExtensions.

Users can define custom actions by deriving from ClaimAction and implementing the abstract Run method.

For more information, see Microsoft.AspNetCore.Authentication.OAuth.Claims.

Add and update user claims

Claims are copied from external providers to the user database on first registration, not on sign in. If additional claims are enabled in an app after a user registers to use the app, call SignInManager.RefreshSignInAsync on a user to force the generation of a new authentication cookie.

In the Development environment working with test user accounts, delete and recreate the user account. For production systems, new claims added to the app can be backfilled into user accounts. After scaffolding the ExternalLogin page into the app at `Areas/Pages/Identity/Account/Manage`, add the following code to the `ExternalLoginModel` in the `ExternalLogin.cshtml.cs` file.

Add a dictionary of added claims. Use the dictionary keys to hold the claim types, and use the values to hold a default value. Add the following line to the top of the class. The following example assumes that one claim is added for the user's Google picture with a generic headshot image as the default value:

C#

```
private readonly IReadOnlyDictionary<string, string> _claimsToSync =
     new Dictionary<string, string>()
     {
             { "urn:google:picture", "https://localhost:5001/headshot.png"
},
     };
```

Replace the default code of the `OnGetCallbackAsync` method with the following code. The code loops through the claims dictionary. Claims are added (backfilled) or updated for each user. When claims are added or updated, the user sign-in is refreshed using the SignInManager<TUser>, preserving the existing authentication properties (`AuthenticationProperties`).

C#

```
private readonly IReadOnlyDictionary<string, string> _claimsToSync =
     new Dictionary<string, string>()
     {
             { "urn:google:picture", "https://localhost:5001/headshot.png"
},
     };

public async Task<IActionResult> OnGetCallbackAsync(string returnUrl = null,
string remoteError = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    if (remoteError != null)
    {
        ErrorMessage = $"Error from external provider: {remoteError}";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }
    var info = await _signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ErrorMessage = "Error loading external login information.";
        return RedirectToPage("./Login", new { ReturnUrl = returnUrl });
    }

    // Sign in the user with this external login provider if the user
already has a login.
    var result = await
_signInManager.ExternalLoginSignInAsync(info.LoginProvider,
info.ProviderKey, isPersistent: false, bypassTwoFactor: true);
    if (result.Succeeded)
    {
        _logger.LogInformation("{Name} logged in with {LoginProvider}
provider.", info.Principal.Identity.Name, info.LoginProvider);
```

```
        if (_claimsToSync.Count > 0)
        {
            var user = await
_userManager.FindByLoginAsync(info.LoginProvider,
                info.ProviderKey);
            var userClaims = await _userManager.GetClaimsAsync(user);
            bool refreshSignIn = false;

            foreach (var addedClaim in _claimsToSync)
            {
                var userClaim = userClaims
                    .FirstOrDefault(c => c.Type == addedClaim.Key);

                if (info.Principal.HasClaim(c => c.Type == addedClaim.Key))
                {
                    var externalClaim =
info.Principal.FindFirst(addedClaim.Key);

                    if (userClaim == null)
                    {
                        await _userManager.AddClaimAsync(user,
                            new Claim(addedClaim.Key, externalClaim.Value));
                        refreshSignIn = true;
                    }
                    else if (userClaim.Value != externalClaim.Value)
                    {
                        await _userManager
                            .ReplaceClaimAsync(user, userClaim,
externalClaim);
                        refreshSignIn = true;
                    }
                }
                else if (userClaim == null)
                {
                    // Fill with a default value
                    await _userManager.AddClaimAsync(user, new
Claim(addedClaim.Key,
                        addedClaim.Value));
                    refreshSignIn = true;
                }
            }

            if (refreshSignIn)
            {
                await _signInManager.RefreshSignInAsync(user);
            }
        }

        return LocalRedirect(returnUrl);
    }
    if (result.IsLockedOut)
    {
        return RedirectToPage("./Lockout");
    }
    else
```

```
    {
        // If the user does not have an account, then ask the user to create
an account.
        ReturnUrl = returnUrl;
        ProviderDisplayName = info.ProviderDisplayName;
        if (info.Principal.HasClaim(c => c.Type == ClaimTypes.Email))
        {
            Input = new InputModel
            {
                Email = info.Principal.FindFirstValue(ClaimTypes.Email)
            };
        }
        return Page();
    }
}
```

A similar approach is taken when claims change while a user is signed in but a backfill step isn't required. To update a user's claims, call the following on the user:

- UserManager.ReplaceClaimAsync on the user for claims stored in the identity database.
- SignInManager.RefreshSignInAsync on the user to force the generation of a new authentication cookie.

Remove claim actions and claims

ClaimActionCollection.Remove(String) removes all claim actions for the given ClaimType from the collection. ClaimActionCollectionMapExtensions.DeleteClaim(ClaimActionCollection, String) deletes a claim of the given ClaimType from the identity. DeleteClaim is primarily used with OpenID Connect (OIDC) to remove protocol-generated claims.

Sample app output

Run the sample app and select the **MyClaims** link:

text

```
User Claims

http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier
    9b342344f-7aab-43c2-1ac1-ba75912ca999
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name
    someone@gmail.com
AspNet.Identity.SecurityStamp
    7D4312MOWRYYBFI1KXRPHGOSTBVWSFDE
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname
```

```
    Judy
urn:google:locale
    en
urn:google:picture
    https://lh4.googleusercontent.com/-XXXXXX/XXXXXX/XXXXXX/XXXXXX/photo.jpg

Authentication Properties

.Token.access_token
    yc23.AlvoZqz56...1lxltXV7D-ZWP9
.Token.token_type
    Bearer
.Token.expires_at
    2019-04-11T22:14:51.0000000+00:00
.Token.TicketCreated
    4/11/2019 9:14:52 PM
.TokenNames
    access_token;token_type;expires_at;TicketCreated
.persistent
.issued
    Thu, 11 Apr 2019 20:51:06 GMT
.expires
    Thu, 25 Apr 2019 20:51:06 GMT
```

Forward request information with a proxy or load balancer

If the app is deployed behind a proxy server or load balancer, some of the original request information might be forwarded to the app in request headers. This information usually includes the secure request scheme (`https`), host, and client IP address. Apps don't automatically read these request headers to discover and use the original request information.

The scheme is used in link generation that affects the authentication flow with external providers. Losing the secure scheme (`https`) results in the app generating incorrect insecure redirect URLs.

Use Forwarded Headers Middleware to make the original request information available to the app for request processing.

For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

View or download sample code (how to download)

Additional resources

- dotnet/AspNetCore engineering SocialSample app: The linked sample app is on the dotnet/AspNetCore GitHub repo's `main` engineering branch. The `main` branch contains code under active development for the next release of ASP.NET Core. To see a version of the sample app for a released version of ASP.NET Core, use the **Branch** drop down list to select a release branch (for example `release/{X.Y}`).

Articles based on ASP.NET Core projects created with individual user accounts

Article • 02/02/2024

ASP.NET Core Identity is included in project templates in Visual Studio with the "Individual User Accounts" option.

The authentication templates are available in .NET CLI with `-au Individual`:

.NET CLI

```
dotnet new mvc -au Individual
dotnet new webapp -au Individual
```

See this GitHub issue for web API authentication.

No Authentication

Authentication is specified in the .NET CLI with the `-au` option. In Visual Studio, the **Change Authentication** dialog is available for new web applications. The default for new web apps in Visual Studio is **No Authentication**.

Projects created with no authentication:

- Don't contain web pages and UI to sign in and sign out.
- Don't contain authentication code.

Windows Authentication

Windows Authentication is specified for new web apps in the .NET CLI with the `-au Windows` option. In Visual Studio, the **Change Authentication** dialog provides the **Windows Authentication** options.

If Windows Authentication is selected, the app is configured to use the Windows Authentication IIS module. Windows Authentication is intended for Intranet web sites.

dotnet new webapp authentication options

The following table shows the authentication options available for new web apps:

Expand table

Option	Type of authentication	Link for more information
None	No authentication.	
Individual	Individual authentication.	Introduction to Identity on ASP.NET Core
IndividualB2C	Cloud-hosted individual authentication with Azure AD B2C.	Azure AD B2C
SingleOrg	Organizational authentication for a single tenant. Entra External ID tenants also use SingleOrg.	Entra ID
MultiOrg	Organizational authentication for multiple tenants.	Entra ID
Windows	Windows authentication.	Windows Authentication

Visual Studio new webapp authentication options

The following table shows the authentication options available when creating a new web app with Visual Studio:

Expand table

Option	Type of authentication	Link for more information
None	No authentication	
Individual User Accounts / Store user accounts in-app	Individual authentication	Introduction to Identity on ASP.NET Core
Individual User Accounts / Connect to an existing user store in the cloud	Cloud-hosted individual authentication with Azure AD B2C	Azure AD B2C
Work or School Cloud / Single Org	Organizational authentication for a single tenant	Azure AD
Work or School Cloud / Multiple Org	Organizational authentication for multiple tenants	Azure AD
Windows	Windows authentication	Windows Authentication

Additional resources

The following articles show how to use the code generated in ASP.NET Core templates that use individual user accounts:

- Account confirmation and password recovery in ASP.NET Core
- Create an ASP.NET Core app with user data protected by authorization

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Microsoft Entra ID with ASP.NET Core

Article • 08/30/2023

These tutorials and samples demonstrate authentication in ASP.NET Core using Microsoft identity platform and Microsoft Entra ID. For additional tutorials and samples using ASP.NET Core with Azure AD, see Microsoft identity platform.

Application Scenarios

- Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app
- Web app that signs in users
- Web app that calls web APIs
- Protected web API
- Web API that calls other web APIs
- Web app that signs in users with Azure AD B2C

Samples

- Enable your ASP.NET Core app to sign-in users and call web APIs using Microsoft identity platform for developers

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Microsoft identity platform documentation

Use the Microsoft identity platform and our open-source authentication libraries to sign in users with Microsoft Entra accounts, Microsoft personal accounts, and social accounts like Facebook and Google. Protect your web APIs and access protected APIs like Microsoft Graph to work with your users' and organization's data.



OVERVIEW
What is the Microsoft identity platform?



CONCEPT
Authentication and authorization basics



CONCEPT
App types and authentication flows



SAMPLE
Code samples



WHAT'S NEW
What's new in docs



CONCEPT
OAuth 2.0 and OpenID Connect (OIDC)



CONCEPT
Migrate apps to MSAL



QUICKSTART
Register an application

Authentication and authorization for any app

Focus on documentation for the app you're building or extending with identity and access management (IAM) support by selecting its type.



Single-page app (SPA)



Web app



Web API

A web app whose code is downloaded and run in the browser itself.

A traditional, or "classic," web app whose code runs on a server, returning page data for the browser to render.

A RESTful web service accessed by apps or other web APIs, typically to work with data served by the API.



Desktop app

Apps with a user interface (UI) whose code runs on a user's desktop, laptop, or notebook computer.



Mobile app

Apps with a UI whose code runs on a user's phone, tablet, or other mobile device.



Background service, daemon, or script

An app or script without a UI that performs non-interactive tasks like server-to-server communication or scheduled...

Get started

Quick access to guidance on adding core IAM features to your apps and best practices for keeping your apps secure and available.

Sign in users

- Single-page web app (SPA)
- Web app
- Desktop app
- Mobile app

Protect a web API

- Configure an app to expose a web API
- Build a protected web API
- Call an API from a non-interactive app or script

Test and deploy apps

- Build a test environment
- Run integration tests

Build for security and resilience

- Build Zero Trust-ready apps
- Prevent an over-privileged app
- Build auth resilience into your apps

Microsoft authentication libraries

The open-source Microsoft Authentication Library (MSAL) is built and supported by Microsoft. We recommend MSAL for any app that uses the Microsoft identity platform for authentication and authorization.

- .NET
- Android
- Angular
- iOS & macOS
- Java
- JavaScript
- Node.js
- Python
- React

Authenticate partners and customers

Sign in users from partner organizations in a business-to-business (B2B) scenario or create custom sign-up and sign-in experiences for your customers in a business-to-customer (B2C)...

External Identities documentation

Connect to Microsoft Graph

Programmatic access to organizational, user, and app data stored in Microsoft Entra ID. Call Microsoft Graph from your app to create and manage Microsoft Entra users and groups, get an...

Microsoft Graph API documentation

Manage and market your apps

Make existing SaaS apps like Dropbox, Salesforce, and ServiceNow available to your organization's users, configure SSO, and manage security. Or, become an independent software vendor (ISV) by...

App management documentation

Manage app users and their access

Automatically create user identities and their roles in your organization's installed SaaS apps. HR-driven provisioning, System for Cross-domain Identity Management (SCIM), and more.

App user and role provisioning documentation

Authentication and authorization in Azure App Service and Azure Functions

Article • 09/27/2024

> **Note**
>
> Starting June 1, 2024, all newly created App Service apps will have the option to generate a unique default hostname using the naming convention `<app-name>-<random-hash>.<region>.azurewebsites.net`. Existing app names will remain unchanged.
>
> Example: `myapp-ds27dh7271aah175.westus-01.azurewebsites.net`
>
> For further details, refer to Unique Default Hostname for App Service Resource.

Azure App Service provides built-in authentication and authorization capabilities (sometimes referred to as "Easy Auth"), so you can sign in users and access data by writing minimal or no code in your web app, RESTful API, and mobile back end, and also Azure Functions. This article describes how App Service helps simplify authentication and authorization for your app.

Why use the built-in authentication?

You're not required to use this feature for authentication and authorization. You can use the bundled security features in your web framework of choice, or you can write your own utilities. However, you will need to ensure that your solution stays up to date with the latest security, protocol, and browser updates.

Implementing a secure solution for authentication (signing-in users) and authorization (providing access to secure data) can take significant effort. You must make sure to follow industry best practices and standards, and keep your implementation up to date. The built-in authentication feature for App Service and Azure Functions can save you time and effort by providing out-of-the-box authentication with federated identity providers, allowing you to focus on the rest of your application.

- Azure App Service allows you to integrate a variety of auth capabilities into your web app or API without implementing them yourself.
- It’s built directly into the platform and doesn’t require any particular language, SDK, security expertise, or even any code to utilize.

- You can integrate with multiple login providers. For example, Microsoft Entra, Facebook, Google, X.

Your app might need to support more complex scenarios such as Visual Studio integration or incremental consent. There are several different authentication solutions available to support these scenarios. To learn more, read Identity scenarios.

Identity providers

App Service uses federated identity, in which a third-party identity provider manages the user identities and authentication flow for you. The following identity providers are available by default:

Expand table

Provider	Sign-in endpoint	How-To guidance
Microsoft Entra	`/.auth/login/aad`	App Service Microsoft Entra platform login
Facebook	`/.auth/login/facebook`	App Service Facebook login
Google	`/.auth/login/google`	App Service Google login
X	`/.auth/login/x`	App Service X login
GitHub	`/.auth/login/github`	App Service GitHub login
Sign in with Apple	`/.auth/login/apple`	App Service Sign in With Apple login (Preview)
Any OpenID Connect provider	`/.auth/login/<providerName>`	App Service OpenID Connect login

When you configure this feature with one of these providers, its sign-in endpoint is available for user authentication and for validation of authentication tokens from the provider. You can provide your users with any number of these sign-in options.

Considerations for using built-in authentication

Enabling this feature will cause all requests to your application to be automatically redirected to HTTPS, regardless of the App Service configuration setting to enforce HTTPS. You can disable this with the `requireHttps` setting in the V2 configuration. However, we do recommend sticking with HTTPS, and you should ensure no security tokens ever get transmitted over non-secure HTTP connections.

App Service can be used for authentication with or without restricting access to your site content and APIs. Access restrictions can be set in the **Authentication** > **Authentication settings** section of your web app. To restrict app access only to authenticated users, set **Action to take when request is not authenticated** to log in with one of the configured identity providers. To authenticate but not restrict access, set **Action to take when request is not authenticated** to "Allow anonymous requests (no action)."

> **Important**
>
> You should give each app registration its own permission and consent. Avoid permission sharing between environments by using separate app registrations for separate deployment slots. When testing new code, this practice can help prevent issues from affecting the production app.

How it works

Feature architecture

Authentication flow

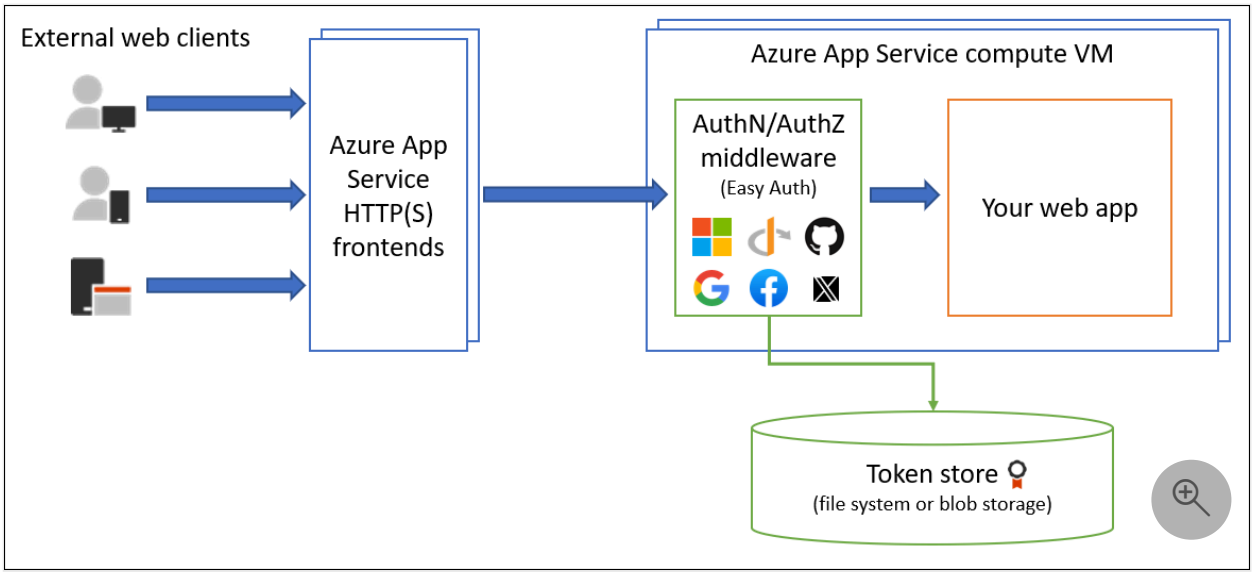
Authorization behavior

Token store

Logging and tracing

Feature architecture

The authentication and authorization middleware component is a feature of the platform that runs on the same VM as your application. When it's enabled, every incoming HTTP request passes through it before being handled by your application.



The platform middleware handles several things for your app:

- Authenticates users and clients with the specified identity provider(s)
- Validates, stores, and refreshes OAuth tokens issued by the configured identity provider(s)
- Manages the authenticated session
- Injects identity information into HTTP request headers

The module runs separately from your application code and can be configured using Azure Resource Manager settings or using a configuration file. No SDKs, specific programming languages, or changes to your application code are required.

Feature architecture on Windows (non-container deployment)

The authentication and authorization module runs as a native IIS module in the same sandbox as your application. When it's enabled, every incoming HTTP request passes through it before being handled by your application.

Feature architecture on Linux and containers

The authentication and authorization module runs in a separate container, isolated from your application code. Using what's known as the Ambassador pattern, it interacts with the incoming traffic to perform similar functionality as on Windows. Because it does not run in-process, no direct integration with specific language frameworks is possible; however, the relevant information that your app needs is passed through using request headers as explained below.

Authentication flow

The authentication flow is the same for all providers, but differs depending on whether you want to sign in with the provider's SDK:

- Without provider SDK: The application delegates federated sign-in to App Service. This is typically the case with browser apps, which can present the provider's login page to the user. The server code manages the sign-in process, so it is also called *server-directed flow* or *server flow*. This case applies to browser apps and mobile apps that use an embedded browser for authentication.
- With provider SDK: The application signs users in to the provider manually and then submits the authentication token to App Service for validation. This is typically the case with browser-less apps, which can't present the provider's sign-in page to the user. The application code manages the sign-in process, so it is also called *client-directed flow* or *client flow*. This case applies to REST APIs, Azure Functions, and JavaScript browser clients, as well as browser apps that need more flexibility in the sign-in process. It also applies to native mobile apps that sign users in using the provider's SDK.

Calls from a trusted browser app in App Service to another REST API in App Service or Azure Functions can be authenticated using the server-directed flow. For more information, see Customize sign-ins and sign-outs.

The table below shows the steps of the authentication flow.

Expand table

Step	Without provider SDK	With provider SDK
1. Sign user in	Redirects client to `/.auth/login/<provider>`.	Client code signs user in directly with provider's SDK and receives an authentication token. For information, see the provider's documentation.
2. Post-authentication	Provider redirects client to `/.auth/login/<provider>/callback`.	Client code posts token from provider to `/.auth/login/<provider>` for validation.
3. Establish authenticated session	App Service adds authenticated cookie to response.	App Service returns its own authentication token to client code.
4. Serve authenticated content	Client includes authentication cookie in subsequent requests (automatically handled by browser).	Client code presents authentication token in `X-ZUMO-AUTH` header.

For client browsers, App Service can automatically direct all unauthenticated users to `/.auth/login/<provider>`. You can also present users with one or more `/.auth/login/<provider>` links to sign in to your app using their provider of choice.

Authorization behavior

> **Important**
>
> By default, this feature only provides authentication, not authorization. Your application may still need to make authorization decisions, in addition to any checks you configure here.

In the Azure portal, you can configure App Service with a number of behaviors when incoming request is not authenticated. The following headings describe the options.

Restrict access

- **Allow unauthenticated requests** This option defers authorization of unauthenticated traffic to your application code. For authenticated requests, App Service also passes along authentication information in the HTTP headers.

  This option provides more flexibility in handling anonymous requests. For example, it lets you present multiple sign-in providers to your users. However, you must write code.

- **Require authentication** This option will reject any unauthenticated traffic to your application. Specific action to take is specified in the **Unauthenticated requests** section.

  With this option, you don't need to write any authentication code in your app. Finer authorization, such as role-specific authorization, can be handled by inspecting the user's claims (see Access user claims).

  > **Caution**
  >
  > Restricting access in this way applies to all calls to your app, which may not be desirable for apps wanting a publicly available home page, as in many single-page applications.

  > **Note**

> When using the Microsoft identity provider for users in your organization, the default behavior is that any user in your Microsoft Entra tenant can request a token for your application. You can configure the application in Microsoft Entra if you want to restrict access to your app to a defined set of users. App Service also offers some basic built-in authorization checks which can help with some validations. To learn more about authorization in Microsoft Entra, see Microsoft Entra authorization basics.

Unauthenticated requests

- **HTTP 302 Found redirect: recommended for websites** Redirects action to one of the configured identity providers. In these cases, a browser client is redirected to `/.auth/login/<provider>` for the provider you choose.
- **HTTP 401 Unauthorized: recommended for APIs** If the anonymous request comes from a native mobile app, the returned response is an `HTTP 401 Unauthorized`. You can also configure the rejection to be an `HTTP 401 Unauthorized` for all requests.
- **HTTP 403 Forbidden** Configures the rejection to be an `HTTP 403 Forbidden` for all requests.
- **HTTP 404 Not found** Configures the rejection to be an `HTTP 404 Not found` for all requests.

Token store

App Service provides a built-in token store, which is a repository of tokens that are associated with the users of your web apps, APIs, or native mobile apps. When you enable authentication with any provider, this token store is immediately available to your app. If your application code needs to access data from these providers on the user's behalf, such as:

- post to the authenticated user's Facebook timeline
- read the user's corporate data using the Microsoft Graph API

You typically must write code to collect, store, and refresh these tokens in your application. With the token store, you just retrieve the tokens when you need them and tell App Service to refresh them when they become invalid.

The ID tokens, access tokens, and refresh tokens are cached for the authenticated session, and they're accessible only by the associated user.

If you don't need to work with tokens in your app, you can disable the token store in your app's **Authentication / Authorization** page.

Logging and tracing

If you enable application logging, you will see authentication and authorization traces directly in your log files. If you see an authentication error that you didn't expect, you can conveniently find all the details by looking in your existing application logs. If you enable failed request tracing, you can see exactly what role the authentication and authorization module may have played in a failed request. In the trace logs, look for references to a module named `EasyAuthModule_32/64`.

Cross-site request forgery mitigation

App Service authentication mitigates CSRF by inspecting client requests for the following conditions:

- It's a POST request that authenticated using a session cookie.
- The request came from a known browser (as determined by the HTTP `User-Agent` header).
- The HTTP `Origin` or HTTP `Referer` header is missing or is not in the configured list of approved external domains for redirection.
- The HTTP `Origin` header is missing or is not in the configured list of CORS origins.

When a request fulfills all these conditions, App Service authentication automatically rejects it. You can workaround this mitigation logic by adding your external domain to the redirect list to **Authentication** > **Edit authentication settings** > **Allowed external redirect URLs**.

Considerations when using Azure Front Door

When using Azure App Service with authentication behind Azure Front Door or other reverse proxies, a few additional things have to be taken into consideration.

- Disable Front Door caching for the authentication workflow.

- Use the Front Door endpoint for redirects.

  App Service is usually not accessible directly when exposed via Azure Front Door. This can be prevented, for example, by exposing App Service via Private Link in Azure Front Door Premium. To prevent the authentication workflow to redirect traffic back to App Service directly, it is important to configure the application to redirect back to `https://<front-door-endpoint>/.auth/login/<provider>/callback`.

- Ensure that App Service is using the right redirect URI

In some configurations, the App Service is using the App Service FQDN as the redirect URI instead of the Front Door FQDN. This will lead to an issue when the client is being redirected to App Service instead of Front Door. To change that, the `forwardProxy` setting needs to be set to `Standard` to make App Service respect the `X-Forwarded-Host` header set by Azure Front Door.

Other reverse proxies like Azure Application Gateway or 3rd-party products might use different headers and need a different forwardProxy setting.

This configuration cannot be done via the Azure portal today and needs to be done via `az rest`:

Export settings

`az rest --uri /subscriptions/REPLACE-ME-SUBSCRIPTIONID/resourceGroups/REPLACE-ME-RESOURCEGROUP/providers/Microsoft.Web/sites/REPLACE-ME-APPNAME/config/authsettingsV2?api-version=2020-09-01 --method get > auth.json`

Update settings

Search for

```JSON
"httpSettings": {
  "forwardProxy": {
    "convention": "Standard"
  }
}
```

and ensure that `convention` is set to `Standard` to respect the `X-Forwarded-Host` header used by Azure Front Door.

Import settings

`az rest --uri /subscriptions/REPLACE-ME-SUBSCRIPTIONID/resourceGroups/REPLACE-ME-RESOURCEGROUP/providers/Microsoft.Web/sites/REPLACE-ME-APPNAME/config/authsettingsV2?api-version=2020-09-01 --method put --body @auth.json`

More resources

- How-To: Configure your App Service or Azure Functions app to use Microsoft Entra login

- Customize sign-ins and sign-outs
- Work with OAuth tokens and sessions
- Access user and application claims
- File-based configuration

Samples:

- Tutorial: Add authentication to your web app running on Azure App Service
- Tutorial: Authenticate and authorize users end-to-end in Azure App Service (Windows or Linux)
- .NET Core integration of Azure AppService EasyAuth (3rd party)
- Getting Azure App Service authentication working with .NET Core (3rd party)

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Web app that signs in users: App registration

Article • 03/25/2024

This article explains the app registration steps for a web app that signs in users.

To register your application, you can use:

- The web app quickstarts. In addition to being a great first experience with creating an application, quickstarts in the Azure portal contain a button named **Make this change for me**. You can use this button to set the properties you need, even for an existing app. Adapt the values of these properties to your own case. In particular, the web API URL for your app is probably going to be different from the proposed default, which will also affect the sign-out URI.
- The Azure portal to register your application manually.
- PowerShell and command-line tools.

Register an app by using the quickstarts

You can use the following link to bootstrap the creation of your web application:

Register an application

Register an app by using the Azure portal

> **Tip**
>
> Steps in this article might vary slightly based on the portal you start from.

> **Note**
>
> The portal to use is different depending on whether your application runs in the Microsoft Azure public cloud or in a national or sovereign cloud. For more information, see National clouds.

1. Sign in to the Microsoft Entra admin center.
2. If you have access to multiple tenants, use the **Settings** icon in the top menu to switch to the tenant in which you want to register the application from the

Directories + subscriptions menu.

3. Browse to **Identity** > **Applications** > **App registrations**, select **New registration**.

ASP.NET Core

1. When the **Register an application** page appears, enter your application's registration information:
   a. Enter a **Name** for your application, for example `AspNetCore-WebApp`. Users of your app might see this name, and you can change it later.
   b. Choose the supported account types for your application. (See Supported account types.)
   c. For **Redirect URI**, add the type of application and the URI destination that will accept returned token responses after successful authentication. For example, enter `https://localhost:44321`.
   d. Select **Register**.
2. Under **Manage**, select **Authentication** and then add the following information:
   a. In the **Web** section, add `https://localhost:44321/signin-oidc` as a **Redirect URI**.
   b. In **Front-channel logout URL**, enter `https://localhost:44321/signout-oidc`.
   c. Select **Save**.

Register an app by using PowerShell

You can also register an application with Microsoft Graph PowerShell, using the New-MgApplication.

Here's an idea of the code. For a fully functioning code, see this sample

PowerShell

```
# Connect to the Microsoft Graph API, non-interactive is not supported for
the moment (Oct 2021)
Write-Host "Connecting to Microsoft Graph"
if ($tenantId -eq "") {
   Connect-MgGraph -Scopes "User.Read.All Organization.Read.All
Application.ReadWrite.All" -Environment $azureEnvironmentName
}
else {
   Connect-MgGraph -TenantId $tenantId -Scopes "User.Read.All
Organization.Read.All Application.ReadWrite.All" -Environment
$azureEnvironmentName
}
```

```
$context = Get-MgContext
$tenantId = $context.TenantId

# Get the user running the script
$currentUserPrincipalName = $context.Account
$user = Get-MgUser -Filter "UserPrincipalName eq '$($context.Account)'"

# get the tenant we signed in to
$Tenant = Get-MgOrganization
$tenantName = $Tenant.DisplayName

$verifiedDomain = $Tenant.VerifiedDomains | where {$_.Isdefault -eq $true}
$verifiedDomainName = $verifiedDomain.Name
$tenantId = $Tenant.Id

Write-Host ("Connected to Tenant {0} ({1}) as account '{2}'. Domain is
'{3}'" -f  $Tenant.DisplayName, $Tenant.Id, $currentUserPrincipalName,
$verifiedDomainName)

# Create the webApp AAD application
Write-Host "Creating the AAD application (WebApp)"
# create the application
$webAppAadApplication = New-MgApplication -DisplayName "WebApp" `
                                                    -Web `
                                                    @{ `
                                                          RedirectUris =
"https://localhost:44321/", "https://localhost:44321/signin-oidc"; `
                                                          HomePageUrl =
"https://localhost:44321/"; `
                                                          LogoutUrl =
"https://localhost:44321/signout-oidc"; `
                                                       } `
                                                       -SignInAudience
AzureADandPersonalMicrosoftAccount `
                                                    #end of command

$currentAppId = $webAppAadApplication.AppId
$currentAppObjectId = $webAppAadApplication.Id

$tenantName = (Get-MgApplication -ApplicationId
$currentAppObjectId).PublisherDomain
#Update-MgApplication -ApplicationId $currentAppObjectId -IdentifierUris
@("https://$tenantName/WebApp")

# create the service principal of the newly created application
$webAppServicePrincipal = New-MgServicePrincipal -AppId $currentAppId -Tags
{WindowsAzureActiveDirectoryIntegratedApp}

# add the user running the script as an app owner if needed
$owner = Get-MgApplicationOwner -ApplicationId $currentAppObjectId
if ($owner -eq $null)
{
   New-MgApplicationOwnerByRef -ApplicationId $currentAppObjectId  -
BodyParameter = @{"@odata.id" =
"https://graph.microsoft.com/v1.0/directoryObjects/$user.ObjectId"}
```

```
    Write-Host "'$($user.UserPrincipalName)' added as an application owner to
app '$($webAppServicePrincipal.DisplayName)'"
}
Write-Host "Done creating the webApp application (WebApp)"
```

Next step

ASP.NET Core

Move on to the next article in this scenario, App's code configuration.

Feedback

Was this page helpful? Yes No

Provide product feedback

A web app that calls web APIs: App registration

Article • 10/23/2023

A web app that calls web APIs has the same registration as a web app that signs users in. So, follow the instructions in A web app that signs in users: App registration.

However, because the web app now also calls web APIs, it becomes a confidential client application. That's why some extra registration is required. The app must share client credentials, or *secrets*, with the Microsoft identity platform.

Add a client secret or certificate

As with any confidential client application, you need to add a secret or certificate to act as that application's *credentials* so it can authenticate as itself, without user interaction.

You can add credentials to your client app's registration by using the Azure portal or by using a command-line tool like PowerShell.

Add client credentials by using the Azure portal

To add credentials to your confidential client application's app registration, follow the steps in Quickstart: Register an application with the Microsoft identity platform for the type of credential you want to add:

- Add a client secret
- Add a certificate

Add client credentials by using PowerShell

Alternatively, you can add credentials when you register your application with the Microsoft identity platform by using PowerShell.

The active-directory-dotnetcore-daemon-v2 code sample on GitHub shows how to add an application secret or certificate when registering an application:

- For details on how to add a **client secret** with PowerShell, see AppCreationScripts/Configure.ps1.
- For details on how to add a **certificate** with PowerShell, see AppCreationScripts-withCert/Configure.ps1.

API permissions

Web apps call APIs on behalf of the signed-in user. To do that, they must request *delegated permissions*. For details, see Add permissions to access your web API.

Next steps

Move on to the next article in this scenario, Code configuration.

Feedback

Was this page helpful? Yes No

Provide product feedback

Protected web API: App registration

Article • 05/28/2024

This article explains how to register an application for a protected web API.

For the common steps to register an app, see Quickstart: Register an application with the Microsoft identity platform.

Accepted token version

The Microsoft identity platform can issue v1.0 tokens and v2.0 tokens. For more information about these tokens, refer to Access tokens.

The token version your API may accept depends on your **Supported account types** selection when you create your web API application registration in the Azure portal.

- If the value of **Supported account types** is **Accounts in any organizational directory and personal Microsoft accounts (such as Skype, Xbox, Outlook.com)**, the accepted token version must be v2.0.
- Otherwise, the accepted token version can be v1.0.

After you create the application, you can determine or change the accepted token version by following these steps:

1. In the Microsoft Entra admin center, select your app and then select **Manifest**.
2. Find the property **accessTokenAcceptedVersion** in the manifest.
3. The value specifies to Microsoft Entra which token version the web API accepts.

    - If the value is 2, the web API accepts v2.0 tokens.
    - If the value is **null**, the web API accepts v1.0 tokens.

4. If you changed the token version, select **Save**.

The web API specifies which token version it accepts. When a client requests a token for your web API from the Microsoft identity platform, the client gets a token that indicates which token version the web API accepts.

No redirect URI

Web APIs don't need to register a redirect URI because no user is interactively signed in.

Exposed API

Other settings specific to web APIs are the exposed API and the exposed scopes or app roles.

Scopes and the Application ID URI

Scopes usually have the form `resourceURI/scopeName`. For Microsoft Graph, the scopes have shortcuts. For example, `User.Read` is a shortcut for `https://graph.microsoft.com/user.read`.

During app registration, define these parameters:

- The resource URI
- One or more scopes
- One or more app roles

By default, the application registration portal recommends that you use the resource URI `api://{clientId}`. This URI is unique but not human readable. If you change the URI, make sure the new value is unique. The application registration portal will ensure that you use a configured publisher domain.

To client applications, scopes show up as *delegated permissions* and app roles show up as *application permissions* for your web API.

Scopes also appear on the consent window that's presented to users of your app. Therefore, provide the corresponding strings that describe the scope:

- As seen by a user.
- As seen by a tenant admin, who can grant admin consent.

App roles cannot be consented to by a user (as they're used by an application that calls the web API on behalf of itself). A tenant administrator will need to consent to client applications of your web API exposing app roles. See Admin consent for details.

Expose delegated permissions (scopes)

To expose delegated permissions, or *scopes*, follow the steps in Configure an application to expose a web API.

If you're following along with the web API scenario described in this set of articles, use these settings:

- **Application ID URI**: Accept the proposed application ID URI (*api://<clientId>*) (if prompted)
- **Scope name**: *access_as_user*
- **Who can consent**: *Admins and users*
- **Admin consent display name**: *Access TodoListService as a user*
- **Admin consent description**: *Accesses the TodoListService web API as a user*
- **User consent display name**: *Access TodoListService as a user*
- **User consent description**: *Accesses the TodoListService web API as a user*
- **State**: *Enabled*

> **Tip**
>
> For the **Application ID URI**, you have the option to set it to the physical authority of the API, for example `https://graph.microsoft.com`. This can be useful if the URL of the API that needs to be called is known.

If your web API is called by a service or daemon app

Expose *application permissions* instead of delegated permissions if your API should be accessed by daemons, services, or other non-interactive (by a human) applications. Because daemon- and service-type applications run unattended and authenticate with their own identity, there is no user to "delegate" their permission.

Expose application permissions (app roles)

To expose application permissions, follow the steps in Add app roles to your app.

In the **Create app role** pane under **Allowed member types**, select **Applications**. Or, add the role by using the **Application manifest editor** as described in the article.

Restrict access tokens to specific clients apps

App roles are the mechanism an application developer uses to expose their app's permissions. Your web API's code should check for app roles in the access tokens it receives from callers.

To add another layer of security, a Microsoft Entra tenant administrator can configure their tenant so the Microsoft identity platform issues security tokens *only* to the client apps they've approved for API access.

To increase security by restricting token issuance only to client apps that have been assigned app roles:

1. In the Microsoft Entra admin center, select your app under **Identity** > **Applications** > **App registrations**.
2. On the application's **Overview** page, in **Essentials**, find and select its **Managed application in local directory** link to navigate to its **Enterprise Application Overview** page.
3. Under **Manage**, select **Properties**.
4. Set **Assignment required?** to **No**.
5. Select **Save**.

Microsoft Entra ID will now check for app role assignments of client applications that request access tokens for your web API. If a client app hasn't been assigned any app roles, Microsoft Entra ID returns an error message to the client similar to `_invalid_client: AADSTS501051: Application \<application name\> isn't assigned to a role for the \<web API\>_`.

> ⚠ **Warning**
>
> **DO NOT use AADSTS error codes** or their message strings as literals in your application's code. The "AADSTS" error codes and the error message strings returned by Microsoft Entra ID are *not immutable*, and may be changed by Microsoft at any time and without your knowledge. If you make branching decisions in your code based on the values of either the AADSTS codes or their message strings, you put your application's functionality and stability at risk.

Next step

The next article in this series is App code configuration.

Feedback

Was this page helpful? Yes No

Provide product feedback

A web API that calls web APIs: App registration

Article • 10/23/2023

A web API that calls downstream web APIs has the same registration as a protected web API. Follow the instructions in Protected web API: App registration.

Because the web app now calls web APIs, it becomes a confidential client application. That's why extra registration information is required: the app needs to share secrets (client credentials) with the Microsoft identity platform.

Add a client secret or certificate

As with any confidential client application, you need to add a secret or certificate to act as that application's *credentials* so it can authenticate as itself, without user interaction.

You can add credentials to your client app's registration by using the Azure portal or by using a command-line tool like PowerShell.

Add client credentials by using the Azure portal

To add credentials to your confidential client application's app registration, follow the steps in Quickstart: Register an application with the Microsoft identity platform for the type of credential you want to add:

- Add a client secret
- Add a certificate

Add client credentials by using PowerShell

Alternatively, you can add credentials when you register your application with the Microsoft identity platform by using PowerShell.

The active-directory-dotnetcore-daemon-v2 code sample on GitHub shows how to add an application secret or certificate when registering an application:

- For details on how to add a **client secret** with PowerShell, see AppCreationScripts/Configure.ps1.
- For details on how to add a **certificate** with PowerShell, see AppCreationScripts-withCert/Configure.ps1.

API permissions

Web apps call APIs on behalf of users for whom the bearer token was received. The web apps need to request delegated permissions. For more information, see Add permissions to access your web API.

Next steps

Move on to the next article in this scenario, App Code configuration.

Feedback

Was this page helpful? Yes No

Provide product feedback

Cloud authentication with Azure Active Directory B2C in ASP.NET Core

Article • 02/15/2024

By Damien Bod

Azure Active Directory B2C (Azure AD B2C) is a cloud identity management solution for web and mobile apps. The service provides authentication for apps hosted in the cloud and on-premises. Authentication types include individual accounts, social network accounts, and federated enterprise accounts. Additionally, Azure AD B2C can provide multi-factor authentication with minimal configuration.

> **Tip**
>
> Microsoft Entra ID, Microsoft Entra External ID and Azure AD B2C are separate product offerings. An Entra ID tenant generally represents an organization, while an Azure AD B2C tenant or a Microsoft Entra External ID tenant can represent a collection of identities to be used with relying party applications. To learn more, see **Azure AD B2C: Frequently asked questions (FAQ)**.

> **Tip**
>
> **Microsoft Entra External ID for customers** is Microsoft's new customer identity and access management (CIAM) solution.

In this tutorial, you'll learn how to configure an ASP.NET Core app for authentication with Azure AD B2C.

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- .NET SDK. Install the latest .NET SDK for your platform.

Preparation

1. Create an Azure Active Directory B2C tenant.

2. Create a new ASP.NET Core Razor pages app:

.NET CLI

```
dotnet new razor -o azure-ad-b2c
```

The previous command creates a Razor pages app in a directory named *azure-ad-b2c*.

> **Tip**
>
> You may prefer to use Visual Studio to create your app.

3. Create a web app registration in the tenant. For **Redirect URI**, use `https://localhost:5001/signin-oidc`. Replace `5001` with the port used by your app when using Visual Studio generated ports.

Modify the app

1. Add the `Microsoft.Identity.Web` and `Microsoft.Identity.Web.UI` packages to the project. If you're using Visual Studio, you can use NuGet Package Manager.

.NET CLI

```
dotnet add package Microsoft.Identity.Web
dotnet add package Microsoft.Identity.Web.UI
```

In the preceding:

- `Microsoft.Identity.Web` includes the basic set of dependencies for authenticating with the Microsoft identity platform.
- `Microsoft.Identity.Web.UI` includes UI functionality encapsulated in an area named `MicrosoftIdentity`.

2. Add an `AzureADB2C` object to `appsettings.json`.

> **Note**
>
> When using Azure B2C user flows, you need to set the **Instance** and the PolicyId of the type of flow.

JSON

```
{
  "AzureADB2C": {
    "Instance": "https://--your-domain--.b2clogin.com",
    "Domain": "[Enter the domain of your B2C tenant, e.g.
contoso.onmicrosoft.com]",
    "TenantId": "[Enter 'common', or 'organizations' or the Tenant Id
(Obtained from the Azure portal. Select 'Endpoints' from the 'App
registrations' blade and use the GUID in any of the URLs), e.g.
da41245a5-11b3-996c-00a8-4d99re19f292]",
    "ClientId": "[Enter the Client Id (Application ID obtained from the
Azure portal), e.g. ba74781c2-53c2-442a-97c2-3d60re42f403]",
    // Use either a secret or a certificate. ClientCertificates are
recommended.
    "ClientSecret": "[Copy the client secret added to the app from the
Azure portal]",
    "ClientCertificates": [
    ],
    // the following is required to handle Continuous Access Evaluation
challenges
    "ClientCapabilities": [ "cp1" ],
    "CallbackPath": "/signin-oidc",
    // Add your policy here
    "SignUpSignInPolicyId": "B2C_1_signup_signin",
    "SignedOutCallbackPath": "/signout-callback-oidc"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

- For **Domain**, use the domain of your Azure AD B2C tenant.
- For **ClientId**, use the **Application (client) ID** from the app registration you created in your tenant.
- For **Instance**, use the domain of your Azure AD B2C tenant.
- For **SignUpSignInPolicyId**, use the user flow policy defined in the Azure B2C tenant
- Use either the **ClientSecret** or the **ClientCertificates** configuration. ClientCertificates are recommended.
- Leave all other values as they are.

3. In *Pages/Shared*, create a file named `_LoginPartial.cshtml`. Include the following code:

```
razor
```

```
@using System.Security.Principal

<ul class="navbar-nav">
@if (User.Identity?.IsAuthenticated == true)
{
        <span class="navbar-text text-dark">Hello @User.Identity?.Name!
</span>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity"
asp-controller="Account" asp-action="SignOut">Sign out</a>
        </li>
}
else
{
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="MicrosoftIdentity"
asp-controller="Account" asp-action="SignIn">Sign in</a>
        </li>
}
</ul>
```

The preceding code:

- Checks if the user is authenticated.
- Renders a **Sign out** or **Sign in** link as appropriate.
  - The link points to an action method on the `Account` controller in the `MicrosoftIdentity` area.

4. In *Pages/Shared/_Layout.cshtml*, add the highlighted line within the `<header>` element:

razor

```
<header>
    <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
        <div class="container">
            <a class="navbar-brand" asp-area="" asp-
page="/Index">azure_ad_b2c</a>
            <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                    aria-expanded="false" aria-label="Toggle
navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                <ul class="navbar-nav flex-grow-1">
                    <li class="nav-item">
```

```
                        <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-
page="/Privacy">Privacy</a>
                    </li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>
</header>
```

Adding `<partial name="_LoginPartial" />` renders the `_LoginPartial.cshtml` partial view in every page request that uses this layout.

5. In *Program.cs*, make the following changes:

   a. Add the following `using` directives:

   C#

   ```
   using Microsoft.Identity.Web;
   using Microsoft.Identity.Web.UI;
   using Microsoft.AspNetCore.Authentication.OpenIdConnect;
   ```

   The preceding code resolves references used in the next steps.

   b. Update the `builder.Services` lines with the following code:

   C#

   ```
   builder.Services.AddAuthentication(OpenIdConnectDefaults.Authenticat
   ionScheme)

   .AddMicrosoftIdentityWebApp(builder.Configuration.GetSection("AzureA
   DB2C"));

   builder.Services.AddAuthorization(options =>
   {
       // By default, all incoming requests will be authorized
   according to
       // the default policy
       options.FallbackPolicy = options.DefaultPolicy;
   });
   builder.Services.AddRazorPages(options => {
       options.Conventions.AllowAnonymousToPage("/Index");
   })
   ```

```
.AddMvcOptions(options => { })
.AddMicrosoftIdentityUI();
```

In the preceding code:

- Calls to the `AddAuthentication` and `AddMicrosoftIdentityWebApp` methods configure the app to use Open ID Connect, specifically configured for the Microsoft identity platform.
- `AddAuthorization` initializes ASP.NET Core authorization.
- The `AddRazorPages` call configures the app so anonymous browsers can view the Index page. All other requests require authentication.
- `AddMvcOptions` and `AddMicrosoftIdentityUI` add the required UI components for redirecting to/from Azure AD B2C.

c. Update the highlighted line to the `Configure` method:

C#

```
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
```

The preceding code enables authentication in ASP.NET Core.

Run the app

> **Note**
>
> Use the profile which matches the Azure App registration **Redirect URIs**

1. Run the app.

.NET CLI

```
dotnet run --launch-profile https
```

2. Browse to the app's secure endpoint, for example, `https://localhost:5001/`.

- The Index page renders with no authentication challenge.
- The header includes a **Sign in** link because you're not authenticated.

3. Select the **Privacy** link.

    - The browser is redirected to your tenant's configured authentication method.
    - After signing in, the header displays a welcome message and a **Sign out** link.

Next steps

In this tutorial, you learned how to configure an ASP.NET Core app for authentication with Azure AD B2C.

Now that the ASP.NET Core app is configured to use Azure AD B2C for authentication, the Authorize attribute can be used to secure your app. Continue developing your app by learning to:

- Customize the Azure AD B2C user interface.
- Configure password complexity requirements.
- Enable multi-factor authentication.
- Configure additional identity providers, such as Microsoft, Facebook, Google, Amazon, Twitter, and others.
- Use the Microsoft Graph API to retrieve additional user information, such as group membership, from the Azure AD B2C tenant.
- How to secure a Web API built with ASP.NET Core using the Azure AD B2C .
- Tutorial: Grant access to an ASP.NET web API using Azure Active Directory B2C.

Enable authentication in your own web API by using Azure AD B2C

Article • 01/11/2024

To authorize access to a web API, you can serve only requests that include a valid access token that Azure Active Directory B2C (Azure AD B2C) issues. This article shows you how to enable Azure AD B2C authorization to your web API. After you complete the steps in this article, only users who obtain a valid access token will be authorized to call your web API endpoints.

Prerequisites

Before you begin, read one of the following articles, which discuss how to configure authentication for apps that call web APIs. Then, follow the steps in this article to replace the sample web API with your own web API.

- Configure authentication in a sample ASP.NET Core application
- Configure authentication in a sample single-page application (SPA)

Overview

Token-based authentication ensures that requests to a web API includes a valid access token.

The app completes the following steps:

1. It authenticates users with Azure AD B2C.
2. It acquires an access token with the required permissions (scopes) for the web API endpoint.
3. It passes the access token as a bearer token in the authentication header of the HTTP request by using this format:

   ```HTTP
   Authorization: Bearer <access token>
   ```

The web API completes the following steps:

1. It reads the bearer token from the authorization header in the HTTP request.

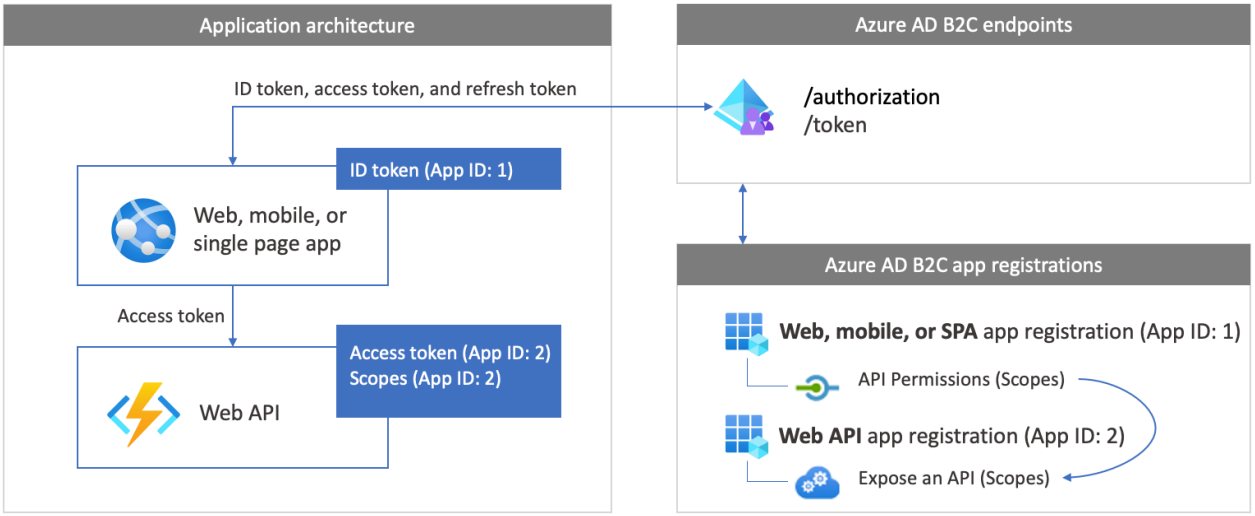
2. It validates the token.
3. It validates the permissions (scopes) in the token.
4. It reads the claims that are encoded in the token (optional).
5. It responds to the HTTP request.

App registration overview

To enable your app to sign in with Azure AD B2C and call a web API, you need to register two applications in the Azure AD B2C directory.

- The *web, mobile, or SPA application* registration enables your app to sign in with Azure AD B2C. The app registration process generates an *Application ID*, also known as the *client ID*, which uniquely identifies your application (for example, *App ID: 1*).
- The *web API* registration enables your app to call a protected web API. The registration exposes the web API permissions (scopes). The app registration process generates an *Application ID*, which uniquely identifies your web API (for example, *App ID: 2*). Grant your app (App ID: 1) permissions to the web API scopes (App ID: 2).

The application registrations and the application architecture are described in the following diagram:



Prepare your development environment

In the next sections, you create a new web API project. Select your programming language, ASP.NET Core or Node.js. Make sure you have a computer that's running either of the following software:

ASP.NET Core

- Visual Studio Code
- C# for Visual Studio Code (latest version)
- .NET 5.0 SDK

Step 1: Create a protected web API

Create a new web API project. First, select the programming language you want to use, **ASP.NET Core** or **Node.js**.

ASP.NET Core

Use the dotnet new command. The `dotnet new` command creates a new folder named *TodoList* with the web API project assets. Open the directory, and then open Visual Studio Code.

.NET CLI

```
dotnet new webapi -o TodoList
cd TodoList
code .
```

When you're prompted to "add required assets to the project," select **Yes**.

Step 2: Install the dependencies

Add the authentication library to your web API project. The authentication library parses the HTTP authentication header, validates the token, and extracts claims. For more information, review the documentation for the library.

ASP.NET Core

To add the authentication library, install the package by running the following command:

.NET CLI

```
dotnet add package Microsoft.Identity.Web
```

Step 3: Initiate the authentication library

Add the necessary code to initiate the authentication library.

ASP.NET Core

Open *Startup.cs* and then, at the beginning of the class, add the following `using` declarations:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Identity.Web;
```

Find the `ConfigureServices(IServiceCollection services)` function. Then, before the `services.AddControllers();` line of code, add the following code snippet:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Adds Microsoft Identity platform (Azure AD B2C) support to
protect this Api
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddMicrosoftIdentityWebApi(options =>
    {
        Configuration.Bind("AzureAdB2C", options);

        options.TokenValidationParameters.NameClaimType = "name";
    },
    options => { Configuration.Bind("AzureAdB2C", options); });
    // End of the Microsoft Identity platform block

    services.AddControllers();
}
```

Find the `Configure` function. Then, immediately after the `app.UseRouting();` line of code, add the following code snippet:

C#

```
app.UseAuthentication();
```

After the change, your code should look like the following snippet:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    // Add the following line
    app.UseAuthentication();
    // End of the block you add

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Step 4: Add the endpoints

Add two endpoints to your web API:

- Anonymous `/public` endpoint. This endpoint returns the current date and time. Use it to debug your web API with anonymous calls.
- Protected `/hello` endpoint. This endpoint returns the value of the `name` claim within the access token.

To add the anonymous endpoint:

ASP.NET Core

Under the */Controllers* folder, add a *PublicController.cs* file, and then add it to the following code snippet:

C#

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;

namespace TodoList.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class PublicController : ControllerBase
    {
        private readonly ILogger<PublicController> _logger;

        public PublicController(ILogger<PublicController> logger)
        {
            _logger = logger;
        }

        [HttpGet]
        public ActionResult Get()
        {
            return Ok( new {date = DateTime.UtcNow.ToString()});
        }
    }
}
```

To add the protected endpoint:

ASP.NET Core

Under the */Controllers* folder, add a *HelloController.cs* file, and then add it to the following code:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using Microsoft.Identity.Web.Resource;

namespace TodoList.Controllers
{
    [Authorize]
    [RequiredScope("tasks.read")]
    [ApiController]
    [Route("[controller]")]
    public class HelloController : ControllerBase
    {
```

```
        private readonly ILogger<HelloController> _logger;
        private readonly IHttpContextAccessor _contextAccessor;

        public HelloController(ILogger<HelloController> logger,
IHttpContextAccessor contextAccessor)
        {
            _logger = logger;
            _contextAccessor = contextAccessor;
        }

        [HttpGet]
        public ActionResult Get()
        {
            return Ok( new { name = User.Identity.Name});
        }
    }
}
```

The `HelloController` controller is decorated with the AuthorizeAttribute, which limits access to authenticated users only.

The controller is also decorated with the `[RequiredScope("tasks.read")]`. The RequiredScopeAttribute verifies that the web API is called with the right scopes, `tasks.read`.

Step 5: Configure the web server

In a development environment, set the web API to listen on incoming HTTP or HTTPS requests port number. In this example, use HTTP port 6000 and HTTPS port 6001. The base URI of the web API will be `http://localhost:6000` for HTTP and `https://localhost:6001` for HTTPS.

ASP.NET Core

Add the following JSON snippet to the *appsettings.json* file.

JSON

```
"Kestrel": {
    "EndPoints": {
      "Http": {
        "Url": "http://localhost:6000"
      },
      "Https": {
         "Url": "https://localhost:6001"
      }
```

```
    }
  }
```

Step 6: Configure the web API

Add configurations to a configuration file. The file contains information about your Azure AD B2C identity provider. The web API app uses this information to validate the access token that the web app passes as a bearer token.

ASP.NET Core

Under the project root folder, open the *appsettings.json* file, and then add the following settings:

JSON

```
{
  "AzureAdB2C": {
    "Instance": "https://contoso.b2clogin.com",
    "Domain": "contoso.onmicrosoft.com",
    "ClientId": "<web-api-app-application-id>",
    "SignedOutCallbackPath": "/signout/<your-sign-up-in-policy>",
    "SignUpSignInPolicyId": "<your-sign-up-in-policy>"
  },
  // More settings here
}
```

In the *appsettings.json* file, update the following properties:

Expand table

Section	Key	Value
AzureAdB2C	Instance	The first part of your Azure AD B2C tenant name (for example, `https://contoso.b2clogin.com`).
AzureAdB2C	Domain	Your Azure AD B2C tenant full tenant name (for example, `contoso.onmicrosoft.com`).
AzureAdB2C	ClientId	The web API application ID. In the preceding diagram, it's the application with *App ID: 2*. To learn how to get your web API application registration ID, see Prerequisites.
AzureAdB2C	SignUpSignInPolicyId	The user flows, or custom policy. To learn how to get

Section	Key	Value
		your user flow or policy, see Prerequisites.

Step 7: Run and test the web API

Finally, run the web API with your Azure AD B2C environment settings.

ASP.NET Core

In the command shell, start the web app by running the following command:

```bush
 dotnet run
```

You should see the following output, which means that your app is up and running and ready to receive requests.

```
Now listening on: http://localhost:6000
```

To stop the program, in the command shell, select Ctrl+C. You can rerun the app by using the `node app.js` command.

> **Tip**
>
> Alternatively, to run the `dotnet run` command, you can use the Visual Studio Code debugger. Visual Studio Code's built-in debugger helps accelerate your edit, compile, and debug loop.

Open a browser and go to `http://localhost:6000/public`. In the browser window, you should see the following text displayed, along with the current date and time.

Step 8: Call the web API from your app

Try to call the protected web API endpoint without an access token. Open a browser and go to `http://localhost:6000/hello`. The API will return an unauthorized HTTP error

message, confirming that web API is protected with a bearer token.

Continue to configure your app to call the web API. For guidance, see the Prerequisites section.

Watch this video to learn about some best practices when you integrate Azure AD B2C with an API.
https://www.youtube-nocookie.com/embed/wuUu71RcsIo

Next steps

Get the complete example on GitHub:

ASP.NET Core

- Get the web API by using the Microsoft identity library.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Use cookie authentication without ASP.NET Core Identity

Article • 06/03/2022

By Rick Anderson

ASP.NET Core Identity is a complete, full-featured authentication provider for creating and maintaining logins. However, a cookie-based authentication provider without ASP.NET Core Identity can be used. For more information, see Introduction to Identity on ASP.NET Core.

View or download sample code (how to download)

For demonstration purposes in the sample app, the user account for the hypothetical user, Maria Rodriguez, is hardcoded into the app. Use the **Email** address `maria.rodriguez@contoso.com` and any password to sign in the user. The user is authenticated in the `AuthenticateUser` method in the `Pages/Account/Login.cshtml.cs` file. In a real-world example, the user would be authenticated against a datastore.

Add cookie authentication

- Add the Authentication Middleware services with the AddAuthentication and AddCookie methods.
- Call UseAuthentication and UseAuthorization to set the `HttpContext.User` property and run the Authorization Middleware for requests. `UseAuthentication` and `UseAuthorization` must be called before `Map` methods such as MapRazorPages and MapDefaultControllerRoute

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthentication(CookieAuthenticationDefaults.Authenticati
onScheme)
    .AddCookie();

builder.Services.AddHttpContextAccessor();
```

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

AuthenticationScheme passed to `AddAuthentication` sets the default authentication scheme for the app. `AuthenticationScheme` is useful when there are multiple instances of cookie authentication and the app needs to authorize with a specific scheme. Setting the `AuthenticationScheme` to CookieAuthenticationDefaults.AuthenticationScheme provides a value of `"Cookies"` for the scheme. Any string value can be used that distinguishes the scheme.

The app's authentication scheme is different from the app's cookie authentication scheme. When a cookie authentication scheme isn't provided to AddCookie, it uses CookieAuthenticationDefaults.AuthenticationScheme. The CookieAuthenticationDefaults.AuthenticationScheme GitHub Source shows it's set to `"Cookies"`.

The authentication cookie's IsEssential property is set to `true` by default. Authentication cookies are allowed when a site visitor hasn't consented to data collection. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

The CookieAuthenticationOptions class is used to configure the authentication provider options.

Configure CookieAuthenticationOptions in the AddCookie method:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
builder.Services.AddControllersWithViews();

builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.ExpireTimeSpan = TimeSpan.FromMinutes(20);
        options.SlidingExpiration = true;
        options.AccessDeniedPath = "/Forbidden/";
    });

builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Cookie Policy Middleware

The Cookie Policy Middleware (GitHub Source) UseCookiePolicy enables cookie policy capabilities. Middleware is processed in the order it's added:

```
C#

app.UseCookiePolicy(cookiePolicyOptions);
```

Use CookiePolicyOptions provided to the Cookie Policy Middleware to control global characteristics of cookie processing and hook into cookie processing handlers when cookies are appended or deleted.

The default MinimumSameSitePolicy value is `SameSiteMode.Lax` to permit OAuth2 authentication. To strictly enforce a same-site policy of `SameSiteMode.Strict`, set the `MinimumSameSitePolicy`. Although this setting breaks OAuth2 and other cross-origin

authentication schemes, it elevates the level of cookie security for other types of apps that don't rely on cross-origin request processing.

```
C#

var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

The Cookie Policy Middleware setting for `MinimumSameSitePolicy` can affect the setting of `Cookie.SameSite` in `CookieAuthenticationOptions` settings according to the matrix below.

Expand table

MinimumSameSitePolicy	Cookie.SameSite	Resultant Cookie.SameSite setting
SameSiteMode.None	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Lax SameSiteMode.Lax SameSiteMode.Strict
SameSiteMode.Strict	SameSiteMode.None SameSiteMode.Lax SameSiteMode.Strict	SameSiteMode.Strict SameSiteMode.Strict SameSiteMode.Strict

Create an authentication cookie

To create a cookie holding user information, construct a ClaimsPrincipal. The user information is serialized and stored in the cookie.

Create a ClaimsIdentity with any required Claims and call SignInAsync to sign in the user. `Login.cshtml.cs` in the sample app contains the following code:

```
C#

public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    ReturnUrl = returnUrl;

    if (ModelState.IsValid)
    {
```

```
        // Use Input.Email and Input.Password to authenticate the user
        // with your custom authentication logic.
        //
        // For demonstration purposes, the sample validates the user
        // on the email address maria.rodriguez@contoso.com with
        // any password that passes model validation.

        var user = await AuthenticateUser(Input.Email, Input.Password);

        if (user == null)
        {
            ModelState.AddModelError(string.Empty, "Invalid login
attempt.");
            return Page();
        }

        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name, user.Email),
            new Claim("FullName", user.FullName),
            new Claim(ClaimTypes.Role, "Administrator"),
        };

        var claimsIdentity = new ClaimsIdentity(
            claims, CookieAuthenticationDefaults.AuthenticationScheme);

        var authProperties = new AuthenticationProperties
        {
            //AllowRefresh = <bool>,
            // Refreshing the authentication session should be allowed.

            //ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
            // The time at which the authentication ticket expires. A
            // value set here overrides the ExpireTimeSpan option of
            // CookieAuthenticationOptions set with AddCookie.

            //IsPersistent = true,
            // Whether the authentication session is persisted across
            // multiple requests. When used with cookies, controls
            // whether the cookie's lifetime is absolute (matching the
            // lifetime of the authentication ticket) or session-based.

            //IssuedUtc = <DateTimeOffset>,
            // The time at which the authentication ticket was issued.

            //RedirectUri = <string>
            // The full path or absolute URI to be used as an http
            // redirect response value.
        };

        await HttpContext.SignInAsync(
            CookieAuthenticationDefaults.AuthenticationScheme,
            new ClaimsPrincipal(claimsIdentity),
            authProperties);
```

```
            _logger.LogInformation("User {Email} logged in at {Time}.",
                user.Email, DateTime.UtcNow);

            return LocalRedirect(Url.GetLocalUrl(returnUrl));
        }

        // Something failed. Redisplay the form.
        return Page();
    }
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

`SignInAsync` creates an encrypted cookie and adds it to the current response. If `AuthenticationScheme` isn't specified, the default scheme is used.

RedirectUri is only used on a few specific paths by default, for example, the login path and logout paths. For more information see the CookieAuthenticationHandler source.

ASP.NET Core's Data Protection system is used for encryption. For an app hosted on multiple machines, load balancing across apps, or using a web farm, configure data protection to use the same key ring and app identifier.

Sign out

To sign out the current user and delete their cookie, call SignOutAsync:

C#

```
public async Task OnGetAsync(string returnUrl = null)
{
    if (!string.IsNullOrEmpty(ErrorMessage))
    {
        ModelState.AddModelError(string.Empty, ErrorMessage);
    }

    // Clear the existing external cookie
    await HttpContext.SignOutAsync(
        CookieAuthenticationDefaults.AuthenticationScheme);

    ReturnUrl = returnUrl;
}
```

If `CookieAuthenticationDefaults.AuthenticationScheme` or "Cookies" isn't used as the scheme, supply the scheme used when configuring the authentication provider. Otherwise, the default scheme is used. For example, if "ContosoCookie" is used as the scheme, supply the scheme used when configuring the authentication provider.

When the browser closes it automatically deletes session based cookies (non-persistent cookies), but no cookies are cleared when an individual tab is closed. The server is not notified of tab or browser close events.

React to back-end changes

Once a cookie is created, the cookie is the single source of identity. If a user account is disabled in back-end systems:

- The app's cookie authentication system continues to process requests based on the authentication cookie.
- The user remains signed into the app as long as the authentication cookie is valid.

The ValidatePrincipal event can be used to intercept and override validation of the cookie identity. Validating the cookie on every request mitigates the risk of revoked users accessing the app.

One approach to cookie validation is based on keeping track of when the user database changes. If the database hasn't been changed since the user's cookie was issued, there's no need to re-authenticate the user if their cookie is still valid. In the sample app, the database is implemented in `IUserRepository` and stores a `LastChanged` value. When a user is updated in the database, the `LastChanged` value is set to the current time.

In order to invalidate a cookie when the database changes based on the `LastChanged` value, create the cookie with a `LastChanged` claim containing the current `LastChanged` value from the database:

C#

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};

var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

To implement an override for the `ValidatePrincipal` event, write a method with the following signature in a class that derives from CookieAuthenticationEvents:

C#

```
ValidatePrincipal(CookieValidatePrincipalContext)
```

The following is an example implementation of `CookieAuthenticationEvents`:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;

public class CustomCookieAuthenticationEvents : CookieAuthenticationEvents
{
    private readonly IUserRepository _userRepository;

    public CustomCookieAuthenticationEvents(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    public override async Task
ValidatePrincipal(CookieValidatePrincipalContext context)
    {
        var userPrincipal = context.Principal;

        // Look for the LastChanged claim.
        var lastChanged = (from c in userPrincipal.Claims
                           where c.Type == "LastChanged"
                           select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !_userRepository.ValidateLastChanged(lastChanged))
        {
            context.RejectPrincipal();

            await context.HttpContext.SignOutAsync(
                CookieAuthenticationDefaults.AuthenticationScheme);
        }
    }
}
```

Register the events instance during cookie service registration. Provide a scoped service registration for your `CustomCookieAuthenticationEvents` class:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
builder.Services.AddControllersWithViews();

builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.EventsType = typeof(CustomCookieAuthenticationEvents);
    });

builder.Services.AddScoped<CustomCookieAuthenticationEvents>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Consider a situation in which the user's name is updated—a decision that doesn't affect security in any way. If you want to non-destructively update the user principal, call `context.ReplacePrincipal` and set the `context.ShouldRenew` property to `true`.

> ⚠ **Warning**
>
> The approach described here is triggered on every request. Validating authentication cookies for all users on every request can result in a large performance penalty for the app.

Persistent cookies

You may want the cookie to persist across browser sessions. This persistence should only be enabled with explicit user consent with a "Remember Me" checkbox on sign in or a similar mechanism.

The following code snippet creates an identity and corresponding cookie that survives through browser closures. Any sliding expiration settings previously configured are honored. If the cookie expires while the browser is closed, the browser clears the cookie once it's restarted.

Set IsPersistent to `true` in AuthenticationProperties:

```
C#

// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

Absolute cookie expiration

An absolute expiration time can be set with ExpiresUtc. To create a persistent cookie, `IsPersistent` must also be set. Otherwise, the cookie is created with a session-based lifetime and could expire either before or after the authentication ticket that it holds. When `ExpiresUtc` is set, it overrides the value of the ExpireTimeSpan option of CookieAuthenticationOptions, if set.

The following code snippet creates an identity and corresponding cookie that lasts for 20 minutes. This ignores any sliding expiration settings previously configured.

```
C#

// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Configure OpenID Connect Web (UI) authentication in ASP.NET Core

Article • 12/20/2024

By Damien Bowden

View or download sample code

This article covers the following areas:

- What is an OpenID Connect confidential interactive client
- Create an OpenID Connect client in ASP.NET Core
- Examples of OpenID Connect client with code snippets
- Using third party OpenID Connect provider clients
- Backend for frontend (BFF) security architecture
- Advanced features, standards, extending the an OpenID Connect client

For an alternative experience using Microsoft Authentication Library for .NET, Microsoft Identity Web, and Microsoft Entra ID, see Quickstart: Sign in users and call the Microsoft Graph API from an ASP.NET Core web app (Azure documentation).

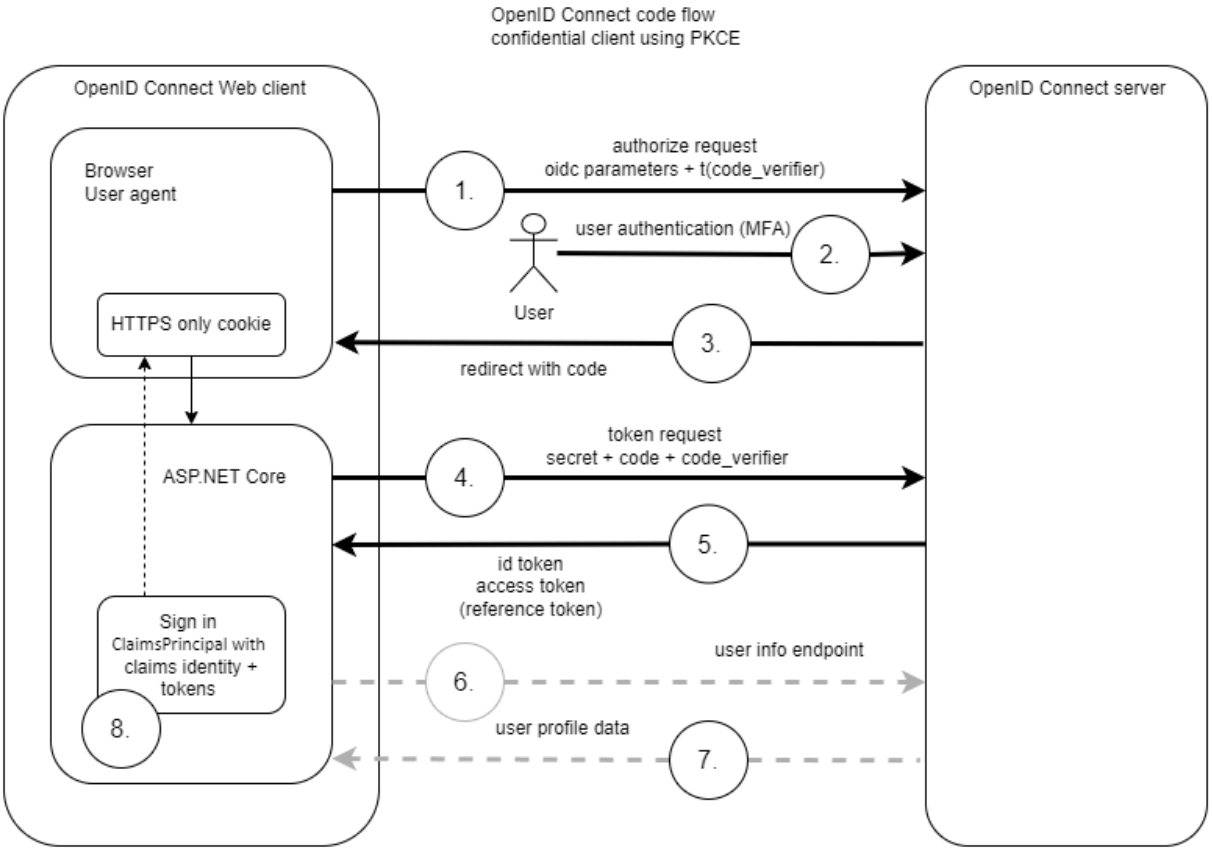
For an example using Microsoft Entra External ID OIDC server, see Sign in users for a sample ASP.NET Core web app in an external tenant and An ASP.NET Core web app authenticating users against Microsoft Entra External ID using Microsoft Identity Web.

What is an OpenID Connect confidential interactive client

OpenID Connect can be used to implement authentication in ASP.NET Core applications. The recommended way is to use an OpenID Connect confidential client using the code flow. Using the Proof Key for Code Exchange by OAuth Public Clients (PKCE) is recommended for this implementation. Both the application client and the user of the application are authenticated in the confidential flow. The application client uses a client secret or a client assertion to authenticate.

Public OpenID Connect/OAuth clients are no longer recommended for web applications.

The default flow works as shown in the following diagram:



OpenID Connect comes in many variations and all server implementations have slightly different parameters and requirements. Some servers don't support the user info endpoint, some still don't support PKCE and others require special parameters in the token request. Client assertions can be used instead of client secrets. New standards also exist which add extra security on top of the OpenID Connect Core, for example FAPI, CIBA or DPoP for downstream APIs.

> **Note**
>
> From .NET 9, OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126 is used per default, if the OpenID Connect server supports this. This is a three step flow and not a two step flow as shown above. (User Info request is an optional step.)

Create an Open ID Connect code flow client using Razor Pages

The following section shows how to implement an OpenID Connect client in an empty ASP.NET Core Razor page project. The same logic can be applied to any ASP.NET Core web project with only the UI integration being different.

Add OpenID Connect support

Add the Microsoft.AspNetCore.Authentication.OpenIdConnect Nuget packages to the ASP.NET Core project.

Setup the OpenID Connect client

Add the authentication to the web application using the builder.Services in the `Program.cs` file. The configuration is dependent on the OpenID Connect server. Each OpenID Connect server requires small differences in the setup.

The OpenID Connect handler is used for challenges and signout. The cookie is used to handle the session in the web application. The default schemes for the authentication can be specified as required.

For more information, see the ASP.NET Core authentication-handler guidance.

C#

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    var oidcConfig =
builder.Configuration.GetSection("OpenIDConnectSettings");

    options.Authority = oidcConfig["Authority"];
    options.ClientId = oidcConfig["ClientId"];
    options.ClientSecret = oidcConfig["ClientSecret"];

    options.SignInScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.ResponseType = OpenIdConnectResponseType.Code;

    options.SaveTokens = true;
    options.GetClaimsFromUserInfoEndpoint = true;

    options.MapInboundClaims = false;
    options.TokenValidationParameters.NameClaimType =
JwtRegisteredClaimNames.Name;
    options.TokenValidationParameters.RoleClaimType = "roles";
});
```

For details on the different OpenID Connect options, see Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC).

For the different claims mapping possibilities, see Mapping, customizing, and transforming claims in ASP.NET Core.

> **Note**
>
> The following namespaces are required:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;
using Microsoft.IdentityModel.Tokens;
```

Setup the configuration properties

Add the OpenID Connect client settings to the application configuration properties. The settings must match the client configuration in the OpenID Connect server. No secrets should be persisted in application settings where they might get accidentally checked in. Secrets should be stored in a secure location like Azure Key Vault in production environments or in user secrets in a development environment. For more informaiton, see Safe storage of app secrets in development in ASP.NET Core.

JSON

```
"OpenIDConnectSettings": {
  // OpenID Connect URL. (The base URL for the /.well-known/openid-
configuration)
  "Authority": "<Authority>",
  // client ID from the OpenID Connect server
  "ClientId": "<Client ID>",
  //"ClientSecret": "--stored-in-user-secrets-or-key-vault--"
},
```

Signed-out callback path configuration

The SignedOutCallbackPath (configuration key: "`SignedOutCallbackPath`") is the request path within the app's base path intercepted by the OpenID Connect handler where the user agent is first returned after signing out from the identity provider. The sample app doesn't set a value for the path because the default value of "`/signout-callback-oidc`"

is used. After intercepting the request, the OpenID Connect handler redirects to the SignedOutRedirectUri or RedirectUri, if specified.

Configure the signed-out callback path in the app's OIDC provider registration. In the following example, the `{PORT}` placeholder is the app's port:

> https://localhost:{PORT}/signout-callback-oidc

> **Note**
>
> When using Microsoft Entra ID, set the path in the **Web** platform configuration's **Redirect URI** entries in the Entra or Azure portal. A port isn't required for `localhost` addresses when using Entra. Most other OIDC providers require the correct port. If you don't add the signed-out callback path URI to the app's registration in Entra, Entra refuses to redirect the user back to the app and merely asks them to close their browser window.

Update the ASP.NET Core pipeline method in the program class.

The `UseRouting` method must be implemented before the `UseAuthorization` method.

C#

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();
app.UseAuthentication();
// Authorization is applied for middleware after the UseAuthorization method
app.UseAuthorization();
app.MapRazorPages();
```

Force authorization

Add the [Authorize] attribute to the protected Razor pages:

C#

```
[Authorize]
```

A better approach is to force authorization for the whole app and opt out for unsecure pages:

C#

```
var requireAuthPolicy = new AuthorizationPolicyBuilder()
    .RequireAuthenticatedUser()
    .Build();

builder.Services.AddAuthorizationBuilder()
    .SetFallbackPolicy(requireAuthPolicy);
```

Add a new `Logout.cshtml` and `SignedOut.cshtml` Razor pages to the project

A logout is required to sign out both the cookie session and the OpenID Connect session. The whole app needs to redirect to the OpenID Connect server to sign out. After a successful sign out, the app opens the `RedirectUri` route.

Implement a default sign-out page and change the `Logout` razor page code to the following:

C#

```
[Authorize]
public class LogoutModel : PageModel
{
    public IActionResult OnGetAsync()
    {
        return SignOut(new AuthenticationProperties
        {
            RedirectUri = "/SignedOut"
        },
        // Clear auth cookie
        CookieAuthenticationDefaults.AuthenticationScheme,
        // Redirect to OIDC provider signout endpoint
        OpenIdConnectDefaults.AuthenticationScheme);
    }
}
```

The `SignedOut.cshtml` requires the [AllowAnonymous] attribute:

C#

```
[AllowAnonymous]
public class SignedOutModel : PageModel
{
```

```
    public void OnGet()
    {
    }
}
```

Implement `Login` page

A `Login` Razor page can also be implemented to call the `ChallengeAsync` directly with the required `AuthProperties`. This isn't required if the web app requires authentication and the default challenge is used.

The `Login.cshtml` page requires the [AllowAnonymous] attribute:

CSHTML

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace RazorPageOidc.Pages;

[AllowAnonymous]
public class LoginModel : PageModel
{
    [BindProperty(SupportsGet = true)]
    public string? ReturnUrl { get; set; }

    public async Task OnGetAsync()
    {
        var properties = GetAuthProperties(ReturnUrl);
        await HttpContext.ChallengeAsync(properties);
    }

    private static AuthenticationProperties GetAuthProperties(string?
returnUrl)
    {
        const string pathBase = "/";

        // Prevent open redirects.
        if (string.IsNullOrEmpty(returnUrl))
        {
            returnUrl = pathBase;
        }
        else if (!Uri.IsWellFormedUriString(returnUrl, UriKind.Relative))
        {
            returnUrl = new Uri(returnUrl, UriKind.Absolute).PathAndQuery;
        }
        else if (returnUrl[0] != '/')
        {
            returnUrl = $"{pathBase}{returnUrl}";
```

```
        }

        return new AuthenticationProperties { RedirectUri = returnUrl };
    }
}
```

Add a login and logout button for the user

```
@if (Context.User.Identity!.IsAuthenticated)
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-
page="/Logout">Logout</a>
    </li>

    <span class="nav-link text-dark">Hi @Context.User.Identity.Name</span>
}
else
{
    <li class="nav-item">
        <a class="nav-link text-dark" asp-area="" asp-
page="/Index">Login</a>
    </li>
}
```

Examples with code snippets

Example Using User Info endpoint

The OpenID Connect options can be used to map claims, implement handlers or even save the tokens in the session for later usage.

The `Scope` option can be used to request different claims or a refresh token which is sent as information to the OpenID Connect server. Requesting the `offline_access` is asking the server to return a reference token which can be used to refresh the session without authenticating the user of the application again.

C#

```
services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
```

```
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(OpenIdConnectDefaults.AuthenticationScheme, options =>
{
    var oidcConfig =
builder.Configuration.GetSection("OpenIDConnectSettings");
    options.Authority = oidcConfig["IdentityProviderUrl"];
    options.ClientSecret = oidcConfig["ClientSecret"];
    options.ClientId = oidcConfig["Audience"];
    options.ResponseType = OpenIdConnectResponseType.Code;

    options.Scope.Clear();
    options.Scope.Add("openid");
    options.Scope.Add("profile");
    options.Scope.Add("email");
    options.Scope.Add("offline_access");

    options.ClaimActions.Remove("amr");
    options.ClaimActions.MapUniqueJsonKey("website", "website");

    options.GetClaimsFromUserInfoEndpoint = true;
    options.SaveTokens = true;

    // .NET 9 feature
    options.PushedAuthorizationBehavior =
PushedAuthorizationBehavior.Require;

    options.TokenValidationParameters.NameClaimType = "name";
    options.TokenValidationParameters.RoleClaimType = "role";
});
```

Implementing Microsoft identity providers

Microsoft has multiple identity providers and OpenID Connect implementations. Microsoft has different OpenID Connect servers:

- Microsoft Entra ID
- Microsoft Entra External ID
- Azure AD B2C

If authenticating using one of the Microsoft identity providers in ASP.NET Core, it is recommended to use the Microsoft.Identity.Web Nuget packages.

The `Microsoft.Identity.Web` Nuget packages is a Microsoft specific client built on top on the ASP.NET Core OpenID Connect client with some changes to the default client.

Using third party OpenID Connect provider clients

Many OpenID Connect server implementations create Nuget packages which are optimized for the same OpenID Connect implementation. These packages implement the OpenID Connect client specifics with the extras required by the specific OpenID Connect server. `Microsoft.Identity.Web` is one example of this.

If implementing multiple OpenID Connect clients from different OpenID Connect servers in a single application, it's normally better to revert to the default ASP.NET Core implementation as the different clients overwrite some options which affect the other clients.

OpenIddict Web providers is a client implementation which supports many different server implementations.

IdentityModel is a .NET standard helper library for claims-based identity, OAuth 2.0 and OpenID Connect. This can also be used to help with the client implementation.

Backend for frontend (BFF) security architecture

It is no longer recommended to implement OpenID Connect public clients for any web apps.

For more information, see the draft OAuth 2.0 for Browser-Based Applications.

If implementing **web** applications which have no independent backend, we recommend using the Backend for Frontend (BFF) pattern security architecture. This pattern can be implemented in different ways, but the authentication is always implemented in the backend, and no sensitive data is sent to the web client for further authorization or authentication flows.

Advanced features, standards, extending the OIDC client

Logging

Debugging OpenID Connect clients can be hard. Personally identifiable information (PII) data is not logged by default. If debugging in development mode, the

`IdentityModelEventSource.ShowPII` can be used to log sensitive personal data. Don't deploy an app with `IdentityModelEventSource.ShowPII` to productive servers.

C#

```
//using ...

using Microsoft.IdentityModel.Logging;

var builder = WebApplication.CreateBuilder(args);

//... code

var app = builder.Build();

IdentityModelEventSource.ShowPII = true;

//... code

app.Run();
```

For more information, see Logging.

> ⓘ Note
>
> You may want to lower the configured log level to see all the required logs.

OIDC and OAuth Parameter Customization

The OAuth and OIDC authentication handlers (AdditionalAuthorizationParameters) option allows customization of authorization message parameters that are usually included as part of the redirect query string.

Map claims from OpenID Connect

For more information, see Mapping, customizing, and transforming claims in ASP.NET Core.

Blazor OpenID Connect

For more information, see Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC).

Standards

- OpenID Connect 1.0
- Proof Key for Code Exchange by OAuth Public Clients
- The OAuth 2.0 Authorization Framework
- OAuth 2.0 Pushed Authorization Requests (PAR) RFC 9126

Configure JWT bearer authentication in ASP.NET Core

Article • 01/19/2025

By Damien Bowden

JWT (JSON Web Token) Bearer Authentication is commonly utilized for APIs. While it operates similarly to cookie authentication, the identity provider issues a JWT or tokens upon a successful authentication. These tokens can then be sent to other servers to authenticate, unlike cookies which are only sent back to the issuing domain. A JWT is a self-contained token that encapsulates information for an API resource or a client. The client which requested the JWT can request data from an API resource using the Authorization header and a bearer token.

JWT Bearer Authentication provides:

- **Authentication**: When using the `JwtBearerHandler`, bearer tokens are essential for authentication. The `JwtBearerHandler` validates the token and extracts the user's identity from its claims.
- **Authorization**: Bearer tokens enable authorization by providing a collection of claims representing the user's or application's permissions, much like a cookie.
- **Delegated Authorization**: When a user-specific access token is used to authenticate between APIs instead of an application-wide access token, this process is known as *delegated authorization*.

For an introduction to JWT Bearer Authentication, see JSON Web Tokens. View or download sample code

This article covers the following areas:

- Token types
- Using JWT tokens to secure an APIs
- How OIDC/OAuth fits into this?
- Implementing JWT bearer token authentication
- Recommended approaches to create a JWT

Token types

There are numerous types of tokens and formats. Generating your own access tokens or ID tokens is discouraged, except for testing purposes. Self-created tokens that do not adhere to established standards:

- Can lead to security vulnerabilities.
- Are only suitable for closed systems.

We recommend using OpenID Connect 1.0 or an OAuth standard for creating access tokens intended for API access.

Access tokens

Access tokens:

- Are strings used by a client app to make requests to the server implementing an API.
- Can vary in format. Different APIs may use different formats for the tokens.
- Can be encrypted.
- Should never be read or interpreted by a web client or UI app holding the access token.
- Are intended solely for making requests to an API.
- Are typically sent to the API in the **Authorization** request header as a bearer token.

See The OAuth 2.0 Authorization Framework

Application access tokens and delegated access tokens

Access tokens can be either *application access tokens* or *delegated access tokens*. The tokens have different claims and are managed and stored differently. An *application access token* is typically stored once in the app until it expires, while a *delegated access token* is stored per user, either in a cookie or in a secure server cache.

We recommend using delegated user access tokens whenever a user is involved. Downstream APIs can request a delegated user access token on behalf of the authenticated user.

Sender constrained access tokens

Access tokens can be used as bearer tokens or sender-constrained tokens to access resources. Sender-constrained tokens require the requesting client to prove possession of a private key to use the token. Proving possession of a private key guarantees the token can't be used independently. Sender-constrained tokens can be implemented in two ways:

- Demonstrating Proof of Possession (DPoP)
- Mutual-TLS (MTLS)

ID tokens

ID tokens are security tokens that confirm a user's successful authentication. The tokens allow the client to verify the user's identity. The JWT token server issues ID tokens containing claims with user information. ID tokens are always in JWT format.

ID tokens ***should never*** be used to access APIs.

Other tokens

There are many types of tokens, including access and ID tokens, as specified by OpenID Connect and OAuth standards. Refresh tokens can be used to refresh a UI app without re-authenticating the user. OAuth JAR tokens can securely send authorization requests. Verifiable credentials flows utilize JWT types for issuing or verifying credentials. It is crucial to use tokens according to the specifications. See the standards links provided later in this article for more information.

Using JWT tokens to secure an API

When using JWT access tokens for API authorization, the API grants or denies access based on the provided token. If the request is not authorized, a 401 or 403 response is returned. The API shouldn't redirect the user to the identity provider to obtain a new token or request additional permissions. The app consuming the API is responsible for acquiring an appropriate token. This ensures a clear separation of concerns between the API (authorization) and the consuming client app (authentication).

> Note HTTP also allows returning 404 for not authorized, so as to not leak information about the existence of resources to unauthorized clients.

401 Unauthorized

A 401 Unauthorized response indicates that the provided access token doesn't meet the required standards. This could be due to several reasons, including:

- **Invalid signature**: The token's signature doesn't match, suggesting potential tampering.
- **Expiration**: The token has expired and is no longer valid.
- **Incorrect claims**: Critical claims within the token, such as the audience (`aud`) or issuer (`iss`), are missing or invalid.

> Note: From the HTTP Semantics RFC 9110: The server generating a 401 response MUST send a WWW-Authenticate header field (Section 11.6.1) containing at least one challenge applicable to the target resource.

The OAuth specifications provide detailed guidelines on the required claims and their validation.

403 Forbidden

A 403 Forbidden response typically indicates that the authenticated user lacks the necessary permissions to access the requested resource. This is distinct from authentication issues, e.g. an invalid token, and is unrelated to the standard claims within the access token.

In ASP.NET Core, you can enforce authorization using:

Requirements and policies: Define custom requirements, e.g., "Must be an administrator" and associate them with policies. Role-based authorization: Assign users to roles e.g., "Admin," "Editor", and restrict access based on those roles.

What role has OIDC and/or OAuth when using bearer tokens?

When an API uses JWT access tokens for authorization, the API only validates the access token, not on how the token was obtained.

OpenID Connect (OIDC) and OAuth 2.0 provide standardized, secure frameworks for token acquisition. Token acquisition varies depending on the type of app. Due to the complexity of secure token acquisition, it's highly recommended to rely on these standards:

- For apps acting on behalf of a user and an application: OIDC is the preferred choice, enabling delegated user access. In web apps, the confidential code flow with Proof Key for Code Exchange (PKCE) is recommended for enhanced security.
  - If the calling app is an ASP.NET Core app with server-side OIDC authentication, you can use the SaveTokens option to store access token in a cookie for later use via HttpContext.GetTokenAsync("access_token").
- If the app has no user: The OAuth 2.0 client credentials flow is suitable for obtaining application access tokens.

Implementing JWT bearer token authentication

The Microsoft.AspNetCore.Authentication.JwtBearer Nuget package can be used to validate the JWT bearer tokens.

JWT bearer tokens should be fully validated in an API. The following should be validated:

- Signature, for trust and integrity. This ensures the token was created by the designated secure token service and has not been tampered with.
- Issuer claim with the expected value.
- Audience claim with the expected value.
- Token expiration.

The following claims are required for OAuth 2.0 access tokens: `iss`, `exp`, `aud`, `sub`, `client_id`, `iat, and` jti`.

If any of these claims or values are incorrect, the API should return a 401 response.

JWT bearer token basic validation

A basic implementation of the AddJwtBearer can validate just the audience and the issuer. The signature must be validated so that the token can be trusted and that it hasn't been tampered with.

C#

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(jwtOptions =>
{
    jwtOptions.Authority = "https://{--your-authority--}";
    jwtOptions.Audience = "https://{--your-audience--}";
});
```

JWT bearer token explicit validation

The AddJwtBearer method provides multiple configurations. Some secure token providers use a non-standard metadata address and the parameter can be setup explicitly. The API can accept multiple issuers or audiences.

Explicitly defining the parameters is not required. The definitions depends on the access token claim values and the secure token server used to validate the access token. You should use the default values if possible.

See Mapping claims MapInboundClaims details.

C#

```
builder.Services.AddAuthentication()
.AddJwtBearer("some-scheme", jwtOptions =>
{
    jwtOptions.MetadataAddress =
builder.Configuration["Api:MetadataAddress"];
    // Optional if the MetadataAddress is specified
    jwtOptions.Authority = builder.Configuration["Api:Authority"];
    jwtOptions.Audience = builder.Configuration["Api:Audience"];
    jwtOptions.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateIssuerSigningKey = true,
        ValidAudiences =
builder.Configuration.GetSection("Api:ValidAudiences").Get<string[]>(),
        ValidIssuers =
builder.Configuration.GetSection("Api:ValidIssuers").Get<string[]>()
    };

    jwtOptions.MapInboundClaims = false;
});
```

JWT with multiple schemes

APIs often need to accommodate access tokens from various issuers. Supporting multiple token issuers in an API can be accomplished by:

- **Separate APIs**: Create distinct APIs with dedicated authentication schemes for each issuer.
- AddPolicyScheme This method can define multiple authentication schemes and implement logic to select the appropriate scheme based on token properties (e.g., issuer, claims). This approach allows for greater flexibility within a single API.

Forcing the bearer authentication

SetDefaultPolicy can be used to require authentication for all requests even to endpoints without an `[Authorize]` attribute. SetDefaultPolicy configures the policy used for endpoints with the `[Authorize]` attribute and already defaults to requiring authenticated users. See the require authenticated users documentation for more details.

C#

```
var requireAuthPolicy = new AuthorizationPolicyBuilder()
    .RequireAuthenticatedUser()
    .Build();

builder.Services.AddAuthorizationBuilder()
    .SetFallbackPolicy(requireAuthPolicy);
```

The Authorize attribute can also be used to force the authentication. If multiple schemes are used, the bearer scheme generally needs to be set as the default authentication scheme or specified via `[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme])`.

Authorization in controllers:

C#

```
[Authorize]
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
```

Authorization in Minimal APIs:

C#

```
app.MapGet("/hello", [Authorize] () => "Hi");
```

Recommended approaches to create a JWT

Insecure handling of access tokens, such as weak authentication or storing tokens in vulnerable client-side storage, can lead to significant security vulnerabilities. For example, storing access tokens directly in the browser using local storage, session storage, or web workers. The following section contains best practices for apps using and creating access tokens.

Use standards

Standards like OpenID Connect or OAuth should **always** be used when creating access tokens. Access tokens should **not** be created in production apps without adhering to the security precautions outlined in this article. Creating access tokens should be limited to test scenarios.

Use asymmetric keys

Asymmetric keys should **always** be used when creating access tokens. The public key is available in the well known endpoints and the API clients can validate the signature of the access token using the public key.

Never create an access token from a username/password request

You should **NOT** create an access token from a username/password request. Username/password requests aren't authenticated and are vunerable to impersonation and phishing attacks. Access tokens should only be created using an OpenID Connect flow or an OAuth standard flow. Deviating from these standards can result in an insecure app.

Use cookies

For secure web apps, a backend is required to store access tokens on a trusted server. Only a secure HTTP only cookie is shared on the client browser. See the OIDC authentication documentation for how to do this in an ASP.NET Core web app.

Downstream APIs

APIs occasionally need to access user data from downstream APIs on behalf of the authenticated user in the calling app. While implementing an OAuth client credentials flow is an option, it necessitates full trust between the two API apps. A more secure approach involves using a zero-trust strategy with a delegated user access token. This approach:

- Enhances security by granting the API only the necessary permissions for that specific user.
- Requires the API to create the new access token for the user calling the app and the API.

There are several ways to implement a zero-trust strategy with a delegated user access token:

Use OAuth 2.0 Token Exchange to request a new delegated access token

This is a good way to implement this requirement but it's complicated if you must implement the OAuth flow.

See OAuth 2.0 Token Exchange

Use Microsoft Identity Web on behalf of flow to request a new delegated access token

Using the Microsoft Identity Web authentication library is the easiest and a secure approach. It only works with Microsoft Entra ID, Microsoft Entra External ID.

For more information, see [Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow

Use the same delegated access token sent to the API

This approach is not difficult to implement but the access token has access to both APIs. Yarp reverse proxy can be used to implement this.

Use OAuth client credentials flow and use an application access token

This is easy to implement but the client application has full application access and not a delegated access token. The token should be cached in the client API application.

> Note Any app-to-app security also works. Certificate authentication, or in Azure, a managed identity can be used.

Handling access tokens

When using access tokens in a client application, the access tokens need to be rotated, persisted and stored somewhere on the server. In a web application, cookies are used to secure the session and can be used to store tokens via SaveTokens option.

`SaveTokens` will not currently refresh access tokens automatically, but this functionality is planned for .NET 10. Follow https://github.com/dotnet/aspnetcore/issues/8175 for updates. In the meantime, you can manually refresh the access token as demonstrated in the Blazor Web App with OIDC documentation or use a third-party NuGet package like Duende.AccessTokenManagement.OpenIdConnect for handling and managing access tokens in the client app. For more information, see Duende token management.

> Note If deploying to production, the cache should work in a mutli-instance deployment. A persistent cache is normally required.

Some secure token servers encrypt the access tokens. Access tokens do not require any format. When using OAuth introspection, a reference token is used instead of an access token. A client (UI) application should never open an access token as the access token is not intended for this. Only an API for which the access token was created for should open the access token.

- Do not open access tokens in a UI application
- Do not send the ID token to the APIs
- Access tokens can have any format
- Access tokens can be encrypted
- Access tokens expire and need to be rotated
- Access tokens are persisted on a secure backend server

YARP (Yet Another Reverse Proxy)

YARP (Yet Another Reverse Proxy) is a good tool for handling HTTP requests and forwarding the requests to other APIs. YARP can implement security logic for acquiring new access credentials. YARP is frequently used in the backend for frontend (BFF) security architecture. The Blazor Web App with OIDC documentation demonstrates using YARP to implement the BFF pattern.

Testing APIs

Integration tests and containers with access tokens can be used to test secure APIs. Access tokens can be created using the dotnet user-jwts tool.

> ⚠ **Warning**
>
> Ensure that security problems are **not** introduced into the API for testing purposes. Testing becomes more challenging when delegated access tokens are used, as these tokens can only be created through a UI and an OpenID Connect flow. If a test tool is used to create delegated access tokens, security features must be disabled for testing. It's essential that these features are only disabled in the test environment.

Create dedicated and isolated test environments where security features can safely be disabled or modified. Ensure these changes are strictly limited to the test environment.

Use Swagger UI, Curl and other API UI tools

Swagger UI and Curl are great UI tools for testing APIs. For the tools to work, the API can produce an OpenAPI document and this can be loaded into the client testing tool. A security flow to acquire a new access token can be added to the API OpenAPI file.

> ⚠ **Warning**
>
> Do not deploy insecure security test flows to production.

When implementing a Swagger UI for an API, you should normally not deploy the UI to production as the security must be weakened to allow this to work.

Map claims from OpenID Connect

Refer to the following document:

Mapping, customizing, and transforming claims in ASP.NET Core

Standards

JSON Web Token (JWT)

The OAuth 2.0 Authorization Framework

OAuth 2.0 Demonstrating Proof of Possession DPoP

OAuth 2.0 JWT-Secured Authorization Request (JAR) RFC 9101

OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens

OpenID Connect 1.0

Microsoft identity platform and OAuth 2.0 On-Behalf-Of flow

OAuth 2.0 Token Exchange

JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens

HTTP Semantics RFC 9110

Configure certificate authentication in ASP.NET Core

Article • 11/03/2023

`Microsoft.AspNetCore.Authentication.Certificate` contains an implementation similar to Certificate Authentication for ASP.NET Core. Certificate authentication happens at the TLS level, long before it ever gets to ASP.NET Core. More accurately, this is an authentication handler that validates the certificate and then gives you an event where you can resolve that certificate to a `ClaimsPrincipal`.

You ***must*** configure your server for certificate authentication, be it IIS, Kestrel, Azure Web Apps, or whatever else you're using.

Proxy and load balancer scenarios

Certificate authentication is a stateful scenario primarily used where a proxy or load balancer doesn't handle traffic between clients and servers. If a proxy or load balancer is used, certificate authentication only works if the proxy or load balancer:

- Handles the authentication.
- Passes the user authentication information to the app (for example, in a request header), which acts on the authentication information.

An alternative to certificate authentication in environments where proxies and load balancers are used is Active Directory Federated Services (ADFS) with OpenID Connect (OIDC).

Get started

Acquire an HTTPS certificate, apply it, and configure your server to require certificates.

In the web app:

- Add a reference to the Microsoft.AspNetCore.Authentication.Certificate NuGet package.
- In `Program.cs`, call `builder.Services.AddAuthentication(CertificateAuthenticationDefaults.AuthenticationScheme).AddCertificate(...);`. Provide a delegate for `OnCertificateValidated` to do any supplementary validation on the client

certificate sent with requests. Turn that information into a `ClaimsPrincipal` and set it on the `context.Principal` property.

If authentication fails, this handler returns a `403 (Forbidden)` response rather a `401 (Unauthorized)`, as you might expect. The reasoning is that the authentication should happen during the initial TLS connection. By the time it reaches the handler, it's too late. There's no way to upgrade the connection from an anonymous connection to one with a certificate.

`UseAuthentication` is required to set `HttpContext.User` to a `ClaimsPrincipal` created from the certificate. For example:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate();

var app = builder.Build();

app.UseAuthentication();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The preceding example demonstrates the default way to add certificate authentication. The handler constructs a user principal using the common certificate properties.

Configure certificate validation

The `CertificateAuthenticationOptions` handler has some built-in validations that are the minimum validations you should perform on a certificate. Each of these settings is enabled by default.

AllowedCertificateTypes = Chained, SelfSigned, or All (Chained | SelfSigned)

Default value: `CertificateTypes.Chained`

This check validates that only the appropriate certificate type is allowed. If the app is using self-signed certificates, this option needs to be set to `CertificateTypes.All` or

`CertificateTypes.SelfSigned`.

ChainTrustValidationMode

Default value: X509ChainTrustMode.System

The certificate presented by the client must chain to a trusted root certificate. This check controls which trust store contains these root certificates.

By default, the handler uses the system trust store. If the presented client certificate needs to chain to a root certificate which doesn't appear in the system trust store, this option can be set to X509ChainTrustMode.CustomRootTrust to make the handler use the `CustomTrustStore`.

CustomTrustStore

Default value: Empty X509Certificate2Collection

If the handler's ChainTrustValidationMode property is set to `X509ChainTrustMode.CustomRootTrust`, this X509Certificate2Collection contains every certificate which will be used to validate the client certificate up to a trusted root, including the trusted root.

When the client presents a certificate which is part of a multi-level certificate chain, `CustomTrustStore` must contain every issuing certificate in the chain.

ValidateCertificateUse

Default value: `true`

This check validates that the certificate presented by the client has the Client Authentication extended key use (EKU), or no EKUs at all. As the specifications say, if no EKU is specified, then all EKUs are deemed valid.

ValidateValidityPeriod

Default value: `true`

This check validates that the certificate is within its validity period. On each request, the handler ensures that a certificate that was valid when it was presented hasn't expired during its current session.

RevocationFlag

Default value: `X509RevocationFlag.ExcludeRoot`

A flag that specifies which certificates in the chain are checked for revocation.

Revocation checks are only performed when the certificate is chained to a root certificate.

RevocationMode

Default value: `X509RevocationMode.Online`

A flag that specifies how revocation checks are performed.

Specifying an online check can result in a long delay while the certificate authority is contacted.

Revocation checks are only performed when the certificate is chained to a root certificate.

Can I configure my app to require a certificate only on certain paths?

This isn't possible. Remember the certificate exchange is done at the start of the HTTPS conversation, it's done by the server before the first request is received on that connection so it's not possible to scope based on any request fields.

Handler events

The handler has two events:

- `OnAuthenticationFailed`: Called if an exception happens during authentication and allows you to react.
- `OnCertificateValidated`: Called after the certificate has been validated, passed validation and a default principal has been created. This event allows you to perform your own validation and augment or replace the principal. For examples include:
    - Determining if the certificate is known to your services.
    - Constructing your own principal. Consider the following example:

C#

```
builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate(options =>
    {
        options.Events = new CertificateAuthenticationEvents
        {
            OnCertificateValidated = context =>
            {
                var claims = new[]
                {
                    new Claim(
                        ClaimTypes.NameIdentifier,
                        context.ClientCertificate.Subject,
                        ClaimValueTypes.String,
context.Options.ClaimsIssuer),
                    new Claim(
                        ClaimTypes.Name,
                        context.ClientCertificate.Subject,
                        ClaimValueTypes.String,
context.Options.ClaimsIssuer)
                };

                context.Principal = new ClaimsPrincipal(
                    new ClaimsIdentity(claims,
context.Scheme.Name));
                context.Success();

                return Task.CompletedTask;
            }
        };
    });
```

If you find the inbound certificate doesn't meet your extra validation, call `context.Fail("failure reason")` with a failure reason.

For better functionality, call a service registered in dependency injection that connects to a database or other type of user store. Access the service by using the context passed into the delegate. Consider the following example:

C#

```
builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate(options =>
    {
        options.Events = new CertificateAuthenticationEvents
        {
            OnCertificateValidated = context =>
            {
```

```
                var validationService = context.HttpContext.RequestServices
                    .GetRequiredService<ICertificateValidationService>();

                if
(validationService.ValidateCertificate(context.ClientCertificate))
                {
                    var claims = new[]
                    {
                        new Claim(
                            ClaimTypes.NameIdentifier,
                            context.ClientCertificate.Subject,
                            ClaimValueTypes.String,
context.Options.ClaimsIssuer),
                        new Claim(
                            ClaimTypes.Name,
                            context.ClientCertificate.Subject,
                            ClaimValueTypes.String,
context.Options.ClaimsIssuer)
                    };

                    context.Principal = new ClaimsPrincipal(
                        new ClaimsIdentity(claims, context.Scheme.Name));
                    context.Success();
                }

                return Task.CompletedTask;
            }
        };
    });
```

Conceptually, the validation of the certificate is an authorization concern. Adding a check on, for example, an issuer or thumbprint in an authorization policy, rather than inside `OnCertificateValidated`, is perfectly acceptable.

Configure your server to require certificates

Kestrel

In `Program.cs`, configure Kestrel as follows:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<KestrelServerOptions>(options =>
{
    options.ConfigureHttpsDefaults(options =>
        options.ClientCertificateMode =
```

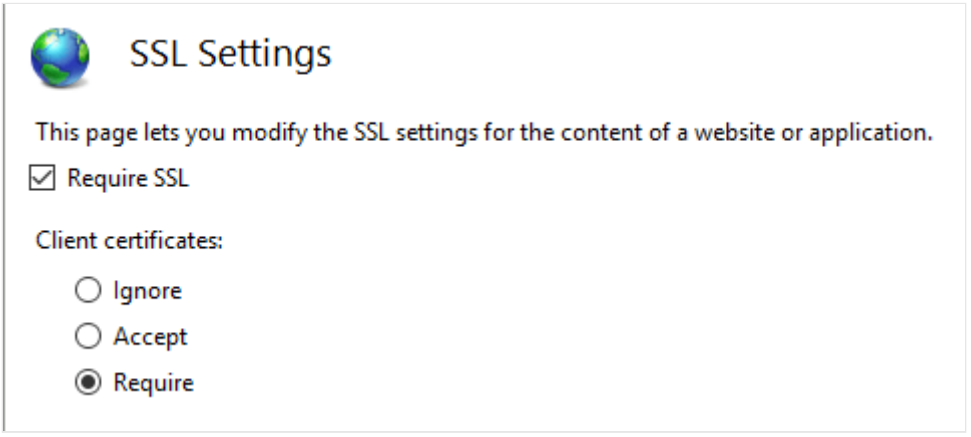
```
ClientCertificateMode.RequireCertificate);
});
```

> ⓘ Note
>
> Endpoints created by calling Listen **before** calling ConfigureHttpsDefaults won't have the defaults applied.

IIS

Complete the following steps in IIS Manager:

1. Select your site from the **Connections** tab.
2. Double-click the **SSL Settings** option in the **Features View** window.
3. Check the **Require SSL** checkbox, and select the **Require** radio button in the **Client certificates** section.



Azure and custom web proxies

See the host and deploy documentation for how to configure the certificate forwarding middleware.

Use certificate authentication in Azure Web Apps

No forwarding configuration is required for Azure. Forwarding configuration is set up by the Certificate Forwarding Middleware.

> ⓘ Note
>
> Certificate Forwarding Middleware is required for this scenario.

For more information, see Use a TLS/SSL certificate in your code in Azure App Service (Azure documentation).

Use certificate authentication in custom web proxies

The `AddCertificateForwarding` method is used to specify:

- The client header name.
- How the certificate is to be loaded (using the `HeaderConverter` property).

In custom web proxies, the certificate is passed as a custom request header, for example `X-SSL-CERT`. To use it, configure certificate forwarding in `Program.cs`:

C#

```
builder.Services.AddCertificateForwarding(options =>
{
    options.CertificateHeader = "X-SSL-CERT";

    options.HeaderConverter = headerValue =>
    {
        X509Certificate2? clientCertificate = null;

        if (!string.IsNullOrWhiteSpace(headerValue))
        {
            clientCertificate = new
X509Certificate2(StringToByteArray(headerValue));
        }

        return clientCertificate!;

        static byte[] StringToByteArray(string hex)
        {
            var numberChars = hex.Length;
            var bytes = new byte[numberChars / 2];

            for (int i = 0; i < numberChars; i += 2)
            {
                bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);
            }

            return bytes;
        }
    };
});
```

If the app is reverse proxied by NGINX with the configuration `proxy_set_header ssl-client-cert $ssl_client_escaped_cert` or deployed on Kubernetes using NGINX Ingress,

the client certificate is passed to the app in URL-encoded form. To use the certificate, decode it as follows:

C#

```
builder.Services.AddCertificateForwarding(options =>
{
    options.CertificateHeader = "ssl-client-cert";

    options.HeaderConverter = (headerValue) =>
    {
        X509Certificate2? clientCertificate = null;

        if (!string.IsNullOrWhiteSpace(headerValue))
        {
            clientCertificate = X509Certificate2.CreateFromPem(
                WebUtility.UrlDecode(headerValue));
        }

        return clientCertificate!;
    };
});
```

Add the middleware in `Program.cs`. `UseCertificateForwarding` is called before the calls to `UseAuthentication` and `UseAuthorization`:

C#

```
var app = builder.Build();

app.UseCertificateForwarding();

app.UseAuthentication();
app.UseAuthorization();
```

A separate class can be used to implement validation logic. Because the same self-signed certificate is used in this example, ensure that only your certificate can be used. Validate that the thumbprints of both the client certificate and the server certificate match, otherwise any certificate can be used and will be enough to authenticate. This would be used inside the `AddCertificate` method. You could also validate the subject or the issuer here if you're using intermediate or child certificates.

C#

```
using System.Security.Cryptography.X509Certificates;

namespace CertAuthSample.Snippets;
```

```
public class SampleCertificateValidationService :
ICertificateValidationService
{
    public bool ValidateCertificate(X509Certificate2 clientCertificate)
    {
        // Don't hardcode passwords in production code.
        // Use a certificate thumbprint or Azure Key Vault.
        var expectedCertificate = new X509Certificate2(
            Path.Combine("/path/to/pfx"), "1234");

        return clientCertificate.Thumbprint ==
expectedCertificate.Thumbprint;
    }
}
```

Implement an HttpClient using a certificate and IHttpClientFactory

In the following example, a client certificate is added to a `HttpClientHandler` using the `ClientCertificates` property from the handler. This handler can then be used in a named instance of an `HttpClient` using the ConfigurePrimaryHttpMessageHandler method. This is setup in `Program.cs`:

C#

```
var clientCertificate =
    new X509Certificate2(
      Path.Combine(_environment.ContentRootPath, "sts_dev_cert.pfx"),
"1234");

builder.Services.AddHttpClient("namedClient", c =>
{
}).ConfigurePrimaryHttpMessageHandler(() =>
{
    var handler = new HttpClientHandler();
    handler.ClientCertificates.Add(clientCertificate);
    return handler;
});
```

The `IHttpClientFactory` can then be used to get the named instance with the handler and the certificate. The `CreateClient` method with the name of the client defined in `Program.cs` is used to get the instance. The HTTP request can be sent using the client as required:

C#

```
public class SampleHttpService
{
    private readonly IHttpClientFactory _httpClientFactory;
```

```
    public SampleHttpService(IHttpClientFactory httpClientFactory)
        => _httpClientFactory = httpClientFactory;

    public async Task<JsonDocument> GetAsync()
    {
        var httpClient = _httpClientFactory.CreateClient("namedClient");
        var httpResponseMessage = await
httpClient.GetAsync("https://example.com");

        if (httpResponseMessage.IsSuccessStatusCode)
        {
            return JsonDocument.Parse(
                await httpResponseMessage.Content.ReadAsStringAsync());
        }

        throw new ApplicationException($"Status code:
{httpResponseMessage.StatusCode}");
    }
}
```

If the correct certificate is sent to the server, the data is returned. If no certificate or the wrong certificate is sent, an HTTP 403 status code is returned.

Create certificates in PowerShell

Creating the certificates is the hardest part in setting up this flow. A root certificate can be created using the `New-SelfSignedCertificate` PowerShell cmdlet. When creating the certificate, use a strong password. It's important to add the `KeyUsageProperty` parameter and the `KeyUsage` parameter as shown.

Create root CA

PowerShell

```
New-SelfSignedCertificate -DnsName "root_ca_dev_damienbod.com",
"root_ca_dev_damienbod.com" -CertStoreLocation "cert:\LocalMachine\My" -
NotAfter (Get-Date).AddYears(20) -FriendlyName "root_ca_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath C:\git\root_ca_dev_damienbod.pfx -Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
root_ca_dev_damienbod.crt
```

> **Note**
>
> The `-DnsName` parameter value must match the deployment target of the app. For example, "localhost" for development.

Install in the trusted root

The root certificate must be trusted on your host system. Only root certificates created by a certificate authority are trusted by default. For information on how to trust the root certificate on Windows, see the Windows documentation or the Import-Certificate PowerShell cmdlet.

Intermediate certificate

An intermediate certificate can now be created from the root certificate. This isn't required for all use cases, but you might need to create many certificates or need to activate or disable groups of certificates. The `TextExtension` parameter is required to set the path length in the basic constraints of the certificate.

The intermediate certificate can then be added to the trusted intermediate certificate in the Windows host system.

PowerShell

```
$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

$parentcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint of
the root..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"intermediate_dev_damienbod.com" -Signer $parentcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "intermediate_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature -
TextExtension @("2.5.29.19={text}CA=1&pathlength=1")

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\intermediate_dev_damienbod.pfx -
Password $mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
intermediate_dev_damienbod.crt
```

Create child certificate from intermediate certificate

A child certificate can be created from the intermediate certificate. This is the end entity and doesn't need to create more child certificates.

PowerShell

```
$parentcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint
from the Intermediate certificate..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_a_dev_damienbod.com" -Signer $parentcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com"

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password
$mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
child_a_dev_damienbod.crt
```

Create child certificate from root certificate

A child certificate can also be created from the root certificate directly.

PowerShell

```
$rootcert = ( Get-ChildItem -Path cert:\LocalMachine\My\"The thumbprint from
the root cert..." )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_a_dev_damienbod.com" -Signer $rootcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com"

$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

Get-ChildItem -Path cert:\localMachine\my\"The thumbprint..." | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password
$mypwd

Export-Certificate -Cert cert:\localMachine\my\"The thumbprint..." -FilePath
child_a_dev_damienbod.crt
```

Example root - intermediate certificate - certificate

PowerShell

```
$mypwdroot = ConvertTo-SecureString -String "1234" -Force -AsPlainText
$mypwd = ConvertTo-SecureString -String "1234" -Force -AsPlainText

New-SelfSignedCertificate -DnsName "root_ca_dev_damienbod.com",
"root_ca_dev_damienbod.com" -CertStoreLocation "cert:\LocalMachine\My" -
NotAfter (Get-Date).AddYears(20) -FriendlyName "root_ca_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature

Get-ChildItem -Path
cert:\localMachine\my\0C89639E4E2998A93E423F919B36D4009A0F9991 | Export-
PfxCertificate -FilePath C:\git\root_ca_dev_damienbod.pfx -Password
$mypwdroot

Export-Certificate -Cert
cert:\localMachine\my\0C89639E4E2998A93E423F919B36D4009A0F9991 -FilePath
root_ca_dev_damienbod.crt

$rootcert = ( Get-ChildItem -Path
cert:\LocalMachine\My\0C89639E4E2998A93E423F919B36D4009A0F9991 )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_a_dev_damienbod.com" -Signer $rootcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_a_dev_damienbod.com" -
KeyUsageProperty All -KeyUsage CertSign, CRLSign, DigitalSignature -
TextExtension @("2.5.29.19={text}CA=1&pathlength=1")

Get-ChildItem -Path
cert:\localMachine\my\BA9BF91ED35538A01375EFC212A2F46104B33A44 | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_a_dev_damienbod.pfx -Password
$mypwd

Export-Certificate -Cert
cert:\localMachine\my\BA9BF91ED35538A01375EFC212A2F46104B33A44 -FilePath
child_a_dev_damienbod.crt

$parentcert = ( Get-ChildItem -Path
cert:\LocalMachine\My\BA9BF91ED35538A01375EFC212A2F46104B33A44 )

New-SelfSignedCertificate -certstorelocation cert:\localmachine\my -dnsname
"child_b_from_a_dev_damienbod.com" -Signer $parentcert -NotAfter (Get-
Date).AddYears(20) -FriendlyName "child_b_from_a_dev_damienbod.com"

Get-ChildItem -Path
cert:\localMachine\my\141594A0AE38CBBECED7AF680F7945CD51D8F28A | Export-
PfxCertificate -FilePath
C:\git\AspNetCoreCertificateAuth\Certs\child_b_from_a_dev_damienbod.pfx -
Password $mypwd

Export-Certificate -Cert
cert:\localMachine\my\141594A0AE38CBBECED7AF680F7945CD51D8F28A -FilePath
child_b_from_a_dev_damienbod.crt
```

When using the root, intermediate, or child certificates, the certificates can be validated using the Thumbprint or PublicKey as required:

C#

```
using System.Security.Cryptography.X509Certificates;

namespace CertAuthSample.Snippets;

public class SampleCertificateThumbprintsValidationService :
ICertificateValidationService
{
    private readonly string[] validThumbprints = new[]
    {
        "141594A0AE38CBBECED7AF680F7945CD51D8F28A",
        "0C89639E4E2998A93E423F919B36D4009A0F9991",
        "BA9BF91ED35538A01375EFC212A2F46104B33A44"
    };

    public bool ValidateCertificate(X509Certificate2 clientCertificate)
        => validThumbprints.Contains(clientCertificate.Thumbprint);
}
```

Certificate validation caching

ASP.NET Core 5.0 and later versions support the ability to enable caching of validation results. The caching dramatically improves performance of certificate authentication, as validation is an expensive operation.

By default, certificate authentication disables caching. To enable caching, call `AddCertificateCache` in `Program.cs`:

C#

```
builder.Services.AddAuthentication(
        CertificateAuthenticationDefaults.AuthenticationScheme)
    .AddCertificate()
    .AddCertificateCache(options =>
    {
        options.CacheSize = 1024;
        options.CacheEntryExpiration = TimeSpan.FromMinutes(2);
    });
```

The default caching implementation stores results in memory. You can provide your own cache by implementing `ICertificateValidationCache` and registering it with dependency injection. For example, `services.AddSingleton<ICertificateValidationCache, YourCache>()`.

Optional client certificates

This section provides information for apps that must protect a subset of the app with a certificate. For example, a Razor Page or controller in the app might require client certificates. This presents challenges as client certificates:

- Are a TLS feature, not an HTTP feature.
- Are negotiated per-connection and usually at the start of the connection before any HTTP data is available.

There are two approaches to implementing optional client certificates:

1. Using separate host names (SNI) and redirecting. While more work to configure, this is recommended because it works in most environments and protocols.
2. Renegotiation during an HTTP request. This has several limitations and is not recommended.

Separate Hosts (SNI)

At the start of the connection, only the Server Name Indication (SNI)† is known. Client certificates can be configured per host name so that one host requires them and another does not.

- Set up binding for the domain and subdomain:
  - For example, set up bindings on `contoso.com` and `myClient.contoso.com`. The `contoso.com` host doesn't require a client certificate but `myClient.contoso.com` does.
  - For more information, see:
    - Kestrel web server in ASP.NET Core:
      - ListenOptions.UseHttps
      - ClientCertificateMode
    - IIS
      - Hosting IIS
      - Configure security on IIS
    - HTTP.sys: Configure Windows Server

ASP.NET Core 5 and later adds more convenient support for redirecting to acquire optional client certificates. For more information, see the Optional certificates sample.

- For requests to the web app that require a client certificate and don't have one:
  - Redirect to the same page using the client certificate protected subdomain.

    - For example, redirect to `myClient.contoso.com/requestedPage`. Because the request to `myClient.contoso.com/requestedPage` is a different hostname than `contoso.com/requestedPage`, the client establishes a different connection and the client certificate is provided.
    - For more information, see Introduction to authorization in ASP.NET Core.

† Server Name Indication (SNI) is a TLS extension to include a virtual domain as a part of SSL negotiation. This effectively means the virtual domain name, or a hostname, can be used to identify the network end point.

Renegotiation

TLS renegotiation is a process by which the client and server can re-assess the encryption requirements for an individual connection, including requesting a client certificate if not previously provided. TLS renegotiation is a security risk and isn't recommended because:

- In HTTP/1.1 the server must first buffer or consume any HTTP data that is in flight such as POST request bodies to make sure the connection is clear for the renegotiation. Otherwise the renegotiation can stop responding or fail.
- HTTP/2 and HTTP/3 explicitly prohibit renegotiation.
- There are security risks associated with renegotiation. TLS 1.3 removed renegotiation of the whole connection and replaced it with a new extension for requesting only the client certificate after the start of the connection. This mechanism is exposed via the same APIs and is still subject to the prior constraints of buffering and HTTP protocol versions.

The implementation and configuration of this feature varies by server and framework version.

IIS

IIS manages the client certificate negotiation on your behalf. A subsection of the application can enable the `SslRequireCert` option to negotiate the client certificate for those requests. See Configuration in the IIS documentation for details.

IIS will automatically buffer any request body data up to a configured size limit before renegotiating. Requests that exceed the limit are rejected with a 413 response. This limit defaults to 48KB and is configurable by setting the uploadReadAheadSize.

HttpSys

HttpSys has two settings which control the client certificate negotiation and both should be set. The first is in netsh.exe under `http add sslcert clientcertnegotiation=enable/disable`. This flag indicates if the client certificate should be negotiated at the start of a connection and it should be set to `disable` for optional client certificates. See the netsh docs for details.

The other setting is ClientCertificateMethod. When set to `AllowRenegotation`, the client certificate can be renegotiated during a request.

NOTE The application should buffer or consume any request body data before attempting the renegotiation, otherwise the request may become unresponsive.

An application can first check the ClientCertificate property to see if the certificate is available. If it is not available, ensure the request body has been consumed before calling GetClientCertificateAsync to negotiate one. Note `GetClientCertificateAsync` can return a null certificate if the client declines to provide one.

NOTE The behavior of the `ClientCertificate` property changed in .NET 6. For more information, see this GitHub issue.

Kestrel

Kestrel controls client certificate negotiation with the ClientCertificateMode option.

ClientCertificateMode.DelayCertificate is new option available in .NET 6 or later. When set, an app can check the ClientCertificate property to see if the certificate is available. If it isn't available, ensure the request body has been consumed before calling GetClientCertificateAsync to negotiate one. Note `GetClientCertificateAsync` can return a null certificate if the client declines to provide one.

NOTE The application should buffer or consume any request body data before attempting the renegotiation, otherwise `GetClientCertificateAsync` may throw `InvalidOperationException: Client stream needs to be drained before renegotiation.`.

If you're programmatically configuring the TLS settings per SNI host name, call the UseHttps overload (.NET 6 or later) that takes TlsHandshakeCallbackOptions and controls client certificate renegotiation via TlsHandshakeCallbackContext.AllowDelayedClientCertificateNegotation.

Leave questions, comments, and other feedback on optional client certificates in this GitHub discussion issue.

Configure Windows Authentication in ASP.NET Core

Article • 04/25/2023

By Rick Anderson and Kirk Larkin

Windows Authentication (also known as Negotiate, Kerberos, or NTLM authentication) can be configured for ASP.NET Core apps hosted with IIS, Kestrel, or HTTP.sys.

Windows Authentication relies on the operating system to authenticate users of ASP.NET Core apps. Windows Authentication is used for servers that run on a corporate network using Active Directory domain identities or Windows accounts to identify users. Windows Authentication is best suited to intranet environments where users, client apps, and web servers belong to the same Windows domain.

> **Note**
>
> Windows Authentication isn't supported with HTTP/2. Authentication challenges can be sent on HTTP/2 responses, but the client must downgrade to HTTP/1.1 before authenticating.

Proxy and load balancer scenarios

Windows Authentication is a stateful scenario primarily used in an intranet, where a proxy or load balancer doesn't usually handle traffic between clients and servers. If a proxy or load balancer is used, Windows Authentication only works if the proxy or load balancer:

- Handles the authentication.
- Passes the user authentication information to the app (for example, in a request header), which acts on the authentication information.

An alternative to Windows Authentication in environments where proxies and load balancers are used is Active Directory Federated Services (ADFS) with OpenID Connect (OIDC).

IIS/IIS Express

Add the NuGet package Microsoft.AspNetCore.Authentication.Negotiate and authentication services by calling AddAuthentication in `Program.cs`:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
    .AddNegotiate();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
});
builder.Services.AddRazorPages();

var app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding code was generated by the ASP.NET Core Razor Pages template with **Windows Authentication** specified.

Launch settings (debugger)

Configuration for launch settings only affects the `Properties/launchSettings.json` file for IIS Express and doesn't configure IIS for Windows Authentication. Server configuration is explained in the IIS section.

The **Web Application** templates available via Visual Studio or the .NET CLI can be configured to support Windows Authentication, which updates the `Properties/launchSettings.json` file automatically.

Visual Studio

New project

Create a new Razor Pages or MVC app. In the **Additional information** dialog, set the **Authentication type** to **Windows**.

Run the app. The username appears in the rendered app's user interface.

Existing project

The project's properties enable Windows Authentication and disable Anonymous Authentication. Open the launch profiles dialog:

1. In Solution Explorer, right click the project and select **Properties**.
2. Select the **Debug** > **General** tab and select **Open debug launch profiles UI**.
3. Clear the checkbox for **Enable Anonymous Authentication**.
4. Select the checkbox for **Enable Windows Authentication**.

Alternatively, the properties can be configured in the `iisSettings` node of the `launchSettings.json` file:

JSON

```
"iisSettings": {
    "windowsAuthentication": true,
    "anonymousAuthentication": false,
    "iisExpress": {
        "applicationUrl": "http://localhost:52171/",
        "sslPort": 44308
    }
}
```

IIS

IIS uses the ASP.NET Core Module to host ASP.NET Core apps. Windows Authentication is configured for IIS via the *web.config* file. The following sections show how to:

- Provide a local *web.config* file that activates Windows Authentication on the server when the app is deployed.
- Use the IIS Manager to configure the *web.config* file of an ASP.NET Core app that has already been deployed to the server.

If you haven't already done so, enable IIS to host ASP.NET Core apps. For more information, see Host ASP.NET Core on Windows with IIS.

Enable the IIS Role Service for Windows Authentication. For more information, see Enable Windows Authentication in IIS Role Services (see Step 2).

IIS Integration Middleware is configured to automatically authenticate requests by default. For more information, see Host ASP.NET Core on Windows with IIS: IIS options (AutomaticAuthentication).

The ASP.NET Core Module is configured to forward the Windows Authentication token to the app by default. For more information, see ASP.NET Core Module configuration reference: Attributes of the aspNetCore element.

Use **either** of the following approaches:

- **Before publishing and deploying the project,** add the following *web.config* file to the project root:

  XML

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <location path="." inheritInChildApplications="false">
      <system.webServer>
        <security>
          <authentication>
            <anonymousAuthentication enabled="false" />
            <windowsAuthentication enabled="true" />
          </authentication>
        </security>
      </system.webServer>
    </location>
  </configuration>
  ```

  When the project is published by the .NET Core SDK (without the `<IsTransformWebConfigDisabled>` property set to `true` in the project file), the published *web.config* file includes the `<location><system.webServer><security><authentication>` section. For more information on the `<IsTransformWebConfigDisabled>` property, see Host ASP.NET Core on Windows with IIS.

- **After publishing and deploying the project,** perform server-side configuration with the IIS Manager:

1. In IIS Manager, select the IIS site under the **Sites** node of the **Connections** sidebar.
2. Double-click **Authentication** in the **IIS** area.
3. Select **Anonymous Authentication**. Select **Disable** in the **Actions** sidebar.
4. Select **Windows Authentication**. Select **Enable** in the **Actions** sidebar.

When these actions are taken, IIS Manager modifies the app's *web.config* file. A `<system.webServer><security><authentication>` node is added with updated settings for `anonymousAuthentication` and `windowsAuthentication`:

XML

```
<system.webServer>
  <security>
    <authentication>
      <anonymousAuthentication enabled="false" />
      <windowsAuthentication enabled="true" />
    </authentication>
  </security>
</system.webServer>
```

The `<system.webServer>` section added to the *web.config* file by IIS Manager is outside of the app's `<location>` section added by the .NET Core SDK when the app is published. Because the section is added outside of the `<location>` node, the settings are inherited by any sub-apps to the current app. To prevent inheritance, move the added `<security>` section inside of the `<location><system.webServer>` section that the .NET Core SDK provided.

When IIS Manager is used to add the IIS configuration, it only affects the app's *web.config* file on the server. A subsequent deployment of the app may overwrite the settings on the server if the server's copy of *web.config* is replaced by the project's *web.config* file. Use **either** of the following approaches to manage the settings:

- Use IIS Manager to reset the settings in the *web.config* file after the file is overwritten on deployment.
- Add a *web.config file* to the app locally with the settings.

Kestrel

The Microsoft.AspNetCore.Authentication.Negotiate NuGet package can be used with Kestrel to support Windows Authentication using Negotiate and Kerberos on Windows, Linux, and macOS.

> **⚠ Warning**
>
> Credentials can be persisted across requests on a connection. *Negotiate authentication must not be used with proxies unless the proxy maintains a 1:1 connection affinity (a persistent connection) with Kestrel.*

> **ⓘ Note**
>
> The Negotiate handler detects if the underlying server supports Windows Authentication natively and if it is enabled. If the server supports Windows Authentication but it is disabled, an error is thrown asking you to enable the server implementation. When Windows Authentication is enabled in the server, the Negotiate handler transparently forwards authentication requests to it.

Authentication is enabled by the following highlighted code to `Program.cs`:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
   .AddNegotiate();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
});
builder.Services.AddRazorPages();

var app = builder.Build();
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();
app.MapRazorPages();
```

```
app.Run();
```

The preceding code was generated by the ASP.NET Core Razor Pages template with **Windows Authentication** specified. The following APIs are used in the preceding code:

- AddAuthentication
- AddNegotiate
- UseAuthentication

Kerberos authentication and role-based access control (RBAC)

Kerberos authentication on Linux or macOS doesn't provide any role information for an authenticated user. To add role and group information to a Kerberos user, the authentication handler must be configured to retrieve the roles from an LDAP domain. The most basic configuration only specifies an LDAP domain to query against and uses the authenticated user's context to query the LDAP domain:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;
using System.Runtime.InteropServices;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
    .AddNegotiate(options =>
    {
        if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
        {
            options.EnableLdap("contoso.com");
        }
    });
```

Some configurations may require specific credentials to query the LDAP domain. The credentials can be specified in the following highlighted options:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;
using System.Runtime.InteropServices;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
        .AddNegotiate(options =>
```

```
        {
            if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
            {
                options.EnableLdap(settings =>
                {
                    settings.Domain = "contoso.com";
                    settings.MachineAccountName = "machineName";
                    settings.MachineAccountPassword =
                                    builder.Configuration["Password"];
                });
            }
        });

builder.Services.AddRazorPages();
```

By default, the negotiate authentication handler resolves nested domains. In a large or complicated LDAP environment, resolving nested domains may result in a slow lookup or a lot of memory being used for each user. Nested domain resolution can be disabled using the `IgnoreNestedGroups` option.

Anonymous requests are allowed. Use ASP.NET Core Authorization to challenge anonymous requests for authentication.

Windows environment configuration

The Microsoft.AspNetCore.Authentication.Negotiate component performs User Mode authentication. Service Principal Names (SPNs) must be added to the user account running the service, not the machine account. Execute `setspn -S HTTP/myservername.mydomain.com myuser` in an administrative command shell.

Kerberos vs NTLM

The Negotiate package on Kestrel for ASP.NET Core attempts to use Kerberos, which is a more secure and peformant authentication scheme than NTLM:

C#

```
using Microsoft.AspNetCore.Authentication.Negotiate;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(NegotiateDefaults.AuthenticationScheme)
   .AddNegotiate();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = options.DefaultPolicy;
```

```
});
builder.Services.AddRazorPages();

var app = builder.Build();
```

NegotiateDefaults.AuthenticationScheme specifies Kerberos because it's the default.

IIS, IISExpress, and Kestrel support both Kerberos and NTLM.

Examining the WWW-Authenticate: header using IIS or IISExpress with a tool like Fiddler shows either `Negotiate` or NTLM.

Kestrel only shows `WWW-Authenticate: Negotiate`

The `WWW-Authenticate: Negotiate` header means that the server can use NTLM or Kerberos. Kestrel requires the Negotiate header prefix, it doesn't support directly specifying `NTLM` in the request or response auth headers. NTLM is supported in Kestrel, but it must be sent as `Negotiate`.

On Kestrel, to see if NTLM or Kerberos is used, Base64 decode the header and it shows either `NTLM` or `HTTP`. `HTTP` indicates Kerberos was used.

Linux and macOS environment configuration

Instructions for joining a Linux or macOS machine to a Windows domain are available in the Connect Azure Data Studio to your SQL Server using Windows authentication - Kerberos article. The instructions create a machine account for the Linux machine on the domain. SPNs must be added to that machine account.

> **Note**
>
> When following the guidance in the Connect Azure Data Studio to your SQL Server using Windows authentication - Kerberos article, replace `python-software-properties` with `python3-software-properties` if needed.

Once the Linux or macOS machine is joined to the domain, additional steps are required to provide a keytab file with the SPNs:

- On the domain controller, add new web service SPNs to the machine account:
  - `setspn -S HTTP/mywebservice.mydomain.com mymachine`
  - `setspn -S HTTP/mywebservice@MYDOMAIN.COM mymachine`
- Use ktpass to generate a keytab file:

    - `ktpass -princ HTTP/mywebservice.mydomain.com@MYDOMAIN.COM -pass myKeyTabFilePassword -mapuser MYDOMAIN\mymachine$ -pType KRB5_NT_PRINCIPAL -out c:\temp\mymachine.HTTP.keytab -crypto AES256-SHA1`
    - Some fields must be specified in uppercase as indicated.
- Copy the keytab file to the Linux or macOS machine.
- Select the keytab file via an environment variable: `export KRB5_KTNAME=/tmp/mymachine.HTTP.keytab`
- Invoke `klist` to show the SPNs currently available for use.

> ⓘ Note
>
> A keytab file contains domain access credentials and must be protected accordingly.

HTTP.sys

HTTP.sys supports Kernel Mode Windows Authentication using Negotiate, NTLM, or Basic authentication.

The following code adds authentication and configures the app's web host to use HTTP.sys with Windows Authentication:

C#

```
using Microsoft.AspNetCore.Server.HttpSys;
using System.Runtime.InteropServices;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);

if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
{
    builder.WebHost.UseHttpSys(options =>
        {
            options.Authentication.Schemes =
                AuthenticationSchemes.NTLM |
                AuthenticationSchemes.Negotiate;
            options.Authentication.AllowAnonymous = false;
        });
}
```

> ⓘ Note

HTTP.sys delegates to Kernel Mode authentication with the Kerberos authentication protocol. User Mode authentication isn't supported with Kerberos and HTTP.sys. The machine account must be used to decrypt the Kerberos token/ticket that's obtained from Active Directory and forwarded by the client to the server to authenticate the user. Register the Service Principal Name (SPN) for the host, not the user of the app.

> **Note**
>
> HTTP.sys isn't supported on Nano Server version 1709 or later. To use Windows Authentication and HTTP.sys with Nano Server, use a Server Core (microsoft/windowsservercore) container (see `https://hub.docker.com/_/microsoft-windows-servercore`). For more information on Server Core, see What is the Server Core installation option in Windows Server?.

Authorize users

The configuration state of anonymous access determines the way in which the [Authorize] and [AllowAnonymous] attributes are used in the app. The following two sections explain how to handle the disallowed and allowed configuration states of anonymous access.

Disallow anonymous access

When Windows Authentication is enabled and anonymous access is disabled, the [Authorize] and [AllowAnonymous] attributes have no effect. If an IIS site is configured to disallow anonymous access, the request never reaches the app. For this reason, the [AllowAnonymous] attribute isn't applicable.

Allow anonymous access

When both Windows Authentication and anonymous access are enabled, use the [Authorize] and [AllowAnonymous] attributes. The [Authorize] attribute allows you to secure endpoints of the app which require authentication. The [AllowAnonymous] attribute overrides the [Authorize] attribute in apps that allow anonymous access. For attribute usage details, see Simple authorization in ASP.NET Core.

> **Note**

By default, users who lack authorization to access a page are presented with an empty HTTP 403 response. The StatusCodePages Middleware can be configured to provide users with a better "Access Denied" experience.

Impersonation

ASP.NET Core doesn't implement impersonation. Apps run with the app's identity for all requests, using app pool or process identity. If the app should perform an action on behalf of a user, use WindowsIdentity.RunImpersonated or RunImpersonatedAsync in a terminal inline middleware in `Program.cs`. Run a single action in this context and then close the context.

C#

```
app.Run(async (context) =>
{
    try
    {
        var user = (WindowsIdentity)context.User.Identity!;

        await context.Response
            .WriteAsync($"User: {user.Name}\tState:
{user.ImpersonationLevel}\n");

        await WindowsIdentity.RunImpersonatedAsync(user.AccessToken, async
() =>
        {
            var impersonatedUser = WindowsIdentity.GetCurrent();
            var message =
                $"User: {impersonatedUser.Name}\t" +
                $"State: {impersonatedUser.ImpersonationLevel}";

            var bytes = Encoding.UTF8.GetBytes(message);
            await context.Response.Body.WriteAsync(bytes, 0, bytes.Length);
        });
    }
    catch (Exception e)
    {
        await context.Response.WriteAsync(e.ToString());
    }
});
```

While the Microsoft.AspNetCore.Authentication.Negotiate package enables authentication on Windows, Linux, and macOS, impersonation is only supported on Windows.

Claims transformations

When hosting with IIS, AuthenticateAsync isn't called internally to initialize a user. Therefore, an IClaimsTransformation implementation used to transform claims after every authentication isn't activated by default. For more information and a code example that activates claims transformations, see Differences between in-process and out-of-process hosting.

Additional resources

- dotnet publish
- Host ASP.NET Core on Windows with IIS
- ASP.NET Core Module (ANCM) for IIS
- Visual Studio publish profiles (.pubxml) for ASP.NET Core app deployment

Authenticate users with WS-Federation in ASP.NET Core

Article • 09/07/2023

This tutorial demonstrates how to enable users to sign in with a WS-Federation authentication provider like Active Directory Federation Services (ADFS) or Microsoft Entra ID. It uses the ASP.NET Core sample app described in Facebook, Google, and external provider authentication.

For ASP.NET Core apps, WS-Federation support is provided by Microsoft.AspNetCore.Authentication.WsFederation. This component is ported from Microsoft.Owin.Security.WsFederation and shares many of that component's mechanics. However, the components differ in a couple of important ways.

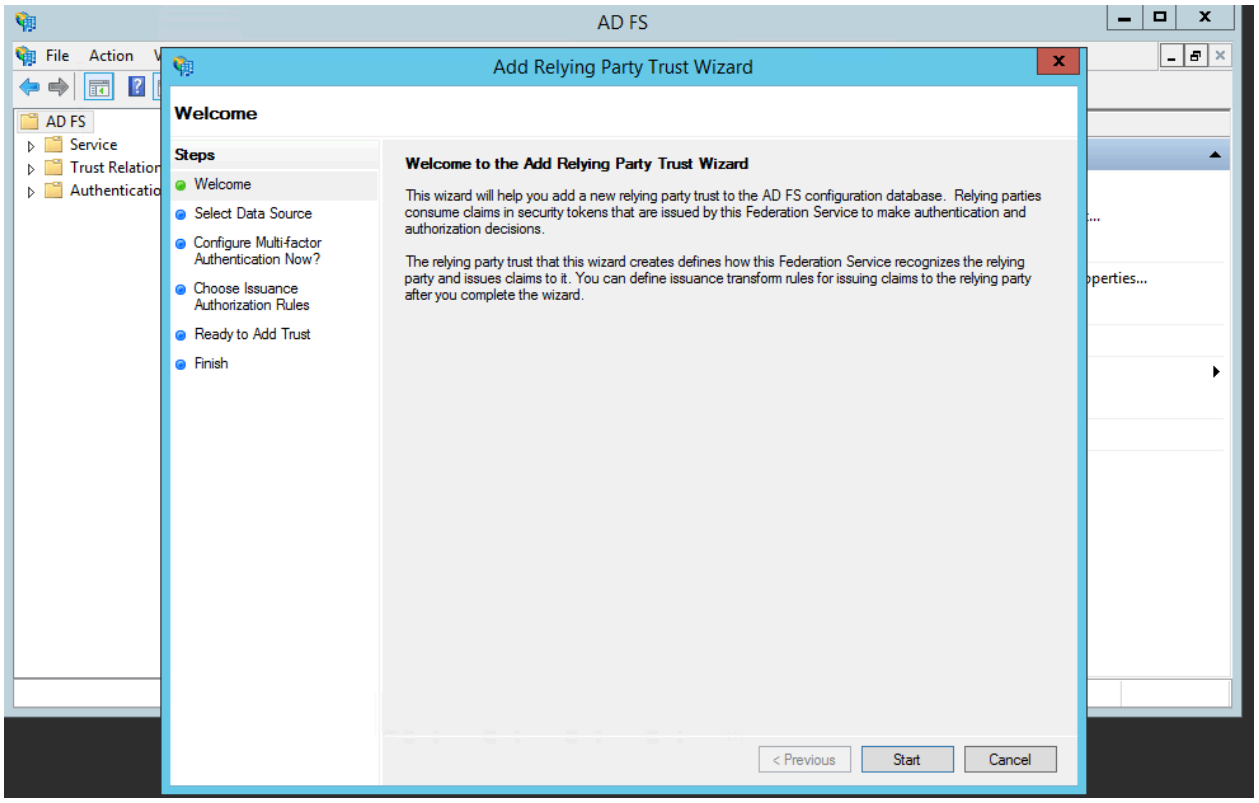
By default, the new middleware:

- Doesn't allow unsolicited logins. This feature of the WS-Federation protocol is vulnerable to XSRF attacks. However, it can be enabled with the `AllowUnsolicitedLogins` option.
- Doesn't check every form post for sign-in messages. Only requests to the `CallbackPath` are checked for sign-ins. `CallbackPath` defaults to `/signin-wsfed` but can be changed via the inherited RemoteAuthenticationOptions.CallbackPath property of the WsFederationOptions class. This path can be shared with other authentication providers by enabling the SkipUnrecognizedRequests option.

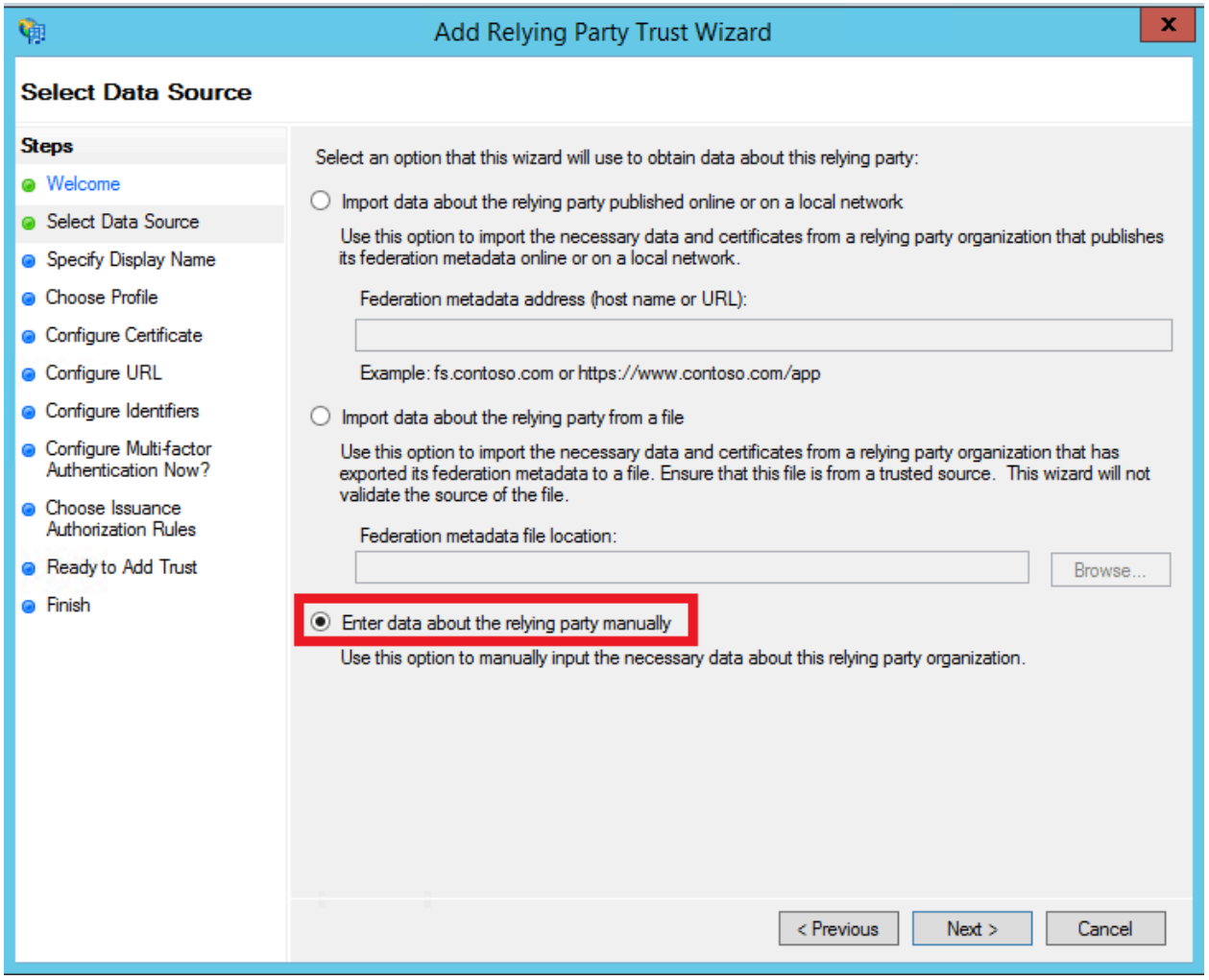
Register the app with Active Directory

Active Directory Federation Services

- Open the server's **Add Relying Party Trust Wizard** from the ADFS Management console:

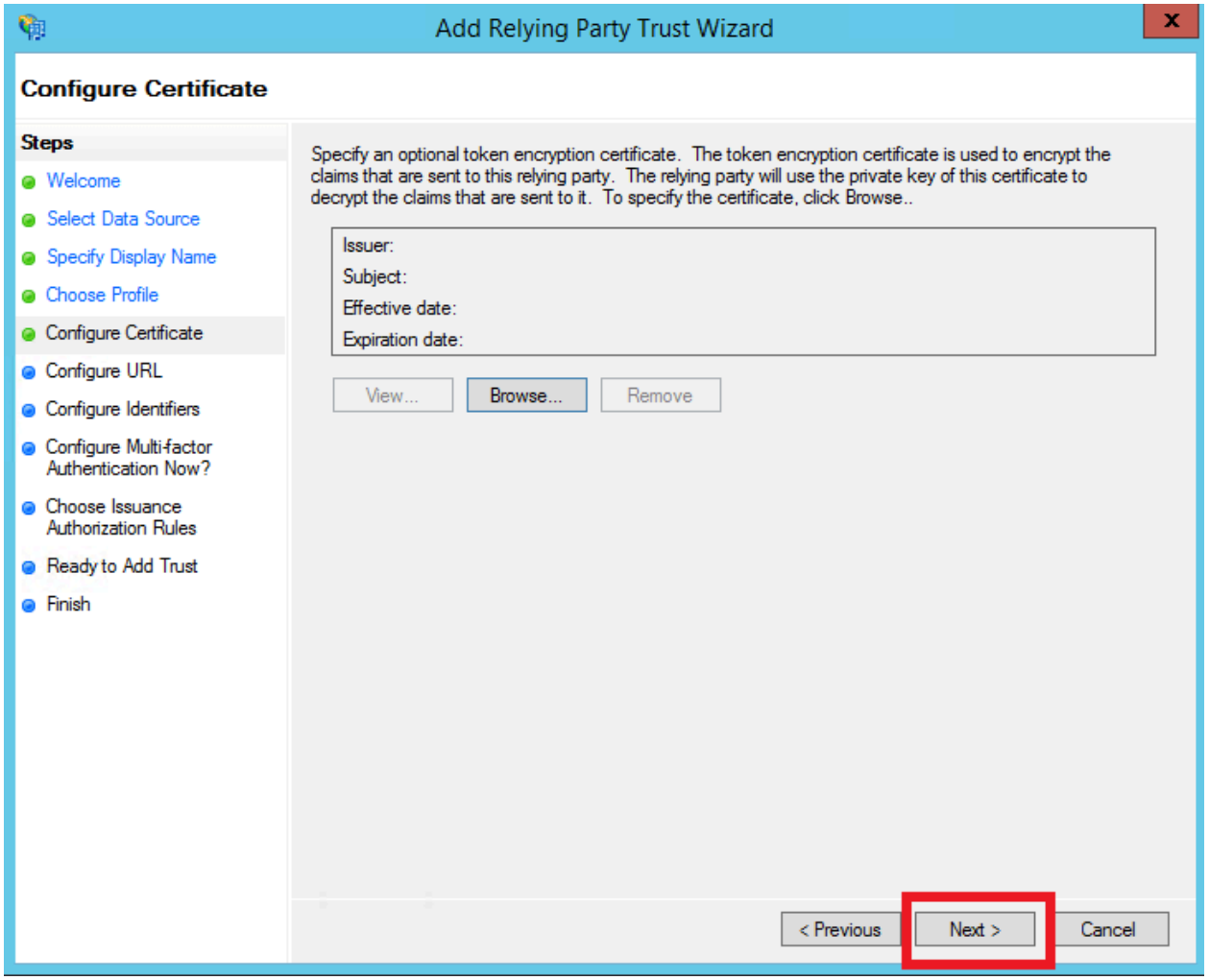


- Choose to enter data manually:

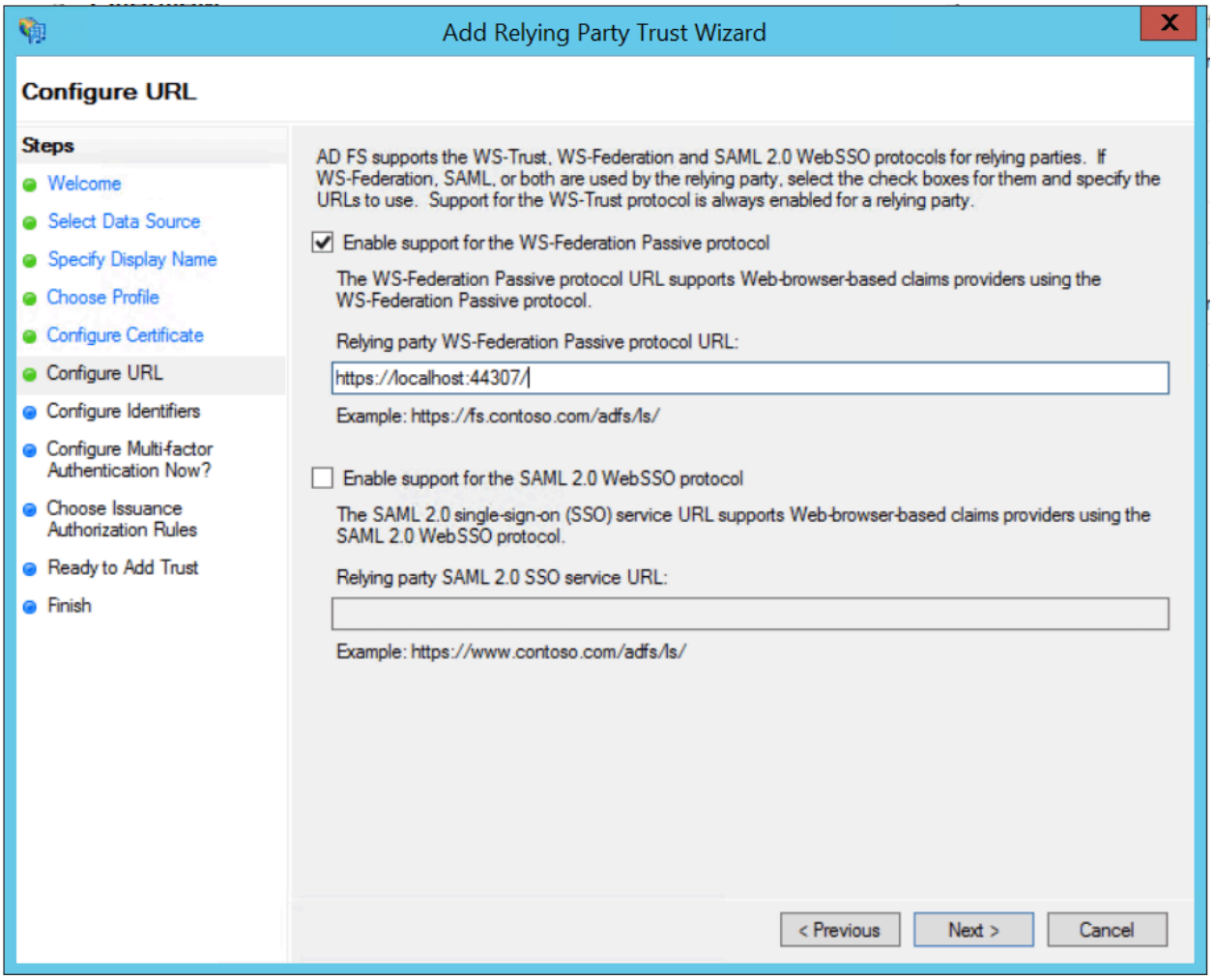


- Enter a display name for the relying party. The name isn't important to the ASP.NET Core app.

- Microsoft.AspNetCore.Authentication.WsFederation lacks support for token encryption, so don't configure a token encryption certificate:

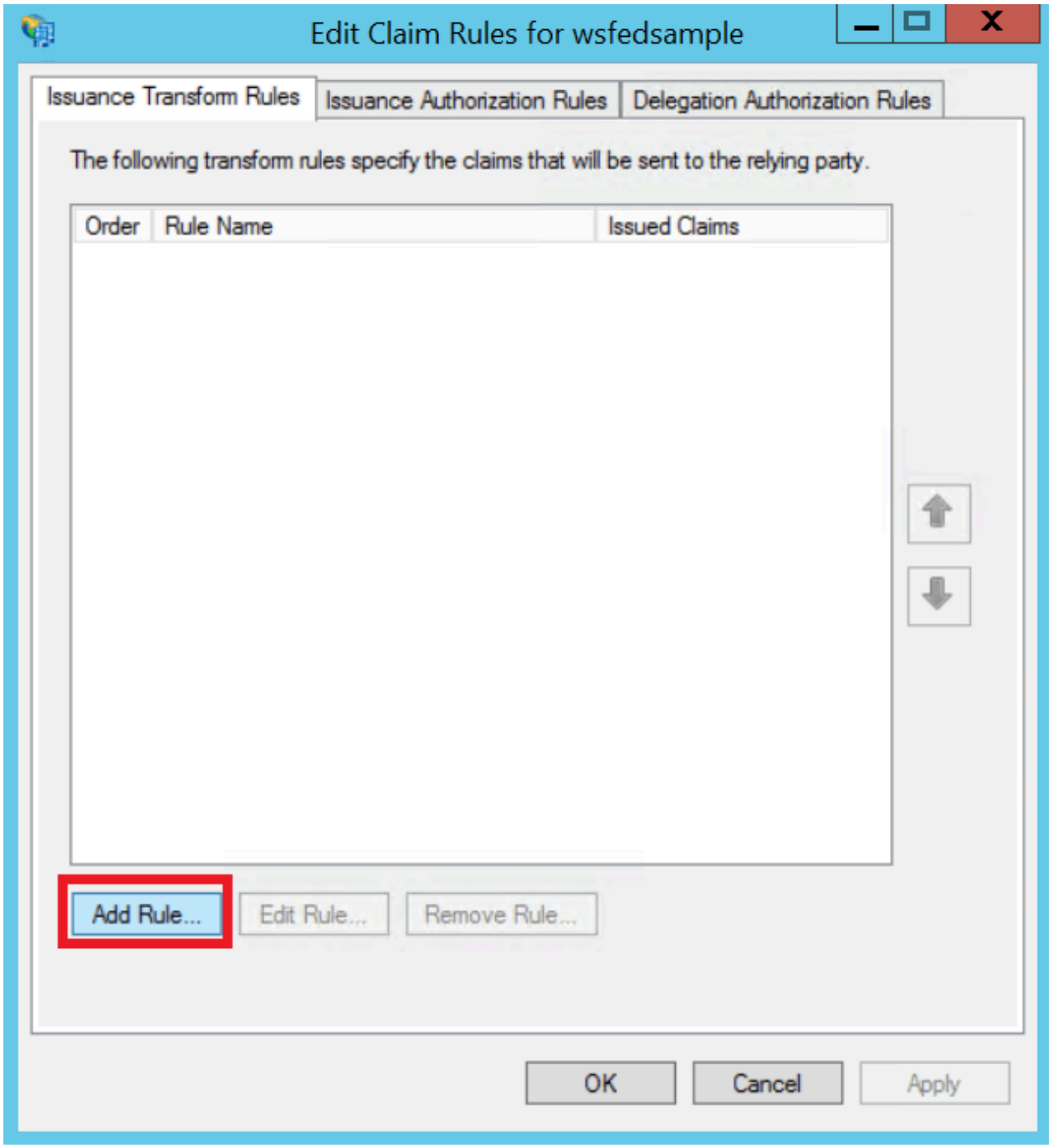


- Enable support for WS-Federation Passive protocol, using the app's URL. Verify the port is correct for the app:

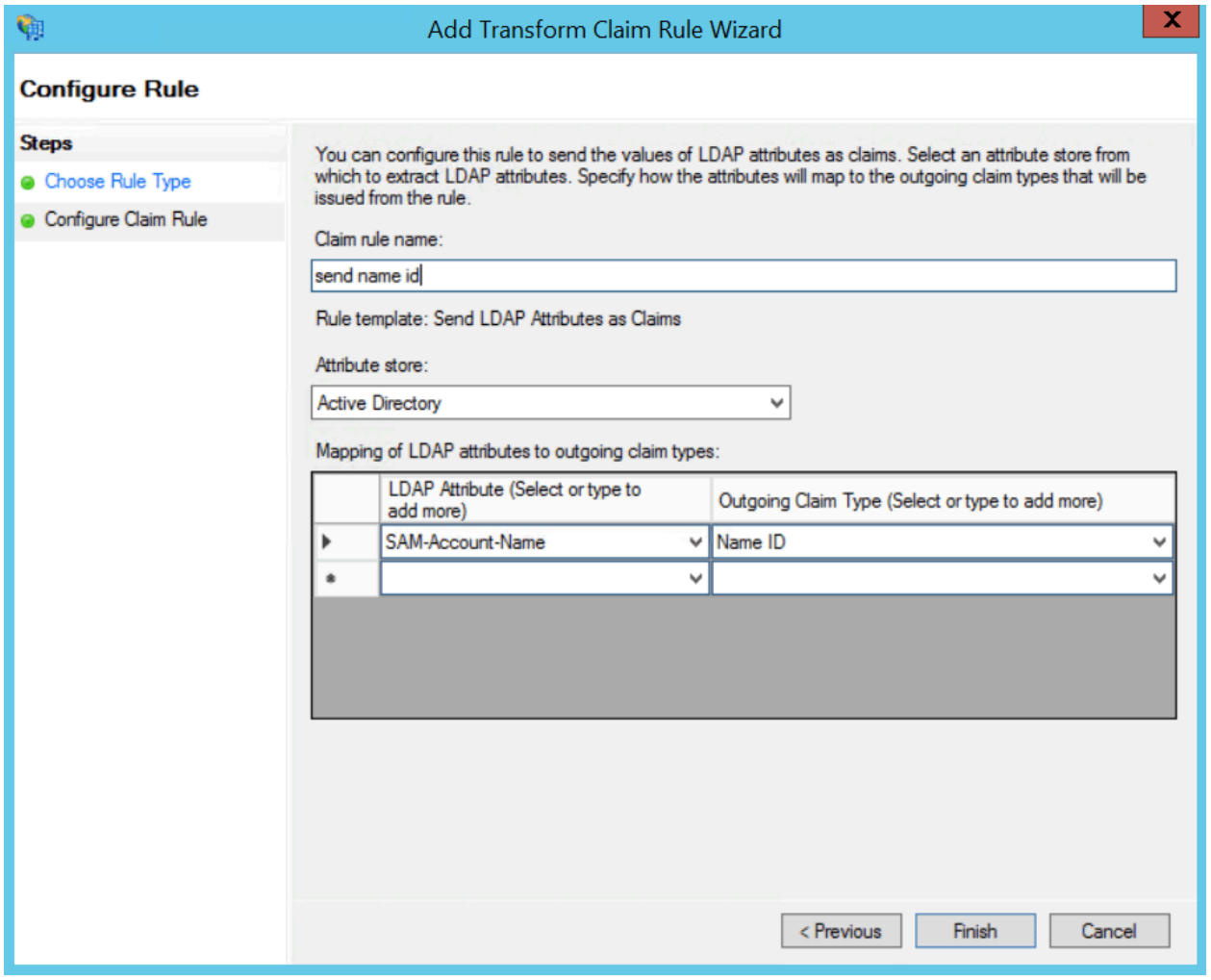


> ⓘ **Note**
>
> This must be an HTTPS URL. IIS Express can provide a self-signed certificate when hosting the app during development. Kestrel requires manual certificate configuration. See the Kestrel documentation for more details.

- Click **Next** through the rest of the wizard and **Close** at the end.
- ASP.NET Core Identity requires a **Name ID** claim. Add one from the **Edit Claim Rules** dialog:



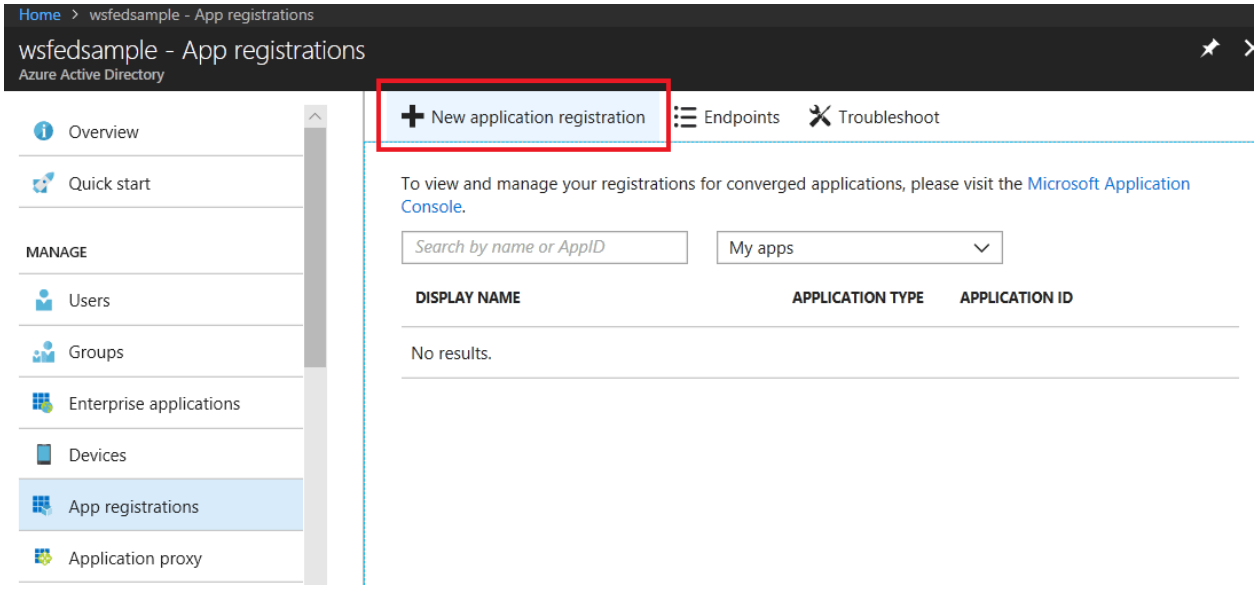
- In the **Add Transform Claim Rule Wizard**, leave the default **Send LDAP Attributes as Claims** template selected, and click **Next**. Add a rule mapping the **SAM-Account-Name** LDAP attribute to the **Name ID** outgoing claim:



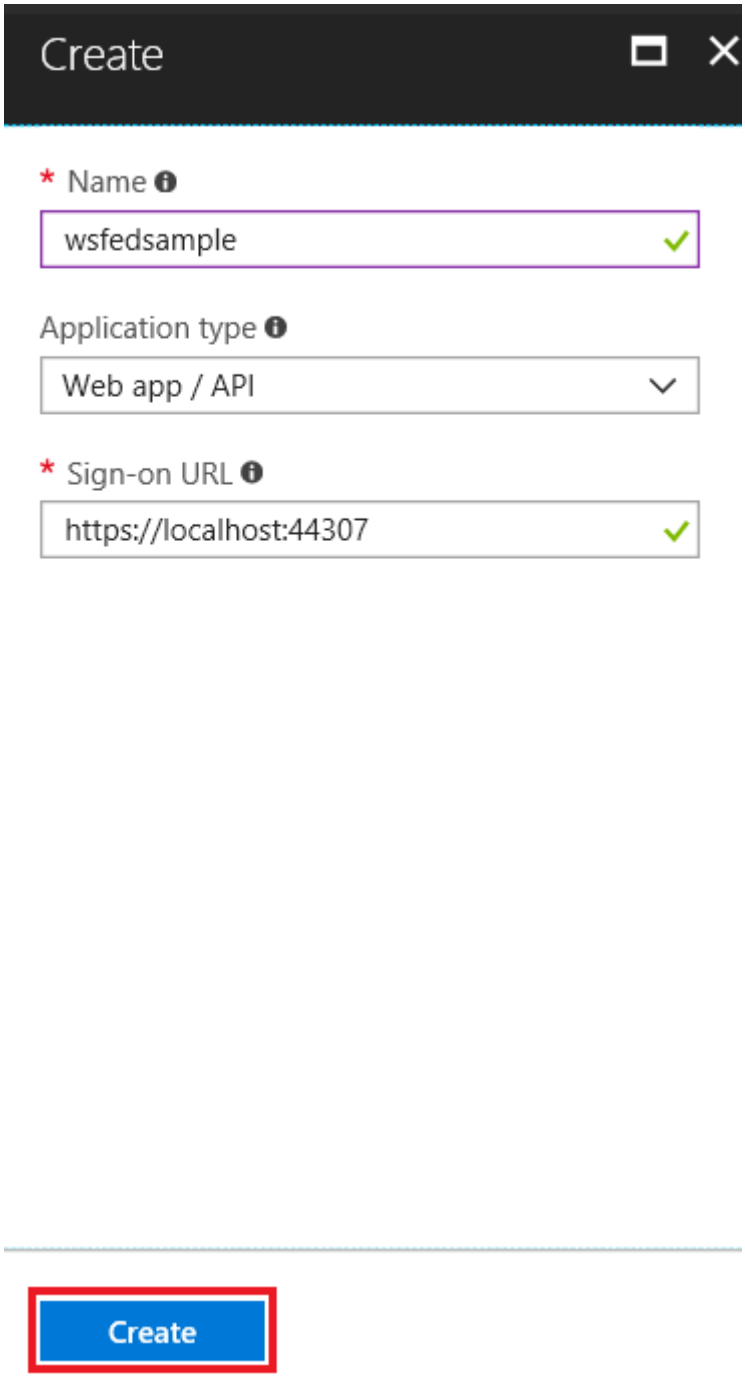
- Click **Finish** > **OK** in the **Edit Claim Rules** window.

Microsoft Entra ID

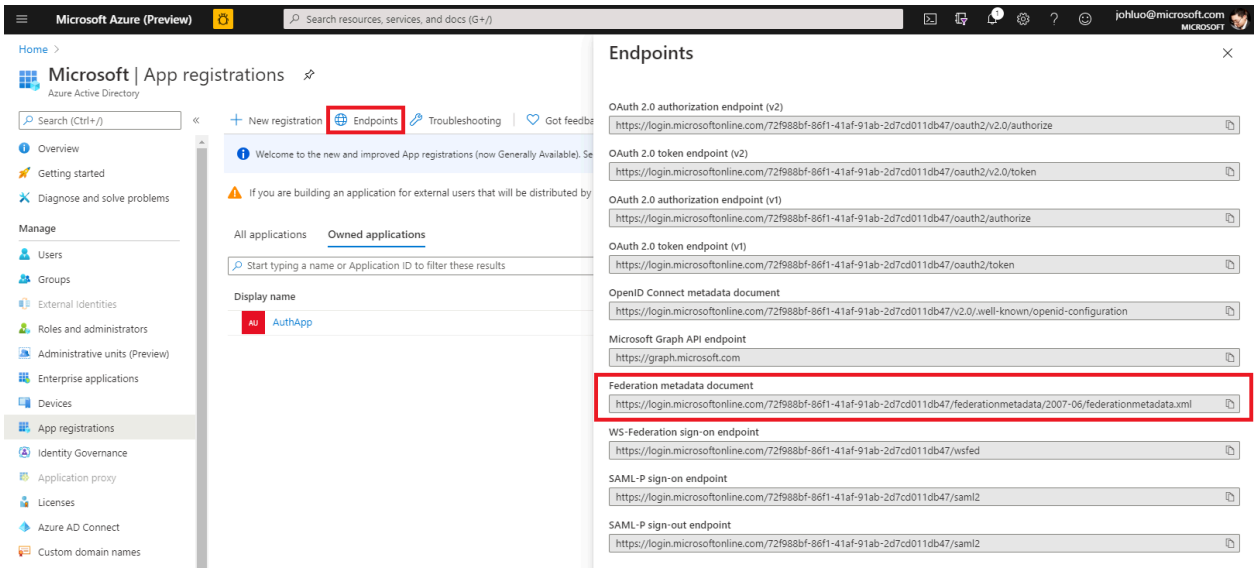
- Navigate to the Microsoft Entra ID tenant's app registrations blade. Click **New application registration**:



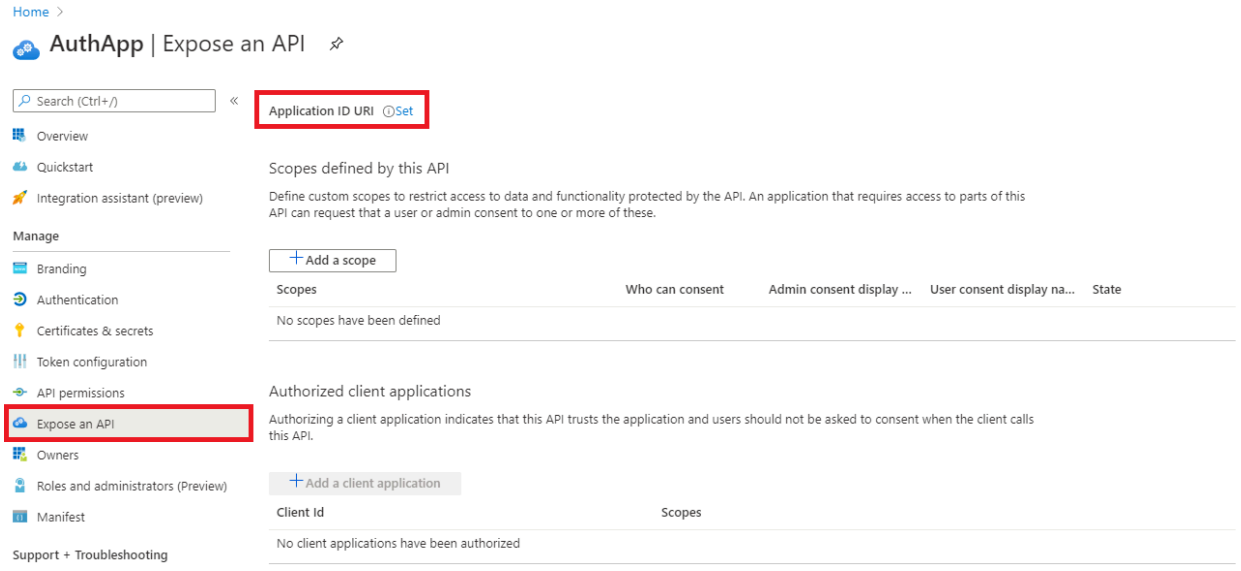
- Enter a name for the app registration. This isn't important to the ASP.NET Core app.
- Enter the URL the app listens on as the **Sign-on URL**:



- Click **Endpoints** and note the **Federation Metadata Document** URL. This is the WS-Federation middleware's `MetadataAddress`:



- Navigate to the new app registration. Click **Expose an API**. Click Application ID URI **Set** > **Save**. Make note of the **Application ID URI**. This is the WS-Federation middleware's `Wtrealm`:



Use WS-Federation without ASP.NET Core Identity

The WS-Federation middleware can be used without Identity. For example:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(sharedOptions =>
    {
        sharedOptions.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        sharedOptions.DefaultChallengeScheme =
WsFederationDefaults.AuthenticationScheme;
    })
```

```
     .AddWsFederation(options =>
     {
         options.Wtrealm = Configuration["wsfed:realm"];
         options.MetadataAddress = Configuration["wsfed:metadata"];
     })
     .AddCookie();

    services.AddControllersWithViews();
    services.AddRazorPages();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}
```

Add WS-Federation as an external login provider for ASP.NET Core Identity

- Add a dependency on Microsoft.AspNetCore.Authentication.WsFederation to the project.
- Add WS-Federation to `Startup.ConfigureServices`:

```
public void ConfigureServices(IServiceCollection services)
{
```

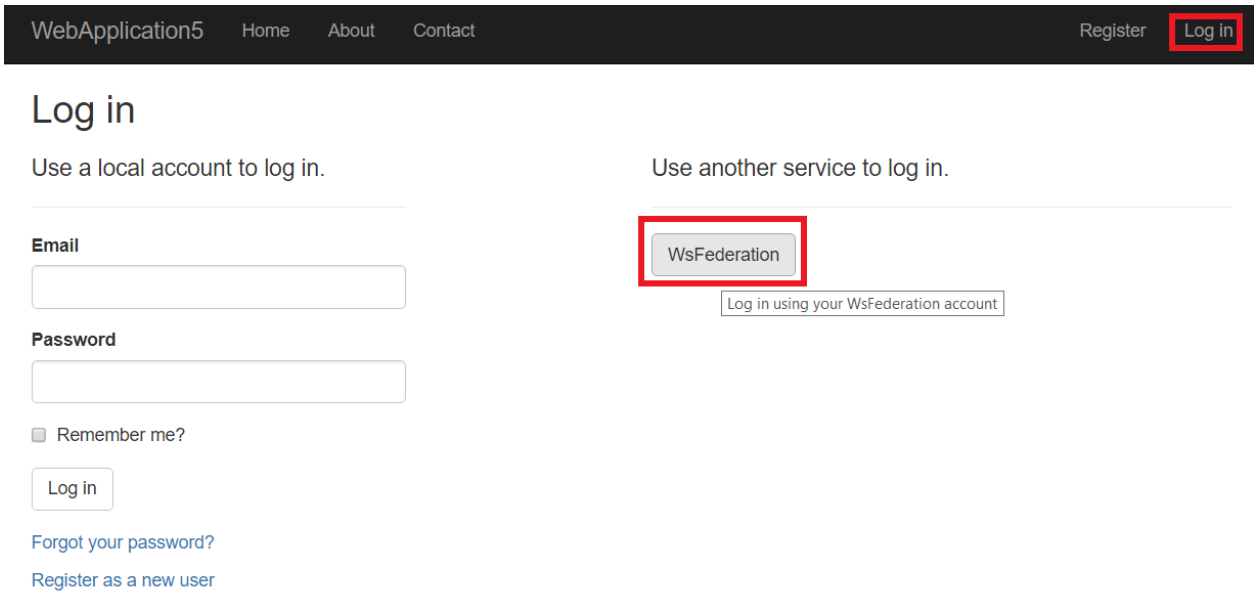
```
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();


    services.AddAuthentication()
        .AddWsFederation(options =>
        {
            // MetadataAddress represents the Active Directory instance used
to authenticate users.
            options.MetadataAddress = "https://<ADFS FQDN or AAD
tenant>/FederationMetadata/2007-06/FederationMetadata.xml";

            // Wtrealm is the app's identifier in the Active Directory
instance.
            // For ADFS, use the relying party's identifier, its WS-
Federation Passive protocol URL:
            options.Wtrealm = "https://localhost:44307/";

            // For AAD, use the Application ID URI from the app
registration's Overview blade:
            options.Wtrealm = "api://bbd35166-7c13-49f3-8041-9551f2847b69";
        });

    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

The AddAuthentication(IServiceCollection, String) overload sets the DefaultScheme property. The AddAuthentication(IServiceCollection, Action<AuthenticationOptions>) overload allows configuring authentication options, which can be used to set up default authentication schemes for different purposes. Subsequent calls to `AddAuthentication` override previously configured AuthenticationOptions properties.

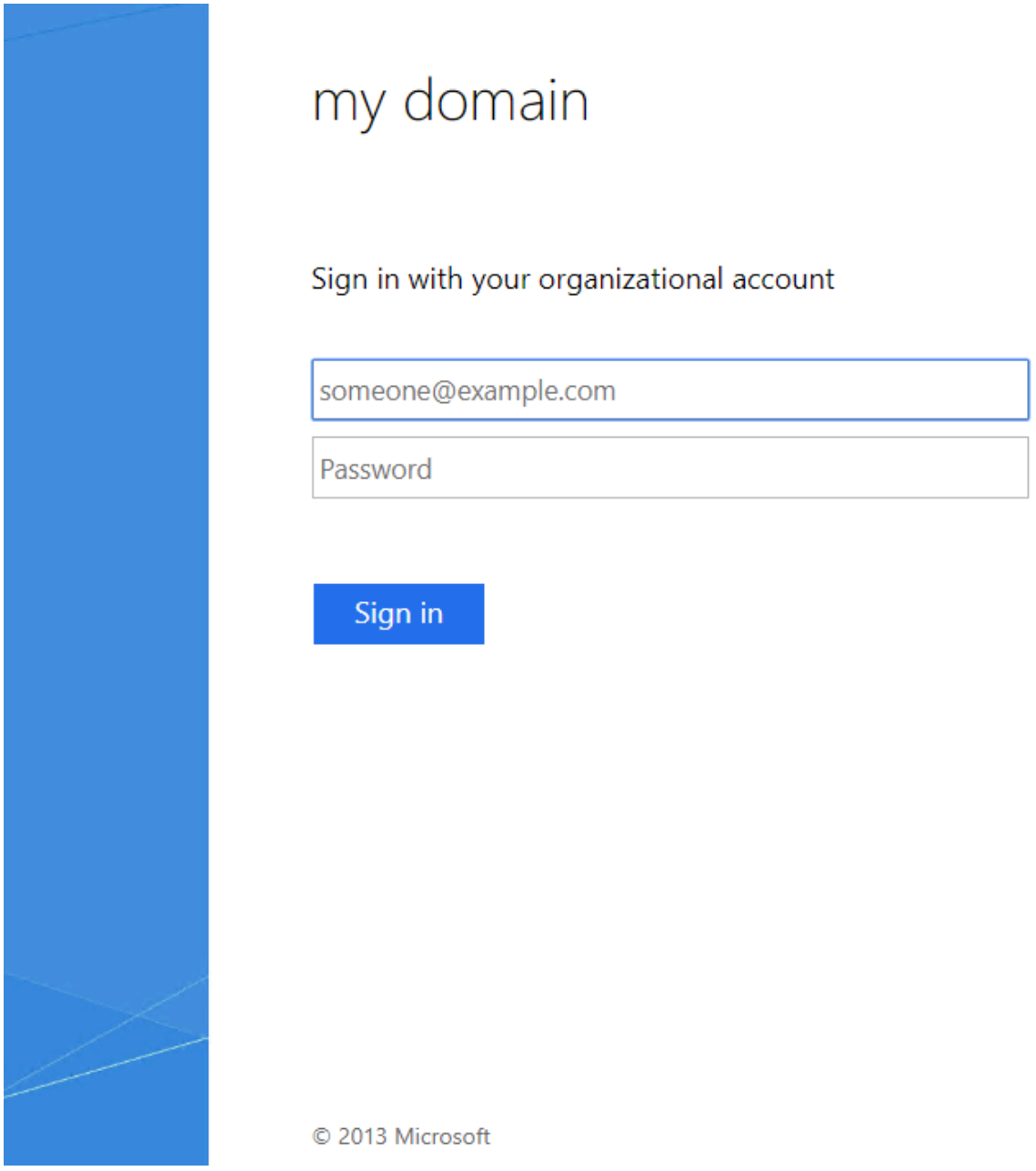
AuthenticationBuilder extension methods that register an authentication handler may only be called once per authentication scheme. Overloads exist that allow configuring the scheme properties, scheme name, and display name.

Log in with WS-Federation

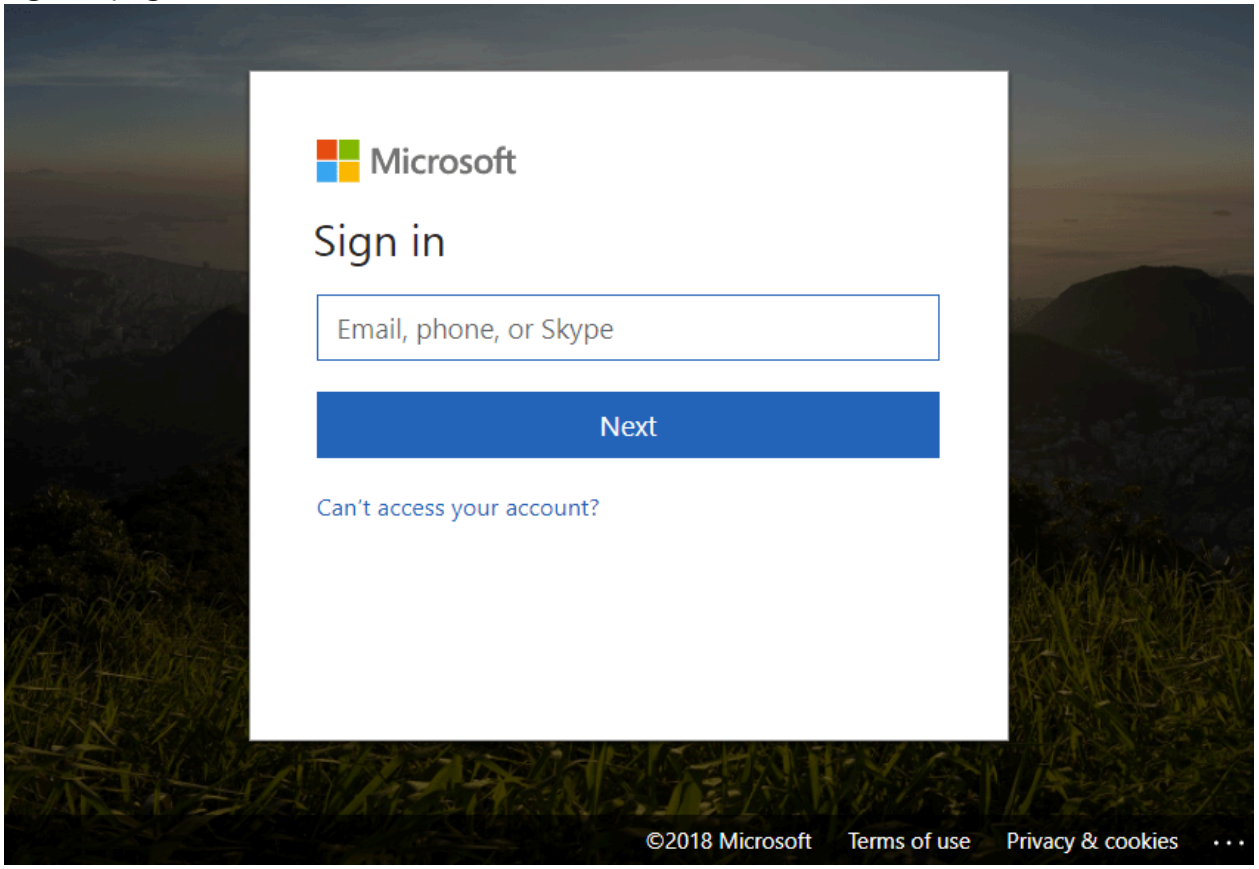
Browse to the app and click the **Log in** link in the nav header. There's an option to log in with WsFederation:



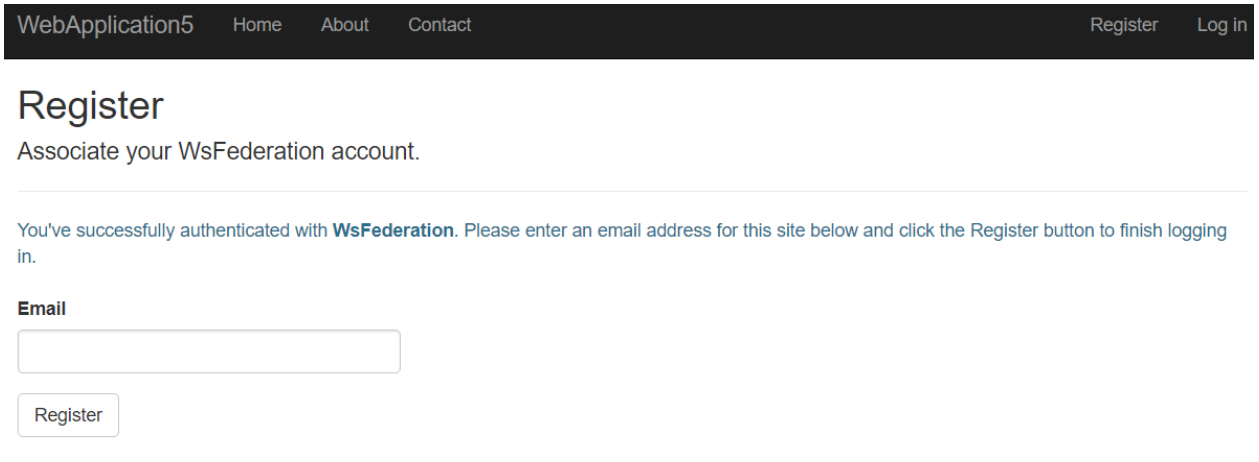
With ADFS as the provider, the button redirects to an ADFS sign-in page:



With Microsoft Entra ID as the provider, the button redirects to a Microsoft Entra ID sign-in page:



A successful sign-in for a new user redirects to the app's user registration page:



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Use social sign-in provider authentication without ASP.NET Core Identity

Article • 06/03/2022

By Kirk Larkin and Rick Anderson

Facebook and Google authentication in ASP.NET Core describes how to enable users to sign in using OAuth 2.0 with credentials from external authentication providers. The approach described in that article includes ASP.NET Core Identity as an authentication provider.

This sample demonstrates how to use an external authentication provider **without** ASP.NET Core Identity. This approach is useful for apps that don't require all of the features of ASP.NET Core Identity, but still require integration with a trusted external authentication provider.

This sample uses Google authentication for authenticating users. Using Google authentication shifts many of the complexities of managing the sign-in process to Google. To integrate with a different external authentication provider, see the following articles:

- Facebook authentication
- Microsoft authentication
- Twitter authentication
- Other providers

Configuration

In `Program.cs`, configure the app's authentication schemes with the AddAuthentication, AddCookie, and AddGoogle methods:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.Google;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddAuthentication(options =>
    {
```

```
        options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
GoogleDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddGoogle(options =>
    {
        options.ClientId =
builder.Configuration["Authentication:Google:ClientId"];
        options.ClientSecret =
builder.Configuration["Authentication:Google:ClientSecret"];
    });

builder.Services.AddRazorPages();
```

The call to AddAuthentication sets the app's DefaultScheme. The `DefaultScheme` is the default scheme used by the following `HttpContext` authentication extension methods:

- AuthenticateAsync
- ChallengeAsync
- ForbidAsync
- SignInAsync
- SignOutAsync

Setting the app's `DefaultScheme` to CookieAuthenticationDefaults.AuthenticationScheme ("Cookies") configures the app to use Cookies as the default scheme for these extension methods. Setting the app's DefaultChallengeScheme to GoogleDefaults.AuthenticationScheme ("Google") configures the app to use Google as the default scheme for calls to `ChallengeAsync`. `DefaultChallengeScheme` overrides `DefaultScheme`. See AuthenticationOptions for more properties that override `DefaultScheme` when set.

In `Program.cs`, call UseAuthentication and UseAuthorization. This middleware combination sets the HttpContext.User property and runs the Authorization Middleware for requests:

C#

```
app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();
```

To learn more about authentication schemes, see Authentication Concepts. To learn more about cookie authentication, see Use cookie authentication without ASP.NET Core Identity.

Apply authorization

Test the app's authentication configuration by applying the [Authorize] attribute to a controller, action, or page. The following code limits access to the *Privacy* page to users that have been authenticated:

C#

```
[Authorize]
public class PrivacyModel : PageModel
{

}
```

Save the access token

SaveTokens defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. `SaveTokens` is set to `false` by default to reduce the size of the final authentication cookie.

To save access and refresh tokens after a successful authorization, set `SaveTokens` to `true` in `Program.cs`:

C#

```
builder.Services
    .AddAuthentication(options =>
    {
        options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
GoogleDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddGoogle(options =>
    {
        options.ClientId =
builder.Configuration["Authentication:Google:ClientId"];
        options.ClientSecret =
builder.Configuration["Authentication:Google:ClientSecret"];
        options.SaveTokens = true;
    });
```

To retrieve a saved token, use GetTokenAsync. The following example retrieves the token named `access_token`:

C#

```
public async Task OnGetAsync()
{
    var accessToken = await HttpContext.GetTokenAsync(
        GoogleDefaults.AuthenticationScheme, "access_token");

    // ...
}
```

Sign out

To sign out the current user and delete their cookie, call SignOutAsync. The following code adds a `Logout` page handler to the *Index* page:

C#

```
public class IndexModel : PageModel
{
    public async Task<IActionResult> OnPostLogoutAsync()
    {
        // using Microsoft.AspNetCore.Authentication;
        await HttpContext.SignOutAsync();
        return RedirectToPage();
    }
}
```

Notice that the call to `SignOutAsync` doesn't specify an authentication scheme. The app uses the `DefaultScheme`, `CookieAuthenticationDefaults.AuthenticationScheme`, as a fallback.

Additional resources

- Simple authorization in ASP.NET Core
- Persist additional claims and tokens from external providers in ASP.NET Core

Policy schemes in ASP.NET Core

Article • 06/03/2022

Authentication policy schemes make it easier to have a single logical authentication scheme potentially use multiple approaches. For example, a policy scheme might use Google authentication for challenges, and cookie authentication for everything else. Authentication policy schemes make it:

- Easy to forward any authentication action to another scheme.
- Forward dynamically based on the request.

All authentication schemes that use derived AuthenticationSchemeOptions and the associated AuthenticationHandler<TOptions>:

- Are automatically policy schemes in ASP.NET Core 2.1 and later.
- Can be enabled via configuring the scheme's options.

C#

```
public class AuthenticationSchemeOptions
{
    /// <summary>
    /// If set, this specifies a default scheme that authentication handlers
should
    /// forward all authentication operations to, by default. The default
forwarding
    /// logic checks in this order:
    /// 1. The most specific
ForwardAuthenticate/Challenge/Forbid/SignIn/SignOut
    /// 2. The ForwardDefaultSelector
    /// 3. ForwardDefault
    /// The first non null result is used as the target scheme to forward
to.
    /// </summary>
    public string ForwardDefault { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// AuthenticateAsync calls to. For example:
    /// Context.AuthenticateAsync("ThisScheme") =>
    ///
Context.AuthenticateAsync("ForwardAuthenticateValue");
    /// Set the target to the current scheme to disable forwarding and allow
    /// normal processing.
    /// </summary>
    public string ForwardAuthenticate { get; set; }

    /// <summary>
```

```
    /// If set, this specifies the target scheme that this scheme should
forward
    /// ChallengeAsync calls to. For example:
    /// Context.ChallengeAsync("ThisScheme") =>
    ///
Context.ChallengeAsync("ForwardChallengeValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardChallenge { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// ForbidAsync calls to.For example:
    /// Context.ForbidAsync("ThisScheme")
    ///                                 =>
Context.ForbidAsync("ForwardForbidValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardForbid { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// SignInAsync calls to. For example:
    /// Context.SignInAsync("ThisScheme") =>
    ///
Context.SignInAsync("ForwardSignInValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardSignIn { get; set; }

    /// <summary>
    /// If set, this specifies the target scheme that this scheme should
forward
    /// SignOutAsync calls to. For example:
    /// Context.SignOutAsync("ThisScheme") =>
    ///
Context.SignOutAsync("ForwardSignOutValue");
    /// Set the target to the current scheme to disable forwarding and allow
normal
    /// processing.
    /// </summary>
    public string ForwardSignOut { get; set; }

    /// <summary>
    /// Used to select a default scheme for the current request that
authentication
    /// handlers should forward all authentication operations to by default.
```

```
The
    /// default forwarding checks in this order:
    /// 1. The most specific
ForwardAuthenticate/Challenge/Forbid/SignIn/SignOut
    /// 2. The ForwardDefaultSelector
    /// 3. ForwardDefault.
    /// The first non null result will be used as the target scheme to
forward to.
    /// </summary>
    public Func<HttpContext, string> ForwardDefaultSelector { get; set; }
}
```

Examples

The following example shows a higher level scheme that combines lower level schemes. Google authentication is used for challenges, and cookie authentication is used for everything else:

C#

```
public void ConfigureServices(IServiceCollection services)
{

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme
)
       .AddCookie(options => options.ForwardChallenge = "Google")
       .AddGoogle(options => { });
}
```

The following example enables dynamic selection of schemes on a per request basis. That is, how to mix cookies and API authentication:

C#

```
public void ConfigureServices(IServiceCollection services)
{

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme
)
        .AddCookie(options =>
        {
            // For example, can foward any requests that start with /api
            // to the api scheme.
            options.ForwardDefaultSelector = ctx =>
               ctx.Request.Path.StartsWithSegments("/api") ? "Api" : null;
        })
        .AddYourApiAuth("Api");
}
```

Manage JSON Web Tokens in development with dotnet user-jwts

Article • 04/11/2023

By Rick Anderson

The `dotnet user-jwts` command line tool can create and manage app specific local JSON Web Tokens (JWTs).

Synopsis

.NET CLI

```
dotnet user-jwts [<PROJECT>] [command]
dotnet user-jwts [command] -h|--help
```

Description

Creates and manages project specific local JSON Web Tokens.

Arguments

`PROJECT | SOLUTION`

The MSBuild project to apply a command on. If a project is not specified, MSBuild searches the current working directory for a file that has a file extension that ends in *proj* and uses that file.

Commands

Expand table

Command	Description
clear	Delete all issued JWTs for a project.
create	Issue a new JSON Web Token.
remove	Delete a given JWT.

Command	Description
key	Display or reset the signing key used to issue JWTs.
list	Lists the JWTs issued for the project.
print	Display the details of a given JWT.

Create

Usage: `dotnet user-jwts create [options]`

Expand table

Option	Description
-p | --project	The path of the project to operate on. Defaults to the project in the current directory.
--scheme	The scheme name to use for the generated token. Defaults to 'Bearer'.
-n | --name	The name of the user to create the JWT for. Defaults to the current environment user.
--audience	The audiences to create the JWT for. Defaults to the URLs configured in the project's launchSettings.json.
--issuer	The issuer of the JWT. Defaults to 'dotnet-user-jwts'.
--scope	A scope claim to add to the JWT. Specify once for each scope.
--role	A role claim to add to the JWT. Specify once for each role.
--claim	Claims to add to the JWT. Specify once for each claim in the format "name=value".
--not-before	The UTC date & time the JWT should not be valid before in the format 'yyyy-MM-dd [[HH:mm[[:ss]]]]'. Defaults to the date & time the JWT is created.
--expires-on	The UTC date & time the JWT should expire in the format 'yyyy-MM-dd [[[[HH:mm]]:ss]]'. Defaults to 6 months after the --not-before date. Do not use this option in conjunction with the --valid-for option.
--valid-for	The period the JWT should expire after. Specify using a number followed by duration type like 'd' for days, 'h' for hours, 'm' for minutes, and 's' for seconds, for example 365d'. Do not use this option in conjunction with the --expires-on option.
-o | --output	The format to use for displaying output from the command. Can be one of 'default', 'token', or 'json'.
-h | --help	Show help information

Examples

Run the following commands to create an empty web project and add the Microsoft.AspNetCore.Authentication.JwtBearer NuGet package:

.NET CLI

```
dotnet new web -o MyJWT
cd MyJWT
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Replace the contents of `Program.cs` with the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();

app.Run();
```

In the preceding code, a GET request to `/secret` returns an `401 Unauthorized` error. A production app might get the JWT from a Security token service (STS), perhaps in response to logging in via a set of credentials. For the purpose of working with the API during local development, the `dotnet user-jwts` command line tool can be used to create and manage app-specific local JWTs.

The `user-jwts` tool is similar in concept to the user-secrets tool, it can be used to manage values for the app that are only valid for the developer on the local machine. In fact, the user-jwts tool utilizes the `user-secrets` infrastructure to manage the key that the JWTs are signed with, ensuring it's stored safely in the user profile.

The `user-jwts` tool hides implementation details, such as where and how the values are stored. The tool can be used without knowing the implementation details. The values

are stored in a JSON file in the local machine's user profile folder:

Windows

File system path:

`%APPDATA%\Microsoft\UserSecrets\<secrets_GUID>\user-jwts.json`

Create a JWT

The following command creates a local JWT:

.NET CLI

```
dotnet user-jwts create
```

The preceding command creates a JWT and updates the project's `appsettings.Development.json` file with JSON similar to the following:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Authentication": {
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "http://localhost:8401",
          "https://localhost:44308",
          "http://localhost:5182",
          "https://localhost:7076"
        ],
        "ValidIssuer": "dotnet-user-jwts"
      }
    }
  }
}
```

Copy the JWT and the `ID` created in the preceding command. Use a tool like Curl to test `/secret`:

.NET CLI

```
curl -i -H "Authorization: Bearer {token}" https://localhost:{port}/secret
```

Where `{token}` is the previously generated JWT.

Display JWT security information

The following command displays the JWT security information, including expiration, scopes, roles, token header and payload, and the compact token:

.NET CLI

```
dotnet user-jwts print {ID} --show-all
```

Create a token for a specific user and scope

See Create in this topic for supported create options.

The following command creates a JWT for a user named `MyTestUser`:

.NET CLI

```
dotnet user-jwts create --name MyTestUser --scope "myapi:secrets"
```

The preceding command has output similar to the following:

.NET CLI

```
New JWT saved with ID '43e0b748'.
Name: MyTestUser
Scopes: myapi:secrets

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.{Remaining token deleted}
```

The preceding token can be used to test the `/secret2` endpoint in the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
```

```
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();
app.MapGet("/secret2", () => "This is a different secret!")
    .RequireAuthorization(p => p.RequireClaim("scope", "myapi:secrets"));

app.Run();
```

Mapping, customizing, and transforming claims in ASP.NET Core

Article • 08/16/2024

By Damien Bowden

Claims can be created from any user or identity data which can be issued using a trusted identity provider or ASP.NET Core identity. A claim is a name value pair that represents what the subject is, not what the subject can do. This article covers the following areas:

- How to configure and map claims using an OpenID Connect client
- Set the name and role claim
- Reset the claims namespaces
- Customize, extend the claims using TransformAsync

Mapping claims using OpenID Connect authentication

The profile claims can be returned in the `id_token`, which is returned after a successful authentication. The ASP.NET Core client app only requires the profile scope. When using the `id_token` for claims, no extra claims mapping is required.

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
   .AddCookie()
   .AddOpenIdConnect(options =>
   {
       options.SignInScheme = "Cookies";
       options.Authority = "-your-identity-provider-";
       options.RequireHttpsMetadata = true;
       options.ClientId = "-your-clientid-";
       options.ClientSecret = "-your-client-secret-from-user-secrets-or-
keyvault";
       options.ResponseType = "code";
       options.UsePkce = true;
       options.Scope.Add("profile");
       options.SaveTokens = true;
   });
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding code requires the Microsoft.AspNetCore.Authentication.OpenIdConnect NuGet package.

Another way to get the user claims is to use the OpenID Connect User Info API. The ASP.NET Core client app uses the `GetClaimsFromUserInfoEndpoint` property to configure this. One important difference from the first settings, is that you must specify the claims you require using the `MapUniqueJsonKey` method, otherwise only the `name`, `given_name` and `email` standard claims will be available in the client app. The claims included in the `id_token` are mapped per default. This is the major difference to the first option. You must explicitly define some of the claims you require.

```
```

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
   .AddCookie()
   .AddOpenIdConnect(options =>
   {
       options.SignInScheme = "Cookies";
       options.Authority = "-your-identity-provider-";
       options.RequireHttpsMetadata = true;
       options.ClientId = "-your-clientid-";
       options.ClientSecret = "-client-secret-from-user-secrets-or-
keyvault";
       options.ResponseType = "code";
       options.UsePkce = true;
       options.Scope.Add("profile");
       options.SaveTokens = true;
       options.GetClaimsFromUserInfoEndpoint = true;
       options.ClaimActions.MapUniqueJsonKey("preferred_username",
                                             "preferred_username");
       options.ClaimActions.MapUniqueJsonKey("gender", "gender");
   });

var app = builder.Build();

// Code removed for brevity.
```

Name claim and role claim mapping

The **Name** claim and the **Role** claim are mapped to default properties in the ASP.NET Core HTTP context. Sometimes it is required to use different claims for the default properties, or the name claim and the role claim do not match the default values. The claims can be mapped using the **TokenValidationParameters** property and set to any claim as required. The values from the claims can be used directly in the HttpContext **User.Identity.Name** property and the roles.

If the `User.Identity.Name` has no value or the roles are missing, please check the values in the returned claims and set the `NameClaimType` and the `RoleClaimType` values. The returned claims from the client authentication can be viewed in the HTTP context.

C#

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
  .AddCookie()
  .AddOpenIdConnect(options =>
  {
       // Other options...
       options.TokenValidationParameters = new TokenValidationParameters
       {
          NameClaimType = "email"
          //, RoleClaimType = "role"
       };
  });
```

Claims namespaces, default namespaces

ASP.NET Core adds default namespaces to some known claims, which might not be required in the app. Optionally, disable these added namespaces and use the exact claims that the OpenID Connect server created.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

JsonWebTokenHandler.DefaultInboundClaimTypeMap.Clear();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
   .AddCookie()
   .AddOpenIdConnect(options =>
   {
       options.SignInScheme = "Cookies";
```

```
        options.Authority = "-your-identity-provider-";
        options.RequireHttpsMetadata = true;
        options.ClientId = "-your-clientid-";
        options.ClientSecret = "-your-client-secret-from-user-secrets-or-
keyvault";
        options.ResponseType = "code";
        options.UsePkce = true;
        options.Scope.Add("profile");
        options.SaveTokens = true;
    });

var app = builder.Build();

// Code removed for brevity.
```

If you need to disable the namespaces per scheme and not globally, you can use the **MapInboundClaims = false** option.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
    .AddCookie()
    .AddOpenIdConnect(options =>
    {
        options.SignInScheme = "Cookies";
        options.Authority = "-your-identity-provider-";
        options.RequireHttpsMetadata = true;
        options.ClientId = "-your-clientid-";
        options.ClientSecret = "-your-client-secret-from-user-secrets-or-
keyvault";
        options.ResponseType = "code";
        options.UsePkce = true;
        options.MapInboundClaims = false;
        options.Scope.Add("profile");
        options.SaveTokens = true;
    });

var app = builder.Build();

// Code removed for brevity.
```

Extend or add custom claims using `IClaimsTransformation`

The IClaimsTransformation interface can be used to add extra claims to the ClaimsPrincipal class. The interface requires a single method TransformAsync. This method might get called multiple times. Only add a new claim if it does not already exist in the `ClaimsPrincipal`. A `ClaimsIdentity` is created to add the new claims and this can be added to the `ClaimsPrincipal`.

C#

```
using Microsoft.AspNetCore.Authentication;
using System.Security.Claims;

public class MyClaimsTransformation : IClaimsTransformation
{
    public Task<ClaimsPrincipal> TransformAsync(ClaimsPrincipal principal)
    {
        ClaimsIdentity claimsIdentity = new ClaimsIdentity();
        var claimType = "myNewClaim";
        if (!principal.HasClaim(claim => claim.Type == claimType))
        {
            claimsIdentity.AddClaim(new Claim(claimType, "myClaimValue"));
        }

        principal.AddIdentity(claimsIdentity);
        return Task.FromResult(principal);
    }
}
```

The IClaimsTransformation interface and the `MyClaimsTransformation` class can be registered as a service:

C#

```
builder.Services.AddTransient<IClaimsTransformation, MyClaimsTransformation>
();
```

Map claims from external identity providers

Refer to the following document:

Persist additional claims and tokens from external providers in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Community OSS authentication options for ASP.NET Core

Article • 05/06/2024

This page shows community-provided open-source software (OSS) authentication options for ASP.NET Core. This page is periodically updated as new providers become available.

OSS authentication providers [.NET]

Expand table

Name	Description
Duende IdentityServer	IdentityServer is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core.
OpenIddict	OAuth 2.0/OpenID Connect server for ASP.NET Core and ASP.NET 4.x.
FIDO2 .NET Library, WebAuthn	FIDO2 .NET library for FIDO2 / WebAuthn Attestation and Assertion using .NET

OSS authentication provider clients [.NET]

Expand table

Name	Description
OpenIddict	OAuth 2.0/OpenID Connect client for ASP.NET Core, ASP.NET 4.x and Windows/Linux desktop apps with built-in integrations for 80+ services such as Auth0, Microsoft Entra ID, GitHub, Google, Twitter or Yahoo.
AspNet.Security.OAuth.Providers	A collection of security middleware for ASP.NET Core apps to support social authentication.
AspNet.Security.OpenId.Providers	A collection of security middleware for ASP.NET Core apps to support OpenID 2.0 authentication providers like Steam.

Other OSS authentication providers

Expand table

Name	Description
Gluu Server	Enterprise ready, open source software for identity, access management (IAM), and single sign-on (SSO). For more information, see the Gluu Product Documentation.
Keycloak	Open Source Identity and Access Management For Modern Applications and Services.
node-oidc-provider	OpenID Certified™ OAuth 2.0 Authorization Server implementation for Node.js.
Authentik	Authentik is an open-source Identity Provider focused on flexibility and versatility.

To add a provider, edit this page.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Identity management solutions for .NET web apps

Article • 10/08/2024

The following table provides an overview of various identity management solutions that can be used in ASP.NET Core apps. These solutions offer features and capabilities to manage user authentication, authorization, and user identity within an app. It includes options for apps that are:

- Container-based
- Self-hosted, where you manage the installation and infrastructure to support it.
- Managed, such as cloud-based services like Microsoft Entra

The following table lists both open source and commercial solutions in alphabetical order. Each line contains details such as license type, website, and documentation that is specific to ASP.NET Core integration. The table can help identify the identity management solutions that best align with your app's needs.

Many of the commercial licenses provide "community" or free options that may be available depending on your company size and app requirements.

Expand table

Name	Type	License Type	Documentation
ASP.NET Core Identity	Self host	OSS (MIT)	Secure a web app with ASP.NET Core Identity
Auth0	Managed	Commercial	Get started
Duende IdentityServer	Self host	Commercial	ASP.NET Identity integration
Keycloak	Container	OSS (Apache 2.0)	Keycloak securing apps documentation
Microsoft Entra ID	Managed	Commercial	Entra documentation
Okta	Managed	Commercial	Okta for ASP.NET Core
OpenIddict	Self host	OSS (Apache 2.0)	OpenIddict documentation

Is there a solution that should be added to this list? Do you have a correction, suggestion, or feedback? We welcome your contributions. Learn how to contribute.

Multi-factor authentication in ASP.NET Core

Article • 10/30/2024

By Damien Bowden

View or download sample code (damienbod/AspNetCoreHybridFlowWithApi GitHub repository)

Multi-factor authentication (MFA) is a process in which a user is requested during a sign-in event for additional forms of identification. This prompt could be to enter a code from a cellphone, use a FIDO2 key, or to provide a fingerprint scan. When you require a second form of authentication, security is enhanced. The additional factor isn't easily obtained or duplicated by a cyberattacker.

This article covers the following areas:

- What is MFA and what MFA flows are recommended
- Configure MFA for administration pages using ASP.NET Core Identity
- Send MFA sign-in requirement to OpenID Connect server
- Force ASP.NET Core OpenID Connect client to require MFA

MFA, 2FA

MFA requires at least two or more types of proof for an identity like something you know, something you possess, or biometric validation for the user to authenticate.

Two-factor authentication (2FA) is like a subset of MFA, but the difference being that MFA can require two or more factors to prove the identity.

2FA is supported by default when using ASP.NET Core Identity. To enable or disable 2FA for a specific user, set the IdentityUser<TKey>.TwoFactorEnabled property. The ASP.NET Core Identity Default UI includes pages for configuring 2FA.

MFA TOTP (Time-based One-time Password Algorithm)

MFA using TOTP is supported by default when using ASP.NET Core Identity. This approach can be used together with any compliant authenticator app, including:

- Microsoft Authenticator
- Google Authenticator

For implementation details, see Enable QR Code generation for TOTP authenticator apps in ASP.NET Core.

To disable support for MFA TOTP, configure authentication using AddIdentity instead of AddDefaultIdentity. `AddDefaultIdentity` calls AddDefaultTokenProviders internally, which registers multiple token providers including one for MFA TOTP. To register only specific token providers, call AddTokenProvider for each required provider. For more information about available token providers, see the AddDefaultTokenProviders source on GitHub.

MFA passkeys/FIDO2 or passwordless

passkeys/FIDO2 is currently:

- The most secure way of achieving MFA.
- MFA that protects against phishing attacks. (As well as certificate authentication and Windows for business)

At present, ASP.NET Core doesn't support passkeys/FIDO2 directly. Passkeys/FIDO2 can be used for MFA or passwordless flows.

Microsoft Entra ID provides support for passkeys/FIDO2 and passwordless flows. For more information, see Passwordless authentication options.

Other forms of passwordless MFA do not or may not protect against phishing.

MFA SMS

MFA with SMS increases security massively compared with password authentication (single factor). However, using SMS as a second factor is no longer recommended. Too many known attack vectors exist for this type of implementation.

NIST guidelines

Configure MFA for administration pages using ASP.NET Core Identity

MFA could be forced on users to access sensitive pages within an ASP.NET Core Identity app. This could be useful for apps where different levels of access exist for the different identities. For example, users might be able to view the profile data using a password login, but an administrator would be required to use MFA to access the administrative pages.

Extend the login with an MFA claim

The demo code is setup using ASP.NET Core with Identity and Razor Pages. The `AddIdentity` method is used instead of `AddDefaultIdentity` one, so an `IUserClaimsPrincipalFactory` implementation can be used to add claims to the identity after a successful login.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

C#

```
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlite(
        Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddIdentity<IdentityUser, IdentityRole>(options =>
        options.SignIn.RequireConfirmedAccount = false)
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

builder.Services.AddSingleton<IEmailSender, EmailSender>();
builder.Services.AddScoped<IUserClaimsPrincipalFactory<IdentityUser>,
    AdditionalUserClaimsPrincipalFactory>();

builder.Services.AddAuthorization(options =>
    options.AddPolicy("TwoFactorEnabled", x => x.RequireClaim("amr",
"mfa")));

builder.Services.AddRazorPages();
```

The `AdditionalUserClaimsPrincipalFactory` class adds the `amr` claim to the user claims only after a successful login. The claim's value is read from the database. The claim is added here because the user should only access the higher protected view if the identity has logged in with MFA. If the database view is read from the database directly instead of using the claim, it's possible to access the view without MFA directly after activating the MFA.

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using System.Collections.Generic;
using System.Security.Claims;
using System.Threading.Tasks;

namespace IdentityStandaloneMfa
{
    public class AdditionalUserClaimsPrincipalFactory :
        UserClaimsPrincipalFactory<IdentityUser, IdentityRole>
    {
        public AdditionalUserClaimsPrincipalFactory(
            UserManager<IdentityUser> userManager,
            RoleManager<IdentityRole> roleManager,
            IOptions<IdentityOptions> optionsAccessor)
            : base(userManager, roleManager, optionsAccessor)
        {
        }

        public async override Task<ClaimsPrincipal> CreateAsync(IdentityUser
user)
        {
            var principal = await base.CreateAsync(user);
            var identity = (ClaimsIdentity)principal.Identity;

            var claims = new List<Claim>();

            if (user.TwoFactorEnabled)
            {
                claims.Add(new Claim("amr", "mfa"));
            }
            else
            {
                claims.Add(new Claim("amr", "pwd"));
            }

            identity.AddClaims(claims);
            return principal;
        }
    }
}
```

Because the Identity service setup changed in the `Startup` class, the layouts of the Identity need to be updated. Scaffold the Identity pages into the app. Define the layout in the `Identity/Account/Manage/_Layout.cshtml` file.

CSHTML

```
@{
    Layout = "/Pages/Shared/_Layout.cshtml";
```

```
}
```

Also assign the layout for all the manage pages from the Identity pages:

CSHTML

```
@{
    Layout = "_Layout.cshtml";
}
```

Validate the MFA requirement in the administration page

The administration Razor Page validates that the user has logged in using MFA. In the `OnGet` method, the identity is used to access the user claims. The `amr` claim is checked for the value `mfa`. If the identity is missing this claim or is `false`, the page redirects to the Enable MFA page. This is possible because the user has logged in already, but without MFA.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityStandaloneMfa
{
    public class AdminModel : PageModel
    {
        public IActionResult OnGet()
        {
            var claimTwoFactorEnabled =
                User.Claims.FirstOrDefault(t => t.Type == "amr");

            if (claimTwoFactorEnabled != null &&
                "mfa".Equals(claimTwoFactorEnabled.Value))
            {
                // You logged in with MFA, do the administrative stuff
            }
            else
            {
                return Redirect(
                    "/Identity/Account/Manage/TwoFactorAuthentication");
            }

            return Page();
        }
```

```
    }
}
```

UI logic to toggle user login information

An authorization policy was added at startup. The policy requires the `amr` claim with the value `mfa`.

C#

```
services.AddAuthorization(options =>
    options.AddPolicy("TwoFactorEnabled",
        x => x.RequireClaim("amr", "mfa")));
```

This policy can then be used in the `_Layout` view to show or hide the **Admin** menu with the warning:

CSHTML

```
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@inject UserManager<IdentityUser> UserManager
@inject IAuthorizationService AuthorizationService
```

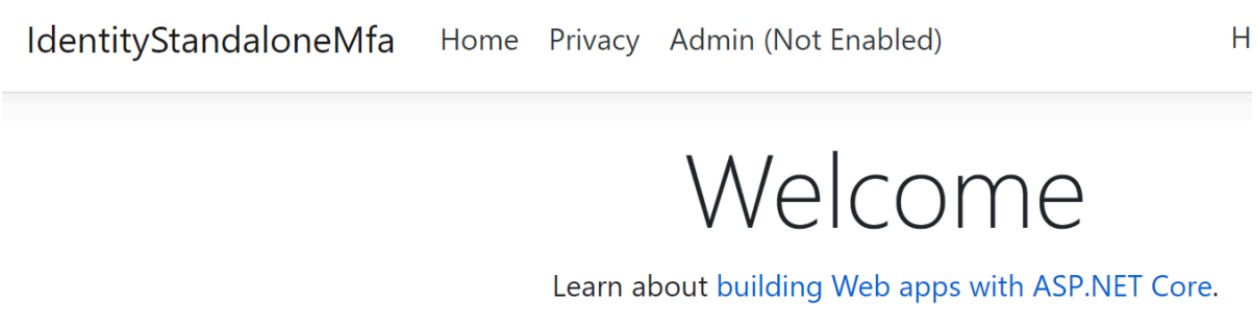
If the identity has logged in using MFA, the **Admin** menu is displayed without the tooltip warning. When the user has logged in without MFA, the **Admin (Not Enabled)** menu is displayed along with the tooltip that informs the user (explaining the warning).

CSHTML

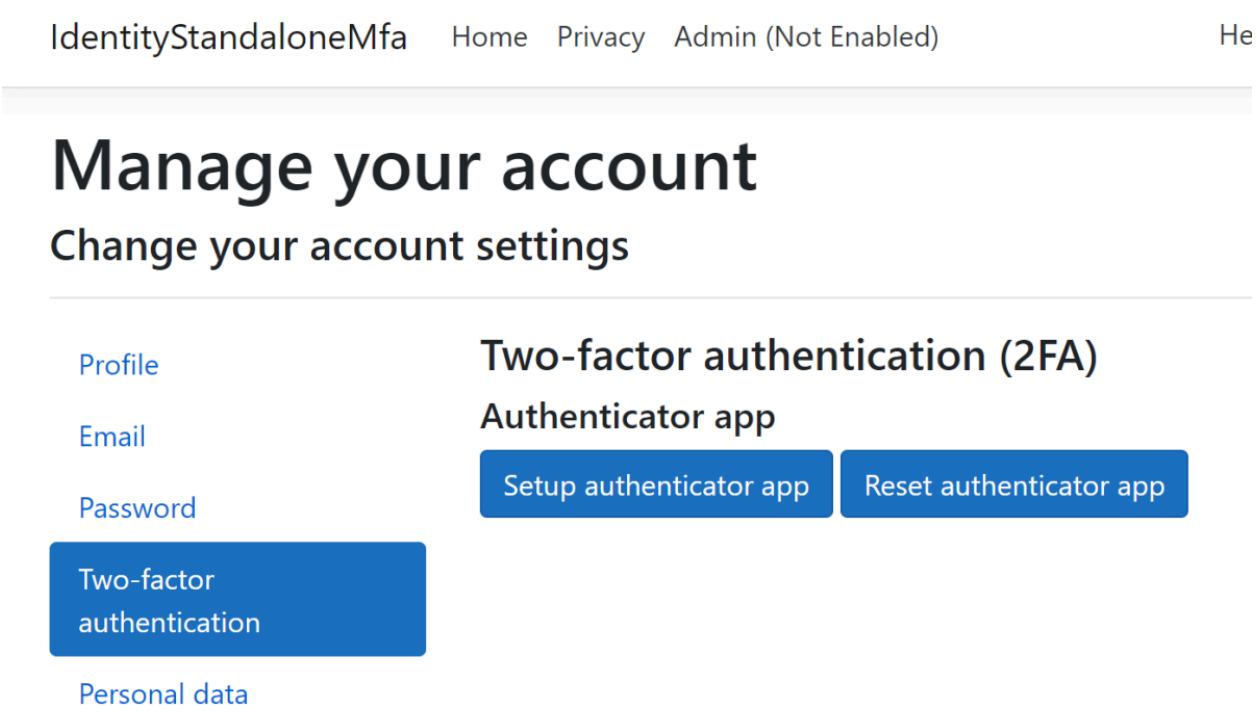
```
@if (SignInManager.IsSignedIn(User))
{
    @if ((AuthorizationService.AuthorizeAsync(User,
"TwoFactorEnabled")).Result.Succeeded)
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-
page="/Admin">Admin</a>
        </li>
    }
    else
    {
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-page="/Admin"
               id="tooltip-demo"
               data-toggle="tooltip"
```

```
                data-placement="bottom"
                title="MFA is NOT enabled. This is required for the Admin
Page. If you have activated MFA, then logout, login again.">
                 Admin (Not Enabled)
            </a>
        </li>
    }
}
```

If the user logs in without MFA, the warning is displayed:



The user is redirected to the MFA enable view when clicking the **Admin** link:



Send MFA sign-in requirement to OpenID Connect server

The `acr_values` parameter can be used to pass the `mfa` required value from the client to the server in an authentication request.

> **Note**
>
> The `acr_values` parameter needs to be handled on the OpenID Connect server for this to work.

OpenID Connect ASP.NET Core client

The ASP.NET Core Razor Pages OpenID Connect client app uses the `AddOpenIdConnect` method to login to the OpenID Connect server. The `acr_values` parameter is set with the `mfa` value and sent with the authentication request. The `OpenIdConnectEvents` is used to add this.

For recommended `acr_values` parameter values, see Authentication Method Reference Values.

C#

```
build.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.SignInScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.Authority = "<OpenID Connect server URL>";
    options.RequireHttpsMetadata = true;
    options.ClientId = "<OpenID Connect client ID>";
    options.ClientSecret = "<>";
    options.ResponseType = "code";
    options.UsePkce = true;
    options.Scope.Add("profile");
    options.Scope.Add("offline_access");
    options.SaveTokens = true;
    options.Events = new OpenIdConnectEvents
    {
```

```
        OnRedirectToIdentityProvider = context =>
        {
            context.ProtocolMessage.SetParameter("acr_values", "mfa");
            return Task.FromResult(0);
        }
    };
});
```

Example OpenID Connect Duende IdentityServer server with ASP.NET Core Identity

On the OpenID Connect server, which is implemented using ASP.NET Core Identity with Razor Pages, a new page named `ErrorEnable2FA.cshtml` is created. The view:

- Displays if the Identity comes from an app that requires MFA but the user hasn't activated this in Identity.
- Informs the user and adds a link to activate this.

CSHTML

```
@{
    ViewData["Title"] = "ErrorEnable2FA";
}

<h1>The client application requires you to have MFA enabled. Enable this,
try login again.</h1>

<br />

You can enable MFA to login here:

<br />

<a href="~/Identity/Account/Manage/TwoFactorAuthentication">Enable MFA</a>
```

In the `Login` method, the `IIdentityServerInteractionService` interface implementation `_interaction` is used to access the OpenID Connect request parameters. The `acr_values` parameter is accessed using the `AcrValues` property. As the client sent this with `mfa` set, this can then be checked.

If MFA is required, and the user in ASP.NET Core Identity has MFA enabled, then the login continues. When the user has no MFA enabled, the user is redirected to the custom view `ErrorEnable2FA.cshtml`. Then ASP.NET Core Identity signs the user in.

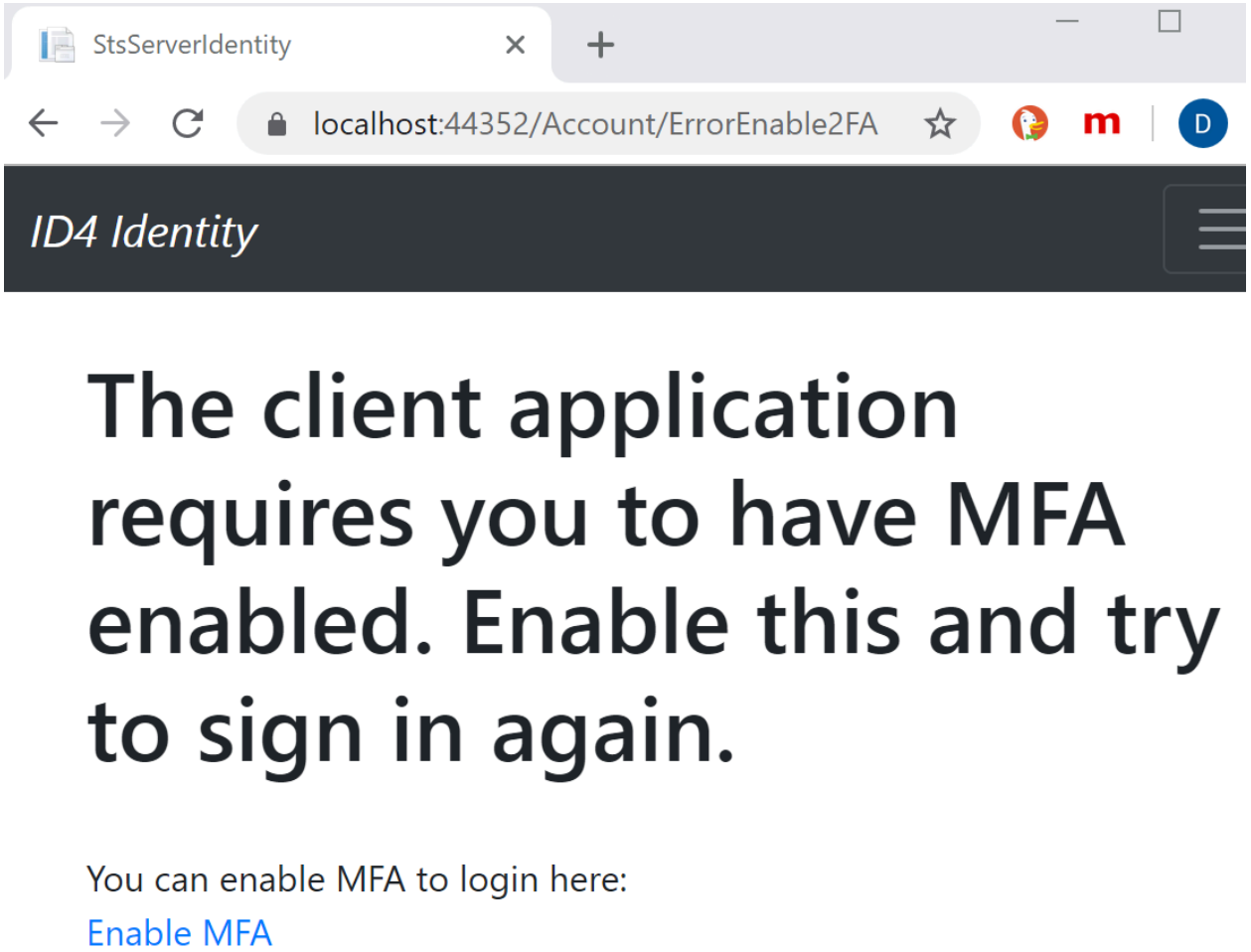
The Fido2Store is used to check if the user has activated MFA using a custom FIDO2 Token Provider.

C#

```
public async Task<IActionResult> OnPost()
{
    // check if we are in the context of an authorization request
    var context = await
_interaction.GetAuthorizationContextAsync(Input.ReturnUrl);

    var requires2Fa = context?.AcrValues.Count(t => t.Contains("mfa")) >= 1;

    var user = await _userManager.FindByNameAsync(Input.Username);
    if (user != null && !user.TwoFactorEnabled && requires2Fa)
    {
        return RedirectToPage("/Home/ErrorEnable2FA/Index");
    }

    // code omitted for brevity

    if (ModelState.IsValid)
    {
        var result = await
_signInManager.PasswordSignInAsync(Input.Username, Input.Password,
Input.RememberLogin, lockoutOnFailure: true);
        if (result.Succeeded)
        {
            // code omitted for brevity
        }
        if (result.RequiresTwoFactor)
        {
            var fido2ItemExistsForUser = await
_fido2Store.GetCredentialsByUserNameAsync(user.UserName);
            if (fido2ItemExistsForUser.Count > 0)
            {
                return RedirectToPage("/Account/LoginFido2Mfa", new { area =
"Identity", Input.ReturnUrl, Input.RememberLogin });
            }

            return RedirectToPage("/Account/LoginWith2fa", new { area =
"Identity", Input.ReturnUrl, RememberMe = Input.RememberLogin });
        }

        await _events.RaiseAsync(new UserLoginFailureEvent(Input.Username,
"invalid credentials", clientId: context?.Client.ClientId));
        ModelState.AddModelError(string.Empty,
LoginOptions.InvalidCredentialsErrorMessage);
    }

    // something went wrong, show form with error
    await BuildModelAsync(Input.ReturnUrl);
    return Page();
}
```

If the user is already logged in, the client app:

- Still validates the `amr` claim.
- Can set up the MFA with a link to the ASP.NET Core Identity view.



Force ASP.NET Core OpenID Connect client to require MFA

This example shows how an ASP.NET Core Razor Page app, which uses OpenID Connect to sign in, can require that users have authenticated using MFA.

To validate the MFA requirement, an `IAuthorizationRequirement` requirement is created. This will be added to the pages using a policy that requires MFA.

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AspNetCoreRequireMfaOidc;

public class RequireMfa : IAuthorizationRequirement{}
```

An `AuthorizationHandler` is implemented that will use the `amr` claim and check for the value `mfa`. The `amr` is returned in the `id_token` of a successful authentication and can have many different values as defined in the Authentication Method Reference Values specification.

The returned value depends on how the identity authenticated and on the OpenID Connect server implementation.

The `AuthorizationHandler` uses the `RequireMfa` requirement and validates the `amr` claim. The OpenID Connect server can be implemented using Duende Identity Server with ASP.NET Core Identity. When a user logs in using TOTP, the `amr` claim is returned with an MFA value. If using a different OpenID Connect server implementation or a different MFA type, the `amr` claim will, or can, have a different value. The code must be extended to accept this as well.

C#

```
public class RequireMfaHandler : AuthorizationHandler<RequireMfa>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        RequireMfa requirement)
    {
        if (context == null)
            throw new ArgumentNullException(nameof(context));
        if (requirement == null)
            throw new ArgumentNullException(nameof(requirement));

        var amrClaim =
            context.User.Claims.FirstOrDefault(t => t.Type == "amr");

        if (amrClaim != null && amrClaim.Value == Amr.Mfa)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

In the program file, the `AddOpenIdConnect` method is used as the default challenge scheme. The authorization handler, which is used to check the `amr` claim, is added to the Inversion of Control container. A policy is then created which adds the `RequireMfa` requirement.

C#

```
builder.Services.ConfigureApplicationCookie(options =>
        options.Cookie.SecurePolicy =
            CookieSecurePolicy.Always);

builder.Services.AddSingleton<IAuthorizationHandler, RequireMfaHandler>();

builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
        OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.SignInScheme =
        CookieAuthenticationDefaults.AuthenticationScheme;
    options.Authority = "https://localhost:44352";
    options.RequireHttpsMetadata = true;
    options.ClientId = "AspNetCoreRequireMfaOidc";
    options.ClientSecret = "AspNetCoreRequireMfaOidcSecret";
    options.ResponseType = "code";
    options.UsePkce = true;
    options.Scope.Add("profile");
    options.Scope.Add("offline_access");
    options.SaveTokens = true;
});

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireMfa", policyIsAdminRequirement =>
    {
        policyIsAdminRequirement.Requirements.Add(new RequireMfa());
    });
});

builder.Services.AddRazorPages();
```

This policy is then used in the Razor page as required. The policy could be added globally for the entire app as well.

C#

```
[Authorize(Policy= "RequireMfa")]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }
}
```

If the user authenticates without MFA, the `amr` claim will probably have a `pwd` value. The request won't be authorized to access the page. Using the default values, the user will be redirected to the *Account/AccessDenied* page. This behavior can be changed or you can implement your own custom logic here. In this example, a link is added so that the valid user can set up MFA for their account.

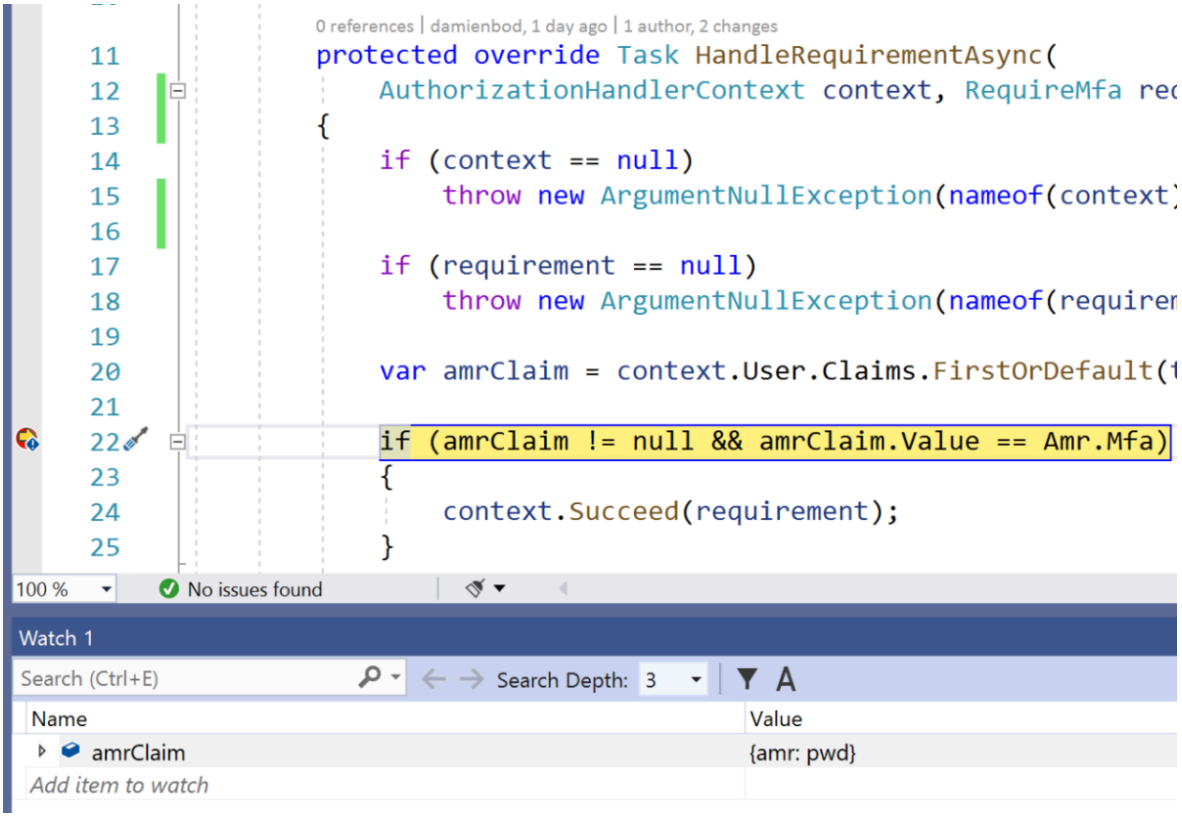
CSHTML

```
@page
@model AspNetCoreRequireMfaOidc.AccessDeniedModel
@{
    ViewData["Title"] = "AccessDenied";
    Layout = "~/Pages/Shared/_Layout.cshtml";
}

<h1>AccessDenied</h1>

You require MFA to login here

<a href="https://localhost:44352/Manage/TwoFactorAuthentication">Enable
MFA</a>
```

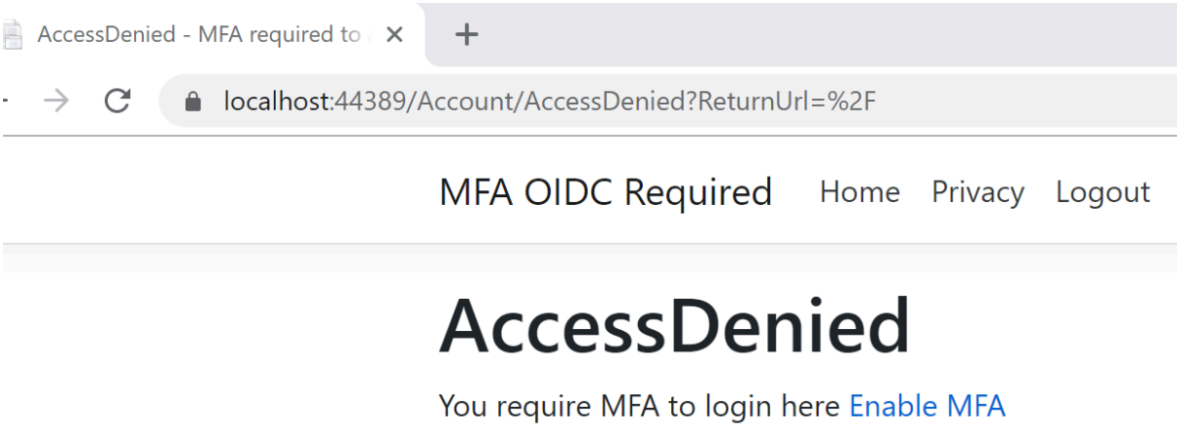
Now only users that authenticate with MFA can access the page or website. If different MFA types are used or if 2FA is okay, the `amr` claim will have different values and needs to be processed correctly. Different OpenID Connect servers also return different values for this claim and might not follow the Authentication Method Reference Values specification.

When logging in without MFA (for example, using just a password):

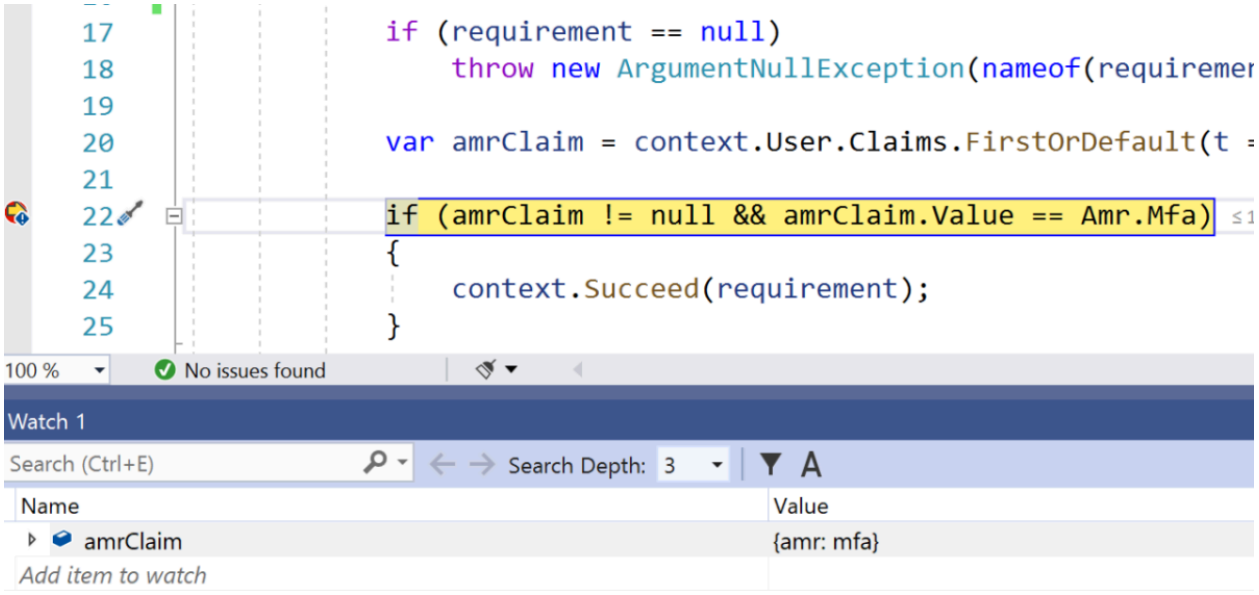
- The `amr` has the `pwd` value:



- Access is denied:



Alternatively, logging in using OTP with Identity:



Additional resources

- Enable QR Code generation for TOTP authenticator apps in ASP.NET Core
- Passwordless authentication options for Azure Active Directory
- FIDO2 .NET library for FIDO2 / WebAuthn Attestation and Assertion using .NET
- WebAuthn Awesome

Introduction to authorization in ASP.NET Core

Article • 12/02/2024

Authorization refers to the process that determines what a user is able to do. For example, an administrative user is allowed to create a document library, add documents, edit documents, and delete them. A non-administrative user working with the library is only authorized to read the documents.

Authorization is separate and distinct from authentication. However, authorization relies on an authentication mechanism. Authentication is the process of verifying a user's identity, which may result in the creation of one or more identity objects for the user.

For more information about authentication in ASP.NET Core, see Overview of ASP.NET Core Authentication.

Authorization types

ASP.NET Core authorization provides a simple, declarative role and a rich policy-based model. Authorization is expressed in requirements, and handlers evaluate a user's claims against requirements. Imperative checks can be based on simple policies or policies which evaluate both the user identity and properties of the resource that the user is attempting to access.

Namespaces

Authorization components, including the `AuthorizeAttribute` and `AllowAnonymousAttribute` attributes, are found in the `Microsoft.AspNetCore.Authorization` namespace.

Consult the documentation on simple authorization.

Create an ASP.NET Core web app with user data protected by authorization

Article • 10/04/2024

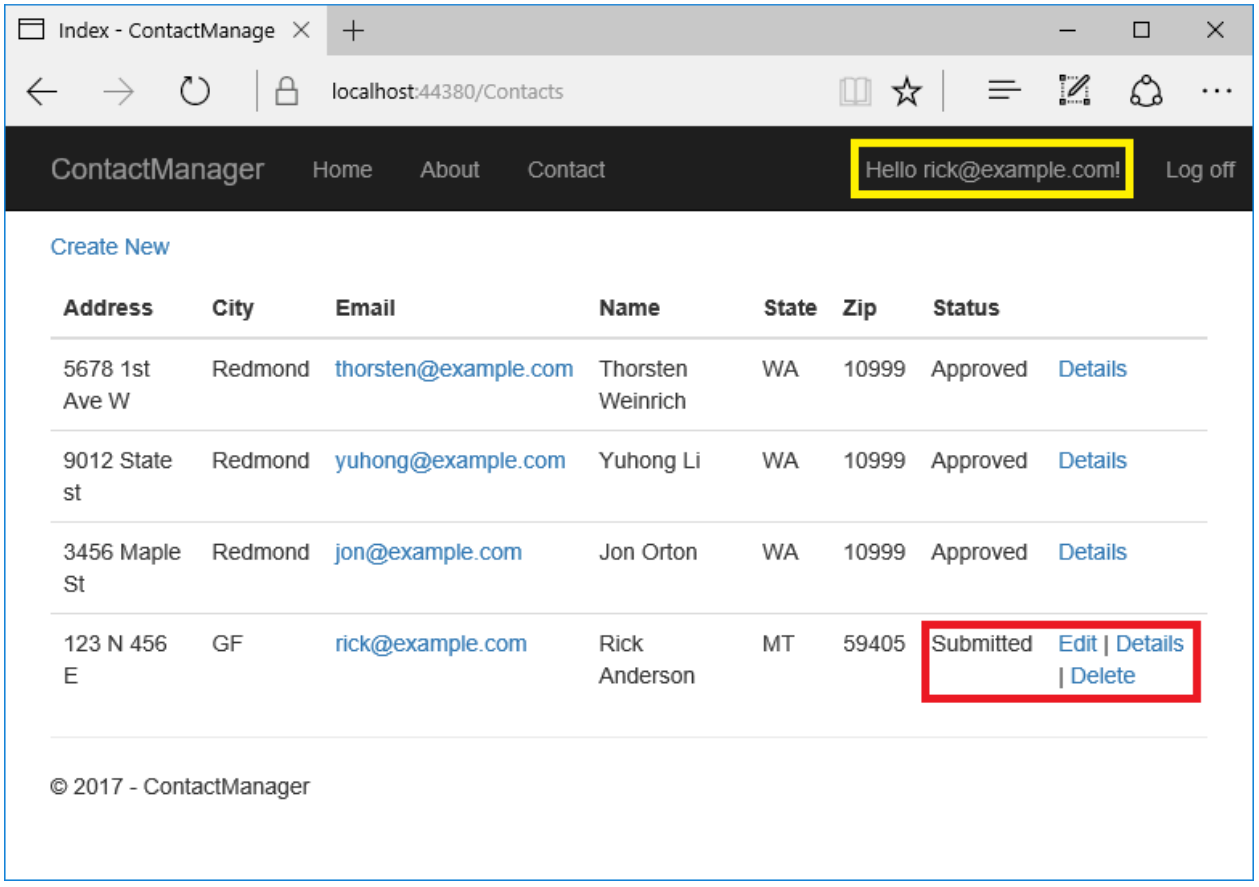
By Rick Anderson and Joe Audette

This tutorial shows how to create an ASP.NET Core web app with user data protected by authorization. It displays a list of contacts that authenticated (registered) users have created. There are three security groups:

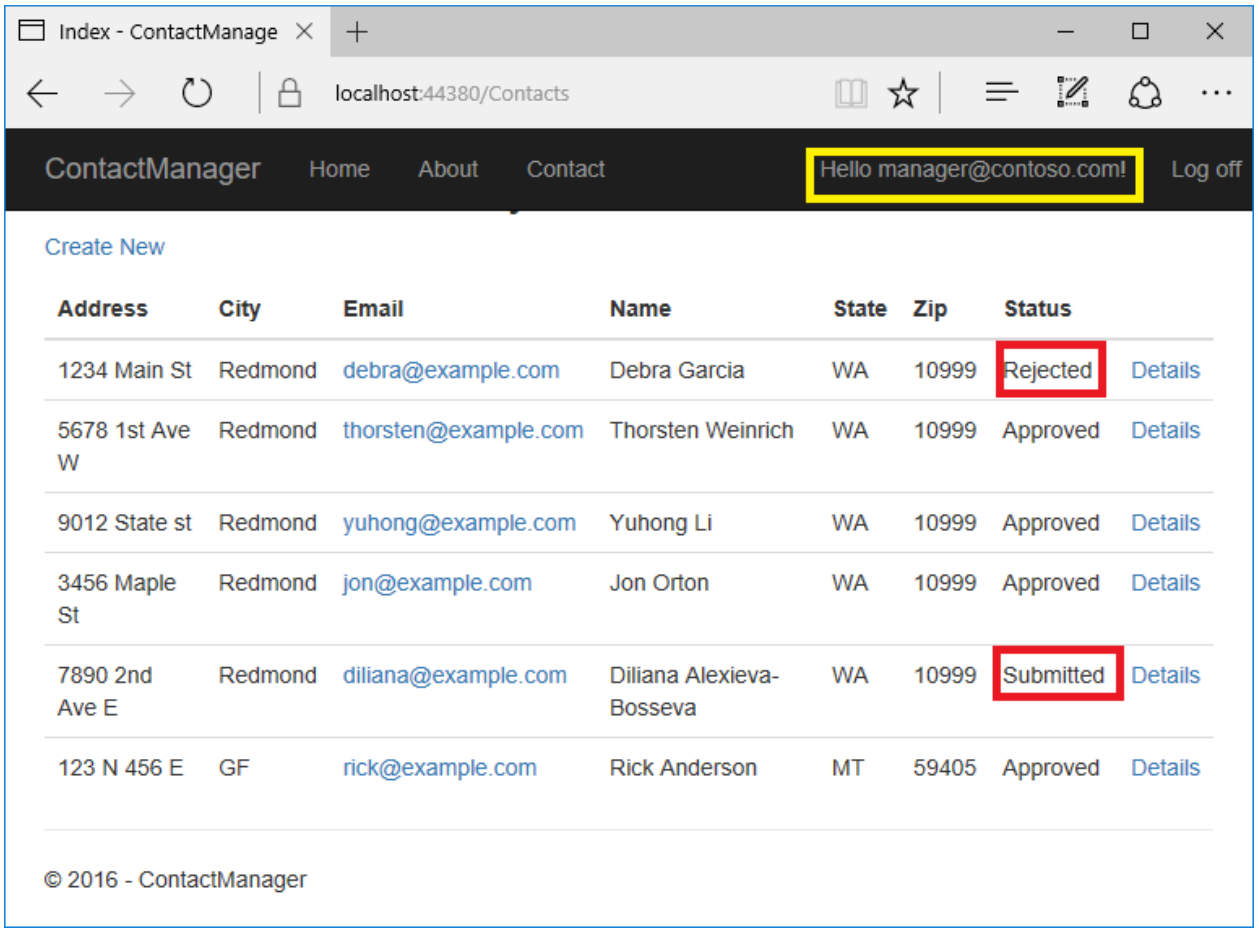
- **Registered users** can view all the approved data and can edit/delete their own data.
- **Managers** can approve or reject contact data. Only approved contacts are visible to users.
- **Administrators** can approve/reject and edit/delete any data.

The images in this document don't exactly match the latest templates.

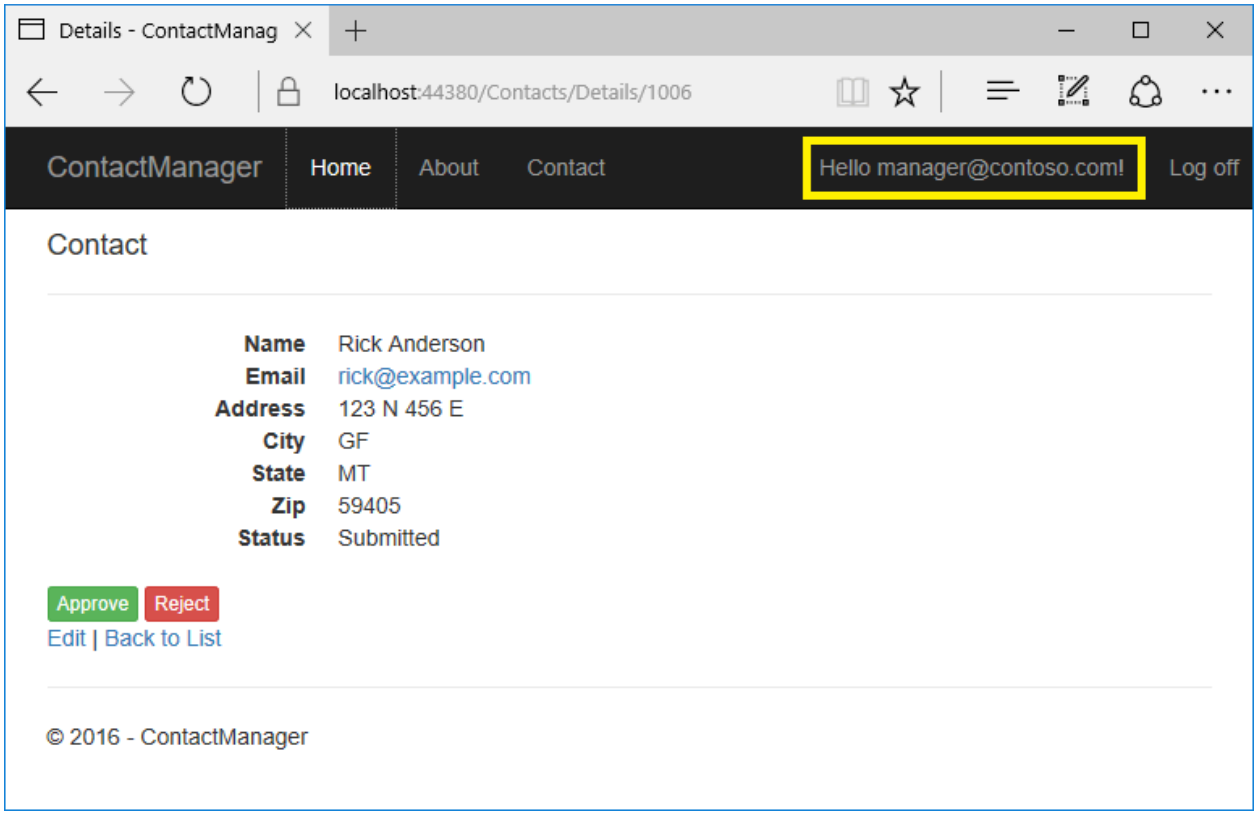
In the following image, user Rick (`rick@example.com`) is signed in. Rick can only view approved contacts and **Edit/Delete/Create New** links for his contacts. Only the last record, created by Rick, displays **Edit** and **Delete** links. Other users won't see the last record until a manager or administrator changes the status to "Approved".



In the following image, `manager@contoso.com` is signed in and in the manager's role:

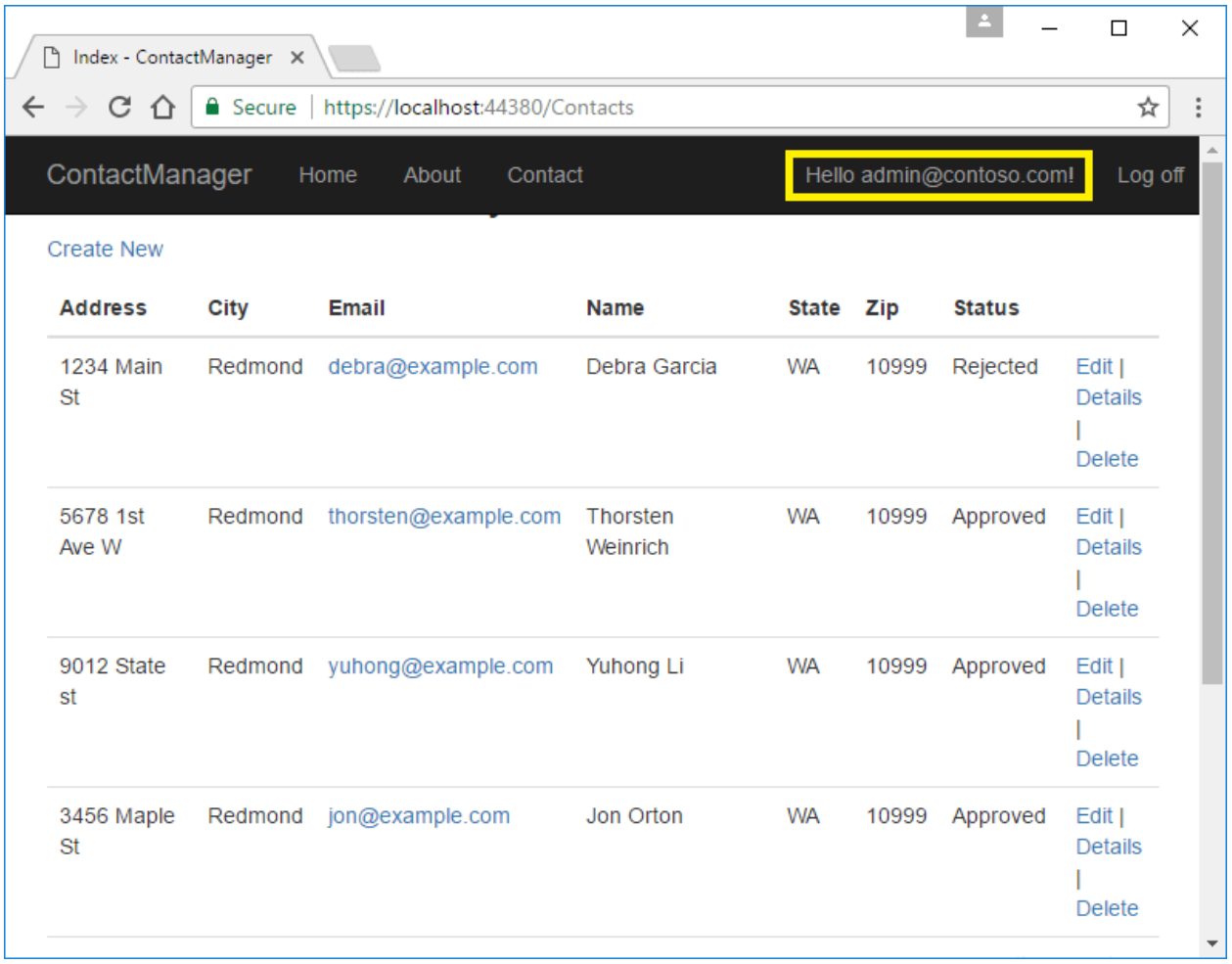


The following image shows the managers details view of a contact:



The **Approve** and **Reject** buttons are only displayed for managers and administrators.

In the following image, `admin@contoso.com` is signed in and in the administrator's role:



The administrator has all privileges. She can read, edit, or delete any contact and change the status of contacts.

The app was created by scaffolding the following `Contact` model:

C#

```
public class Contact
{
    public int ContactId { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

The sample contains the following authorization handlers:

- `ContactIsOwnerAuthorizationHandler`: Ensures that a user can only edit their data.

- `ContactManagerAuthorizationHandler`: Allows managers to approve or reject contacts.
- `ContactAdministratorsAuthorizationHandler`: Allows administrators to approve or reject contacts and to edit/delete contacts.

Prerequisites

This tutorial is advanced. You should be familiar with:

- ASP.NET Core
- Authentication
- Account Confirmation and Password Recovery
- Authorization
- Entity Framework Core

The starter and completed app

Download the completed app. Test the completed app so you become familiar with its security features.

> **Tip**
>
> Use **git sparse-checkout** to only download the sample subfolder. For example:

```
git clone --depth 1 --filter=blob:none
https://github.com/dotnet/AspNetCore.Docs.git --sparse
cd AspNetCore.Docs
git sparse-checkout init --cone
git sparse-checkout set aspnetcore/security/authorization/secure-
data/samples
```

The starter app

Download the starter app.

Run the app, tap the **ContactManager** link, and verify you can create, edit, and delete a contact. To create the starter app, see Create the starter app.

Secure user data

The following sections have all the major steps to create the secure user data app. You may find it helpful to refer to the completed project.

Tie the contact data to the user

Use the ASP.NET Identity user ID to ensure users can edit their data, but not other users data. Add `OwnerID` and `ContactStatus` to the `Contact` model:

C#

```
public class Contact
{
    public int ContactId { get; set; }

    // user ID from AspNetUser table.
    public string? OwnerID { get; set; }

    public string? Name { get; set; }
    public string? Address { get; set; }
    public string? City { get; set; }
    public string? State { get; set; }
    public string? Zip { get; set; }
    [DataType(DataType.EmailAddress)]
    public string? Email { get; set; }

    public ContactStatus Status { get; set; }
}

public enum ContactStatus
{
    Submitted,
    Approved,
    Rejected
}
```

`OwnerID` is the user's ID from the `AspNetUser` table in the Identity database. The `Status` field determines if a contact is viewable by general users.

Create a new migration and update the database:

.NET CLI

```
dotnet ef migrations add userID_Status
dotnet ef database update
```

Add Role services to Identity

Append AddRoles to add Role services:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

Require authenticated users

Set the fallback authorization policy to require users to be authenticated:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});
```

The preceding highlighted code sets the fallback authorization policy. The fallback authorization policy requires ***all*** users to be authenticated, except for Razor Pages,

controllers, or action methods with an authorization attribute. For example, Razor Pages, controllers, or action methods with `[AllowAnonymous]` or `[Authorize(PolicyName="MyPolicy")]` use the applied authorization attribute rather than the fallback authorization policy.

RequireAuthenticatedUser adds DenyAnonymousAuthorizationRequirement to the current instance, which enforces that the current user is authenticated.

The fallback authorization policy:

- Is applied to all requests that don't explicitly specify an authorization policy. For requests served by endpoint routing, this includes any endpoint that doesn't specify an authorization attribute. For requests served by other middleware after the authorization middleware, such as static files, this applies the policy to all requests.

Setting the fallback authorization policy to require users to be authenticated protects newly added Razor Pages and controllers. Having authorization required by default is more secure than relying on new controllers and Razor Pages to include the `[Authorize]` attribute.

The AuthorizationOptions class also contains AuthorizationOptions.DefaultPolicy. The `DefaultPolicy` is the policy used with the `[Authorize]` attribute when no policy is specified. `[Authorize]` doesn't contain a named policy, unlike `[Authorize(PolicyName="MyPolicy")]`.

For more information on policies, see Policy-based authorization in ASP.NET Core.

An alternative way for MVC controllers and Razor Pages to require all users be authenticated is adding an authorization filter:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.Authorization;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();
```

```
builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddControllers(config =>
{
    var policy = new AuthorizationPolicyBuilder()
                     .RequireAuthenticatedUser()
                     .Build();
    config.Filters.Add(new AuthorizeFilter(policy));
});

var app = builder.Build();
```

The preceding code uses an authorization filter, setting the fallback policy uses endpoint routing. Setting the fallback policy is the preferred way to require all users be authenticated.

Add AllowAnonymous to the `Index` and `Privacy` pages so anonymous users can get information about the site before they register:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages;

[AllowAnonymous]
public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {

    }
}
```

Configure the test account

The `SeedData` class creates two accounts: administrator and manager. Use the Secret Manager tool to set a password for these accounts. Set the password from the project directory (the directory containing `Program.cs`):

.NET CLI

```
dotnet user-secrets set SeedUserPW <PW>
```

If a weak password is specified, an exception is thrown when `SeedData.Initialize` is called.

Update the app to use the test password:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});

// Authorization handlers.
builder.Services.AddScoped<IAuthorizationHandler,
                      ContactIsOwnerAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactAdministratorsAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactManagerAuthorizationHandler>();

var app = builder.Build();
```

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<ApplicationDbContext>();
    context.Database.Migrate();
    // requires using Microsoft.Extensions.Configuration;
    // Set password with the Secret Manager tool.
    // dotnet user-secrets set SeedUserPW <pw>

    var testUserPw = builder.Configuration.GetValue<string>("SeedUserPW");

   await SeedData.Initialize(services, testUserPw);
}
```

Create the test accounts and update the contacts

Update the `Initialize` method in the `SeedData` class to create the test accounts:

C#

```
public static async Task Initialize(IServiceProvider serviceProvider, string
testUserPw)
{
    using (var context = new ApplicationDbContext(

serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>
()))
    {
        // For sample purposes seed both with the same password.
        // Password is set with the following:
        // dotnet user-secrets set SeedUserPW <pw>
        // The admin user can do anything

        var adminID = await EnsureUser(serviceProvider, testUserPw,
"admin@contoso.com");
        await EnsureRole(serviceProvider, adminID,
Constants.ContactAdministratorsRole);

        // allowed user can create and edit contacts that they create
        var managerID = await EnsureUser(serviceProvider, testUserPw,
"manager@contoso.com");
        await EnsureRole(serviceProvider, managerID,
Constants.ContactManagersRole);

        SeedDB(context, adminID);
    }
}

private static async Task<string> EnsureUser(IServiceProvider
serviceProvider,
                                            string testUserPw, string
UserName)
```

```
{
    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>
();

    var user = await userManager.FindByNameAsync(UserName);
    if (user == null)
    {
        user = new IdentityUser
        {
            UserName = UserName,
            EmailConfirmed = true
        };
        await userManager.CreateAsync(user, testUserPw);
    }

    if (user == null)
    {
        throw new Exception("The password is probably not strong enough!");
    }

    return user.Id;
}

private static async Task<IdentityResult> EnsureRole(IServiceProvider
serviceProvider,
                                                     string uid,
string role)
{
    var roleManager = serviceProvider.GetService<RoleManager<IdentityRole>>
();

    if (roleManager == null)
    {
        throw new Exception("roleManager null");
    }

    IdentityResult IR;
    if (!await roleManager.RoleExistsAsync(role))
    {
        IR = await roleManager.CreateAsync(new IdentityRole(role));
    }

    var userManager = serviceProvider.GetService<UserManager<IdentityUser>>
();

    //if (userManager == null)
    //{
    //    throw new Exception("userManager is null");
    //}

    var user = await userManager.FindByIdAsync(uid);

    if (user == null)
    {
        throw new Exception("The testUserPw password was probably not strong
```

```
enough!");
    }

    IR = await userManager.AddToRoleAsync(user, role);

    return IR;
}
```

Add the administrator user ID and `ContactStatus` to the contacts. Make one of the contacts "Submitted" and one "Rejected". Add the user ID and status to all the contacts. Only one contact is shown:

C#

```
public static void SeedDB(ApplicationDbContext context, string adminID)
{
    if (context.Contact.Any())
    {
        return;   // DB has been seeded
    }

    context.Contact.AddRange(
        new Contact
        {
            Name = "Debra Garcia",
            Address = "1234 Main St",
            City = "Redmond",
            State = "WA",
            Zip = "10999",
            Email = "debra@example.com",
            Status = ContactStatus.Approved,
            OwnerID = adminID
        },
```

Create owner, manager, and administrator authorization handlers

Create a `ContactIsOwnerAuthorizationHandler` class in the *Authorization* folder. The `ContactIsOwnerAuthorizationHandler` verifies that the user acting on a resource owns the resource.

C#

```
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;
```

```
using System.Threading.Tasks;

namespace ContactManager.Authorization
{
    public class ContactIsOwnerAuthorizationHandler
                : AuthorizationHandler<OperationAuthorizationRequirement,
Contact>
    {
        UserManager<IdentityUser> _userManager;

        public ContactIsOwnerAuthorizationHandler(UserManager<IdentityUser>
            userManager)
        {
            _userManager = userManager;
        }

        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement
requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for CRUD permission, return.

            if (requirement.Name != Constants.CreateOperationName &&
                requirement.Name != Constants.ReadOperationName   &&
                requirement.Name != Constants.UpdateOperationName &&
                requirement.Name != Constants.DeleteOperationName )
            {
                return Task.CompletedTask;
            }

            if (resource.OwnerID == _userManager.GetUserId(context.User))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

The `ContactIsOwnerAuthorizationHandler` calls context.Succeed if the current authenticated user is the contact owner. Authorization handlers generally:

- Call `context.Succeed` when the requirements are met.

- Return `Task.CompletedTask` when requirements aren't met. Returning `Task.CompletedTask` without a prior call to `context.Success` or `context.Fail`, is not a success or failure, it allows other authorization handlers to run.

If you need to explicitly fail, call context.Fail.

The app allows contact owners to edit/delete/create their own data. `ContactIsOwnerAuthorizationHandler` doesn't need to check the operation passed in the requirement parameter.

Create a manager authorization handler

Create a `ContactManagerAuthorizationHandler` class in the *Authorization* folder. The `ContactManagerAuthorizationHandler` verifies the user acting on the resource is a manager. Only managers can approve or reject content changes (new or changed).

C#

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;
using Microsoft.AspNetCore.Identity;

namespace ContactManager.Authorization
{
    public class ContactManagerAuthorizationHandler :
        AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task
            HandleRequirementAsync(AuthorizationHandlerContext context,
                                   OperationAuthorizationRequirement
requirement,
                                   Contact resource)
        {
            if (context.User == null || resource == null)
            {
                return Task.CompletedTask;
            }

            // If not asking for approval/reject, return.
            if (requirement.Name != Constants.ApproveOperationName &&
                requirement.Name != Constants.RejectOperationName)
            {
                return Task.CompletedTask;
            }

            // Managers can approve or reject.
            if (context.User.IsInRole(Constants.ContactManagersRole))
            {
```

```
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Create an administrator authorization handler

Create a `ContactAdministratorsAuthorizationHandler` class in the *Authorization* folder. The `ContactAdministratorsAuthorizationHandler` verifies the user acting on the resource is an administrator. Administrator can do all operations.

C#

```
using System.Threading.Tasks;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public class ContactAdministratorsAuthorizationHandler
                      :
AuthorizationHandler<OperationAuthorizationRequirement, Contact>
    {
        protected override Task HandleRequirementAsync(
                                                AuthorizationHandlerContext
context,
                                     OperationAuthorizationRequirement
requirement,
                                      Contact resource)
        {
            if (context.User == null)
            {
                return Task.CompletedTask;
            }

            // Administrators can do anything.
            if (context.User.IsInRole(Constants.ContactAdministratorsRole))
            {
                context.Succeed(requirement);
            }

            return Task.CompletedTask;
        }
    }
}
```

Register the authorization handlers

Services using Entity Framework Core must be registered for dependency injection using AddScoped. The `ContactIsOwnerAuthorizationHandler` uses ASP.NET Core Identity, which is built on Entity Framework Core. Register the handlers with the service collection so they're available to the `ContactsController` through dependency injection. Add the following code to the end of `ConfigureServices`:

C#

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(
    options => options.SignIn.RequireConfirmedAccount = true)
    .AddRoles<IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();

builder.Services.AddAuthorization(options =>
{
    options.FallbackPolicy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
});

// Authorization handlers.
builder.Services.AddScoped<IAuthorizationHandler,
                      ContactIsOwnerAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactAdministratorsAuthorizationHandler>();

builder.Services.AddSingleton<IAuthorizationHandler,
                      ContactManagerAuthorizationHandler>();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var context = services.GetRequiredService<ApplicationDbContext>();
    context.Database.Migrate();
    // requires using Microsoft.Extensions.Configuration;
    // Set password with the Secret Manager tool.
    // dotnet user-secrets set SeedUserPW <pw>
```

```
    var testUserPw = builder.Configuration.GetValue<string>("SeedUserPW");

   await SeedData.Initialize(services, testUserPw);
}
```

`ContactAdministratorsAuthorizationHandler` and `ContactManagerAuthorizationHandler` are added as singletons. They're singletons because they don't use EF and all the information needed is in the `Context` parameter of the `HandleRequirementAsync` method.

Support authorization

In this section, you update the Razor Pages and add an operations requirements class.

Review the contact operations requirements class

Review the `ContactOperations` class. This class contains the requirements the app supports:

C#

```
using Microsoft.AspNetCore.Authorization.Infrastructure;

namespace ContactManager.Authorization
{
    public static class ContactOperations
    {
        public static OperationAuthorizationRequirement Create =
          new OperationAuthorizationRequirement
{Name=Constants.CreateOperationName};
        public static OperationAuthorizationRequirement Read =
          new OperationAuthorizationRequirement
{Name=Constants.ReadOperationName};
        public static OperationAuthorizationRequirement Update =
          new OperationAuthorizationRequirement
{Name=Constants.UpdateOperationName};
        public static OperationAuthorizationRequirement Delete =
          new OperationAuthorizationRequirement
{Name=Constants.DeleteOperationName};
        public static OperationAuthorizationRequirement Approve =
          new OperationAuthorizationRequirement
{Name=Constants.ApproveOperationName};
        public static OperationAuthorizationRequirement Reject =
          new OperationAuthorizationRequirement
{Name=Constants.RejectOperationName};
    }

    public class Constants
    {
```

```
        public static readonly string CreateOperationName = "Create";
        public static readonly string ReadOperationName = "Read";
        public static readonly string UpdateOperationName = "Update";
        public static readonly string DeleteOperationName = "Delete";
        public static readonly string ApproveOperationName = "Approve";
        public static readonly string RejectOperationName = "Reject";

        public static readonly string ContactAdministratorsRole =

"ContactAdministrators";
        public static readonly string ContactManagersRole =
"ContactManagers";
    }
}
```

Create a base class for the Contacts Razor Pages

Create a base class that contains the services used in the contacts Razor Pages. The base class puts the initialization code in one location:

C#

```
using ContactManager.Data;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ContactManager.Pages.Contacts
{
    public class DI_BasePageModel : PageModel
    {
        protected ApplicationDbContext Context { get; }
        protected IAuthorizationService AuthorizationService { get; }
        protected UserManager<IdentityUser> UserManager { get; }

        public DI_BasePageModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<IdentityUser> userManager) : base()
        {
            Context = context;
            UserManager = userManager;
            AuthorizationService = authorizationService;
        }
    }
}
```

The preceding code:

- Adds the `IAuthorizationService` service to access to the authorization handlers.

- Adds the Identity `UserManager` service.
- Add the `ApplicationDbContext`.

Update the CreateModel

Update the create page model:

- Constructor to use the `DI_BasePageModel` base class.
- `OnPostAsync` method to:
  - Add the user ID to the `Contact` model.
  - Call the authorization handler to verify the user has permission to create contacts.

C#

```
using ContactManager.Authorization;
using ContactManager.Data;
using ContactManager.Models;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace ContactManager.Pages.Contacts
{
    public class CreateModel : DI_BasePageModel
    {
        public CreateModel(
            ApplicationDbContext context,
            IAuthorizationService authorizationService,
            UserManager<IdentityUser> userManager)
            : base(context, authorizationService, userManager)
        {
        }

        public IActionResult OnGet()
        {
            return Page();
        }

        [BindProperty]
        public Contact Contact { get; set; }

        public async Task<IActionResult> OnPostAsync()
        {
            if (!ModelState.IsValid)
            {
                return Page();
            }

            Contact.OwnerID = UserManager.GetUserId(User);
```

```
            var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                        User, Contact,

ContactOperations.Create);
            if (!isAuthorized.Succeeded)
            {
                return Forbid();
            }

            Context.Contact.Add(Contact);
            await Context.SaveChangesAsync();

            return RedirectToPage("./Index");
        }
    }
}
```

Update the IndexModel

Update the `OnGetAsync` method so only approved contacts are shown to general users:

C#

```
public class IndexModel : DI_BasePageModel
{
    public IndexModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public IList<Contact> Contact { get; set; }

    public async Task OnGetAsync()
    {
        var contacts = from c in Context.Contact
                       select c;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||

User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        // Only approved contacts are shown UNLESS you're authorized to see
them
        // or you are the owner.
        if (!isAuthorized)
        {
```

```
            contacts = contacts.Where(c => c.Status ==
ContactStatus.Approved
                                      || c.OwnerID == currentUserId);
        }

        Contact = await contacts.ToListAsync();
    }
}
```

Update the EditModel

Add an authorization handler to verify the user owns the contact. Because resource authorization is being validated, the `[Authorize]` attribute is not enough. The app doesn't have access to the resource when attributes are evaluated. Resource-based authorization must be imperative. Checks must be performed once the app has access to the resource, either by loading it in the page model or by loading it within the handler itself. You frequently access the resource by passing in the resource key.

C#

```
public class EditModel : DI_BasePageModel
{
    public EditModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? contact = await Context.Contact.FirstOrDefaultAsync(
                                                    m => m.ContactId ==
id);
        if (contact == null)
        {
            return NotFound();
        }

        Contact = contact;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, Contact,
                                                ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
```

```
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        // Fetch Contact from DB to get OwnerID.
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                 User, contact,
                                                 ContactOperations.Update);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        Contact.OwnerID = contact.OwnerID;

        Context.Attach(Contact).State = EntityState.Modified;

        if (Contact.Status == ContactStatus.Approved)
        {
            // If the contact is updated after approval,
            // and the user cannot approve,
            // set the status back to submitted so the update can be
            // checked and approved.
            var canApprove = await AuthorizationService.AuthorizeAsync(User,
                                     Contact,
                                     ContactOperations.Approve);

            if (!canApprove.Succeeded)
            {
                Contact.Status = ContactStatus.Submitted;
            }
        }

        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
```

```
    }
}
```

Update the DeleteModel

Update the delete page model to use the authorization handler to verify the user has delete permission on the contact.

C#

```
public class DeleteModel : DI_BasePageModel
{
    public DeleteModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    [BindProperty]
    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? _contact = await Context.Contact.FirstOrDefaultAsync(
                                            m => m.ContactId == id);

        if (_contact == null)
        {
            return NotFound();
        }
        Contact = _contact;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                User, Contact,
                                                ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id)
    {
        var contact = await Context
            .Contact.AsNoTracking()
            .FirstOrDefaultAsync(m => m.ContactId == id);

        if (contact == null)
```

```
        {
            return NotFound();
        }

        var isAuthorized = await AuthorizationService.AuthorizeAsync(
                                                 User, contact,
                                                 ContactOperations.Delete);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }

        Context.Contact.Remove(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Inject the authorization service into the views

Currently, the UI shows edit and delete links for contacts the user can't modify.

Inject the authorization service in the `Pages/_ViewImports.cshtml` file so it's available to all views:

CSHTML

```
@using Microsoft.AspNetCore.Identity
@using ContactManager
@using ContactManager.Data
@namespace ContactManager.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@using ContactManager.Authorization;
@using Microsoft.AspNetCore.Authorization
@using ContactManager.Models
@inject IAuthorizationService AuthorizationService
```

The preceding markup adds several `using` statements.

Update the **Edit** and **Delete** links in `Pages/Contacts/Index.cshtml` so they're only rendered for users with the appropriate permissions:

CSHTML

```
@page
@model ContactManager.Pages.Contacts.IndexModel

@{
```

```
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Address)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].City)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].State)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Zip)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].Email)
            </th>
             <th>
                @Html.DisplayNameFor(model => model.Contact[0].Status)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Contact) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Address)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.City)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.State)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Zip)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Email)
```

```
            </td>
                        <td>
                    @Html.DisplayFor(modelItem => item.Status)
                </td>
                <td>
                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Update)).Succeeded)
                    {
                        <a asp-page="./Edit" asp-route-
id="@item.ContactId">Edit</a>
                        <text> | </text>
                    }

                    <a asp-page="./Details" asp-route-
id="@item.ContactId">Details</a>

                    @if ((await AuthorizationService.AuthorizeAsync(
                     User, item,
                     ContactOperations.Delete)).Succeeded)
                    {
                        <text> | </text>
                        <a asp-page="./Delete" asp-route-
id="@item.ContactId">Delete</a>
                    }
                </td>
            </tr>
        }
    </tbody>
</table>
```

> ⚠ **Warning**
>
> Hiding links from users that don't have permission to change data doesn't secure the app. Hiding links makes the app more user-friendly by displaying only valid links. Users can hack the generated URLs to invoke edit and delete operations on data they don't own. The Razor Page or controller must enforce access checks to secure the data.

Update Details

Update the details view so managers can approve or reject contacts:

CSHTML

```
        @*Preceding markup omitted for brevity.*@
        <dt class="col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.Contact.Email)
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Contact.Email)
        </dd>
    <dt>
            @Html.DisplayNameFor(model => model.Contact.Status)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Contact.Status)
        </dd>
    </dl>
</div>

@if (Model.Contact.Status != ContactStatus.Approved)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Approve)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId"
/>
            <input type="hidden" name="status"
value="@ContactStatus.Approved" />
            <button type="submit" class="btn btn-xs btn-
success">Approve</button>
        </form>
    }
}

@if (Model.Contact.Status != ContactStatus.Rejected)
{
    @if ((await AuthorizationService.AuthorizeAsync(
     User, Model.Contact, ContactOperations.Reject)).Succeeded)
    {
        <form style="display:inline;" method="post">
            <input type="hidden" name="id" value="@Model.Contact.ContactId"
/>
            <input type="hidden" name="status"
value="@ContactStatus.Rejected" />
            <button type="submit" class="btn btn-xs btn-
danger">Reject</button>
        </form>
    }
}

<div>
    @if ((await AuthorizationService.AuthorizeAsync(
         User, Model.Contact,
         ContactOperations.Update)).Succeeded)
    {
        <a asp-page="./Edit" asp-route-
id="@Model.Contact.ContactId">Edit</a>
        <text> | </text>
    }
```

```
    <a asp-page="./Index">Back to List</a>
</div>
```

Update the details page model

C#

```
public class DetailsModel : DI_BasePageModel
{
    public DetailsModel(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? _contact = await Context.Contact.FirstOrDefaultAsync(m =>
m.ContactId == id);

        if (_contact == null)
        {
            return NotFound();
        }
        Contact = _contact;

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||

User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            && currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return Forbid();
        }

        return Page();
    }

    public async Task<IActionResult> OnPostAsync(int id, ContactStatus
status)
    {
        var contact = await Context.Contact.FirstOrDefaultAsync(
                                                    m => m.ContactId == id);
```

```
        if (contact == null)
        {
            return NotFound();
        }

        var contactOperation = (status == ContactStatus.Approved)
                                                   ?
ContactOperations.Approve
                                                   :
ContactOperations.Reject;

        var isAuthorized = await AuthorizationService.AuthorizeAsync(User,
contact,
                                    contactOperation);
        if (!isAuthorized.Succeeded)
        {
            return Forbid();
        }
        contact.Status = status;
        Context.Contact.Update(contact);
        await Context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Add or remove a user to a role

See this issue for information on:

- Removing privileges from a user. For example, muting a user in a chat app.
- Adding privileges to a user.

Differences between Challenge and Forbid

This app sets the default policy to require authenticated users. The following code allows anonymous users. Anonymous users are allowed to show the differences between Challenge vs Forbid.

C#

```
[AllowAnonymous]
public class Details2Model : DI_BasePageModel
{
    public Details2Model(
        ApplicationDbContext context,
        IAuthorizationService authorizationService,
        UserManager<IdentityUser> userManager)
```

```
        : base(context, authorizationService, userManager)
    {
    }

    public Contact Contact { get; set; }

    public async Task<IActionResult> OnGetAsync(int id)
    {
        Contact? _contact = await Context.Contact.FirstOrDefaultAsync(m =>
m.ContactId == id);

        if (_contact == null)
        {
            return NotFound();
        }
        Contact = _contact;

        if (!User.Identity!.IsAuthenticated)
        {
            return Challenge();
        }

        var isAuthorized = User.IsInRole(Constants.ContactManagersRole) ||

User.IsInRole(Constants.ContactAdministratorsRole);

        var currentUserId = UserManager.GetUserId(User);

        if (!isAuthorized
            && currentUserId != Contact.OwnerID
            && Contact.Status != ContactStatus.Approved)
        {
            return Forbid();
        }

        return Page();
    }
}
```

In the preceding code:

- When the user is **not** authenticated, a `ChallengeResult` is returned. When a `ChallengeResult` is returned, the user is redirected to the sign-in page.
- When the user is authenticated, but not authorized, a `ForbidResult` is returned. When a `ForbidResult` is returned, the user is redirected to the access denied page.

Test the completed app

> ⚠ **Warning**

This article uses the Secret Manager tool to store the password for the seeded user accounts. The Secret Manager tool is used to store sensitive data during local development. For information on authentication procedures that can be used when an app is deployed to a test or production environment, see Secure authentication flows.

If you haven't already set a password for seeded user accounts, use the Secret Manager tool to set a password:

- Choose a strong password:
  - At least 12 characters long but 14 or more is better.
  - A combination of uppercase letters, lowercase letters, numbers, and symbols.
  - Not a word that can be found in a dictionary or the name of a person, character, product, or organization.
  - Significantly different from your previous passwords.
  - Easy for you to remember but difficult for others to guess. Consider using a memorable phrase like "6MonkeysRLooking^".
- Execute the following command from the project's folder, where `<PW>` is the password:

  .NET CLI

  ```
  dotnet user-secrets set SeedUserPW <PW>
  ```

If the app has contacts:

- Delete all of the records in the `Contact` table.
- Restart the app to seed the database.

An easy way to test the completed app is to launch three different browsers (or incognito/InPrivate sessions). In one browser, register a new user (for example, `test@example.com`). Sign in to each browser with a different user. Verify the following operations:

- Registered users can view all of the approved contact data.
- Registered users can edit/delete their own data.
- Managers can approve/reject contact data. The `Details` view shows **Approve** and **Reject** buttons.
- Administrators can approve/reject and edit/delete all data.

Expand table

User	Approve or reject contacts	Options
test@example.com	No	Edit and delete their data.
manager@contoso.com	Yes	Edit and delete their data.
admin@contoso.com	Yes	Edit and delete ***all*** data.

Create a contact in the administrator's browser. Copy the URL for delete and edit from the administrator contact. Paste these links into the test user's browser to verify the test user can't perform these operations.

Create the starter app

- Create a Razor Pages app named "ContactManager"
  - Create the app with **Individual User Accounts**.
  - Name it "ContactManager" so the namespace matches the namespace used in the sample.
  - `-uld` specifies LocalDB instead of SQLite

.NET CLI

```
dotnet new webapp -o ContactManager -au Individual -uld
```

- Add `Models/Contact.cs`: secure-data\samples\starter6\ContactManager\Models\Contact.cs

C#

```
using System.ComponentModel.DataAnnotations;

namespace ContactManager.Models
{
    public class Contact
    {
        public int ContactId { get; set; }
        public string? Name { get; set; }
        public string? Address { get; set; }
        public string? City { get; set; }
        public string? State { get; set; }
        public string? Zip { get; set; }
        [DataType(DataType.EmailAddress)]
        public string? Email { get; set; }
    }
}
```

- Scaffold the `Contact` model.
- Create initial migration and update the database:

.NET CLI

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet tool install -g dotnet-aspnet-codegenerator
dotnet-aspnet-codegenerator razorpage -m Contact -udl -dc
ApplicationDbContext -outDir Pages\Contacts --referenceScriptLibraries
dotnet ef database drop -f
dotnet ef migrations add initial
dotnet ef database update
```

> **Note**
>
> By default the architecture of the .NET binaries to install represents the currently running OS architecture. To specify a different OS architecture, see dotnet tool install, --arch option. For more information, see GitHub issue dotnet/AspNetCore.Docs #29262.

- Update the **ContactManager** anchor in the `Pages/Shared/_Layout.cshtml` file:

CSHTML

```
<a class="nav-link text-dark" asp-area="" asp-
page="/Contacts/Index">Contact Manager</a>
```

- Test the app by creating, editing, and deleting a contact

Seed the database

Add the SeedData class to the *Data* folder:

C#

```
using ContactManager.Models;
using Microsoft.EntityFrameworkCore;

// dotnet aspnet-codegenerator razorpage -m Contact -dc ApplicationDbContext
-udl -outDir Pages\Contacts --referenceScriptLibraries

namespace ContactManager.Data
{
    public static class SeedData
    {
```

```
        public static async Task Initialize(IServiceProvider
serviceProvider, string testUserPw="")
        {
            using (var context = new ApplicationDbContext(

serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>
()))
            {
                SeedDB(context, testUserPw);
            }
        }

        public static void SeedDB(ApplicationDbContext context, string
adminID)
        {
            if (context.Contact.Any())
            {
                return;   // DB has been seeded
            }

            context.Contact.AddRange(
                new Contact
                {
                    Name = "Debra Garcia",
                    Address = "1234 Main St",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "debra@example.com"
                },
                new Contact
                {
                    Name = "Thorsten Weinrich",
                    Address = "5678 1st Ave W",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "thorsten@example.com"
                },
                new Contact
                {
                    Name = "Yuhong Li",
                    Address = "9012 State st",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "yuhong@example.com"
                },
                new Contact
                {
                    Name = "Jon Orton",
                    Address = "3456 Maple St",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
```

```
                    Email = "jon@example.com"
                },
                new Contact
                {
                    Name = "Diliana Alexieva-Bosseva",
                    Address = "7890 2nd Ave E",
                    City = "Redmond",
                    State = "WA",
                    Zip = "10999",
                    Email = "diliana@example.com"
                }
             );
            context.SaveChanges();
        }

    }
}
```

Call `SeedData.Initialize` from `Program.cs`:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using ContactManager.Data;

var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
builder.Services.AddRazorPages();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    await SeedData.Initialize(services);
}

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
```

```
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Test that the app seeded the database. If there are any rows in the contact DB, the seed method doesn't run.

Additional resources

- Tutorial: Build an ASP.NET Core and Azure SQL Database app in Azure App Service
- ASP.NET Core Authorization Lab. This lab goes into more detail on the security features introduced in this tutorial.
- Introduction to authorization in ASP.NET Core
- Custom policy-based authorization

Razor Pages authorization conventions in ASP.NET Core

Article • 06/03/2022

One way to control access in your Razor Pages app is to use authorization conventions at startup. These conventions allow you to authorize users and allow anonymous users to access individual pages or folders of pages. The conventions described in this topic automatically apply authorization filters to control access.

View or download sample code (how to download)

The sample app uses cookie authentication without ASP.NET Core Identity. The concepts and examples shown in this topic apply equally to apps that use ASP.NET Core Identity. To use ASP.NET Core Identity, follow the guidance in Introduction to Identity on ASP.NET Core.

Require authorization to access a page

Use the AuthorizePage convention to add an AuthorizeFilter to the page at the specified path:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

To specify an authorization policy, use an AuthorizePage overload:

C#

```
options.Conventions.AuthorizePage("/Contact", "AtLeast21");
```

Note

An **AuthorizeFilter** can be applied to a page model class with the `[Authorize]` filter attribute. For more information, see **Authorize filter attribute**.

Require authorization to access a folder of pages

Use the AuthorizeFolder convention to add an AuthorizeFilter to all of the pages in a folder at the specified path:

C#

```
services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

To specify an authorization policy, use an AuthorizeFolder overload:

C#

```
options.Conventions.AuthorizeFolder("/Private", "AtLeast21");
```

Require authorization to access an area page

Use the AuthorizeAreaPage convention to add an AuthorizeFilter to the area page at the specified path:

C#

```
options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts");
```

The page name is the path of the file without an extension relative to the pages root directory for the specified area. For example, the page name for the file `Areas/Identity/Pages/Manage/Accounts.cshtml` is */Manage/Accounts*.

To specify an authorization policy, use an AuthorizeAreaPage overload:

```
C#

options.Conventions.AuthorizeAreaPage("Identity", "/Manage/Accounts",
"AtLeast21");
```

Require authorization to access a folder of areas

Use the AuthorizeAreaFolder convention to add an AuthorizeFilter to all of the areas in a folder at the specified path:

```
C#

options.Conventions.AuthorizeAreaFolder("Identity", "/Manage");
```

The folder path is the path of the folder relative to the pages root directory for the specified area. For example, the folder path for the files under *Areas/Identity/Pages/Manage/* is */Manage*.

To specify an authorization policy, use an AuthorizeAreaFolder overload:

```
C#

options.Conventions.AuthorizeAreaFolder("Identity", "/Manage", "AtLeast21");
```

Allow anonymous access to a page

Use the AllowAnonymousToPage convention to add an AllowAnonymousFilter to a page at the specified path:

```
C#

services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path without an extension and containing only forward slashes.

Allow anonymous access to a folder of pages

Use the AllowAnonymousToFolder convention to add an AllowAnonymousFilter to all of the pages in a folder at the specified path:

```
C#

services.AddRazorPages(options =>
{
    options.Conventions.AuthorizePage("/Contact");
    options.Conventions.AuthorizeFolder("/Private");
    options.Conventions.AllowAnonymousToPage("/Private/PublicPage");
    options.Conventions.AllowAnonymousToFolder("/Private/PublicPages");
});
```

The specified path is the View Engine path, which is the Razor Pages root relative path.

Note on combining authorized and anonymous access

It's valid to specify that a folder of pages requires authorization and then specify that a page within that folder allows anonymous access:

```
C#

// This works.
.AuthorizeFolder("/Private").AllowAnonymousToPage("/Private/Public")
```

The reverse, however, isn't valid. You can't declare a folder of pages for anonymous access and then specify a page within that folder that requires authorization:

```
C#

// This doesn't work!
.AllowAnonymousToFolder("/Public").AuthorizePage("/Public/Private")
```

Requiring authorization on the Private page fails. When both the AllowAnonymousFilter and AuthorizeFilter are applied to the page, the AllowAnonymousFilter takes precedence and controls access.

Additional resources

- Razor Pages route and app conventions in ASP.NET Core
- PageConventionCollection

Simple authorization in ASP.NET Core

Article • 05/02/2024

Authorization in ASP.NET Core is controlled with the [Authorize] attribute and its various parameters. In its most basic form, applying the `[Authorize]` attribute to a controller, action, or Razor Page, limits access to that component to authenticated users.

Prerequisites

This article assumes that you have a basic understanding of ASP.NET Core Razor Pages and MVC. If you're new to ASP.NET Core, see the following resources:

- Introduction to Razor Pages in ASP.NET Core
- Overview of ASP.NET Core MVC
- Tutorial: Get started with Razor Pages in ASP.NET Core
- Introduction to Identity on ASP.NET Core

Use the `[Authorize]` attribute

The following code limits access to the `AccountController` to authenticated users:

C#

```
[Authorize]
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

If you want to apply authorization to an action rather than the controller, apply the `AuthorizeAttribute` attribute to the action itself:

C#

```
public class AccountController : Controller
{
   public ActionResult Login()
```

```
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

Now only authenticated users can access the `Logout` function.

You can also use the `AllowAnonymous` attribute to allow access by non-authenticated users to individual actions. For example:

C#

```
[Authorize]
public class AccountController : Controller
{
    [AllowAnonymous]
    public ActionResult Login()
    {
    }

    public ActionResult Logout()
    {
    }
}
```

This would allow only authenticated users to the `AccountController`, except for the `Login` action, which is accessible by everyone, regardless of their authenticated or unauthenticated / anonymous status.

> ⚠ **Warning**
>
> `[AllowAnonymous]` bypasses authorization statements. If you combine `[AllowAnonymous]` and an `[Authorize]` attribute, the `[Authorize]` attributes are ignored. For example if you apply `[AllowAnonymous]` at the controller level:
>
> - Any authorization requirements from `[Authorize]` attributes on the same controller or action methods on the controller are ignored.
> - Authentication middleware is not short-circuited but doesn't need to succeed.

The following code limits access to the `LogoutModel` Razor Page to authenticated users:

C#

```
[Authorize]
public class LogoutModel : PageModel
{
    public async Task OnGetAsync()
    {

    }

    public async Task<IActionResult> OnPostAsync()
    {

    }
}
```

For information on how to globally require all users to be authenticated, see Require authenticated users.

Authorize attribute and Razor Pages

The AuthorizeAttribute can ***not*** be applied to Razor Page handlers. For example, `[Authorize]` can't be applied to `OnGet`, `OnPost`, or any other page handler. Consider using an ASP.NET Core MVC controller for pages with different authorization requirements for different handlers. Using an MVC controller when different authorization requirements are required:

- Is the least complex approach.
- Is the approach recommended by Microsoft.

If you decide not to use an MVC controller, the following two approaches can be used to apply authorization to Razor Page handler methods:

- Use separate pages for page handlers requiring different authorization. Move shared content into one or more partial views. When possible, this is the recommended approach.

- For content that must share a common page, write a filter that performs authorization as part of IAsyncPageFilter.OnPageHandlerSelectionAsync. The PageHandlerAuth GitHub project demonstrates this approach:
  - The AuthorizeIndexPageHandlerFilter implements the authorization filter:

    C#

    ```
    [TypeFilter(typeof(AuthorizeIndexPageHandlerFilter))]
    public class IndexModel : PageModel
    ```

```
{
    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {

    }

    public void OnPost()
    {

    }

    [AuthorizePageHandler]
    public void OnPostAuthorized()
    {

    }
}
```

- The [AuthorizePageHandler] attribute is applied to the `OnPostAuthorized` page handler:

C#

```
public class AuthorizeIndexPageHandlerFilter : IAsyncPageFilter,
IOrderedFilter
{
    private readonly IAuthorizationPolicyProvider policyProvider;
    private readonly IPolicyEvaluator policyEvaluator;

    public AuthorizeIndexPageHandlerFilter(
        IAuthorizationPolicyProvider policyProvider,
        IPolicyEvaluator policyEvaluator)
    {
        this.policyProvider = policyProvider;
        this.policyEvaluator = policyEvaluator;
    }

    // Run late in the selection pipeline
    public int Order => 10000;

    public Task OnPageHandlerExecutionAsync(PageHandlerExecutingContext
context, PageHandlerExecutionDelegate next) => next();

    public async Task
OnPageHandlerSelectionAsync(PageHandlerSelectedContext context)
    {
        var attribute =
```

```
context.HandlerMethod?.MethodInfo?.GetCustomAttribute<AuthorizePageHand
lerAttribute>();
        if (attribute is null)
        {
            return;
        }

        var policy = await
AuthorizationPolicy.CombineAsync(policyProvider, new[] { attribute });
        if (policy is null)
        {
            return;
        }

        await AuthorizeAsync(context, policy);
    }

    #region AuthZ - do not change
    private async Task AuthorizeAsync(ActionContext actionContext,
AuthorizationPolicy policy)
    {
        var httpContext = actionContext.HttpContext;
        var authenticateResult = await
policyEvaluator.AuthenticateAsync(policy, httpContext);
        var authorizeResult = await
policyEvaluator.AuthorizeAsync(policy, authenticateResult, httpContext,
actionContext.ActionDescriptor);
        if (authorizeResult.Challenged)
        {
            if (policy.AuthenticationSchemes.Count > 0)
            {
                foreach (var scheme in policy.AuthenticationSchemes)
                {
                    await httpContext.ChallengeAsync(scheme);
                }
            }
            else
            {
                await httpContext.ChallengeAsync();
            }

            return;
        }
        else if (authorizeResult.Forbidden)
        {
            if (policy.AuthenticationSchemes.Count > 0)
            {
                foreach (var scheme in policy.AuthenticationSchemes)
                {
                    await httpContext.ForbidAsync(scheme);
                }
            }
            else
            {
                await httpContext.ForbidAsync();
```

```
            }

            return;
        }
    }
```

> ⚠ **Warning**
>
> The PageHandlerAuth sample approach does ***not***:
>
> - Compose with authorization attributes applied to the page, page model, or globally. Composing authorization attributes results in authentication and authorization executing multiple times when you have one more `AuthorizeAttribute` or `AuthorizeFilter` instances also applied to the page.
> - Work in conjunction with the rest of ASP.NET Core authentication and authorization system. You must verify using this approach works correctly for your application.

There are no plans to support the `AuthorizeAttribute` on Razor Page handlers.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Custom authorization policies with IAuthorizationRequirementData

Article • 06/15/2023

Consider the following sample that implements a custom `MinimumAgeAuthorizationHandler`:

C#

```
using AuthRequirementsData.Authorization;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAuthentication().AddJwtBearer();
builder.Services.AddAuthorization();
builder.Services.AddControllers();
builder.Services.AddSingleton<IAuthorizationHandler,
MinimumAgeAuthorizationHandler>();

var app = builder.Build();

app.MapControllers();

app.Run();
```

The `MinimumAgeAuthorizationHandler` class:

C#

```
using Microsoft.AspNetCore.Authorization;
using System.Globalization;
using System.Security.Claims;

namespace AuthRequirementsData.Authorization;

class MinimumAgeAuthorizationHandler :
AuthorizationHandler<MinimumAgeAuthorizeAttribute>
{
    private readonly ILogger<MinimumAgeAuthorizationHandler> _logger;

    public
MinimumAgeAuthorizationHandler(ILogger<MinimumAgeAuthorizationHandler>
logger)
    {
        _logger = logger;
    }
```

```
    // Check whether a given MinimumAgeRequirement is satisfied or not for a
particular
    // context.
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
                                                MinimumAgeAuthorizeAttribute
requirement)
    {
        // Log as a warning so that it's very clear in sample output which
authorization
        // policies(and requirements/handlers) are in use.
        _logger.LogWarning("Evaluating authorization requirement for age >=
{age}",

requirement.Age);

        // Check the user's age.
        var dateOfBirthClaim = context.User.FindFirst(c => c.Type ==

ClaimTypes.DateOfBirth);
        if (dateOfBirthClaim != null)
        {
            // If the user has a date of birth claim, check their age.
            var dateOfBirth = Convert.ToDateTime(dateOfBirthClaim.Value,

CultureInfo.InvariantCulture);
            var age = DateTime.Now.Year - dateOfBirth.Year;
            if (dateOfBirth > DateTime.Now.AddYears(-age))
            {
                // Adjust age if the user hasn't had a birthday yet this
year.
                age--;
            }

            // If the user meets the age criterion, mark the authorization
requirement
            // succeeded.
            if (age >= requirement.Age)
            {
                _logger.LogInformation(
                    "Minimum age authorization requirement {age} satisfied",
                      requirement.Age);
                context.Succeed(requirement);
            }
            else
            {
                _logger.LogInformation("Current user's DateOfBirth claim
({dateOfBirth})"
                   + " does not satisfy the minimum age authorization
requirement {age}",
                    dateOfBirthClaim.Value,
                    requirement.Age);
            }
        }
        else
```

```
        {
            _logger.LogInformation("No DateOfBirth claim present");
        }

        return Task.CompletedTask;
    }
}
```

The custom `MinimumAgePolicyProvider`:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.Extensions.Options;

namespace AuthRequirementsData.Authorization;

class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";
    public DefaultAuthorizationPolicyProvider FallbackPolicyProvider { get;
}
    public MinimumAgePolicyProvider(IOptions<AuthorizationOptions> options)
    {
        FallbackPolicyProvider = new
DefaultAuthorizationPolicyProvider(options);
    }
    public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
                            FallbackPolicyProvider.GetDefaultPolicyAsync();
    public Task<AuthorizationPolicy?> GetFallbackPolicyAsync() =>
                            FallbackPolicyProvider.GetFallbackPolicyAsync();

    public Task<AuthorizationPolicy?> GetPolicyAsync(string policyName)
    {
        if (policyName.StartsWith(POLICY_PREFIX,
StringComparison.OrdinalIgnoreCase) &&
            int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var
age))
        {
            var policy = new AuthorizationPolicyBuilder(

JwtBearerDefaults.AuthenticationScheme);
            policy.AddRequirements(new MinimumAgeRequirement(age));
            return Task.FromResult<AuthorizationPolicy?>(policy.Build());
        }

        return Task.FromResult<AuthorizationPolicy?>(null);
    }
}
```

ASP.NET Core only uses one authorization policy provider. If the custom implementation doesn't handle all policies, including default policies, etc., it should fall back to an alternate provider. In the preceding sample, a default authorization policy provider is:

- Constructed with options from the dependency injection container.
- Used if this custom provider isn't able to handle a given policy name.

If a custom policy provider is able to handle all expected policy names, setting the fallback policy with GetFallbackPolicyAsync() isn't required.

C#

```
class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";
    public DefaultAuthorizationPolicyProvider FallbackPolicyProvider { get;
}
    public MinimumAgePolicyProvider(IOptions<AuthorizationOptions> options)
    {
        FallbackPolicyProvider = new
DefaultAuthorizationPolicyProvider(options);
    }
    public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
                            FallbackPolicyProvider.GetDefaultPolicyAsync();
    public Task<AuthorizationPolicy?> GetFallbackPolicyAsync() =>
                            FallbackPolicyProvider.GetFallbackPolicyAsync();
```

Policies are looked up by string name, therefore parameters, for example, `age`, are embedded in the policy names. This is abstracted away from developers by the more strongly-typed attributes derived from AuthorizeAttribute. For example, the `[MinimumAgeAuthorize()]` attribute in this sample looks up policies by string name.

C#

```
public Task<AuthorizationPolicy?> GetPolicyAsync(string policyName)
{
    if (policyName.StartsWith(POLICY_PREFIX,
StringComparison.OrdinalIgnoreCase) &&
        int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var
age))
    {
        var policy = new AuthorizationPolicyBuilder(

JwtBearerDefaults.AuthenticationScheme);
        policy.AddRequirements(new MinimumAgeRequirement(age));
        return Task.FromResult<AuthorizationPolicy?>(policy.Build());
    }
```

```
    return Task.FromResult<AuthorizationPolicy?>(null);
}
```

The `MinimumAgeAuthorizeAttribute` uses the IAuthorizationRequirementData interface that allows the attribute definition to specify the requirements associated with the authorization policy:

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AuthRequirementsData.Authorization;

class MinimumAgeAuthorizeAttribute : AuthorizeAttribute,
IAuthorizationRequirement,
                                     IAuthorizationRequirementData
{
    public MinimumAgeAuthorizeAttribute(int age) => Age = age;
    public int Age { get; }

    public IEnumerable<IAuthorizationRequirement> GetRequirements()
    {
        yield return this;
    }
}
```

The `GreetingsController` displays the user's name when they satisfy the minimum age policy:

C#

```
using AuthRequirementsData.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace AuthRequirementsData.Controllers;

[ApiController]
[Route("api/[controller]")]
public class GreetingsController : Controller
{
    [MinimumAgeAuthorize(16)]
    [HttpGet("hello")]
    public string Hello() => $"Hello {(HttpContext.User.Identity?.Name ??
"world")}!";
}
```

The complete sample can be found in the AuthRequirementsData folder of the AspNetCore.Docs.Samples repository.

The sample can be tested with dotnet user-jwts and curl:

- `dotnet user-jwts create --claim http://schemas.xmlsoap.org/ws/2005/05/identity/claims/dateofbirth=1989-01-01`
- `curl -i -H "Authorization: Bearer <token from dotnet user-jwts>" http://localhost:<port>/api/greetings/hello`

Role-based authorization in ASP.NET Core

Article • 10/14/2024

When an identity is created it may belong to one or more roles. For example, Tracy may belong to the `Administrator` and `User` roles while Scott may only belong to the `User` role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the IsInRole method on the ClaimsPrincipal class. AddRoles must be added to Role services.

While roles are claims, not all claims are roles. Depending on the identity issuer a role may be a collection of users that may apply claims for group members, as well as an actual claim on an identity. However, claims are meant to be information about an individual user. Using roles to add claims to a user can confuse the boundary between the user and their individual claims. This confusion is why the SPA templates are not designed around roles. In addition, for organizations migrating from an on-premises legacy system the proliferation of roles over the years can mean a role claim may be too large to be contained within a token usable by SPAs. To secure SPAs, see Use Identity to secure a Web API backend for SPAs.

This article outlines role-based authorization for ASP.NET Core MVC and Razor apps. For Blazor apps, see ASP.NET Core Blazor authentication and authorization and ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles.

Add Role services to Identity

Register role-based authorization services in `Program.cs` by calling AddRoles with the role type in the app's Identity configuration. The role type in the following example is `IdentityRole`:

C#

```
builder.Services.AddDefaultIdentity<IdentityUser>( ... )
    .AddRoles<IdentityRole>()
    ...
```

The preceding code requires the Microsoft.AspNetCore.Identity.UI package and a `using` directive for `Microsoft.AspNetCore.Identity`.

Adding role checks

Role based authorization checks:

- Are declarative and specify roles which the current user must be a member of to access the requested resource.
- Are applied to Razor Pages, controllers, or actions within a controller.
- Can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

For example, the following code limits access to any actions on the `AdministrationController` to users who are a member of the `Administrator` role:

C#

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller
{
    public IActionResult Index() =>
        Content("Administrator");
}
```

Multiple roles can be specified as a comma separated list:

C#

```
[Authorize(Roles = "HRManager,Finance")]
public class SalaryController : Controller
{
    public IActionResult Payslip() =>
                    Content("HRManager || Finance");
}
```

The `SalaryController` is only accessible by users who are members of the `HRManager` role ***or*** the `Finance` role.

When multiple attributes are applied, an accessing user must be a member of ***all*** the roles specified. The following sample requires that a user must be a member of ***both*** the `PowerUser` ***and*** `ControlPanelUser` role:

C#

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
```

```
    public IActionResult Index() =>
        Content("PowerUser && ControlPanelUser");
}
```

Access to an action can be limited by applying additional role authorization attributes at the action level:

C#

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlAllPanelController : Controller
{
    public IActionResult SetTime() =>
        Content("Administrator || PowerUser");

    [Authorize(Roles = "Administrator")]
    public IActionResult ShutDown() =>
        Content("Administrator only");
}
```

In the preceding `ControlAllPanelController` controller:

- Members of the `Administrator` role or the `PowerUser` role can access the controller and the `ShutDown` action.
- Only members of the `Administrator` role can access the `SetTime` action.

A controller can be secured but allow anonymous, unauthenticated access to individual actions:

C#

```
[Authorize]
public class Control3PanelController : Controller
{
    public IActionResult SetTime() =>
        Content("[Authorize]");

    [AllowAnonymous]
    public IActionResult Login() =>
        Content("[AllowAnonymous]");
}
```

For Razor Pages, [Authorize] can be applied by either:

- Using a convention, or
- Applying the [Authorize] to the `PageModel` instance:

```
```

C#

```
[Authorize(Policy = "RequireAdministratorRole")]
public class UpdateModel : PageModel
{
    public IActionResult OnPost() =>
        Content("OnPost RequireAdministratorRole");
}
```

> **ⓘ Important**
>
> Filter attributes, including `AuthorizeAttribute`, can only be applied to PageModel and cannot be applied to specific page handler methods.

Policy based role checks

Role requirements can also be expressed using the Policy syntax, where a developer registers a policy at application startup as part of the Authorization service configuration. This typically occurs in the `Program.cs` file:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireAdministratorRole",
        policy => policy.RequireRole("Administrator"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapDefaultControllerRoute();
```

```
app.MapRazorPages();

app.Run();
```

Policies are applied using the Policy property on the [Authorize] attribute:

```
C#

[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown()
{
    return View();
}
```

To specify multiple allowed roles in a requirement, specify them as parameters to the RequireRole method:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("ElevatedRights", policy =>
          policy.RequireRole("Administrator", "PowerUser",
"BackupAdministrator"));
});

var app = builder.Build();
```

The preceding code authorizes users who belong to the `Administrator`, `PowerUser` or `BackupAdministrator` roles.

Claims-based authorization in ASP.NET Core

Article • 11/10/2023

When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name value pair that represents what the subject is, not what the subject can do. For example, you may have a driver's license, issued by a local driving license authority. Your driver's license has your date of birth on it. In this case the claim name would be `DateOfBirth`, the claim value would be your date of birth, for example `8th June 1970` and the issuer would be the driving license authority. Claims-based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value. For example if you want access to a night club the authorization process might be:

The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

Claim-based authorization checks:

- Are declarative.
- Are applied to Razor Pages, controllers, or actions within a controller.
- Can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

Claims in code specify claims which the current user must possess, and optionally the value the claim must hold to access the requested resource. Claims requirements are policy based; the developer must build and register a policy expressing the claims requirements.

The simplest type of claim policy looks for the presence of a claim and doesn't check the value.

Build and register the policy and call UseAuthorization. Registering the policy takes place as part of the Authorization service configuration, typically in the `Program.cs` file:

```
C#
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
   options.AddPolicy("EmployeeOnly", policy =>
policy.RequireClaim("EmployeeNumber"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.Run();
```

In this case the `EmployeeOnly` policy checks for the presence of an `EmployeeNumber` claim on the current identity.

Apply the policy using the `Policy` property on the [Authorize] attribute to specify the policy name.

C#

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    return View();
}
```

The `[Authorize]` attribute can be applied to an entire controller or Razor Page, in which case only identities matching the policy are allowed access to any Action on the controller.

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public ActionResult VacationBalance()
    {
        return View();
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
        return View();
    }
}
```

The following code applies the `[Authorize]` attribute to a Razor Page:

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class IndexModel : PageModel
{
   public void OnGet()
    {

    }
}
```

Policies can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

If you have a controller that's protected by the `[Authorize]` attribute but want to allow anonymous access to particular actions, you apply the `AllowAnonymousAttribute` attribute.

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
```

```
    public ActionResult VacationBalance()
    {
        return View();
    }

    [AllowAnonymous]
    public ActionResult VacationPolicy()
    {
        return View();
    }
}
```

Because policies can ***not*** be applied at the Razor Page handler level, we recommend using a controller when policies must be applied at the page handler level. The rest of the app that doesn't require policies at the Razor Page handler level can use Razor Pages.

Most claims come with a value. You can specify a list of allowed values when creating the policy. The following example would only succeed for employees whose employee number was 1, 2, 3, 4, or 5.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Founders", policy =>
                      policy.RequireClaim("EmployeeNumber", "1", "2", "3",
"4", "5"));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseAuthentication();
app.UseAuthorization();

app.MapDefaultControllerRoute();
```

```
app.MapRazorPages();

app.Run();
```

Add a generic claim check

If the claim value isn't a single value or a transformation is required, use RequireAssertion. For more information, see Use a func to fulfill a policy.

Multiple Policy Evaluation

If multiple policies are applied at the controller and action levels, ***all*** policies must pass before access is granted:

C#

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller
{
    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Payslip()
    {
        return View();
    }

    [Authorize(Policy = "HumanResources")]
    public IActionResult UpdateSalary()
    {
        return View();
    }
}
```

In the preceding example any identity which fulfills the `EmployeeOnly` policy can access the `Payslip` action as that policy is enforced on the controller. However, in order to call the `UpdateSalary` action the identity must fulfill *both* the `EmployeeOnly` policy and the `HumanResources` policy.

If you want more complicated policies, such as taking a date of birth claim, calculating an age from it, and then checking that the age is 21 or older, then you need to write custom policy handlers.

In the following sample, both page handler methods must fulfill *both* the `EmployeeOnly` policy and the `HumanResources` policy:

C#

```
[Authorize(Policy = "EmployeeOnly")]
[Authorize(Policy = "HumanResources")]
public class SalaryModel : PageModel
{
    public ContentResult OnGetPayStub()
    {
        return Content("OnGetPayStub");
    }

    public ContentResult OnGetSalary()
    {
        return Content("OnGetSalary");
    }
}
```


Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Policy-based authorization in ASP.NET Core

Article • 11/22/2023

Underneath the covers, role-based authorization and claims-based authorization use a requirement, a requirement handler, and a preconfigured policy. These building blocks support the expression of authorization evaluations in code. The result is a richer, reusable, testable authorization structure.

An authorization policy consists of one or more requirements. Register it as part of the authorization service configuration, in the app's `Program.cs` file:

C#

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("AtLeast21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});
```

In the preceding example, an "AtLeast21" policy is created. It has a single requirement—that of a minimum age, which is supplied as a parameter to the requirement.

IAuthorizationService

The primary service that determines if authorization is successful is IAuthorizationService:

C#

```
/// <summary>
/// Checks policy based permissions for a user
/// </summary>
public interface IAuthorizationService
{
    /// <summary>
    /// Checks if a user meets a specific set of requirements for the
specified resource
    /// </summary>
    /// <param name="user">The user to evaluate the requirements against.
</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass
null as the value
```

```
    /// </param>
    /// <param name="requirements">The requirements to evaluate.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// This value is <value>true</value> when the user fulfills the policy;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure
that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user, object
resource,
                                        IEnumerable<IAuthorizationRequirement>
requirements);

    /// <summary>
    /// Checks if a user meets a specific authorization policy
    /// </summary>
    /// <param name="user">The user to check the policy against.</param>
    /// <param name="resource">
    /// An optional resource the policy should be checked with.
    /// If a resource is not required for policy evaluation you may pass
null as the value
    /// </param>
    /// <param name="policyName">The name of the policy to check against a
specific
    /// context.</param>
    /// <returns>
    /// A flag indicating whether authorization has succeeded.
    /// Returns a flag indicating whether the user, and optional resource
has fulfilled
    /// the policy.
    /// <value>true</value> when the policy has been fulfilled;
    /// otherwise <value>false</value>.
    /// </returns>
    /// <remarks>
    /// Resource is an optional parameter and may be null. Please ensure
that you check
    /// it is not null before acting upon it.
    /// </remarks>
    Task<AuthorizationResult> AuthorizeAsync(
                                    ClaimsPrincipal user, object resource,
string policyName);
}
```

The preceding code highlights the two methods of the IAuthorizationService.

IAuthorizationRequirement is a marker service with no methods, and the mechanism for tracking whether authorization is successful.

Each IAuthorizationHandler is responsible for checking if requirements are met:

C#

```
/// <summary>
/// Classes implementing this interface are able to make a decision if
authorization
/// is allowed.
/// </summary>
public interface IAuthorizationHandler
{
    /// <summary>
    /// Makes a decision if authorization is allowed.
    /// </summary>
    /// <param name="context">The authorization information.</param>
    Task HandleAsync(AuthorizationHandlerContext context);
}
```

The AuthorizationHandlerContext class is what the handler uses to mark whether requirements have been met:

C#

```
 context.Succeed(requirement)
```

The following code shows the simplified (and annotated with comments) default implementation of the authorization service:

C#

```
public async Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
             object resource, IEnumerable<IAuthorizationRequirement>
requirements)
{
    // Create a tracking context from the authorization inputs.
    var authContext = _contextFactory.CreateContext(requirements, user,
resource);

    // By default this returns an IEnumerable<IAuthorizationHandler> from
DI.
    var handlers = await _handlers.GetHandlersAsync(authContext);

    // Invoke all handlers.
    foreach (var handler in handlers)
    {
        await handler.HandleAsync(authContext);
    }

    // Check the context, by default success is when all requirements have
been met.
```

```
    return _evaluator.Evaluate(authContext);
}
```

The following code shows a typical authorization service configuration:

C#

```
// Add all of your handlers to DI.
builder.Services.AddSingleton<IAuthorizationHandler, MyHandler1>();
// MyHandler2, ...

builder.Services.AddSingleton<IAuthorizationHandler, MyHandlerN>();

// Configure your policies
builder.Services.AddAuthorization(options =>
      options.AddPolicy("Something",
      policy => policy.RequireClaim("Permission", "CanViewPage",
"CanViewAnything")));
```

Use IAuthorizationService, `[Authorize(Policy = "Something")]`, or `RequireAuthorization("Something")` for authorization.

Apply policies to MVC controllers

For apps that use Razor Pages, see the Apply policies to Razor Pages section.

Apply policies to controllers by using the `[Authorize]` attribute with the policy name:

C#

```
[Authorize(Policy = "AtLeast21")]
public class AtLeast21Controller : Controller
{
    public IActionResult Index() => View();
}
```

If multiple policies are applied at the controller and action levels, ***all*** policies must pass before access is granted:

C#

```
[Authorize(Policy = "AtLeast21")]
public class AtLeast21Controller2 : Controller
{
    [Authorize(Policy = "IdentificationValidated")]
```

```
    public IActionResult Index() => View();
}
```

Apply policies to Razor Pages

Apply policies to Razor Pages by using the `[Authorize]` attribute with the policy name. For example:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace AuthorizationPoliciesSample.Pages;

[Authorize(Policy = "AtLeast21")]
public class AtLeast21Model : PageModel { }
```

Policies can ***not*** be applied at the Razor Page handler level, they must be applied to the Page.

Policies can also be applied to Razor Pages by using an authorization convention.

Apply policies to endpoints

Apply policies to endpoints by using RequireAuthorization with the policy name. For example:

C#

```
app.MapGet("/helloworld", () => "Hello World!")
    .RequireAuthorization("AtLeast21");
```

Requirements

An authorization requirement is a collection of data parameters that a policy can use to evaluate the current user principal. In our "AtLeast21" policy, the requirement is a single parameter—the minimum age. A requirement implements IAuthorizationRequirement, which is an empty marker interface. A parameterized minimum age requirement could be implemented as follows:

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Requirements;

public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement(int minimumAge) =>
        MinimumAge = minimumAge;

    public int MinimumAge { get; }
}
```

If an authorization policy contains multiple authorization requirements, all requirements must pass in order for the policy evaluation to succeed. In other words, multiple authorization requirements added to a single authorization policy are treated on an **AND** basis.

> ⓘ **Note**
>
> A requirement doesn't need to have data or properties.

Authorization handlers

An authorization handler is responsible for the evaluation of a requirement's properties. The authorization handler evaluates the requirements against a provided AuthorizationHandlerContext to determine if access is allowed.

A requirement can have multiple handlers. A handler may inherit AuthorizationHandler<TRequirement>, where `TRequirement` is the requirement to be handled. Alternatively, a handler may implement IAuthorizationHandler directly to handle more than one type of requirement.

Use a handler for one requirement

The following example shows a one-to-one relationship in which a minimum age handler handles a single requirement:

C#

```
using System.Security.Claims;
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;
```

```
namespace AuthorizationPoliciesSample.Policies.Handlers;

public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, MinimumAgeRequirement
requirement)
    {
        var dateOfBirthClaim = context.User.FindFirst(
            c => c.Type == ClaimTypes.DateOfBirth && c.Issuer ==
"http://contoso.com");

        if (dateOfBirthClaim is null)
        {
            return Task.CompletedTask;
        }

        var dateOfBirth = Convert.ToDateTime(dateOfBirthClaim.Value);
        int calculatedAge = DateTime.Today.Year - dateOfBirth.Year;
        if (dateOfBirth > DateTime.Today.AddYears(-calculatedAge))
        {
            calculatedAge--;
        }

        if (calculatedAge >= requirement.MinimumAge)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

The preceding code determines if the current user principal has a date of birth claim that has been issued by a known and trusted Issuer. Authorization can't occur when the claim is missing, in which case a completed task is returned. When a claim is present, the user's age is calculated. If the user meets the minimum age defined by the requirement, authorization is considered successful. When authorization is successful, `context.Succeed` is invoked with the satisfied requirement as its sole parameter.

Use a handler for multiple requirements

The following example shows a one-to-many relationship in which a permission handler can handle three different types of requirements:

C#

```
using System.Security.Claims;
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;
```

```
namespace AuthorizationPoliciesSample.Policies.Handlers;

public class PermissionHandler : IAuthorizationHandler
{
    public Task HandleAsync(AuthorizationHandlerContext context)
    {
        var pendingRequirements = context.PendingRequirements.ToList();

        foreach (var requirement in pendingRequirements)
        {
            if (requirement is ReadPermission)
            {
                if (IsOwner(context.User, context.Resource)
                    || IsSponsor(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
            else if (requirement is EditPermission || requirement is
DeletePermission)
            {
                if (IsOwner(context.User, context.Resource))
                {
                    context.Succeed(requirement);
                }
            }
        }

        return Task.CompletedTask;
    }

    private static bool IsOwner(ClaimsPrincipal user, object? resource)
    {
        // Code omitted for brevity
        return true;
    }

    private static bool IsSponsor(ClaimsPrincipal user, object? resource)
    {
        // Code omitted for brevity
        return true;
    }
}
```

The preceding code traverses PendingRequirements—a property containing requirements not marked as successful. For a `ReadPermission` requirement, the user must be either an owner or a sponsor to access the requested resource. For an `EditPermission` or `DeletePermission` requirement, they must be an owner to access the requested resource.

Handler registration

Register handlers in the services collection during configuration. For example:

```
C#

builder.Services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
```

The preceding code registers `MinimumAgeHandler` as a singleton. Handlers can be registered using any of the built-in service lifetimes.

It's possible to bundle both a requirement and a handler into a single class implementing both IAuthorizationRequirement and IAuthorizationHandler. This bundling creates a tight coupling between the handler and requirement and is only recommended for simple requirements and handlers. Creating a class that implements both interfaces removes the need to register the handler in DI because of the built-in PassThroughAuthorizationHandler that allows requirements to handle themselves.

See the AssertionRequirement class for a good example where the `AssertionRequirement` is both a requirement and the handler in a fully self-contained class.

What should a handler return?

Note that the `Handle` method in the handler example returns no value. How is a status of either success or failure indicated?

- A handler indicates success by calling `context.Succeed(IAuthorizationRequirement requirement)`, passing the requirement that has been successfully validated.
- A handler doesn't need to handle failures generally, as other handlers for the same requirement may succeed.
- To guarantee failure, even if other requirement handlers succeed, call `context.Fail`.

If a handler calls `context.Succeed` or `context.Fail`, all other handlers are still called. This allows requirements to produce side effects, such as logging, which takes place even if another handler has successfully validated or failed a requirement. When set to `false`, the InvokeHandlersAfterFailure property short-circuits the execution of handlers when `context.Fail` is called. `InvokeHandlersAfterFailure` defaults to `true`, in which case all handlers are called.

> ⓘ Note
>
> Authorization handlers are called even if authentication fails. Also handlers can execute in any order, so do ***not*** depend on them being called in any particular order.

Why would I want multiple handlers for a requirement?

In cases where you want evaluation to be on an **OR** basis, implement multiple handlers for a single requirement. For example, Microsoft has doors that only open with key cards. If you leave your key card at home, the receptionist prints a temporary sticker and opens the door for you. In this scenario, you'd have a single requirement, *BuildingEntry*, but multiple handlers, each one examining a single requirement.

`BuildingEntryRequirement.cs`

C#

```
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Requirements;

public class BuildingEntryRequirement : IAuthorizationRequirement { }
```

`BadgeEntryHandler.cs`

C#

```
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Handlers;

public class BadgeEntryHandler :
AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, BuildingEntryRequirement
requirement)
    {
        if (context.User.HasClaim(
            c => c.Type == "BadgeId" && c.Issuer ==
"https://microsoftsecurity"))
        {
```

```
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

`TemporaryStickerHandler.cs`

C#

```
using AuthorizationPoliciesSample.Policies.Requirements;
using Microsoft.AspNetCore.Authorization;

namespace AuthorizationPoliciesSample.Policies.Handlers;

public class TemporaryStickerHandler :
AuthorizationHandler<BuildingEntryRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, BuildingEntryRequirement
requirement)
    {
        if (context.User.HasClaim(
            c => c.Type == "TemporaryBadgeId" && c.Issuer ==
"https://microsoftsecurity"))
        {
            // Code to check expiration date omitted for brevity.
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

Ensure that both handlers are registered. If either handler succeeds when a policy evaluates the `BuildingEntryRequirement`, the policy evaluation succeeds.

Use a func to fulfill a policy

There may be situations in which fulfilling a policy is simple to express in code. It's possible to supply a `Func<AuthorizationHandlerContext, bool>` when configuring a policy with the `RequireAssertion` policy builder.

For example, the previous `BadgeEntryHandler` could be rewritten as follows:

C#

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("BadgeEntry", policy =>
        policy.RequireAssertion(context => context.User.HasClaim(c =>
            (c.Type == "BadgeId" || c.Type == "TemporaryBadgeId")
            && c.Issuer == "https://microsoftsecurity")));
});
```

Access MVC request context in handlers

The `HandleRequirementAsync` method has two parameters: an `AuthorizationHandlerContext` and the `TRequirement` being handled. Frameworks such as MVC or SignalR are free to add any object to the `Resource` property on the `AuthorizationHandlerContext` to pass extra information.

When using endpoint routing, authorization is typically handled by the Authorization Middleware. In this case, the `Resource` property is an instance of HttpContext. The context can be used to access the current endpoint, which can be used to probe the underlying resource to which you're routing. For example:

C#

```
if (context.Resource is HttpContext httpContext)
{
    var endpoint = httpContext.GetEndpoint();
    var actionDescriptor =
endpoint.Metadata.GetMetadata<ControllerActionDescriptor>();
    ...
}
```

With traditional routing, or when authorization happens as part of MVC's authorization filter, the value of `Resource` is an AuthorizationFilterContext instance. This property provides access to `HttpContext`, `RouteData`, and everything else provided by MVC and Razor Pages.

The use of the `Resource` property is framework-specific. Using information in the `Resource` property limits your authorization policies to particular frameworks. Cast the `Resource` property using the `is` keyword, and then confirm the cast has succeeded to ensure your code doesn't crash with an `InvalidCastException` when run on other frameworks:

C#

```
// Requires the following import:
//     using Microsoft.AspNetCore.Mvc.Filters;
if (context.Resource is AuthorizationFilterContext mvcContext)
{
    // Examine MVC-specific things like routing data.
}
```

Globally require all users to be authenticated

For information on how to globally require all users to be authenticated, see Require authenticated users.

Authorization with external service sample

The sample code on AspNetCore.Docs.Samples shows how to implement additional authorization requirements with an external authorization service. The sample `Contoso.API` project is secured with Azure AD. An additional authorization check from the `Contoso.Security.API` project returns a payload describing whether the `Contoso.API` client app can invoke the `GetWeather` API.

Configure the sample

- Create an application registration in your Microsoft Entra ID tenant:
- Assign it an AppRole.
- Under API permissions, add the AppRole as a permission and grant Admin consent. Note that in this setup, this app registration represents both the API and the client invoking the API. If you like, you can create two app registrations. If you are using this setup, be sure to only perform the API permissions, add AppRole as a permission step for only the client. Only the client app registration requires a client secret to be generated.
- Configure the `Contoso.API` project with the following settings:

C#

```
{
  "AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "Domain": "<Tenant name from AAD properties>.onmicrosoft.com">,
    "TenantId": "<Tenant Id from AAD properties>",
    "ClientId": "<Client Id from App Registration representing the API>"
```

```
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

- Configure `Contoso.Security.API` with the following settings:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "AllowedClients": [
    "<Use the appropriate Client Id representing the Client calling the
API>"
  ]
}
```

- Open the ContosoAPI.collection.json file and configure an environment with the following:
  - `ClientId`: Client Id from app registration representing the client calling the API.
  - `clientSecret`: Client Secret from app registration representing the client calling the API.
  - `TenantId`: Tenant Id from AAD properties

- Extract the commands from the `ContosoAPI.collection.json` file and use them to construct cURL commands to test the app.

- Run the solution and use cURL to invoke the API. You can add breakpoints in the `Contoso.Security.API.SecurityPolicyController` and observe the client Id is being passed in that is used to assert whether it is allowed to Get Weather.

Additional resources

- Quickstart: Configure an application to expose a web API
- AspNetCore.Docs.Samples code

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Custom Authorization Policy Providers using IAuthorizationPolicyProvider in ASP.NET Core

Article • 06/03/2022

By Mike Rousos

Typically when using policy-based authorization, policies are registered by calling `AuthorizationOptions.AddPolicy` as part of authorization service configuration. In some scenarios, it may not be possible (or desirable) to register all authorization policies in this way. In those cases, you can use a custom IAuthorizationPolicyProvider to control how authorization policies are supplied.

Examples of scenarios where a custom IAuthorizationPolicyProvider may be useful include:

- Using an external service to provide policy evaluation.
- Using a large range of policies (for different room numbers or ages, for example), so it doesn't make sense to add each individual authorization policy with an `AuthorizationOptions.AddPolicy` call.
- Creating policies at runtime based on information in an external data source (like a database) or determining authorization requirements dynamically through another mechanism.

View or download sample code from the AspNetCore GitHub repository. Download the dotnet/AspNetCore repository ZIP file. Unzip the file. Navigate to the *src/Security/samples/CustomPolicyProvider* project folder.

Customize policy retrieval

ASP.NET Core apps use an implementation of the `IAuthorizationPolicyProvider` interface to retrieve authorization policies. By default, DefaultAuthorizationPolicyProvider is registered and used. `DefaultAuthorizationPolicyProvider` returns policies from the `AuthorizationOptions` provided in an `IServiceCollection.AddAuthorization` call.

Customize this behavior by registering a different `IAuthorizationPolicyProvider` implementation in the app's dependency injection container.

The `IAuthorizationPolicyProvider` interface contains three APIs:

- GetPolicyAsync returns an authorization policy for a given name.
- GetDefaultPolicyAsync returns the default authorization policy (the policy used for `[Authorize]` attributes without a policy specified).
- GetFallbackPolicyAsync returns the fallback authorization policy (the policy used by the Authorization Middleware when no policy is specified).

By implementing these APIs, you can customize how authorization policies are provided.

Parameterized authorize attribute example

One scenario where `IAuthorizationPolicyProvider` is useful is enabling custom `[Authorize]` attributes whose requirements depend on a parameter. For example, in policy-based authorization documentation, an age-based ("AtLeast21") policy was used as a sample. If different controller actions in an app should be made available to users of *different* ages, it might be useful to have many different age-based policies. Instead of registering all the different age-based policies that the application will need in `AuthorizationOptions`, you can generate the policies dynamically with a custom `IAuthorizationPolicyProvider`. To make using the policies easier, you can annotate actions with custom authorization attribute like `[MinimumAgeAuthorize(20)]`.

Custom Authorization attributes

Authorization policies are identified by their names. The custom `MinimumAgeAuthorizeAttribute` described previously needs to map arguments into a string that can be used to retrieve the corresponding authorization policy. You can do this by deriving from `AuthorizeAttribute` and making the `Age` property wrap the `AuthorizeAttribute.Policy` property.

C#

```
internal class MinimumAgeAuthorizeAttribute : AuthorizeAttribute
{
    const string POLICY_PREFIX = "MinimumAge";

    public MinimumAgeAuthorizeAttribute(int age) => Age = age;

    // Get or set the Age property by manipulating the underlying Policy
property
    public int Age
    {
        get
```

```
        {
            if (int.TryParse(Policy.Substring(POLICY_PREFIX.Length), out var
age))
            {
                return age;
            }
            return default(int);
        }
        set
        {
            Policy = $"{POLICY_PREFIX}{value.ToString()}";
        }
    }
}
```

This attribute type has a `Policy` string based on the hard-coded prefix (`"MinimumAge"`) and an integer passed in via the constructor.

You can apply it to actions in the same way as other `Authorize` attributes except that it takes an integer as a parameter.

C#

```
[MinimumAgeAuthorize(10)]
public IActionResult RequiresMinimumAge10()
```

Custom IAuthorizationPolicyProvider

The custom `MinimumAgeAuthorizeAttribute` makes it easy to request authorization policies for any minimum age desired. The next problem to solve is making sure that authorization policies are available for all of those different ages. This is where an `IAuthorizationPolicyProvider` is useful.

When using `MinimumAgeAuthorizationAttribute`, the authorization policy names will follow the pattern `"MinimumAge" + Age`, so the custom `IAuthorizationPolicyProvider` should generate authorization policies by:

- Parsing the age from the policy name.
- Using `AuthorizationPolicyBuilder` to create a new `AuthorizationPolicy`
- In this and following examples it will be assumed that the user is authenticated via a cookie. The `AuthorizationPolicyBuilder` should either be constructed with at least one authorization scheme name or always succeed. Otherwise there is no information on how to provide a challenge to the user and an exception will be thrown.

- Adding requirements to the policy based on the age with `AuthorizationPolicyBuilder.AddRequirements`. In other scenarios, you might use `RequireClaim`, `RequireRole`, or `RequireUserName` instead.

C#

```
internal class MinimumAgePolicyProvider : IAuthorizationPolicyProvider
{
    const string POLICY_PREFIX = "MinimumAge";

    // Policies are looked up by string name, so expect 'parameters' (like
age)
    // to be embedded in the policy names. This is abstracted away from
developers
    // by the more strongly-typed attributes derived from AuthorizeAttribute
    // (like [MinimumAgeAuthorize()] in this sample)
    public Task<AuthorizationPolicy> GetPolicyAsync(string policyName)
    {
        if (policyName.StartsWith(POLICY_PREFIX,
StringComparison.OrdinalIgnoreCase) &&
            int.TryParse(policyName.Substring(POLICY_PREFIX.Length), out var
age))
        {
            var policy = new
AuthorizationPolicyBuilder(CookieAuthenticationDefaults.AuthenticationScheme
);
            policy.AddRequirements(new MinimumAgeRequirement(age));
            return Task.FromResult(policy.Build());
        }

        return Task.FromResult<AuthorizationPolicy>(null);
    }
}
```

Multiple authorization policy providers

When using custom `IAuthorizationPolicyProvider` implementations, keep in mind that ASP.NET Core only uses one instance of `IAuthorizationPolicyProvider`. If a custom provider isn't able to provide authorization policies for all policy names that will be used, it should defer to a backup provider.

For example, consider an application that needs both custom age policies and more traditional role-based policy retrieval. Such an app could use a custom authorization policy provider that:

- Attempts to parse policy names.

- Calls into a different policy provider (like `DefaultAuthorizationPolicyProvider`) if the policy name doesn't contain an age.

The example `IAuthorizationPolicyProvider` implementation shown above can be updated to use the `DefaultAuthorizationPolicyProvider` by creating a backup policy provider in its constructor (to be used in case the policy name doesn't match its expected pattern of 'MinimumAge' + age).

```
C#

private DefaultAuthorizationPolicyProvider BackupPolicyProvider { get; }

public MinimumAgePolicyProvider(IOptions<AuthorizationOptions> options)
{
    // ASP.NET Core only uses one authorization policy provider, so if the
custom implementation
    // doesn't handle all policies it should fall back to an alternate
provider.
    BackupPolicyProvider = new DefaultAuthorizationPolicyProvider(options);
}
```

Then, the `GetPolicyAsync` method can be updated to use the `BackupPolicyProvider` instead of returning null:

```
C#

...
return BackupPolicyProvider.GetPolicyAsync(policyName);
```

Default policy

In addition to providing named authorization policies, a custom `IAuthorizationPolicyProvider` needs to implement `GetDefaultPolicyAsync` to provide an authorization policy for `[Authorize]` attributes without a policy name specified.

In many cases, this authorization attribute only requires an authenticated user, so you can make the necessary policy with a call to `RequireAuthenticatedUser`:

```
C#

public Task<AuthorizationPolicy> GetDefaultPolicyAsync() =>
    Task.FromResult(new
AuthorizationPolicyBuilder(CookieAuthenticationDefaults.AuthenticationScheme
).RequireAuthenticatedUser().Build());
```

As with all aspects of a custom `IAuthorizationPolicyProvider`, you can customize this, as needed. In some cases, it may be desirable to retrieve the default policy from a fallback `IAuthorizationPolicyProvider`.

Fallback policy

A custom `IAuthorizationPolicyProvider` can optionally implement `GetFallbackPolicyAsync` to provide a policy that's used when combining policies and when no policies are specified. If `GetFallbackPolicyAsync` returns a non-null policy, the returned policy is used by the Authorization Middleware when no policies are specified for the request.

If no fallback policy is required, the provider can return `null` or defer to the fallback provider:

```
C#

public Task<AuthorizationPolicy> GetFallbackPolicyAsync() =>
    Task.FromResult<AuthorizationPolicy>(null);
```

Use a custom IAuthorizationPolicyProvider

To use custom policies from an `IAuthorizationPolicyProvider`, you ***must***:

- Register the appropriate `AuthorizationHandler` types with dependency injection (described in policy-based authorization), as with all policy-based authorization scenarios.
- Register the custom `IAuthorizationPolicyProvider` type in the app's dependency injection service collection in `Startup.ConfigureServices` to replace the default policy provider.

  ```
  C#

  services.AddSingleton<IAuthorizationPolicyProvider,
  MinimumAgePolicyProvider>();
  ```

A complete custom `IAuthorizationPolicyProvider` sample is available in the dotnet/aspnetcore GitHub repository.

Customize the behavior of `AuthorizationMiddleware`

Article • 06/03/2022

Apps can register an IAuthorizationMiddlewareResultHandler to customize how AuthorizationMiddleware handles authorization results. Apps can use `IAuthorizationMiddlewareResultHandler` to:

- Return customized responses.
- Enhance the default challenge or forbid responses.

The following code shows an example implementation of `IAuthorizationMiddlewareResultHandler` that returns a custom response for specific authorization failures:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authorization.Policy;

public class SampleAuthorizationMiddlewareResultHandler :
IAuthorizationMiddlewareResultHandler
{
    private readonly AuthorizationMiddlewareResultHandler defaultHandler =
new();

    public async Task HandleAsync(
        RequestDelegate next,
        HttpContext context,
        AuthorizationPolicy policy,
        PolicyAuthorizationResult authorizeResult)
    {
        // If the authorization was forbidden and the resource had a
specific requirement,
        // provide a custom 404 response.
        if (authorizeResult.Forbidden
            && authorizeResult.AuthorizationFailure!.FailedRequirements
                .OfType<Show404Requirement>().Any())
        {
            // Return a 404 to make it appear as if the resource doesn't
exist.
            context.Response.StatusCode = StatusCodes.Status404NotFound;
            return;
        }

        // Fall back to the default implementation.
        await defaultHandler.HandleAsync(next, context, policy,
authorizeResult);
```

```
    }
}

public class Show404Requirement : IAuthorizationRequirement { }
```

Register this implementation of `IAuthorizationMiddlewareResultHandler` in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<
    IAuthorizationMiddlewareResultHandler,
SampleAuthorizationMiddlewareResultHandler>();

var app = builder.Build();
```

Dependency injection in requirement handlers in ASP.NET Core

Article • 06/03/2022

Authorization handlers must be registered in the service collection during configuration using dependency injection.

Suppose you had a repository of rules you wanted to evaluate inside an authorization handler and that repository was registered in the service collection. Authorization resolves and injects that into the constructor.

For example, to use the .NET logging infrastructure, inject ILoggerFactory into the handler, as shown in the following example:

C#

```
public class SampleAuthorizationHandler :
AuthorizationHandler<SampleRequirement>
{
    private readonly ILogger _logger;

    public SampleAuthorizationHandler(ILoggerFactory loggerFactory)
        => _logger = loggerFactory.CreateLogger(GetType().FullName);

    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context, SampleRequirement requirement)
    {
        _logger.LogInformation("Inside my handler");

        // ...

        return Task.CompletedTask;
    }
}
```

The preceding handler can be registered with any service lifetime. The following code uses AddSingleton to register the preceding handler:

C#

```
builder.Services.AddSingleton<IAuthorizationHandler,
SampleAuthorizationHandler>();
```

An instance of the handler is created when the app starts, and DI injects the registered `ILoggerFactory` into its constructor.

> **Note**
>
> Don't register authorization handlers that use Entity Framework (EF) as singletons.

Resource-based authorization in ASP.NET Core

Article • 06/03/2022

Authorization approach depends on the resource. For example, only the author of a document is authorized to update the document. Consequently, the document must be retrieved from the data store before authorization evaluation can occur.

Attribute evaluation occurs before data binding and before execution of the page handler or action that loads the document. For these reasons, declarative authorization with an `[Authorize]` attribute doesn't suffice. Instead, you can invoke a custom authorization method—a style known as *imperative authorization*.

View or download sample code (how to download).

Create an ASP.NET Core app with user data protected by authorization contains a sample app that uses resource-based authorization.

Use imperative authorization

Authorization is implemented as an IAuthorizationService service and is registered in the service collection at application startup. The service is made available via dependency injection to page handlers or actions.

C#

```
public class DocumentController : Controller
{
    private readonly IAuthorizationService _authorizationService;
    private readonly IDocumentRepository _documentRepository;

    public DocumentController(IAuthorizationService authorizationService,
                              IDocumentRepository documentRepository)
    {
        _authorizationService = authorizationService;
        _documentRepository = documentRepository;
    }
```

`IAuthorizationService` has two `AuthorizeAsync` method overloads: one accepting the resource and the policy name and the other accepting the resource and a list of requirements to evaluate.

C#

```
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          IEnumerable<IAuthorizationRequirement>
requirements);
Task<AuthorizationResult> AuthorizeAsync(ClaimsPrincipal user,
                          object resource,
                          string policyName);
```

In the following example, the resource to be secured is loaded into a custom `Document` object. An `AuthorizeAsync` overload is invoked to determine whether the current user is allowed to edit the provided document. A custom "EditPolicy" authorization policy is factored into the decision. See Custom policy-based authorization for more on creating authorization policies.

> **Note**
>
> The following code samples assume authentication has run and set the `User` property.

C#

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, "EditPolicy");

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
        return new ForbidResult();
    }
    else
    {
        return new ChallengeResult();
    }
}
```

Write a resource-based handler

Writing a handler for resource-based authorization isn't much different than writing a plain requirements handler. Create a custom requirement class, and implement a requirement handler class. For more information on creating a requirement class, see Requirements.

The handler class specifies both the requirement and resource type. For example, a handler utilizing a `SameAuthorRequirement` and a `Document` resource follows:

C#

```
public class DocumentAuthorizationHandler :
    AuthorizationHandler<SameAuthorRequirement, Document>
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,
                                                    SameAuthorRequirement
requirement,
                                                    Document resource)
    {
        if (context.User.Identity?.Name == resource.Author)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

public class SameAuthorRequirement : IAuthorizationRequirement { }
```

In the preceding example, imagine that `SameAuthorRequirement` is a special case of a more generic `SpecificAuthorRequirement` class. The `SpecificAuthorRequirement` class (not shown) contains a `Name` property representing the name of the author. The `Name` property could be set to the current user.

Register the requirement and handler in `Program.cs`:

C#

```
builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();
```

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("EditPolicy", policy =>
        policy.Requirements.Add(new SameAuthorRequirement()));
});

builder.Services.AddSingleton<IAuthorizationHandler,
DocumentAuthorizationHandler>();
builder.Services.AddSingleton<IAuthorizationHandler,
DocumentAuthorizationCrudHandler>();
builder.Services.AddScoped<IDocumentRepository, DocumentRepository>();
```

Operational requirements

If you're making decisions based on the outcomes of CRUD (Create, Read, Update, Delete) operations, use the OperationAuthorizationRequirement helper class. This class enables you to write a single handler instead of an individual class for each operation type. To use it, provide some operation names:

C#

```
public static class Operations
{
    public static OperationAuthorizationRequirement Create =
        new OperationAuthorizationRequirement { Name = nameof(Create) };
    public static OperationAuthorizationRequirement Read =
        new OperationAuthorizationRequirement { Name = nameof(Read) };
    public static OperationAuthorizationRequirement Update =
        new OperationAuthorizationRequirement { Name = nameof(Update) };
    public static OperationAuthorizationRequirement Delete =
        new OperationAuthorizationRequirement { Name = nameof(Delete) };
}
```

The handler is implemented as follows, using an `OperationAuthorizationRequirement` requirement and a `Document` resource:

C#

```
public class DocumentAuthorizationCrudHandler :
    AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task
HandleRequirementAsync(AuthorizationHandlerContext context,

OperationAuthorizationRequirement requirement,
                                                    Document resource)
    {
        if (context.User.Identity?.Name == resource.Author &&
            requirement.Name == Operations.Read.Name)
```

```
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}
```

The preceding handler validates the operation using the resource, the user's identity, and the requirement's `Name` property.

Challenge and forbid with an operational resource handler

This section shows how the challenge and forbid action results are processed and how challenge and forbid differ.

To call an operational resource handler, specify the operation when invoking `AuthorizeAsync` in your page handler or action. The following example determines whether the authenticated user is permitted to view the provided document.

> ⓘ Note
>
> The following code samples assume authentication has run and set the `User` property.

C#

```
public async Task<IActionResult> OnGetAsync(Guid documentId)
{
    Document = _documentRepository.Find(documentId);

    if (Document == null)
    {
        return new NotFoundResult();
    }

    var authorizationResult = await _authorizationService
            .AuthorizeAsync(User, Document, Operations.Read);

    if (authorizationResult.Succeeded)
    {
        return Page();
    }
    else if (User.Identity.IsAuthenticated)
    {
```

```
            return new ForbidResult();
        }
        else
        {
            return new ChallengeResult();
        }
    }
```

If authorization succeeds, the page for viewing the document is returned. If authorization fails but the user is authenticated, returning `ForbidResult` informs any authentication middleware that authorization failed. A `ChallengeResult` is returned when authentication must be performed. For interactive browser clients, it may be appropriate to redirect the user to a login page.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

View-based authorization in ASP.NET Core MVC

Article • 09/27/2024

A developer often wants to show, hide, or otherwise modify a UI based on the current user identity. You can access the authorization service within MVC views via dependency injection. To inject the authorization service into a Razor view, use the `@inject` directive:

CSHTML

```
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService
```

If you want the authorization service in every view, place the `@inject` directive into the `_ViewImports.cshtml` file of the `Views` directory. For more information, see Dependency injection into views.

Use the injected authorization service to invoke `AuthorizeAsync` in exactly the same way you would check during resource-based authorization:

CSHTML

```
@if ((await AuthorizationService.AuthorizeAsync(User,
"PolicyName")).Succeeded)
{
    <p>This paragraph is displayed because you fulfilled PolicyName.</p>
}
```

In some cases, the resource will be your view model. Invoke `AuthorizeAsync` in exactly the same way you would check during resource-based authorization:

CSHTML

```
@if ((await AuthorizationService.AuthorizeAsync(User, Model,
Operations.Edit)).Succeeded)
{
    <p><a class="btn btn-default" role="button"
        href="@Url.Action("Edit", "Document", new { id = Model.Id
})">Edit</a></p>
}
```

In the preceding code, the model is passed as a resource the policy evaluation should take into consideration.

> **⚠ Warning**
>
> Don't rely on toggling visibility of your app's UI elements as the sole authorization check. Hiding a UI element may not completely prevent access to its associated controller action. For example, consider the button in the preceding code snippet. A user can invoke the `Edit` action method if they know the relative resource URL is `/Document/Edit/1`. For this reason, the `Edit` action method should perform its own authorization check.

Authorize with a specific scheme in ASP.NET Core

Article • 06/03/2022

For an introduction to authentication schemes in ASP.NET Core, see Authentication scheme.

In some scenarios, such as Single Page Applications (SPAs), it's common to use multiple authentication methods. For example, the app may use cookie-based authentication to log in and JWT bearer authentication for JavaScript requests. In some cases, the app may have multiple instances of an authentication handler. For example, two cookie handlers where one contains a basic identity and one is created when a multi-factor authentication (MFA) has been triggered. MFA may be triggered because the user requested an operation that requires extra security. For more information on enforcing MFA when a user requests a resource that requires MFA, see the GitHub issue Protect section with MFA.

An authentication scheme is named when the authentication service is configured during authentication. For example:

C#

```
using Microsoft.AspNetCore.Authentication;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication()
        .AddCookie(options =>
        {
            options.LoginPath = "/Account/Unauthorized/";
            options.AccessDeniedPath = "/Account/Forbidden/";
        })
        .AddJwtBearer(options =>
        {
            options.Audience = "http://localhost:5001/";
            options.Authority = "http://localhost:5000/";
        });

builder.Services.AddAuthentication()
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();
```

```
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

In the preceding code, two authentication handlers have been added: one for cookies and one for bearer.

> **Note**
>
> Specifying the default scheme results in the `HttpContext.User` property being set to that identity. If that behavior isn't desired, disable it by invoking the parameterless form of `AddAuthentication`.

Selecting the scheme with the Authorize attribute

At the point of authorization, the app indicates the handler to be used. Select the handler with which the app will authorize by passing a comma-delimited list of authentication schemes to `[Authorize]`. The [Authorize] attribute specifies the authentication scheme or schemes to use regardless of whether a default is configured. For example:

```
C#
```

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Mvc;

namespace AuthScheme.Controllers;

[Authorize(AuthenticationSchemes = AuthSchemes)]
public class MixedController : Controller
{
    private const string AuthSchemes =
        CookieAuthenticationDefaults.AuthenticationScheme + "," +
        JwtBearerDefaults.AuthenticationScheme;
    public ContentResult Index() => Content(MyWidgets.GetMyContent());

}
```

In the preceding example, both the cookie and bearer handlers run and have a chance to create and append an identity for the current user. By specifying a single scheme only, the corresponding handler runs:

C#

```
[Authorize(AuthenticationSchemes=JwtBearerDefaults.AuthenticationScheme)]
public class Mixed2Controller : Controller
{
    public ContentResult Index() => Content(MyWidgets.GetMyContent());
}
```

In the preceding code, only the handler with the "Bearer" scheme runs. Any cookie-based identities are ignored.

Selecting the scheme with policies

If you prefer to specify the desired schemes in policy, you can set the AuthenticationSchemes collection when adding a policy:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.JwtBearer;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy =>
    {

policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new MinimumAgeRequirement(18));
    });
});
builder.Services.AddAuthentication()
                .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

In the preceding example, the "Over18" policy only runs against the identity created by the "Bearer" handler. Use the policy by setting the `[Authorize]` attribute's `Policy` property:

C#

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace AuthScheme.Controllers;
```

```
[Authorize(Policy = "Over18")]
public class RegistrationController : Controller
{
    // Do Registration
```

Use multiple authentication schemes

Some apps may need to support multiple types of authentication. For example, your app might authenticate users from Azure Active Directory and from a users database. Another example is an app that authenticates users from both Active Directory Federation Services and Azure Active Directory B2C. In this case, the app should accept a JWT bearer token from several issuers.

Add all authentication schemes you'd like to accept. For example, the following code adds two JWT bearer authentication schemes with different issuers:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

// Authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://localhost:5000/identity/";
        })
        .AddJwtBearer("AzureAD", options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://login.microsoftonline.com/eb971100-
7f436/";
        });

// Authorization
builder.Services.AddAuthorization(options =>
{
    var defaultAuthorizationPolicyBuilder = new AuthorizationPolicyBuilder(
        JwtBearerDefaults.AuthenticationScheme,
        "AzureAD");
    defaultAuthorizationPolicyBuilder =
        defaultAuthorizationPolicyBuilder.RequireAuthenticatedUser();
    options.DefaultPolicy = defaultAuthorizationPolicyBuilder.Build();
});

builder.Services.AddAuthentication()
```

```
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

> **Note**
>
> Only one JWT bearer authentication is registered with the default authentication scheme `JwtBearerDefaults.AuthenticationScheme`. Additional authentication has to be registered with a unique authentication scheme.

Update the default authorization policy to accept both authentication schemes. For example:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;

var builder = WebApplication.CreateBuilder(args);

// Authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
```

```
        .AddJwtBearer(options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://localhost:5000/identity/";
        })
        .AddJwtBearer("AzureAD", options =>
        {
            options.Audience = "https://localhost:5000/";
            options.Authority = "https://login.microsoftonline.com/eb971100-
7f436/";
        });

// Authorization
builder.Services.AddAuthorization(options =>
{
    var defaultAuthorizationPolicyBuilder = new AuthorizationPolicyBuilder(
        JwtBearerDefaults.AuthenticationScheme,
        "AzureAD");
    defaultAuthorizationPolicyBuilder =
        defaultAuthorizationPolicyBuilder.RequireAuthenticatedUser();
    options.DefaultPolicy = defaultAuthorizationPolicyBuilder.Build();
});

builder.Services.AddAuthentication()
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute();
app.MapRazorPages();

app.MapFallbackToFile("index.html");

app.Run();
```

As the default authorization policy is overridden, it's possible to use the `[Authorize]` attribute in controllers. The controller then accepts requests with JWT issued by the first or second issuer.

See this GitHub issue on using multiple authentication schemes.

The following example uses Azure Active Directory B2C and another Azure Active Directory tenant:

C#

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.IdentityModel.Tokens;
using Microsoft.Net.Http.Headers;
using System.IdentityModel.Tokens.Jwt;

var builder = WebApplication.CreateBuilder(args);

// Authentication
```

```
builder.Services.AddAuthentication(options =>
{
    options.DefaultScheme = "B2C_OR_AAD";
    options.DefaultChallengeScheme = "B2C_OR_AAD";
})
.AddJwtBearer("B2C", jwtOptions =>
{
    jwtOptions.MetadataAddress = "B2C-MetadataAddress";
    jwtOptions.Authority = "B2C-Authority";
    jwtOptions.Audience = "B2C-Audience";
})
.AddJwtBearer("AAD", jwtOptions =>
{
    jwtOptions.MetadataAddress = "AAD-MetadataAddress";
    jwtOptions.Authority = "AAD-Authority";
    jwtOptions.Audience = "AAD-Audience";
    jwtOptions.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateIssuerSigningKey = true,
        ValidAudiences =
builder.Configuration.GetSection("ValidAudiences").Get<string[]>(),
        ValidIssuers =
builder.Configuration.GetSection("ValidIssuers").Get<string[]>()
    };
})
.AddPolicyScheme("B2C_OR_AAD", "B2C_OR_AAD", options =>
{
    options.ForwardDefaultSelector = context =>
    {
        string authorization =
context.Request.Headers[HeaderNames.Authorization];
        if (!string.IsNullOrEmpty(authorization) &&
authorization.StartsWith("Bearer "))
        {
            var token = authorization.Substring("Bearer ".Length).Trim();
            var jwtHandler = new JwtSecurityTokenHandler();

            return (jwtHandler.CanReadToken(token) &&
jwtHandler.ReadJwtToken(token).Issuer.Equals("B2C-Authority"))
                ? "B2C" : "AAD";
        }
        return "AAD";
    };
});
```

```
builder.Services.AddAuthentication()
        .AddIdentityServerJwt();

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

var app = builder.Build();
```

```
if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthentication();
app.UseIdentityServer();
app.UseAuthorization();

app.MapDefaultControllerRoute().RequireAuthorization();
app.MapRazorPages().RequireAuthorization();

app.MapFallbackToFile("index.html");

app.Run();
```

In the preceding code, ForwardDefaultSelector is used to select a default scheme for the current request that authentication handlers should forward all authentication operations to by default. The default forwarding logic checks the most specific ForwardAuthenticate, ForwardChallenge, ForwardForbid, ForwardSignIn, and ForwardSignOut setting first, followed by checking the ForwardDefaultSelector, followed by ForwardDefault. The first non null result is used as the target scheme to forward to. For more information, see Policy schemes in ASP.NET Core.

ASP.NET Core Data Protection Overview

Article • 05/09/2024

ASP.NET Core provides a cryptographic API to protect data, including key management and rotation.

Web apps often need to store sensitive data. The Windows data protection API (DPAPI) isn't intended for use in web apps.

The ASP.NET Core data protection stack was designed to:

- Provide a built in solution for most Web scenarios.
- Address many of the deficiencies of the previous encryption system.
- Serve as the replacement for the `<machineKey>` element in ASP.NET 1.x - 4.x.

Problem statement

I need to persist trusted information for later retrieval, but I don't trust the persistence mechanism. In web terms, this might be written as *I need to round-trip trusted state via an untrusted client.*

Authenticity, integrity, and tamper-proofing is a requirement. The canonical example of this is an authentication cookie or bearer token. The server generates an ***I am Groot and have xyz permissions*** token and sends it to the client. The client presents that token back to the server, but the server needs some kind of assurance that the client hasn't forged the token.

Confidentiality is a requirement. Since the persisted state is trusted by the server, this state could contain information that shouldn't be disclosed to an untrusted client. For example:

- A file path.
- A permission.
- A handle or other indirect reference.
- Some server-specific data.

Isolation is a requirement. Since modern apps are componentized, individual components want to take advantage of this system without regard to other components in the system. For instance, consider a bearer token component using this stack. It should operate without any interference, for example, from an anti-CSRF mechanism also using the same stack.

Some common assumptions can narrow the scope of requirements:

- All services operating within the cryptosystem are equally trusted.
- The data doesn't need to be generated or consumed outside of the services under our direct control.
- Operations must be fast since each request to the web service might go through the cryptosystem one or more times. The speed requirement makes symmetric cryptography ideal. Asymmetric cryptography isn't used until it's required.

Design philosophy

ASP.NET Core data protection is an easy to use data protection stack. It's based on the following principles:

- Ease of configuration. The system strives for zero configuration. In situations where developers need to configure a specific aspect, such as the key repository, those specific configurations aren't difficult.
- Offer a basic consumer-facing API. The APIs are straight forward to use correctly and difficult to use incorrectly.
- Developers don't have to learn key management principles. The system handles algorithm selection and key lifetime on behalf of the developer. The developer doesn't have access to the raw key material.
- Keys are protected at rest as much as possible. The system figures out an appropriate default protection mechanism and applies it automatically.

The data protection APIs aren't primarily intended for indefinite persistence of confidential payloads. Other technologies, such as Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage. They have correspondingly strong key management capabilities. That said, the ASP.NET Core data protection APIs can be used for long-term protection of confidential data.

Audience

The data protection system provides APIs that target three main audiences:

1. The consumer APIs target application and framework developers.

   I don't want to learn about how the stack operates or about how it's configured. I just want to perform some operation with high probability of using the APIs successfully.

2. The configuration APIs target app developers and system administrators.

I need to tell the data protection system that my environment requires non-default paths or settings.

3. The extensibility APIs target developers in charge of implementing custom policy. Usage of these APIs is limited to rare situations and developers with security experience.

   I need to replace an entire component within the system because I have truly unique behavioral requirements. I'm willing to learn uncommonly used parts of the API surface in order to build a plugin that fulfills my requirements.

Package layout

The data protection stack consists of five packages:

- Microsoft.AspNetCore.DataProtection.Abstractions contains:
  - IDataProtectionProvider and IDataProtector interfaces to create data protection services.
  - Useful extension methods for working with these types. for example, IDataProtector.Protect

  If the data protection system is instantiated elsewhere and you're consuming the API, reference `Microsoft.AspNetCore.DataProtection.Abstractions`.

- Microsoft.AspNetCore.DataProtection contains the core implementation of the data protection system, including:
  - Core cryptographic operations.
  - Key management.
  - Configuration and extensibility.

  To instantiate the data protection system, reference `Microsoft.AspNetCore.DataProtection`. You might need to reference the data protection system when:
  - Adding it to an IServiceCollection.
  - Modifying or extending its behavior.

- Microsoft.AspNetCore.DataProtection.Extensions contains additional APIs which developers might find useful but which don't belong in the core package. For instance, this package contains:
  - Factory methods to instantiate the data protection system to store keys at a location on the file system without dependency injection. See DataProtectionProvider.

    - Extension methods for limiting the lifetime of protected payloads. See ITimeLimitedDataProtector.

- Microsoft.AspNetCore.DataProtection.SystemWeb can be installed into an existing ASP.NET 4.x app to redirect its `<machineKey>` operations to use the new ASP.NET Core data protection stack. For more information, see Replace the ASP.NET machineKey in ASP.NET Core.

- Microsoft.AspNetCore.Cryptography.KeyDerivation provides an implementation of the PBKDF2 password hashing routine and can be used by systems that must handle user passwords securely. For more information, see Hash passwords in ASP.NET Core.

Additional resources

- Get started with the Data Protection APIs in ASP.NET Core
- Host ASP.NET Core in a web farm

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Get started with the Data Protection APIs in ASP.NET Core

Article • 06/03/2022

Basically, protecting data consists of the following steps:

1. Create a data protector from a data protection provider.
2. Call the `Protect` method with the data you want to protect.
3. Call the `Unprotect` method with the data you want to turn back into plain text.

Most frameworks and app models, such as ASP.NET Core or SignalR, already configure the data protection system and add it to a service container that is accessed via dependency injection. The following sample demonstrates:

- Configuring a service container for dependency injection and registering the data protection stack.
- Receiving the data protection provider via DI.
- Creating a protector.
- Protecting then unprotecting data.

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
```

```
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload =
_protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

When you create a protector you must provide one or more Purpose Strings. A purpose string provides isolation between consumers. For example, a protector created with a purpose string of "green" wouldn't be able to unprotect data provided by a protector with a purpose of "purple".

> **Tip**
>
> Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It's intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`.
>
> A call to `Unprotect` will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Use AddOptions to configure custom repository

Consider the following code which uses a service provider because the implementation of `IXmlRepository` has a dependency on a singleton service:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    var sp = services.BuildServiceProvider();
    services.AddDataProtection()
      .AddKeyManagementOptions(o => o.XmlRepository =
sp.GetService<IXmlRepository>());
}
```

The preceding code logs the following warning:

Calling 'BuildServiceProvider' from application code results in an additional copy of singleton services being created. Consider alternatives such as dependency injecting services as parameters to 'Configure'.

The following code provides the `IXmlRepository` implementation without having to build the service provider and therefore making additional copies of singleton services:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<DataProtectionDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    // Register XmlRepository for data protection.
    services.AddOptions<KeyManagementOptions>()
    .Configure<IServiceScopeFactory>((options, factory) =>
    {
        options.XmlRepository = new CustomXmlRepository(factory);
    });

    services.AddRazorPages();
}
```

The preceding code removes the call to `GetService` and hides `IConfigureOptions<T>`.

The following code shows the custom XML repository:

C#

```
using CustomXMLrepo.Data;
using Microsoft.AspNetCore.DataProtection.Repositories;
using Microsoft.Extensions.DependencyInjection;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

public class CustomXmlRepository : IXmlRepository
{
    private readonly IServiceScopeFactory factory;

    public CustomXmlRepository(IServiceScopeFactory factory)
    {
        this.factory = factory;
    }

    public IReadOnlyCollection<XElement> GetAllElements()
    {
        using (var scope = factory.CreateScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<DataProtectionDbContext>();
            var keys = context.XmlKeys.ToList()
                .Select(x => XElement.Parse(x.Xml))
                .ToList();
            return keys;
        }
    }

    public void StoreElement(XElement element, string friendlyName)
    {
        var key = new XmlKey
        {
            Xml = element.ToString(SaveOptions.DisableFormatting)
        };

        using (var scope = factory.CreateScope())
        {
            var context =
scope.ServiceProvider.GetRequiredService<DataProtectionDbContext>();
            context.XmlKeys.Add(key);
            context.SaveChanges();
        }
    }
}
```

The following code shows the XmlKey class:

C#

```
public class XmlKey
{
    public Guid Id { get; set; }
    public string Xml { get; set; }

    public XmlKey()
    {
        this.Id = Guid.NewGuid();
    }
}
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Consumer APIs overview for ASP.NET Core

Article • 04/27/2023

The `IDataProtectionProvider` and `IDataProtector` interfaces are the basic interfaces through which consumers use the data protection system. They're located in the Microsoft.AspNetCore.DataProtection.Abstractions package.

IDataProtectionProvider

The provider interface represents the root of the data protection system. It cannot directly be used to protect or unprotect data. Instead, the consumer must get a reference to an `IDataProtector` by calling `IDataProtectionProvider.CreateProtector(purpose)`, where purpose is a string that describes the intended consumer use case. See Purpose Strings for much more information on the intent of this parameter and how to choose an appropriate value.

IDataProtector

The protector interface is returned by a call to `CreateProtector`, and it's this interface which consumers can use to perform protect and unprotect operations.

To protect a piece of data, pass the data to the `Protect` method. The basic interface defines a method which converts byte[] -> byte[], but there's also an overload (provided as an extension method) which converts string -> string. The security offered by the two methods is identical; the developer should choose whichever overload is most convenient for their use case. Irrespective of the overload chosen, the value returned by the Protect method is now protected (enciphered and tamper-proofed), and the application can send it to an untrusted client.

To unprotect a previously-protected piece of data, pass the protected data to the `Unprotect` method. (There are byte[]-based and string-based overloads for developer convenience.) If the protected payload was generated by an earlier call to `Protect` on this same `IDataProtector`, the `Unprotect` method will return the original unprotected payload. If the protected payload has been tampered with or was produced by a different `IDataProtector`, the `Unprotect` method will throw CryptographicException.

The concept of same vs. different `IDataProtector` ties back to the concept of purpose. If two `IDataProtector` instances were generated from the same root `IDataProtectionProvider` but via different purpose strings in the call to `IDataProtectionProvider.CreateProtector`, then they're considered different protectors, and one won't be able to unprotect payloads generated by the other.

Consuming these interfaces

For a DI-aware component, the intended usage is that the component takes an `IDataProtectionProvider` parameter in its constructor and that the DI system automatically provides this service when the component is instantiated.

> **Note**
>
> Some applications (such as console applications or ASP.NET 4.x applications) might not be DI-aware so cannot use the mechanism described here. For these scenarios consult the Non DI Aware Scenarios document for more information on getting an instance of an `IDataProtection` provider without going through DI.

The following sample demonstrates three concepts:

1. Add the data protection system to the service container,
2. Using DI to receive an instance of an `IDataProtectionProvider`, and
3. Creating an `IDataProtector` from an `IDataProtectionProvider` and using it to protect and unprotect data.

Console app

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
```

```
        var services = serviceCollection.BuildServiceProvider();

        // create an instance of MyClass using the service provider
        var instance = ActivatorUtilities.CreateInstance<MyClass>(services);
        instance.RunSample();
    }

    public class MyClass
    {
        IDataProtector _protector;

        // the 'provider' parameter is provided by DI
        public MyClass(IDataProtectionProvider provider)
        {
            _protector = provider.CreateProtector("Contoso.MyClass.v1");
        }

        public void RunSample()
        {
            Console.Write("Enter input: ");
            string input = Console.ReadLine();

            // protect the payload
            string protectedPayload = _protector.Protect(input);
            Console.WriteLine($"Protect returned: {protectedPayload}");

            // unprotect the payload
            string unprotectedPayload =
_protector.Unprotect(protectedPayload);
            Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8ICcgQwZZhlAlTZT...OdfH66i1PnGmpCR5e441xQ
 * Unprotect returned: Hello world!
 */
```

Web app

Call AddDataProtection(IServiceCollection, Action<DataProtectionOptions>) in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);
```

```
// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddDataProtection();

var app = builder.Build();
```

The following highlighted code shows how to use IDataProtector in a controller:

C#

```
public class HomeController : Controller
{
    private readonly IDataProtector _dataProtector;

    public HomeController(IDataProtectionProvider dataProtectionProvider)
    {
        _dataProtector =
dataProtectionProvider.CreateProtector("HomeControllerPurpose");
    }

    // ...

    public IActionResult Privacy()
    {
        // The original data to protect
        string originalData = "original data";

        // Protect the data (encrypt)
        string protectedData = _dataProtector.Protect(originalData);
        Console.WriteLine($"Protected Data: {protectedData}");

        // Unprotect the data (decrypt)
        string unprotectedData = _dataProtector.Unprotect(protectedData);
        Console.WriteLine($"Unprotected Data: {unprotectedData}");

        return View();
    }

    // ...
```

The package `Microsoft.AspNetCore.DataProtection.Abstractions` contains an extension method GetDataProtector as a developer convenience. It encapsulates as a single operation both retrieving an IDataProtectionProvider from the service provider and calling IDataProtectionProvider.CreateProtector. The following sample demonstrates its usage:

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;
```

```
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        // add data protection services
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection();
        var services = serviceCollection.BuildServiceProvider();

        // get an IDataProtector from the IServiceProvider
        var protector = services.GetDataProtector("Contoso.Example.v2");
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
    }
}
```

Tip

Instances of `IDataProtectionProvider` and `IDataProtector` are thread-safe for multiple callers. It's intended that once a component gets a reference to an `IDataProtector` via a call to `CreateProtector`, it will use that reference for multiple calls to `Protect` and `Unprotect`. A call to `Unprotect` will throw CryptographicException if the protected payload cannot be verified or deciphered. Some components may wish to ignore errors during unprotect operations; a component which reads authentication cookies might handle this error and treat the request as if it had no cookie at all rather than fail the request outright. Components which want this behavior should specifically catch CryptographicException instead of swallowing all exceptions.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Purpose strings in ASP.NET Core

Article • 09/07/2022

Components which consume `IDataProtectionProvider` must pass a unique *purposes* parameter to the `CreateProtector` method. The purposes *parameter* is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root cryptographic keys are the same.

When a consumer specifies a purpose, the purpose string is used along with the root cryptographic keys to derive cryptographic subkeys unique to that consumer. This isolates the consumer from all other cryptographic consumers in the application: no other component can read its payloads, and it cannot read any other component's payloads. This isolation also renders infeasible entire categories of attack against the component.



In the diagram above, `IDataProtector` instances A and B **cannot** read each other's payloads, only their own.

The purpose string doesn't have to be secret. It should simply be unique in the sense that no other well-behaved component will ever provide the same purpose string.

> **Tip**
>
> Using the namespace and type name of the component consuming the data protection APIs is a good rule of thumb, as in practice this information will never conflict.
>
> A Contoso-authored component which is responsible for minting bearer tokens might use Contoso.Security.BearerToken as its purpose string. Or - even better - it might use Contoso.Security.BearerToken.v1 as its purpose string. Appending the version number allows a future version to use Contoso.Security.BearerToken.v2 as

its purpose, and the different versions would be completely isolated from one another as far as payloads go.

Since the purposes parameter to `CreateProtector` is a string array, the above could've been instead specified as `[ "Contoso.Security.BearerToken", "v1" ]`. This allows establishing a hierarchy of purposes and opens up the possibility of multi-tenancy scenarios with the data protection system.

> ⚠ **Warning**
>
> Components shouldn't allow untrusted user input to be the sole source of input for the purposes chain.
>
> For example, consider a component Contoso.Messaging.SecureMessage which is responsible for storing secure messages. If the secure messaging component were to call `CreateProtector([ username ])`, then a malicious user might create an account with username "Contoso.Security.BearerToken" in an attempt to get the component to call `CreateProtector([ "Contoso.Security.BearerToken" ])`, thus inadvertently causing the secure messaging system to mint payloads that could be perceived as authentication tokens.
>
> A better purposes chain for the messaging component would be `CreateProtector([ "Contoso.Messaging.SecureMessage", $"User: {username}" ])`, which provides proper isolation.

The isolation provided by and behaviors of `IDataProtectionProvider`, `IDataProtector`, and purposes are as follows:

- For a given `IDataProtectionProvider` object, the `CreateProtector` method will create an `IDataProtector` object uniquely tied to both the `IDataProtectionProvider` object which created it and the purposes parameter which was passed into the method.
- The purpose parameter must not be null. (If purposes is specified as an array, this means that the array must not be of zero length and all elements of the array must be non-null.) An empty string purpose is technically allowed but is discouraged.
- Two purposes arguments are equivalent if and only if they contain the same strings (using an ordinal comparer) in the same order. A single purpose argument is equivalent to the corresponding single-element purposes array.

- Two `IDataProtector` objects are equivalent if and only if they're created from equivalent `IDataProtectionProvider` objects with equivalent purposes parameters.
- For a given `IDataProtector` object, a call to `Unprotect(protectedData)` will return the original `unprotectedData` if and only if `protectedData := Protect(unprotectedData)` for an equivalent `IDataProtector` object.

> **Note**
>
> We're not considering the case where some component intentionally chooses a purpose string which is known to conflict with another component. Such a component would essentially be considered malicious, and this system isn't intended to provide security guarantees in the event that malicious code is already running inside of the worker process.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

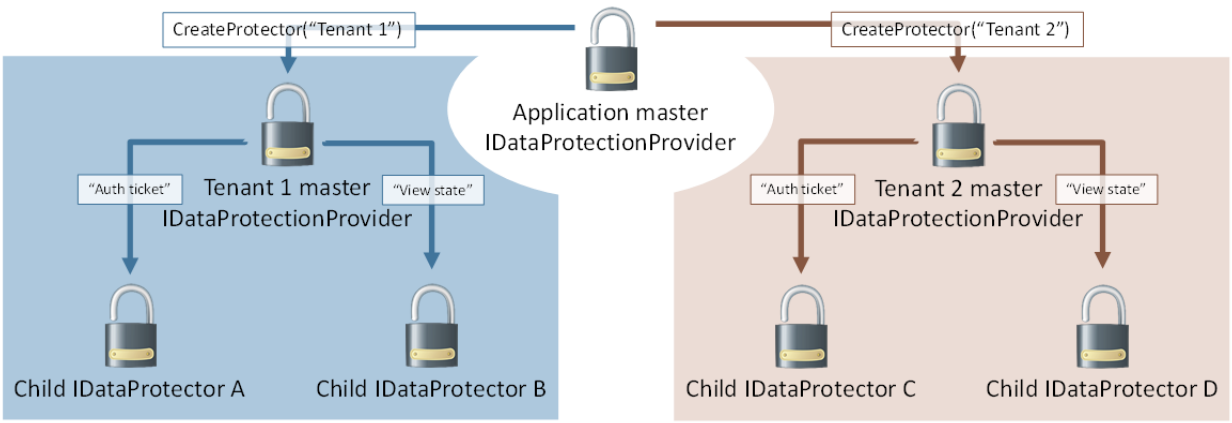
Purpose hierarchy and multi-tenancy in ASP.NET Core

Article • 06/03/2022

Since an `IDataProtector` is also implicitly an `IDataProtectionProvider`, purposes can be chained together. In this sense, `provider.CreateProtector([ "purpose1", "purpose2" ])` is equivalent to `provider.CreateProtector("purpose1").CreateProtector("purpose2")`.

This allows for some interesting hierarchical relationships through the data protection system. In the earlier example of Contoso.Messaging.SecureMessage, the SecureMessage component can call `provider.CreateProtector("Contoso.Messaging.SecureMessage")` once up-front and cache the result into a private `_myProvider` field. Future protectors can then be created via calls to `_myProvider.CreateProtector("User: username")`, and these protectors would be used for securing the individual messages.

This can also be flipped. Consider a single logical application which hosts multiple tenants (a CMS seems reasonable), and each tenant can be configured with its own authentication and state management system. The umbrella application has a single master provider, and it calls `provider.CreateProtector("Tenant 1")` and `provider.CreateProtector("Tenant 2")` to give each tenant its own isolated slice of the data protection system. The tenants could then derive their own individual protectors based on their own needs, but no matter how hard they try they cannot create protectors which collide with any other tenant in the system. Graphically, this is represented as below.



> ⚠ **Warning**
>
> This assumes the umbrella application controls which APIs are available to individual tenants and that tenants cannot execute arbitrary code on the server. If a

tenant can execute arbitrary code, they could perform private reflection to break the isolation guarantees, or they could just read the master keying material directly and derive whatever subkeys they desire.

The data protection system actually uses a sort of multi-tenancy in its default out-of-the-box configuration. By default master keying material is stored in the worker process account's user profile folder (or the registry, for IIS application pool identities). But it's actually fairly common to use a single account to run multiple applications, and thus all these applications would end up sharing the master keying material. To solve this, the data protection system automatically inserts a unique-per-application identifier as the first element in the overall purpose chain. This implicit purpose serves to isolate individual applications from one another by effectively treating each application as a unique tenant within the system, and the protector creation process looks identical to the image above.

Hash passwords in ASP.NET Core

Article • 07/19/2022

This article shows how to call the KeyDerivation.Pbkdf2 method which allows hashing a password using the PBKDF2 algorithm.

> **⚠ Warning**
>
> The `KeyDerivation.Pbkdf2` API is a low-level cryptographic primitive and is intended to be used to integrate apps into an existing protocol or cryptographic system. `KeyDerivation.Pbkdf2` should not be used in new apps which support password based login and need to store hashed passwords in a datastore. New apps should use **PasswordHasher**. For more information on `PasswordHasher`, see **Exploring the ASP.NET Core Identity PasswordHasher**.

The data protection code base includes a NuGet package Microsoft.AspNetCore.Cryptography.KeyDerivation which contains cryptographic key derivation functions. This package is a standalone component and has no dependencies on the rest of the data protection system. It can be used independently. The source exists alongside the data protection code base as a convenience.

> **⚠ Warning**
>
> The following code shows how to use `KeyDerivation.Pbkdf2` to generate a shared secret key. It should not be used to hash a password for storage in a datastore.

```
using Microsoft.AspNetCore.Cryptography.KeyDerivation;
using System.Security.Cryptography;

Console.Write("Enter a password: ");
string? password = Console.ReadLine();

// Generate a 128-bit salt using a sequence of
// cryptographically strong random bytes.
byte[] salt = RandomNumberGenerator.GetBytes(128 / 8); // divide by 8 to
convert bits to bytes
Console.WriteLine($"Salt: {Convert.ToBase64String(salt)}");

// derive a 256-bit subkey (use HMACSHA256 with 100,000 iterations)
string hashed = Convert.ToBase64String(KeyDerivation.Pbkdf2(
    password: password!,
    salt: salt,
```

```
    prf: KeyDerivationPrf.HMACSHA256,
    iterationCount: 100000,
    numBytesRequested: 256 / 8));

Console.WriteLine($"Hashed: {hashed}");

/*
 * SAMPLE OUTPUT
 *
 * Enter a password: Xtw9NMgx
 * Salt: CGYzqeN4plZekNC88Umm1Q==
 * Hashed: Gt9Yc4AiIvmsC1QQbe2RZsCIqvoYlst2xbz0Fs8aHnw=
 */
```

See the source code for ASP.NET Core Identity's `PasswordHasher` type for a real-world use case.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see **How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205)**.

Limit the lifetime of protected payloads in ASP.NET Core

Article • 06/03/2022

There are scenarios where the application developer wants to create a protected payload that expires after a set period of time. For instance, the protected payload might represent a password reset token that should only be valid for one hour. It's certainly possible for the developer to create their own payload format that contains an embedded expiration date, and advanced developers may wish to do this anyway, but for the majority of developers managing these expirations can grow tedious.

To make this easier for our developer audience, the package Microsoft.AspNetCore.DataProtection.Extensions contains utility APIs for creating payloads that automatically expire after a set period of time. These APIs implement the `ITimeLimitedDataProtector` interface.

API usage

The `ITimeLimitedDataProtector` interface is the core interface for protecting and unprotecting time-limited / self-expiring payloads. To create an instance of an `ITimeLimitedDataProtector`, you'll first need an instance of a regular IDataProtector constructed with a specific purpose. Once the `IDataProtector` instance is available, call the `IDataProtector.ToTimeLimitedDataProtector` extension method to get back a protector with built-in expiration capabilities.

`ITimeLimitedDataProtector` exposes the following API surface and extension methods:

- CreateProtector(string purpose) : ITimeLimitedDataProtector - This API is similar to the existing `IDataProtectionProvider.CreateProtector` in that it can be used to create purpose chains from a root time-limited protector.
- Protect(byte[] plaintext, DateTimeOffset expiration) : byte[]
- Protect(byte[] plaintext, TimeSpan lifetime) : byte[]
- Protect(byte[] plaintext) : byte[]
- Protect(string plaintext, DateTimeOffset expiration) : string
- Protect(string plaintext, TimeSpan lifetime) : string

- Protect(string plaintext) : string

In addition to the core `Protect` methods which take only the plaintext, there are new overloads which allow specifying the payload's expiration date. The expiration date can be specified as an absolute date (via a `DateTimeOffset`) or as a relative time (from the current system time, via a `TimeSpan`). If an overload which doesn't take an expiration is called, the payload is assumed never to expire.

- Unprotect(byte[] protectedData, out DateTimeOffset expiration) : byte[]
- Unprotect(byte[] protectedData) : byte[]
- Unprotect(string protectedData, out DateTimeOffset expiration) : string
- Unprotect(string protectedData) : string

The `Unprotect` methods return the original unprotected data. If the payload hasn't yet expired, the absolute expiration is returned as an optional out parameter along with the original unprotected data. If the payload is expired, all overloads of the Unprotect method will throw CryptographicException.

> ⚠ **Warning**
>
> It's not advised to use these APIs to protect payloads which require long-term or indefinite persistence. "Can I afford for the protected payloads to be permanently unrecoverable after a month?" can serve as a good rule of thumb; if the answer is no then developers should consider alternative APIs.

The sample below uses the non-DI code paths for instantiating the data protection system. To run this sample, ensure that you have first added a reference to the Microsoft.AspNetCore.DataProtection.Extensions package.

C#

```
using System;
using System.IO;
using System.Threading;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // create a protector for my application

        var provider = DataProtectionProvider.Create(new
```

```
DirectoryInfo(@"c:\myapp-keys\"));
        var baseProtector =
provider.CreateProtector("Contoso.TimeLimitedSample");

        // convert the normal protector into a time-limited protector
        var timeLimitedProtector =
baseProtector.ToTimeLimitedDataProtector();

        // get some input and protect it for five seconds
        Console.Write("Enter input: ");
        string input = Console.ReadLine();
        string protectedData = timeLimitedProtector.Protect(input, lifetime:
TimeSpan.FromSeconds(5));
        Console.WriteLine($"Protected data: {protectedData}");

        // unprotect it to demonstrate that round-tripping works properly
        string roundtripped = timeLimitedProtector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped data: {roundtripped}");

        // wait 6 seconds and perform another unprotect, demonstrating that
the payload self-expires
        Console.WriteLine("Waiting 6 seconds...");
        Thread.Sleep(6000);
        timeLimitedProtector.Unprotect(protectedData);
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello!
 * Protected data: CfDJ8Hu5z0zwxn...nLk7Ok
 * Round-tripped data: Hello!
 * Waiting 6 seconds...
 * <<throws CryptographicException with message 'The payload expired at
...'>>

 */
```

Unprotect payloads whose keys have been revoked in ASP.NET Core

Article • 09/27/2024

The ASP.NET Core data protection APIs are not primarily intended for indefinite persistence of confidential payloads. Other technologies like Windows CNG DPAPI and Azure Rights Management are more suited to the scenario of indefinite storage, and they have correspondingly strong key management capabilities. That said, there's nothing prohibiting a developer from using the ASP.NET Core data protection APIs for long-term protection of confidential data. Keys are never removed from the key ring, so `IDataProtector.Unprotect` can always recover existing payloads as long as the keys are available and valid.

However, an issue arises when the developer tries to unprotect data that has been protected with a revoked key, as `IDataProtector.Unprotect` will throw an exception in this case. This might be fine for short-lived or transient payloads (like authentication tokens), as these kinds of payloads can easily be recreated by the system, and at worst the site visitor might be required to log in again. But for persisted payloads, having `Unprotect` throw could lead to unacceptable data loss.

IPersistedDataProtector

To support the scenario of allowing payloads to be unprotected even in the face of revoked keys, the data protection system contains an `IPersistedDataProtector` type. To get an instance of `IPersistedDataProtector`, simply get an instance of `IDataProtector` in the normal fashion and try casting the `IDataProtector` to `IPersistedDataProtector`.

> **Note**
>
> Not all `IDataProtector` instances can be cast to `IPersistedDataProtector`. Developers should use the C# as operator or similar to avoid runtime exceptions caused by invalid casts, and they should be prepared to handle the failure case appropriately.

`IPersistedDataProtector` exposes the following API surface:

```
C#
```

```
DangerousUnprotect(byte[] protectedData, bool ignoreRevocationErrors,
     out bool requiresMigration, out bool wasRevoked) : byte[]
```

This API takes the protected payload (as a byte array) and returns the unprotected payload. There's no string-based overload. The two out parameters are as follows.

- `requiresMigration`: Is set to `true` if the key used to protect this payload is no longer the active default key. For example, the key used to protect this payload is old and a key rolling operation has since taken place. The caller may wish to consider reprotecting the payload depending on their business needs.
- `wasRevoked`: Is set to `true` if the key used to protect this payload was revoked.

> ⚠ **Warning**
>
> Exercise extreme caution when passing `ignoreRevocationErrors: true` to the `DangerousUnprotect` method. If after calling this method the `wasRevoked` value is true, then the key used to protect this payload was revoked, and the payload's authenticity should be treated as suspect. In this case, only continue operating on the unprotected payload if you have some separate assurance that it's authentic, for example that it's coming from a secure database rather than being sent by an untrusted web client.

C#

```
using System;
using System.IO;
using System.Text;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // get a protector and perform a protect operation
        var protector =
services.GetDataProtector("Sample.DangerousUnprotect");
```

```
        Console.Write("Input: ");
        byte[] input = Encoding.UTF8.GetBytes(Console.ReadLine());
        var protectedData = protector.Protect(input);
        Console.WriteLine($"Protected payload:
{Convert.ToBase64String(protectedData)}");

        // demonstrate that the payload round-trips properly
        var roundTripped = protector.Unprotect(protectedData);
        Console.WriteLine($"Round-tripped payload:
{Encoding.UTF8.GetString(roundTripped)}");

        // get a reference to the key manager and revoke all keys in the key
ring
        var keyManager = services.GetService<IKeyManager>();
        Console.WriteLine("Revoking all keys in the key ring...");
        keyManager.RevokeAllKeys(DateTimeOffset.Now, "Sample revocation.");

        // try calling Protect - this should throw
        Console.WriteLine("Calling Unprotect...");
        try
        {
            var unprotectedPayload = protector.Unprotect(protectedData);
            Console.WriteLine($"Unprotected payload:
{Encoding.UTF8.GetString(unprotectedPayload)}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
        }

        // try calling DangerousUnprotect
        Console.WriteLine("Calling DangerousUnprotect...");
        try
        {
            IPersistedDataProtector persistedProtector = protector as
IPersistedDataProtector;
            if (persistedProtector == null)
            {
                throw new Exception("Can't call DangerousUnprotect.");
            }

            bool requiresMigration, wasRevoked;
            var unprotectedPayload = persistedProtector.DangerousUnprotect(
                protectedData: protectedData,
                ignoreRevocationErrors: true,
                requiresMigration: out requiresMigration,
                wasRevoked: out wasRevoked);
            Console.WriteLine($"Unprotected payload:
{Encoding.UTF8.GetString(unprotectedPayload)}");
            Console.WriteLine($"Requires migration = {requiresMigration},
was revoked = {wasRevoked}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
```

```
        }
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Input: Hello!
 * Protected payload: CfDJ8LHIzUCX1ZVBn2BZ...
 * Round-tripped payload: Hello!
 * Revoking all keys in the key ring...
 * Calling Unprotect...
 * CryptographicException: The key {...} has been revoked.
 * Calling DangerousUnprotect...
 * Unprotected payload: Hello!
 * Requires migration = True, was revoked = True
 */
```

Data Protection configuration in ASP.NET Core

Article • 06/03/2022

Visit these topics to learn about Data Protection configuration in ASP.NET Core:

- Configure ASP.NET Core Data Protection
  An overview on configuring ASP.NET Core Data Protection.
- Data Protection key management and lifetime
  Information on Data Protection key management and lifetime.
- Data Protection machine-wide policy support
  Details on setting a default machine-wide policy for all apps that use Data Protection.
- Non-DI aware scenarios for Data Protection in ASP.NET Core
  How to use the DataProtectionProvider concrete type to use Data Protection without going through DI-specific code paths.

Configure ASP.NET Core Data Protection

Article • 10/30/2024

When the Data Protection system is initialized, it applies default settings based on the operational environment. These settings are appropriate for apps running on a single machine. However, there are cases where a developer may want to change the default settings:

- The app is spread across multiple machines.
- For compliance reasons.

For these scenarios, the Data Protection system offers a rich configuration API.

> ⚠ **Warning**
>
> Similar to configuration files, the data protection key ring should be protected using appropriate permissions. You can choose to encrypt keys at rest, but this doesn't prevent cyberattackers from creating new keys. Consequently, your app's security is impacted. The storage location configured with Data Protection should have its access limited to the app itself, similar to the way you would protect configuration files. For example, if you choose to store your key ring on disk, use file system permissions. Ensure only the identity under which your web app runs has read, write, and create access to that directory. If you use Azure Blob Storage, only the web app should have the ability to read, write, or create new entries in the blob store, etc.
>
> The extension method AddDataProtection returns an IDataProtectionBuilder. `IDataProtectionBuilder` exposes extension methods that you can chain together to configure Data Protection options.

> ⓘ **Note**
>
> This article was written for an app that runs within a docker container. In a docker container the app always has the same path and, therefore, the same application discriminator. Apps that need to run in multiple environments (for example local and deployed), must set the default application discriminator for the environment. Running an app in multiple environments is beyond the scope of this article.

The following NuGet packages are required for the Data Protection extensions used in this article:

- Azure.Extensions.AspNetCore.DataProtection.Blobs
- Azure.Extensions.AspNetCore.DataProtection.Keys

ProtectKeysWithAzureKeyVault

Sign in to Azure using the CLI, for example:

Azure CLI

```
az login
```

To manage keys with Azure Key Vault, configure the system with ProtectKeysWithAzureKeyVault in `Program.cs`. `blobUriWithSasToken` is the full URI where the key file should be stored. The URI must contain the SAS token as a query string parameter:

C#

```
builder.Services.AddDataProtection()
    .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
    .ProtectKeysWithAzureKeyVault(new Uri("<keyIdentifier>"), new
DefaultAzureCredential());
```

For an app to communicate and authorize itself with KeyVault, the Azure.Identity package must be added.

Set the key ring storage location (for example, PersistKeysToAzureBlobStorage). The location must be set because calling `ProtectKeysWithAzureKeyVault` implements an IXmlEncryptor that disables automatic data protection settings, including the key ring storage location. The preceding example uses Azure Blob Storage to persist the key ring. For more information, see Key storage providers: Azure Storage. You can also persist the key ring locally with PersistKeysToFileSystem.

The `keyIdentifier` is the key vault key identifier used for key encryption. For example, a key created in key vault named `dataprotection` in the `contosokeyvault` has the key identifier `https://contosokeyvault.vault.azure.net/keys/dataprotection/`. Provide the app with **Get**, **Unwrap Key** and **Wrap Key** permissions to the key vault.

`ProtectKeysWithAzureKeyVault` overloads:

- ProtectKeysWithAzureKeyVault(IDataProtectionBuilder, Uri, TokenCredential) permits the use of a keyIdentifier Uri and a tokenCredential to enable the data protection system to use the key vault.

- ProtectKeysWithAzureKeyVault(IDataProtectionBuilder, String, IKeyEncryptionKeyResolver) permits the use of a keyIdentifier string and IKeyEncryptionKeyResolver to enable the data protection system to use the key vault.

If the app uses the older Azure packages (Microsoft.AspNetCore.DataProtection.AzureStorage and Microsoft.AspNetCore.DataProtection.AzureKeyVault), we recommend ***removing*** these references and upgrading to the Azure.Extensions.AspNetCore.DataProtection.Blobs and Azure.Extensions.AspNetCore.DataProtection.Keys. These packages are where new updates are provided, and address some key security and stability issues with the older packages.

C#

```
builder.Services.AddDataProtection()
    // This blob must already exist before the application is run
    .PersistKeysToAzureBlobStorage("<storageAccountConnectionString", "
<containerName>", "<blobName>")
    // Removing this line below for an initial run will ensure the file is
created correctly
    .ProtectKeysWithAzureKeyVault(new Uri("<keyIdentifier>"), new
DefaultAzureCredential());
```

PersistKeysToFileSystem

To store keys on a UNC share instead of at the *%LOCALAPPDATA%* default location, configure the system with PersistKeysToFileSystem:

C#

```
builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"));
```

> ⚠ **Warning**
>
> If you change the key persistence location, the system no longer automatically encrypts keys at rest, since it doesn't know whether DPAPI is an appropriate encryption mechanism.

PersistKeysToDbContext

To store keys in a database using EntityFramework, configure the system with the Microsoft.AspNetCore.DataProtection.EntityFrameworkCore package:

```
C#

builder.Services.AddDataProtection()
    .PersistKeysToDbContext<SampleDbContext>();
```

The preceding code stores the keys in the configured database. The database context being used must implement `IDataProtectionKeyContext`. `IDataProtectionKeyContext` exposes the property `DataProtectionKeys`

```
C#

public DbSet<DataProtectionKey> DataProtectionKeys { get; set; } = null!;
```

This property represents the table in which the keys are stored. Create the table manually or with `DbContext` Migrations. For more information, see DataProtectionKey.

ProtectKeysWith*

You can configure the system to protect keys at rest by calling any of the ProtectKeysWith* configuration APIs. Consider the example below, which stores keys on a UNC share and encrypts those keys at rest with a specific X.509 certificate:

```
C#

builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"))

.ProtectKeysWithCertificate(builder.Configuration["CertificateThumbprint"]);
```

You can provide an X509Certificate2 to ProtectKeysWithCertificate, such as a certificate loaded from a file:

```
C#

builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"))
    .ProtectKeysWithCertificate(
        new X509Certificate2("certificate.pfx",
builder.Configuration["CertificatePassword"]));
```

See Key Encryption At Rest for more examples and discussion on the built-in key encryption mechanisms.

UnprotectKeysWithAnyCertificate

You can rotate certificates and decrypt keys at rest using an array of X509Certificate2 certificates with UnprotectKeysWithAnyCertificate:

C#

```
builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\directory\"))
    .ProtectKeysWithCertificate(
        new X509Certificate2("certificate.pfx",
builder.Configuration["CertificatePassword"]))
    .UnprotectKeysWithAnyCertificate(
        new X509Certificate2("certificate_1.pfx",
builder.Configuration["CertificatePassword_1"]),
        new X509Certificate2("certificate_2.pfx",
builder.Configuration["CertificatePassword_2"]));
```

SetDefaultKeyLifetime

To configure the system to use a key lifetime of 14 days instead of the default 90 days, use SetDefaultKeyLifetime:

C#

```
builder.Services.AddDataProtection()
    .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

SetApplicationName

By default, the Data Protection system isolates apps from one another based on their content root paths, even if they share the same physical key repository. This isolation prevents the apps from understanding each other's protected payloads.

To share protected payloads among apps:

- Configure SetApplicationName in each app with the same value.
- Use the same version of the Data Protection API stack across the apps. Perform **either** of the following in the apps' project files:

- Reference the same shared framework version via the Microsoft.AspNetCore.App metapackage.
- Reference the same Data Protection package version.

```
C#

builder.Services.AddDataProtection()
    .SetApplicationName("<sharedApplicationName>");
```

SetApplicationName internally sets DataProtectionOptions.ApplicationDiscriminator. For troubleshooting purposes, the value assigned to the discriminator by the framework can be logged with the following code placed after the WebApplication is built in `Program.cs`:

```
C#

var discriminator =
app.Services.GetRequiredService<IOptions<DataProtectionOptions>>()
    .Value.ApplicationDiscriminator;
app.Logger.LogInformation("ApplicationDiscriminator:
{ApplicationDiscriminator}", discriminator);
```

For more information on how the discriminator is used, see the following sections later in this article:

- Per-application isolation
- Data Protection and app isolation

> ⚠ **Warning**
>
> In .NET 6, WebApplicationBuilder normalizes the content root path to end with a DirectorySeparatorChar. For example, on Windows the content root path ends in `\` and on Linux `/`. Other hosts don't normalize the path. Most apps migrating from HostBuilder or WebHostBuilder won't share the same app name because they won't have the terminating `DirectorySeparatorChar`. To work around this issue, remove the directory separator character and set the app name manually, as shown in the following code:
>
> ```
> C#
>
> using Microsoft.AspNetCore.DataProtection;
> using System.Reflection;
>
> var builder = WebApplication.CreateBuilder(args);
> var trimmedContentRootPath =
> ```

```
builder.Environment.ContentRootPath.TrimEnd(Path.DirectorySeparatorChar
);
builder.Services.AddDataProtection()
 .SetApplicationName(trimmedContentRootPath);
var app = builder.Build();

app.MapGet("/", () => Assembly.GetEntryAssembly()!.GetName().Name);

app.Run();
```

DisableAutomaticKeyGeneration

You may have a scenario where you don't want an app to automatically roll keys (create new keys) as they approach expiration. One example of this scenario might be apps set up in a primary/secondary relationship, where only the primary app is responsible for key management concerns and secondary apps simply have a read-only view of the key ring. The secondary apps can be configured to treat the key ring as read-only by configuring the system with DisableAutomaticKeyGeneration:

C#

```
builder.Services.AddDataProtection()
    .DisableAutomaticKeyGeneration();
```

Per-application isolation

When the Data Protection system is provided by an ASP.NET Core host, it automatically isolates apps from one another, even if those apps are running under the same worker process account and are using the same master keying material. This is similar to the IsolateApps modifier from System.Web's `<machineKey>` element.

The isolation mechanism works by considering each app on the local machine as a unique tenant, thus the IDataProtector rooted for any given app automatically includes the app ID as a discriminator (ApplicationDiscriminator). The app's unique ID is the app's physical path:

- For apps hosted in IIS, the unique ID is the IIS physical path of the app. If an app is deployed in a web farm environment, this value is stable assuming that the IIS environments are configured similarly across all machines in the web farm.
- For self-hosted apps running on the Kestrel server, the unique ID is the physical path to the app on disk.

The unique identifier is designed to survive resets—both of the individual app and of the machine itself.

This isolation mechanism assumes that the apps aren't malicious. A malicious app can always impact any other app running under the same worker process account. In a shared hosting environment where apps are mutually untrusted, the hosting provider should take steps to ensure OS-level isolation between apps, including separating the apps' underlying key repositories.

If the Data Protection system isn't provided by an ASP.NET Core host (for example, if you instantiate it via the `DataProtectionProvider` concrete type) app isolation is disabled by default. When app isolation is disabled, all apps backed by the same keying material can share payloads as long as they provide the appropriate purposes. To provide app isolation in this environment, call the SetApplicationName method on the configuration object and provide a unique name for each app.

Data Protection and app isolation

Consider the following points for app isolation:

- When multiple apps are pointed at the same key repository, the intention is that the apps share the same master key material. Data Protection is developed with the assumption that all apps sharing a key ring can access all items in that key ring. The application unique identifier is used to isolate application specific keys derived from the key ring provided keys. It doesn't expect item level permissions, such as those provided by Azure KeyVault to be used to enforce extra isolation. Attempting item level permissions generates application errors. If you don't want to rely on the built-in application isolation, separate key store locations should be used and not shared between applications.

- The application discriminator (ApplicationDiscriminator) is used to allow different apps to share the same master key material but to keep their cryptographic payloads distinct from one another. For the apps to be able to read each other's cryptographic payloads, they must have the same application discriminator, which can be set by calling SetApplicationName.

- If an app is compromised (for example, by an RCE attack), all master key material accessible to that app must also be considered compromised, regardless of its protection-at-rest state. This implies that if two apps are pointed at the same repository, even if they use different app discriminators, a compromise of one is functionally equivalent to a compromise of both.

This "functionally equivalent to a compromise of both" clause holds even if the two apps use different mechanisms for key protection at rest. Typically, this isn't an expected configuration. The protection-at-rest mechanism is intended to provide protection in the event a cyberattacker gains read access to the repository. A cyberattacker who gains write access to the repository (perhaps because they attained code execution permission within an app) can insert malicious keys into storage. The Data Protection system intentionally doesn't provide protection against a cyberattacker who gains write access to the key repository.

- If apps need to remain truly isolated from one another, they should use different key repositories. This naturally falls out of the definition of "isolated". Apps are ***not*** isolated if they all have Read and Write access to each other's data stores.

Changing algorithms with UseCryptographicAlgorithms

The Data Protection stack allows you to change the default algorithm used by newly generated keys. The simplest way to do this is to call UseCryptographicAlgorithms from the configuration callback:

C#

```
builder.Services.AddDataProtection()
    .UseCryptographicAlgorithms(new AuthenticatedEncryptorConfiguration
    {
        EncryptionAlgorithm = EncryptionAlgorithm.AES_256_CBC,
        ValidationAlgorithm = ValidationAlgorithm.HMACSHA256
    });
```

The default EncryptionAlgorithm is AES-256-CBC, and the default ValidationAlgorithm is HMACSHA256. The default policy can be set by a system administrator via a machine-wide policy, but an explicit call to `UseCryptographicAlgorithms` overrides the default policy.

Calling `UseCryptographicAlgorithms` allows you to specify the desired algorithm from a predefined built-in list. You don't need to worry about the implementation of the algorithm. In the scenario above, the Data Protection system attempts to use the CNG implementation of AES if running on Windows. Otherwise, it falls back to the managed System.Security.Cryptography.Aes class.

You can manually specify an implementation via a call to UseCustomCryptographicAlgorithms.

> **Tip**
>
> Changing algorithms doesn't affect existing keys in the key ring. It only affects newly-generated keys.

Specifying custom managed algorithms

To specify custom managed algorithms, create a ManagedAuthenticatedEncryptorConfiguration instance that points to the implementation types:

C#

```
builder.Services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new
ManagedAuthenticatedEncryptorConfiguration
    {
        // A type that subclasses SymmetricAlgorithm
        EncryptionAlgorithmType = typeof(Aes),

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // A type that subclasses KeyedHashAlgorithm
        ValidationAlgorithmType = typeof(HMACSHA256)
    });
```

Generally the *Type properties must point to concrete, instantiable (via a public parameterless ctor) implementations of SymmetricAlgorithm and KeyedHashAlgorithm, though the system special-cases some values like `typeof(Aes)` for convenience.

> **Note**
>
> The SymmetricAlgorithm must have a key length of ≥ 128 bits and a block size of ≥ 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The KeyedHashAlgorithm must have a digest size of >= 128 bits, and it must support keys of length equal to the hash algorithm's digest length. The KeyedHashAlgorithm isn't strictly required to be HMAC.

Specifying custom Windows CNG algorithms

To specify a custom Windows CNG algorithm using CBC-mode encryption with HMAC validation, create a CngCbcAuthenticatedEncryptorConfiguration instance that contains the algorithmic information:

C#

```
builder.Services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new
CngCbcAuthenticatedEncryptorConfiguration
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
        EncryptionAlgorithmKeySize = 256,

        // Passed to BCryptOpenAlgorithmProvider
        HashAlgorithm = "SHA256",
        HashAlgorithmProvider = null
    });
```

> **Note**
>
> The symmetric block cipher algorithm must have a key length of >= 128 bits, a block size of >= 64 bits, and it must support CBC-mode encryption with PKCS #7 padding. The hash algorithm must have a digest size of >= 128 bits and must support being opened with the BCRYPT_ALG_HANDLE_HMAC_FLAG flag. The *Provider properties can be set to null to use the default provider for the specified algorithm. For more information, see the BCryptOpenAlgorithmProvider documentation.

To specify a custom Windows CNG algorithm using Galois/Counter Mode encryption with validation, create a CngGcmAuthenticatedEncryptorConfiguration instance that contains the algorithmic information:

C#

```
builder.Services.AddDataProtection()
    .UseCustomCryptographicAlgorithms(new
CngGcmAuthenticatedEncryptorConfiguration
    {
        // Passed to BCryptOpenAlgorithmProvider
        EncryptionAlgorithm = "AES",
        EncryptionAlgorithmProvider = null,

        // Specified in bits
```

```
        EncryptionAlgorithmKeySize = 256
    });
```

> **Note**
>
> The symmetric block cipher algorithm must have a key length of >= 128 bits, a block size of exactly 128 bits, and it must support GCM encryption. You can set the EncryptionAlgorithmProvider property to null to use the default provider for the specified algorithm. For more information, see the BCryptOpenAlgorithmProvider documentation.

Specifying other custom algorithms

Though not exposed as a first-class API, the Data Protection system is extensible enough to allow specifying almost any kind of algorithm. For example, it's possible to keep all keys contained within a Hardware Security Module (HSM) and to provide a custom implementation of the core encryption and decryption routines. For more information, see IAuthenticatedEncryptor in Core cryptography extensibility.

Persisting keys when hosting in a Docker container

When hosting in a Docker container, keys should be maintained in either:

- A folder that's a Docker volume that persists beyond the container's lifetime, such as a shared volume or a host-mounted volume.
- An external provider, such as Azure Blob Storage (shown in the ProtectKeysWithAzureKeyVault section) or Redis.

Persisting keys with Redis

Only Redis versions supporting Redis Data Persistence should be used to store keys. Azure Blob storage is persistent and can be used to store keys. For more information, see this GitHub issue.

Logging DataProtection

Enable `Information` level logging of DataProtection to help diagnosis problem. The following `appsettings.json` file enables information logging of the DataProtection API:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.AspNetCore.DataProtection": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

For more information on logging, see Logging in .NET Core and ASP.NET Core.

Additional resources

- Non-DI aware scenarios for Data Protection in ASP.NET Core
- Data Protection machine-wide policy support in ASP.NET Core
- Host ASP.NET Core in a web farm
- Key storage providers in ASP.NET Core

Data Protection key management and lifetime in ASP.NET Core

Article • 08/21/2024

By Rick Anderson

Key management

The app attempts to detect its operational environment and handle key configuration on its own.

1. If the app is hosted in Azure Apps, keys are persisted to the *%HOME%\ASP.NET\DataProtection-Keys* folder. This folder is backed by network storage and is synchronized across all machines hosting the app.

   - Keys aren't protected at rest.
   - The *DataProtection-Keys* folder supplies the key ring to all instances of an app in a single deployment slot.
   - Separate deployment slots, such as Staging and Production, don't share a key ring. When you swap between deployment slots, for example swapping Staging to Production or using A/B testing, any app using Data Protection won't be able to decrypt stored data using the key ring inside the previous slot. This leads to users being logged out of an app that uses the standard ASP.NET Core cookie authentication, as it uses Data Protection to protect its cookies. If you desire slot-independent key rings, use an external key ring provider, such as Azure Blob Storage, Azure Key Vault, a SQL store, or Redis cache.

2. If the user profile is available, keys are persisted to the *%LOCALAPPDATA%\ASP.NET\DataProtection-Keys* folder. If the operating system is Windows, the keys are encrypted at rest using DPAPI.

   The app pool's setProfileEnvironment attribute must also be enabled. The default value of `setProfileEnvironment` is `true`. In some scenarios (for example, Windows OS), `setProfileEnvironment` is set to `false`. If keys aren't stored in the user profile directory as expected:
   a. Navigate to the *%windir%/system32/inetsrv/config* folder.
   b. Open the *applicationHost.config* file.
   c. Locate the `<system.applicationHost><applicationPools><applicationPoolDefaults><processModel>` element.

   d. Confirm that the `setProfileEnvironment` attribute isn't present, which defaults the value to `true`, or explicitly set the attribute's value to `true`.

3. If the app is hosted in IIS, keys are persisted to the HKLM registry in a special registry key that's ACLed only to the worker process account. Keys are encrypted at rest using DPAPI.

4. If none of these conditions match, keys aren't persisted outside of the current process. When the process shuts down, all generated keys are lost.

The developer is always in full control and can override how and where keys are stored. The first three options above should provide good defaults for most apps similar to how the ASP.NET **<machineKey>** auto-generation routines worked in the past. The final, fallback option is the only scenario that requires the developer to specify configuration upfront if they want key persistence, but this fallback only occurs in rare situations.

When hosting in a Docker container, keys should be persisted in a folder that's a Docker volume (a shared volume or a host-mounted volume that persists beyond the container's lifetime) or in an external provider, such as Azure Key Vault or Redis. An external provider is also useful in web farm scenarios if apps can't access a shared network volume (see PersistKeysToFileSystem for more information).

> ⚠ **Warning**
>
> If the developer overrides the rules outlined above and points the Data Protection system at a specific key repository, automatic encryption of keys at rest is disabled. At-rest protection can be re-enabled via **configuration**.

Key lifetime

Keys have a 90-day lifetime by default. When a key expires, the app automatically generates a new key and sets the new key as the active key. As long as retired keys remain on the system, your app can decrypt any data protected with them. See key management for more information.

Default algorithms

The default payload protection algorithm used is AES-256-CBC for confidentiality and HMACSHA256 for authenticity. A 512-bit master key, changed every 90 days, is used to derive the two sub-keys used for these algorithms on a per-payload basis. See subkey derivation for more information.

Additional resources

- Key management extensibility in ASP.NET Core
- Host ASP.NET Core in a web farm

Data Protection machine-wide policy support in ASP.NET Core

Article • 06/03/2022

By Rick Anderson

When running on Windows, the Data Protection system has limited support for setting a default machine-wide policy for all apps that consume ASP.NET Core Data Protection. The general idea is that an administrator might wish to change a default setting, such as the algorithms used or key lifetime, without the need to manually update every app on the machine.

> ⚠ **Warning**
>
> The system administrator can set default policy, but they can't enforce it. The app developer can always override any value with one of their own choosing. The default policy only affects apps where the developer hasn't specified an explicit value for a setting.

Setting default policy

To set default policy, an administrator can set known values in the system registry under the following registry key:

HKLM\SOFTWARE\Microsoft\DotNetPackages\Microsoft.AspNetCore.DataProtection

If you're on a 64-bit operating system and want to affect the behavior of 32-bit apps, remember to configure the Wow6432Node equivalent of the above key.

The supported values are shown below.

Expand table

Value	Type	Description
EncryptionType	string	Specifies which algorithms should be used for data protection. The value must be CNG-CBC, CNG-GCM, or Managed and is described in more detail below.
DefaultKeyLifetime	DWORD	Specifies the lifetime for newly-generated keys. The value is specified in days and must be >= 7.

Value	Type	Description
KeyEscrowSinks	string	Specifies the types that are used for key escrow. The value is a semicolon-delimited list of key escrow sinks, where each element in the list is the assembly-qualified name of a type that implements IKeyEscrowSink.

Encryption types

If EncryptionType is CNG-CBC, the system is configured to use a CBC-mode symmetric block cipher for confidentiality and HMAC for authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngCbcAuthenticatedEncryptionSettings type.

Expand table

Value	Type	Description
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in CBC mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm EncryptionAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.
HashAlgorithm	string	The name of a hash algorithm understood by CNG. This algorithm is opened in HMAC mode.
HashAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm HashAlgorithm.

If EncryptionType is CNG-GCM, the system is configured to use a Galois/Counter Mode symmetric block cipher for confidentiality and authenticity with services provided by Windows CNG (see Specifying custom Windows CNG algorithms for more details). The following additional values are supported, each of which corresponds to a property on the CngGcmAuthenticatedEncryptionSettings type.

Expand table

Value	Type	Description
EncryptionAlgorithm	string	The name of a symmetric block cipher algorithm understood by CNG. This algorithm is opened in Galois/Counter Mode.
EncryptionAlgorithmProvider	string	The name of the CNG provider implementation that can produce the algorithm EncryptionAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric block cipher algorithm.

If EncryptionType is Managed, the system is configured to use a managed SymmetricAlgorithm for confidentiality and KeyedHashAlgorithm for authenticity (see Specifying custom managed algorithms for more details). The following additional values are supported, each of which corresponds to a property on the ManagedAuthenticatedEncryptionSettings type.

Expand table

Value	Type	Description
EncryptionAlgorithmType	string	The assembly-qualified name of a type that implements SymmetricAlgorithm.
EncryptionAlgorithmKeySize	DWORD	The length (in bits) of the key to derive for the symmetric encryption algorithm.
ValidationAlgorithmType	string	The assembly-qualified name of a type that implements KeyedHashAlgorithm.

If EncryptionType has any other value other than null or empty, the Data Protection system throws an exception at startup.

> ⚠ **Warning**
>
> When configuring a default policy setting that involves type names (EncryptionAlgorithmType, ValidationAlgorithmType, KeyEscrowSinks), the types must be available to the app. This means that for apps running on Desktop CLR, the assemblies that contain these types should be present in the Global Assembly Cache (GAC). For ASP.NET Core apps running on .NET Core, the packages that contain these types should be installed.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Non-DI aware scenarios for Data Protection in ASP.NET Core

Article • 06/03/2022

By Rick Anderson

The ASP.NET Core Data Protection system is normally added to a service container and consumed by dependent components via dependency injection (DI). However, there are cases where this isn't feasible or desired, especially when importing the system into an existing app.

To support these scenarios, the Microsoft.AspNetCore.DataProtection.Extensions package provides a concrete type, DataProtectionProvider, which offers a simple way to use Data Protection without relying on DI. The `DataProtectionProvider` type implements IDataProtectionProvider. Constructing `DataProtectionProvider` only requires providing a DirectoryInfo instance to indicate where the provider's cryptographic keys should be stored, as seen in the following code sample:

C#

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder));

        var protector = dataProtectionProvider.CreateProtector("Program.No-
DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
```

```
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Enter input: Hello world!
 * Protect returned: CfDJ8FWbAn6...ch3hAPm1NJA
 * Unprotect returned: Hello world!
 *
 * Press any key...
*/
```

By default, the `DataProtectionProvider` concrete type doesn't encrypt raw key material before persisting it to the file system. This is to support scenarios where the developer points to a network share and the Data Protection system can't automatically deduce an appropriate at-rest key encryption mechanism.

Additionally, the `DataProtectionProvider` concrete type doesn't isolate apps by default. All apps using the same key directory can share payloads as long as their purpose parameters match.

The DataProtectionProvider constructor accepts an optional configuration callback that can be used to adjust the behaviors of the system. The sample below demonstrates restoring isolation with an explicit call to SetApplicationName. The sample also demonstrates configuring the system to automatically encrypt persisted keys using Windows DPAPI. If the directory points to a UNC share, you may wish to distribute a shared certificate across all relevant machines and to configure the system to use certificate-based encryption with a call to ProtectKeysWithCertificate.

C#

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        // Get the path to %LOCALAPPDATA%\myapp-keys
        var destFolder = Path.Combine(
            System.Environment.GetEnvironmentVariable("LOCALAPPDATA"),
```

```
            "myapp-keys");

        // Instantiate the data protection system at this folder
        var dataProtectionProvider = DataProtectionProvider.Create(
            new DirectoryInfo(destFolder),
            configuration =>
            {
                configuration.SetApplicationName("my app name");
                configuration.ProtectKeysWithDpapi();
            });

        var protector = dataProtectionProvider.CreateProtector("Program.No-
DI");
        Console.Write("Enter input: ");
        var input = Console.ReadLine();

        // Protect the payload
        var protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // Unprotect the payload
        var unprotectedPayload = protector.Unprotect(protectedPayload);
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        Console.WriteLine();
        Console.WriteLine("Press any key...");
        Console.ReadKey();
    }
}
```

Tip

Instances of the `DataProtectionProvider` concrete type are expensive to create. If an app maintains multiple instances of this type and if they're all using the same key storage directory, app performance might degrade. If you use the `DataProtectionProvider` type, we recommend that you create this type once and reuse it as much as possible. The `DataProtectionProvider` type and all IDataProtector instances created from it are thread-safe for multiple callers.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

ASP.NET Core Data Protection extensibility APIs

Article • 06/03/2022

- Core cryptography extensibility
- Key management extensibility
- Miscellaneous APIs

Core cryptography extensibility in ASP.NET Core

Article • 10/06/2022

> ⚠ **Warning**
>
> Types that implement any of the following interfaces should be thread-safe for multiple callers.

IAuthenticatedEncryptor

The **IAuthenticatedEncryptor** interface is the basic building block of the cryptographic subsystem. There's generally one IAuthenticatedEncryptor per key, and the IAuthenticatedEncryptor instance wraps all cryptographic key material and algorithmic information necessary to perform cryptographic operations.

As its name suggests, the type is responsible for providing authenticated encryption and decryption services. It exposes the following two APIs.

- `Decrypt(ArraySegment<byte> ciphertext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`
- `Encrypt(ArraySegment<byte> plaintext, ArraySegment<byte> additionalAuthenticatedData) : byte[]`

The Encrypt method returns a blob that includes the enciphered plaintext and an authentication tag. The authentication tag must encompass the additional authenticated data (AAD), though the AAD itself need not be recoverable from the final payload. The Decrypt method validates the authentication tag and returns the deciphered payload. All failures (except ArgumentNullException and similar) should be homogenized to CryptographicException.

> ⓘ **Note**
>
> The IAuthenticatedEncryptor instance itself doesn't actually need to contain the key material. For example, the implementation could delegate to an HSM for all operations.

How to create an IAuthenticatedEncryptor

ASP.NET Core 2.x

The **IAuthenticatedEncryptorFactory** interface represents a type that knows how to create an IAuthenticatedEncryptor instance. Its API is as follows.

- CreateEncryptorInstance(IKey key) : IAuthenticatedEncryptor

For any given IKey instance, any authenticated encryptors created by its CreateEncryptorInstance method should be considered equivalent, as in the below code sample.

C#

```
// we have an IAuthenticatedEncryptorFactory instance and an IKey
instance
IAuthenticatedEncryptorFactory factory = ...;
IKey key = ...;

// get an encryptor instance and perform an authenticated encryption
operation
ArraySegment<byte> plaintext = new ArraySegment<byte>
(Encoding.UTF8.GetBytes("plaintext"));
ArraySegment<byte> aad = new ArraySegment<byte>
(Encoding.UTF8.GetBytes("AAD"));
var encryptor1 = factory.CreateEncryptorInstance(key);
byte[] ciphertext = encryptor1.Encrypt(plaintext, aad);

// get another encryptor instance and perform an authenticated
decryption operation
var encryptor2 = factory.CreateEncryptorInstance(key);
byte[] roundTripped = encryptor2.Decrypt(new ArraySegment<byte>
(ciphertext), aad);


// the 'roundTripped' and 'plaintext' buffers should be equivalent
```

IAuthenticatedEncryptorDescriptor (ASP.NET Core 2.x only)

ASP.NET Core 2.x

The **IAuthenticatedEncryptorDescriptor** interface represents a type that knows how to export itself to XML. Its API is as follows.

- ExportToXml() : XmlSerializedDescriptorInfo

XML Serialization

The primary difference between IAuthenticatedEncryptor and IAuthenticatedEncryptorDescriptor is that the descriptor knows how to create the encryptor and supply it with valid arguments. Consider an IAuthenticatedEncryptor whose implementation relies on SymmetricAlgorithm and KeyedHashAlgorithm. The encryptor's job is to consume these types, but it doesn't necessarily know where these types came from, so it can't really write out a proper description of how to recreate itself if the application restarts. The descriptor acts as a higher level on top of this. Since the descriptor knows how to create the encryptor instance (for example it knows how to create the required algorithms), it can serialize that knowledge in XML form so that the encryptor instance can be recreated after an application reset.

The descriptor can be serialized via its ExportToXml routine. This routine returns an XmlSerializedDescriptorInfo which contains two properties: the XElement representation of the descriptor and the Type which represents an IAuthenticatedEncryptorDescriptorDeserializer which can be used to resurrect this descriptor given the corresponding XElement.

The serialized descriptor may contain sensitive information such as cryptographic key material. The data protection system has built-in support for encrypting information before it's persisted to storage. To take advantage of this, the descriptor should mark the element which contains sensitive information with the attribute name "requiresEncryption" (xmlns `"<http://schemas.asp.net/2015/03/dataProtection>"`), value "true".

> **Tip**
>
> There's a helper API for setting this attribute. Call the extension method XElement.MarkAsRequiresEncryption() located in namespace Microsoft.AspNetCore.DataProtection.AuthenticatedEncryption.ConfigurationModel .

There can also be cases where the serialized descriptor doesn't contain sensitive information. Consider again the case of a cryptographic key stored in an HSM. The descriptor cannot write out the key material when serializing itself since the HSM won't expose the material in plaintext form. Instead, the descriptor might write out the key-

wrapped version of the key (if the HSM allows export in this fashion) or the HSM's own unique identifier for the key.

IAuthenticatedEncryptorDescriptorDeserializer

The **IAuthenticatedEncryptorDescriptorDeserializer** interface represents a type that knows how to deserialize an IAuthenticatedEncryptorDescriptor instance from an XElement. It exposes a single method:

- ImportFromXml(XElement element) : IAuthenticatedEncryptorDescriptor

The ImportFromXml method takes the XElement that was returned by IAuthenticatedEncryptorDescriptor.ExportToXml and creates an equivalent of the original IAuthenticatedEncryptorDescriptor.

Types which implement IAuthenticatedEncryptorDescriptorDeserializer should have one of the following two public constructors:

- .ctor(IServiceProvider)
- .ctor()

> ⓘ **Note**
>
> The IServiceProvider passed to the constructor may be null.

The top-level factory

ASP.NET Core 2.x

The **AlgorithmConfiguration** class represents a type which knows how to create IAuthenticatedEncryptorDescriptor instances. It exposes a single API.

- CreateNewDescriptor() : IAuthenticatedEncryptorDescriptor

Think of AlgorithmConfiguration as the top-level factory. The configuration serves as a template. It wraps algorithmic information (for example this configuration produces descriptors with an AES-128-GCM master key), but it's not yet associated with a specific key.

When CreateNewDescriptor is called, fresh key material is created solely for this call, and a new IAuthenticatedEncryptorDescriptor is produced which wraps this key

material and the algorithmic information required to consume the material. The key material could be created in software (and held in memory), it could be created and held within an HSM, and so on. The crucial point is that any two calls to CreateNewDescriptor should never create equivalent IAuthenticatedEncryptorDescriptor instances.

The AlgorithmConfiguration type serves as the entry point for key creation routines such as automatic key rolling. To change the implementation for all future keys, set the AuthenticatedEncryptorConfiguration property in KeyManagementOptions.

Key management extensibility in ASP.NET Core

Article • 06/03/2022

Read the key management section before reading this section, as it explains some of the fundamental concepts behind these APIs.

Warning: Types that implement any of the following interfaces should be thread-safe for multiple callers.

Key

The `IKey` interface is the basic representation of a key in cryptosystem. The term key is used here in the abstract sense, not in the literal sense of "cryptographic key material". A key has the following properties:

- Activation, creation, and expiration dates
- Revocation status
- Key identifier (a GUID)

Additionally, `IKey` exposes a `CreateEncryptor` method which can be used to create an IAuthenticatedEncryptor instance tied to this key.

> **Note**
>
> There's no API to retrieve the raw cryptographic material from an `IKey` instance.

IKeyManager

The `IKeyManager` interface represents an object responsible for general key storage, retrieval, and manipulation. It exposes three high-level operations:

- Create a new key and persist it to storage.
- Get all keys from storage.
- Revoke one or more keys and persist the revocation information to storage.

> ⚠ **Warning**
>
> Writing an `IKeyManager` is a very advanced task, and the majority of developers shouldn't attempt it. Instead, most developers should take advantage of the facilities offered by the XmlKeyManager class.

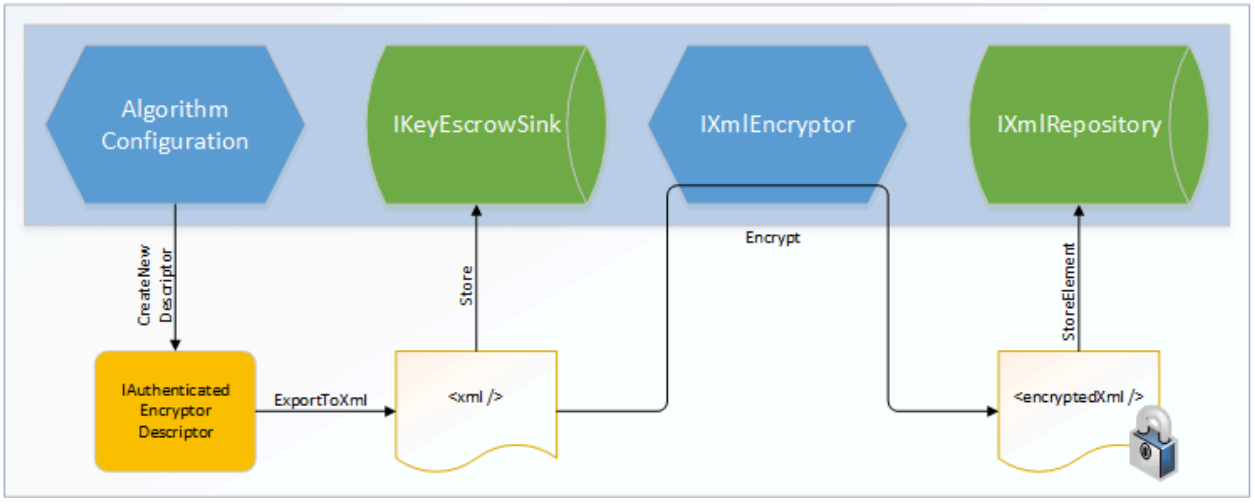
XmlKeyManager

The `XmlKeyManager` type is the in-box concrete implementation of `IKeyManager`. It provides several useful facilities, including key escrow and encryption of keys at rest. Keys in this system are represented as XML elements (specifically, XElement).

`XmlKeyManager` depends on several other components in the course of fulfilling its tasks:

- `AlgorithmConfiguration`, which dictates the algorithms used by new keys.
- `IXmlRepository`, which controls where keys are persisted in storage.
- `IXmlEncryptor` [optional], which allows encrypting keys at rest.
- `IKeyEscrowSink` [optional], which provides key escrow services.

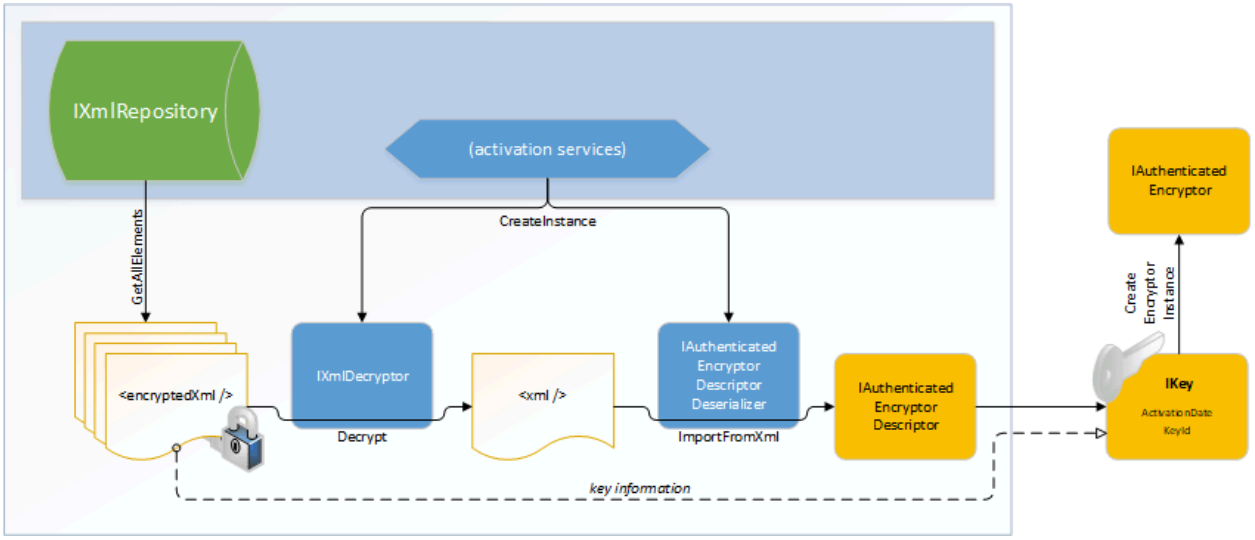
Below are high-level diagrams which indicate how these components are wired together within `XmlKeyManager`.



Key Creation / CreateNewKey

In the implementation of `CreateNewKey`, the `AlgorithmConfiguration` component is used to create a unique `IAuthenticatedEncryptorDescriptor`, which is then serialized as XML. If a key escrow sink is present, the raw (unencrypted) XML is provided to the sink for long-term storage. The unencrypted XML is then run through an `IXmlEncryptor` (if

required) to generate the encrypted XML document. This encrypted document is persisted to long-term storage via the `IXmlRepository`. (If no `IXmlEncryptor` is configured, the unencrypted document is persisted in the `IXmlRepository`.)



Key Retrieval / GetAllKeys

In the implementation of `GetAllKeys`, the XML documents representing keys and revocations are read from the underlying `IXmlRepository`. If these documents are encrypted, the system will automatically decrypt them. `XmlKeyManager` creates the appropriate `IAuthenticatedEncryptorDescriptorDeserializer` instances to deserialize the documents back into `IAuthenticatedEncryptorDescriptor` instances, which are then wrapped in individual `IKey` instances. This collection of `IKey` instances is returned to the caller.

Further information on the particular XML elements can be found in the key storage format document.

IXmlRepository

The `IXmlRepository` interface represents a type that can persist XML to and retrieve XML from a backing store. It exposes two APIs:

- `GetAllElements` : `IReadOnlyCollection<XElement>`
- `StoreElement(XElement element, string friendlyName)`

Implementations of `IXmlRepository` don't need to parse the XML passing through them. They should treat the XML documents as opaque and let higher layers worry about generating and parsing the documents.

There are four built-in concrete types which implement `IXmlRepository`:

- FileSystemXmlRepository
- RegistryXmlRepository
- AzureStorage.AzureBlobXmlRepository
- RedisXmlRepository

See the key storage providers document for more information.

Registering a custom `IXmlRepository` is appropriate when using a different backing store (for example, Azure Table Storage).

To change the default repository application-wide, register a custom `IXmlRepository` instance:

```
C#

services.Configure<KeyManagementOptions>(options => options.XmlRepository =
new MyCustomXmlRepository());
```

IXmlEncryptor

The `IXmlEncryptor` interface represents a type that can encrypt a plaintext XML element. It exposes a single API:

- Encrypt(XElement plaintextElement) : EncryptedXmlInfo

If a serialized `IAuthenticatedEncryptorDescriptor` contains any elements marked as "requires encryption", then `XmlKeyManager` will run those elements through the configured `IXmlEncryptor`'s `Encrypt` method, and it will persist the enciphered element rather than the plaintext element to the `IXmlRepository`. The output of the `Encrypt` method is an `EncryptedXmlInfo` object. This object is a wrapper which contains both the resultant enciphered `XElement` and the Type which represents an `IXmlDecryptor` which can be used to decipher the corresponding element.

There are four built-in concrete types which implement `IXmlEncryptor`:

- CertificateXmlEncryptor
- DpapiNGXmlEncryptor
- DpapiXmlEncryptor
- NullXmlEncryptor

See the key encryption at rest document for more information.

To change the default key-encryption-at-rest mechanism application-wide, register a custom `IXmlEncryptor` instance:

```
C#

services.Configure<KeyManagementOptions>(options => options.XmlEncryptor =
new MyCustomXmlEncryptor());
```

IXmlDecryptor

The `IXmlDecryptor` interface represents a type that knows how to decrypt an `XElement` that was enciphered via an `IXmlEncryptor`. It exposes a single API:

- Decrypt(XElement encryptedElement) : XElement

The `Decrypt` method undoes the encryption performed by `IXmlEncryptor.Encrypt`. Generally, each concrete `IXmlEncryptor` implementation will have a corresponding concrete `IXmlDecryptor` implementation.

Types which implement `IXmlDecryptor` should have one of the following two public constructors:

- .ctor(IServiceProvider)
- .ctor()

> ⓘ **Note**
>
> The `IServiceProvider` passed to the constructor may be null.

IKeyEscrowSink

The `IKeyEscrowSink` interface represents a type that can perform escrow of sensitive information. Recall that serialized descriptors might contain sensitive information (such as cryptographic material), and this is what led to the introduction of the IXmlEncryptor type in the first place. However, accidents happen, and key rings can be deleted or become corrupted.

The escrow interface provides an emergency escape hatch, allowing access to the raw serialized XML before it's transformed by any configured IXmlEncryptor. The interface exposes a single API:

- Store(Guid keyId, XElement element)

It's up to the `IKeyEscrowSink` implementation to handle the provided element in a secure manner consistent with business policy. One possible implementation could be for the escrow sink to encrypt the XML element using a known corporate X.509 certificate where the certificate's private key has been escrowed; the `CertificateXmlEncryptor` type can assist with this. The `IKeyEscrowSink` implementation is also responsible for persisting the provided element appropriately.

By default no escrow mechanism is enabled, though server administrators can configure this globally. It can also be configured programmatically via the `IDataProtectionBuilder.AddKeyEscrowSink` method as shown in the sample below. The `AddKeyEscrowSink` method overloads mirror the `IServiceCollection.AddSingleton` and `IServiceCollection.AddInstance` overloads, as `IKeyEscrowSink` instances are intended to be singletons. If multiple `IKeyEscrowSink` instances are registered, each one will be called during key generation, so keys can be escrowed to multiple mechanisms simultaneously.

There's no API to read material from an `IKeyEscrowSink` instance. This is consistent with the design theory of the escrow mechanism: it's intended to make the key material accessible to a trusted authority, and since the application is itself not a trusted authority, it shouldn't have access to its own escrowed material.

The following sample code demonstrates creating and registering an `IKeyEscrowSink` where keys are escrowed such that only members of "CONTOSODomain Admins" can recover them.

> **Note**
>
> To run this sample, you must be on a domain-joined Windows 8 / Windows Server 2012 machine, and the domain controller must be Windows Server 2012 or later.

C#

```
using System;
using System.IO;
using System.Xml.Linq;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.XmlEncryption;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

public class Program
```

```
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi()
            .AddKeyEscrowSink(sp => new MyKeyEscrowSink(sp));
        var services = serviceCollection.BuildServiceProvider();

        // get a reference to the key manager and force a new key to be
generated
        Console.WriteLine("Generating new key...");
        var keyManager = services.GetService<IKeyManager>();
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddDays(7));
    }

    // A key escrow sink where keys are escrowed such that they
    // can be read by members of the CONTOSO\Domain Admins group.
    private class MyKeyEscrowSink : IKeyEscrowSink
    {
        private readonly IXmlEncryptor _escrowEncryptor;

        public MyKeyEscrowSink(IServiceProvider services)
        {
            // Assuming I'm on a machine that's a member of the CONTOSO
            // domain, I can use the Domain Admins SID to generate an
            // encrypted payload that only they can read. Sample SID from
            // https://technet.microsoft.com/library/cc778824(v=ws.10).aspx.
            _escrowEncryptor = new DpapiNGXmlEncryptor(
                "SID=S-1-5-21-1004336348-1177238915-682003330-512",
                DpapiNGProtectionDescriptorFlags.None,
                new LoggerFactory());
        }

        public void Store(Guid keyId, XElement element)
        {
            // Encrypt the key element to the escrow encryptor.
            var encryptedXmlInfo = _escrowEncryptor.Encrypt(element);

            // A real implementation would save the escrowed key to a
            // write-only file share or some other stable storage, but
            // in this sample we'll just write it out to the console.
            Console.WriteLine($"Escrowing key {keyId}");
            Console.WriteLine(encryptedXmlInfo.EncryptedElement);

            // Note: We cannot read the escrowed key material ourselves.
            // We need to get a member of CONTOSO\Domain Admins to read
            // it for us in the event we need to recover it.
        }
    }
}
```

```
/*
 * SAMPLE OUTPUT
 *
 * Generating new key...
 * Escrowing key 38e74534-c1b8-4b43-aea1-79e856a822e5
 * <encryptedKey>
 *   <!-- This key is encrypted with Windows DPAPI-NG. -->
 *   <!-- Rule: SID=S-1-5-21-1004336348-1177238915-682003330-512 -->
 *   <value>MIIIfAYJKoZIhvcNAQcDoIIIbTCCCGkCAQ...T5rA4g==</value>
 * </encryptedKey>
 */
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Miscellaneous ASP.NET Core Data Protection APIs

Article • 06/03/2022

> ⚠ **Warning**
>
> Types that implement any of the following interfaces should be thread-safe for multiple callers.

ISecret

The `ISecret` interface represents a secret value, such as cryptographic key material. It contains the following API surface:

- `Length`: `int`
- `Dispose()`: `void`
- `WriteSecretIntoBuffer(ArraySegment<byte> buffer)`: `void`

The `WriteSecretIntoBuffer` method populates the supplied buffer with the raw secret value. The reason this API takes the buffer as a parameter rather than returning a `byte[]` directly is that this gives the caller the opportunity to pin the buffer object, limiting secret exposure to the managed garbage collector.

The `Secret` type is a concrete implementation of `ISecret` where the secret value is stored in in-process memory. On Windows platforms, the secret value is encrypted via CryptProtectMemory.

ASP.NET Core Data Protection implementation

Article • 06/03/2022

- Authenticated encryption details
- Subkey Derivation and Authenticated Encryption
- Context headers
- Key Management
- Key Storage Providers
- Key Encryption At Rest
- Key immutability and settings
- Key Storage Format
- Ephemeral data protection providers

Authenticated encryption details in ASP.NET Core

Article • 10/21/2024

Calls to IDataProtector.Protect are authenticated encryption operations. The Protect method offers both confidentiality and authenticity, and it's tied to the purpose chain that was used to derive this particular IDataProtector instance from its root IDataProtectionProvider.

IDataProtector.Protect takes a byte[] plaintext parameter and produces a byte[] protected payload, whose format is described below. (There's also an extension method overload which takes a string plaintext parameter and returns a string protected payload. If this API is used the protected payload format will still have the below structure, but it will be base64url-encoded.)

Protected payload format

The protected payload format consists of three primary components:

- A 32-bit magic header that identifies the version of the data protection system.
- A 128-bit key id that identifies the key used to protect this particular payload.
- The remainder of the protected payload is specific to the encryptor encapsulated by this key. In the example below, the key represents an AES-256-CBC + HMACSHA256 encryptor, and the payload is further subdivided as follows:
  - A 128-bit key modifier.
  - A 128-bit initialization vector.
  - 48 bytes of AES-256-CBC output.
  - An HMACSHA256 authentication tag.

A sample protected payload is illustrated below.

```
09 F0 C9 F0 80 9C 81 0C 19 66 19 40 95 36 53 F8
AA FF EE 57 57 2F 40 4C 3F 7F CC 9D CC D9 32 3E
84 17 99 16 EC BA 1F 4A A1 18 45 1F 2D 13 7A 28
79 6B 86 9C F8 B7 84 F9 26 31 FC B1 86 0A F1 56
61 CF 14 58 D3 51 6F CF 36 50 85 82 08 2D 3F 73
5F B0 AD 9E 1A B2 AE 13 57 90 C8 F5 7C 95 4E 6A
8A AA 06 EF 43 CA 19 62 84 7C 11 B2 C8 71 9D AA
```

```
52 19 2E 5B 4C 1E 54 F0 55 BE 88 92 12 C1 4B 5E
52 C9 74 A0
```

From the payload format above the first 32 bits, or 4 bytes are the magic header identifying the version (09 F0 C9 F0)

The next 128 bits, or 16 bytes is the key identifier (80 9C 81 0C 19 66 19 40 95 36 53 F8 AA FF EE 57)

The remainder contains the payload and is specific to the format used.

> ⚠ **Warning**
>
> All payloads protected to a given key will begin with the same 20-byte (magic value, key id) header. Administrators can use this fact for diagnostic purposes to approximate when a payload was generated. For example, the payload above corresponds to key {aaaaaaaa-0b0b-1c1c-2d2d-333333333333}. If after checking the key repository you find that this specific key's activation date was 2015-01-01 and its expiration date was 2015-03-01, then it's reasonable to assume that the payload (if not tampered with) was generated within that window, give or take a small fudge factor on either side.

Subkey derivation and authenticated encryption in ASP.NET Core

Article • 01/04/2023

Most keys in the key ring will contain some form of entropy and will have algorithmic information stating "CBC-mode encryption + HMAC validation" or "GCM encryption + validation". In these cases, we refer to the embedded entropy as the master keying material (or KM) for this key, and we perform a key derivation function to derive the keys that will be used for the actual cryptographic operations.

> ⓘ **Note**
>
> Keys are abstract, and a custom implementation might not behave as below. If the key provides its own implementation of `IAuthenticatedEncryptor` rather than using one of our built-in factories, the mechanism described in this section no longer applies.

Additional authenticated data and subkey derivation

The `IAuthenticatedEncryptor` interface serves as the core interface for all authenticated encryption operations. Its `Encrypt` method takes two buffers: plaintext and additionalAuthenticatedData (AAD). The plaintext contents flow unchanged the call to `IDataProtector.Protect`, but the AAD is generated by the system and consists of three components:

1. The 32-bit magic header 09 F0 C9 F0 that identifies this version of the data protection system.
2. The 128-bit key id.
3. A variable-length string formed from the purpose chain that created the `IDataProtector` that's performing this operation.

Because the AAD is unique for the tuple of all three components, we can use it to derive new keys from KM instead of using KM itself in all of our cryptographic operations. For every call to `IAuthenticatedEncryptor.Encrypt`, the following key derivation process takes place:

```
( K_E, K_H ) = SP800_108_CTR_HMACSHA512(K_M, AAD, contextHeader || keyModifier)
```

Here, we're calling the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with the following parameters:

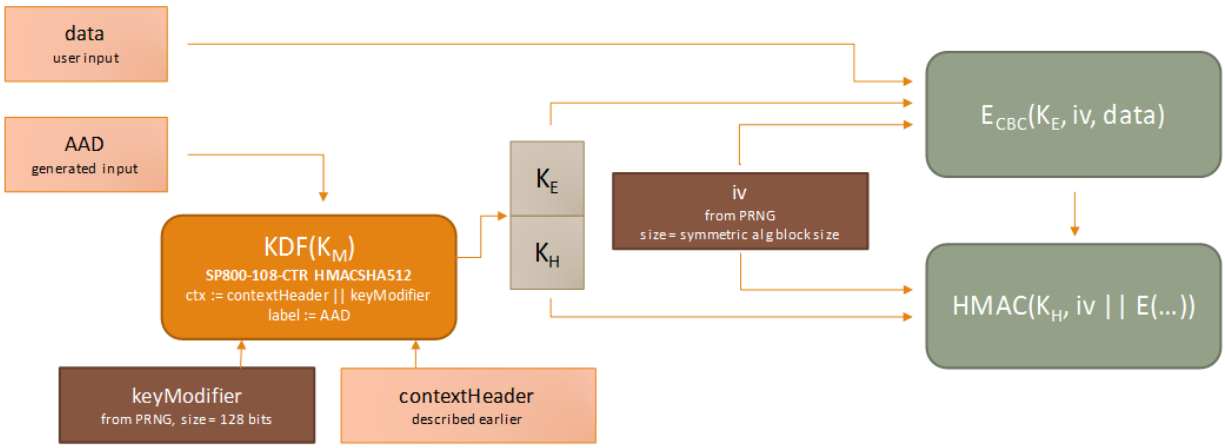
- Key derivation key (KDK) = `K_M`
- PRF = HMACSHA512
- label = additionalAuthenticatedData
- context = contextHeader || keyModifier

The context header is of variable length and essentially serves as a thumbprint of the algorithms for which we're deriving `K_E` and `K_H`. The key modifier is a 128-bit string randomly generated for each call to `Encrypt` and serves to ensure with overwhelming probability that KE and KH are unique for this specific authentication encryption operation, even if all other input to the KDF is constant.

For CBC-mode encryption + HMAC validation operations, `| K_E |` is the length of the symmetric block cipher key, and `| K_H |` is the digest size of the HMAC routine. For GCM encryption + validation operations, `| K_H | = 0`.

CBC-mode encryption + HMAC validation

Once `K_E` is generated via the above mechanism, we generate a random initialization vector and run the symmetric block cipher algorithm to encipher the plaintext. The initialization vector and ciphertext are then run through the HMAC routine initialized with the key `K_H` to produce the MAC. This process and the return value is represented graphically below.

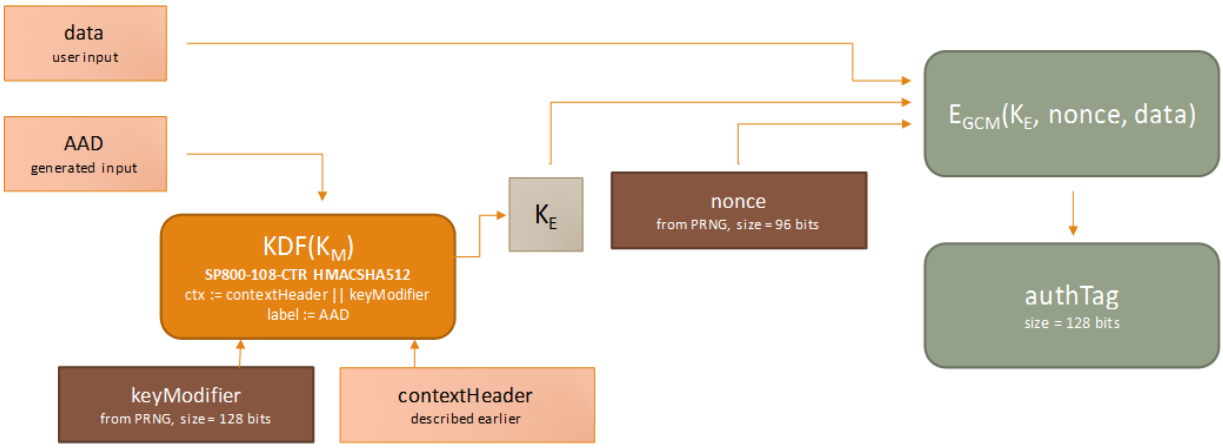


`output:= keyModifier || iv || E_cbc (K_E,iv,data) || HMAC(K_H, iv || E_cbc (K_E,iv,data))`

> ⓘ Note
>
> The `IDataProtector.Protect` implementation will **prepend the magic header and key id** to output before returning it to the caller. Because the magic header and key id are implicitly part of **AAD**, and because the key modifier is fed as input to the KDF, this means that every single byte of the final returned payload is authenticated by the MAC.

Galois/Counter Mode encryption + validation

Once `K_E` is generated via the above mechanism, we generate a random 96-bit nonce and run the symmetric block cipher algorithm to encipher the plaintext and produce the 128-bit authentication tag.



`output := keyModifier || nonce || E_gcm (K_E,nonce,data) || authTag`

> ⓘ Note
>
> Even though GCM natively supports the concept of AAD, we're still feeding AAD only to the original KDF, opting to pass an empty string into GCM for its AAD parameter. The reason for this is two-fold. First, **to support agility** we never want to use `K_M` directly as the encryption key. Additionally, GCM imposes very strict uniqueness requirements on its inputs. The probability that the GCM encryption routine is ever invoked on two or more distinct sets of input data with the same (key, nonce) pair must not exceed 2^-32. If we fix `K_E` we cannot perform more than 2^32 encryption operations before we run afoul of the 2^-32 limit. This might

seem like a very large number of operations, but a high-traffic web server can go through 4 billion requests in mere days, well within the normal lifetime for these keys. To stay compliant of the 2^-32 probability limit, we continue to use a 128-bit key modifier and 96-bit nonce, which radically extends the usable operation count for any given `K_M`. For simplicity of design we share the KDF code path between CBC and GCM operations, and since AAD is already considered in the KDF there's no need to forward it to the GCM routine.

Context headers in ASP.NET Core

Article • 06/03/2022

Background and theory

In the data protection system, a "key" means an object that can provide authenticated encryption services. Each key is identified by a unique id (a GUID), and it carries with it algorithmic information and entropic material. It's intended that each key carry unique entropy, but the system cannot enforce that, and we also need to account for developers who might change the key ring manually by modifying the algorithmic information of an existing key in the key ring. To achieve our security requirements given these cases the data protection system has a concept of cryptographic agility, which allows securely using a single entropic value across multiple cryptographic algorithms.

Most systems which support cryptographic agility do so by including some identifying information about the algorithm inside the payload. The algorithm's OID is generally a good candidate for this. However, one problem that we ran into is that there are multiple ways to specify the same algorithm: "AES" (CNG) and the managed Aes, AesManaged, AesCryptoServiceProvider, AesCng, and RijndaelManaged (given specific parameters) classes are all actually the same thing, and we'd need to maintain a mapping of all of these to the correct OID. If a developer wanted to provide a custom algorithm (or even another implementation of AES!), they'd have to tell us its OID. This extra registration step makes system configuration particularly painful.

Stepping back, we decided that we were approaching the problem from the wrong direction. An OID tells you what the algorithm is, but we don't actually care about this. If we need to use a single entropic value securely in two different algorithms, it's not necessary for us to know what the algorithms actually are. What we actually care about is how they behave. Any decent symmetric block cipher algorithm is also a strong pseudorandom permutation (PRP): fix the inputs (key, chaining mode, IV, plaintext) and the ciphertext output will with overwhelming probability be distinct from any other symmetric block cipher algorithm given the same inputs. Similarly, any decent keyed hash function is also a strong pseudorandom function (PRF), and given a fixed input set its output will overwhelmingly be distinct from any other keyed hash function.

We use this concept of strong PRPs and PRFs to build up a context header. This context header essentially acts as a stable thumbprint over the algorithms in use for any given operation, and it provides the cryptographic agility needed by the data protection system. This header is reproducible and is used later as part of the subkey derivation

process. There are two different ways to build the context header depending on the modes of operation of the underlying algorithms.

CBC-mode encryption + HMAC authentication

The context header consists of the following components:

- [16 bits] The value 00 00, which is a marker meaning "CBC encryption + HMAC authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The key length (in bytes, big-endian) of the HMAC algorithm. (Currently the key size always matches the digest size.)
- [32 bits] The digest size (in bytes, big-endian) of the HMAC algorithm.
- `EncCBC(K_E, IV, "")`, which is the output of the symmetric block cipher algorithm given an empty string input and where IV is an all-zero vector. The construction of `K_E` is described below.
- `MAC(K_H, "")`, which is the output of the HMAC algorithm given an empty string input. The construction of `K_H` is described below.

Ideally, we could pass all-zero vectors for `K_E` and `K_H`. However, we want to avoid the situation where the underlying algorithm checks for the existence of weak keys before performing any operations (notably DES and 3DES), which precludes using a simple or repeatable pattern like an all-zero vector.

Instead, we use the NIST SP800-108 KDF in Counter Mode (see NIST SP800-108, Sec. 5.1) with a zero-length key, label, and context and HMACSHA512 as the underlying PRF. We derive `| K_E | + | K_H |` bytes of output, then decompose the result into `K_E` and `K_H` themselves. Mathematically, this is represented as follows.

```
( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")
```

Example: AES-192-CBC + HMACSHA256

As an example, consider the case where the symmetric block cipher algorithm is AES-192-CBC and the validation algorithm is HMACSHA256. The system would generate the context header using the following steps.

First, let `( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 192 bits` and `| K_H | = 256 bits` per the specified algorithms. This leads to `K_E = 5BB6..21DD` and `K_H = A04A..00A9` in the example below:

```
5B B6 C9 83 13 78 22 1D 8E 10 73 CA CF 65 8E B0
61 62 42 71 CB 83 21 DD A0 4A 05 00 5B AB C0 A2
49 6F A5 61 E3 E2 49 87 AA 63 55 CD 74 0A DA C4
B7 92 3D BF 59 90 00 A9
```

Next, compute `Enc_CBC (K_E, IV, "")` for AES-192-CBC given `IV = 0*` and `K_E` as above.

`result := F474B1872B3B53E4721DE19C0841DB6F`

Next, compute `MAC(K_H, "")` for HMACSHA256 given `K_H` as above.

`result := D4791184B996092EE1202F36E8608FA8FBD98ABDFF5402F264B1D7211536220C`

This produces the full context header below:

```
00 00 00 00 00 18 00 00 00 10 00 00 00 20 00 00
00 20 F4 74 B1 87 2B 3B 53 E4 72 1D E1 9C 08 41
DB 6F D4 79 11 84 B9 96 09 2E E1 20 2F 36 E8 60
8F A8 FB D9 8A BD FF 54 02 F2 64 B1 D7 21 15 36
22 0C
```

This context header is the thumbprint of the authenticated encryption algorithm pair (AES-192-CBC encryption + HMACSHA256 validation). The components, as described above are:

- the marker `(00 00)`
- the block cipher key length `(00 00 00 18)`
- the block cipher block size `(00 00 00 10)`
- the HMAC key length `(00 00 00 20)`

- the HMAC digest size `(00 00 00 20)`
- the block cipher PRP output `(F4 74 - DB 6F)` and
- the HMAC PRF output `(D4 79 - end)`.

> ⓘ **Note**
>
> The CBC-mode encryption + HMAC authentication context header is built the same way regardless of whether the algorithms implementations are provided by Windows CNG or by managed SymmetricAlgorithm and KeyedHashAlgorithm types. This allows applications running on different operating systems to reliably produce the same context header even though the implementations of the algorithms differ between OSes. (In practice, the KeyedHashAlgorithm doesn't have to be a proper HMAC. It can be any keyed hash algorithm type.)

Example: 3DES-192-CBC + HMACSHA1

First, let `( K_E || K_H ) = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 192 bits` and `| K_H | = 160 bits` per the specified algorithms. This leads to `K_E = A219..E2BB` and `K_H = DC4A..B464` in the example below:

```
A2 19 60 2F 83 A9 13 EA B0 61 3A 39 B8 A6 7E 22
61 D9 F8 6C 10 51 E2 BB DC 4A 00 D7 03 A2 48 3E
D1 F7 5A 34 EB 28 3E D7 D4 67 B4 64
```

Next, compute `Enc_CBC (K_E, IV, "")` for 3DES-192-CBC given `IV = 0*` and `K_E` as above.

`result := ABB100F81E53E10E`

Next, compute `MAC(K_H, "")` for HMACSHA1 given `K_H` as above.

`result := 76EB189B35CF03461DDF877CD9F4B1B4D63A7555`

This produces the full context header which is a thumbprint of the authenticated encryption algorithm pair (3DES-192-CBC encryption + HMACSHA1 validation), shown below:

```
00 00 00 00 00 18 00 00 00 08 00 00 00 14 00 00
00 14 AB B1 00 F8 1E 53 E1 0E 76 EB 18 9B 35 CF
03 46 1D DF 87 7C D9 F4 B1 B4 D6 3A 75 55
```

The components break down as follows:

- the marker `(00 00)`
- the block cipher key length `(00 00 00 18)`
- the block cipher block size `(00 00 00 08)`
- the HMAC key length `(00 00 00 14)`
- the HMAC digest size `(00 00 00 14)`
- the block cipher PRP output `(AB B1 - E1 0E)` and
- the HMAC PRF output `(76 EB - end)`.

Galois/Counter Mode encryption + authentication

The context header consists of the following components:

- [16 bits] The value 00 01, which is a marker meaning "GCM encryption + authentication".
- [32 bits] The key length (in bytes, big-endian) of the symmetric block cipher algorithm.
- [32 bits] The nonce size (in bytes, big-endian) used during authenticated encryption operations. (For our system, this is fixed at nonce size = 96 bits.)
- [32 bits] The block size (in bytes, big-endian) of the symmetric block cipher algorithm. (For GCM, this is fixed at block size = 128 bits.)
- [32 bits] The authentication tag size (in bytes, big-endian) produced by the authenticated encryption function. (For our system, this is fixed at tag size = 128 bits.)
- [128 bits] The tag of `Enc_GCM (K_E, nonce, "")`, which is the output of the symmetric block cipher algorithm given an empty string input and where nonce is a 96-bit all-zero vector.

`K_E` is derived using the same mechanism as in the CBC encryption + HMAC authentication scenario. However, since there's no `K_H` in play here, we essentially have `| K_H | = 0`, and the algorithm collapses to the below form.

```
K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")
```

Example: AES-256-GCM

First, let `K_E = SP800_108_CTR(prf = HMACSHA512, key = "", label = "", context = "")`, where `| K_E | = 256 bits`.

```
K_E := 22BC6F1B171C08C4AE2F27444AF8FC8B3087A90006CAEA91FDCFB47C1B8733B8
```

Next, compute the authentication tag of `Enc_GCM (K_E, nonce, "")` for AES-256-GCM given `nonce = 096` and `K_E` as above.

```
result := E7DCCE66DF855A323A6BB7BD7A59BE45
```

This produces the full context header below:

```

00 01 00 00 00 20 00 00 00 0C 00 00 00 10 00 00
00 10 E7 DC CE 66 DF 85 5A 32 3A 6B B7 BD 7A 59
BE 45
```

The components break down as follows:

- the marker `(00 01)`
- the block cipher key length `(00 00 00 20)`
- the nonce size `(00 00 00 0C)`
- the block cipher block size `(00 00 00 10)`
- the authentication tag size `(00 00 00 10)` and
- the authentication tag from running the block cipher `(E7 DC - end)`.

Key management in ASP.NET Core

Article • 06/03/2022

The data protection system automatically manages the lifetime of master keys used to protect and unprotect payloads. Each key can exist in one of four stages:

- Created - the key exists in the key ring but has not yet been activated. The key shouldn't be used for new Protect operations until sufficient time has elapsed that the key has had a chance to propagate to all machines that are consuming this key ring.
- Active - the key exists in the key ring and should be used for all new Protect operations.
- Expired - the key has run its natural lifetime and should no longer be used for new Protect operations.
- Revoked - the key is compromised and must not be used for new Protect operations.

Created, active, and expired keys may all be used to unprotect incoming payloads. Revoked keys by default may not be used to unprotect payloads, but the application developer can override this behavior if necessary.

> ⚠ **Warning**
>
> The developer might be tempted to delete a key from the key ring (for example by deleting the corresponding file from the file system). At that point, all data protected by the key is permanently undecipherable, and there's no emergency override like there's with revoked keys. Deleting a key is truly destructive behavior.

Default key selection

When the data protection system reads the key ring from the backing repository, it will attempt to locate a "default" key from the key ring. The default key is used for new Protect operations.

The general heuristic is that the data protection system chooses the key with the most recent activation date as the default key. (There's a small fudge factor to allow for server-to-server clock skew.) If the key is expired or revoked, and if the application has

not disabled automatic key generation, then a new key will be generated with immediate activation per the key expiration and rolling policy below.

The reason the data protection system generates a new key immediately rather than falling back to a different key is that new key generation should be treated as an implicit expiration of all keys that were activated prior to the new key. The general idea is that new keys may have been configured with different algorithms or encryption-at-rest mechanisms than old keys, and the system should prefer the current configuration over falling back.

There's an exception. If the application developer has disabled automatic key generation, then the data protection system must choose something as the default key. In this fallback scenario, the system will choose the non-revoked key with the most recent activation date, with preference given to keys that have had time to propagate to other machines in the cluster. The fallback system may end up choosing an expired default key as a result. The fallback system will never choose a revoked key as the default key, and if the key ring is empty or every key has been revoked then the system will produce an error upon initialization.

Key expiration and rolling

When a key is created, it's automatically given an activation date of { now + 2 days } and an expiration date of { now + 90 days }. The 2-day delay before activation gives the key time to propagate through the system. That is, it allows other applications pointing at the backing store to observe the key at their next auto-refresh period, thus maximizing the chances that when the key ring does become active it has propagated to all applications that might need to use it.

If the default key will expire within 2 days and if the key ring doesn't already have a key that will be active upon expiration of the default key, then the data protection system will automatically persist a new key to the key ring. This new key has an activation date of { default key's expiration date } and an expiration date of { now + 90 days }. This allows the system to automatically roll keys on a regular basis with no interruption of service.

There might be circumstances where a key will be created with immediate activation. One example would be when the application hasn't run for a time and all keys in the key ring are expired. When this happens, the key is given an activation date of { now } without the normal 2-day activation delay.

The default key lifetime is 90 days, though this is configurable as in the following example.

C#

```
services.AddDataProtection()
       // use 14-day lifetime instead of 90-day lifetime
       .SetDefaultKeyLifetime(TimeSpan.FromDays(14));
```

An administrator can also change the default system-wide, though an explicit call to `SetDefaultKeyLifetime` will override any system-wide policy. The default key lifetime cannot be shorter than 7 days.

Automatic key ring refresh

When the data protection system initializes, it reads the key ring from the underlying repository and caches it in memory. This cache allows Protect and Unprotect operations to proceed without hitting the backing store. The system will automatically check the backing store for changes approximately every 24 hours or when the current default key expires, whichever comes first.

> ⚠ **Warning**
>
> Developers should very rarely (if ever) need to use the key management APIs directly. The data protection system will perform automatic key management as described above.

The data protection system exposes an interface `IKeyManager` that can be used to inspect and make changes to the key ring. The DI system that provided the instance of `IDataProtectionProvider` can also provide an instance of `IKeyManager` for your consumption. Alternatively, you can pull the `IKeyManager` straight from the `IServiceProvider` as in the example below.

Any operation which modifies the key ring (creating a new key explicitly or performing a revocation) will invalidate the in-memory cache. The next call to `Protect` or `Unprotect` will cause the data protection system to reread the key ring and recreate the cache.

The sample below demonstrates using the `IKeyManager` interface to inspect and manipulate the key ring, including revoking existing keys and generating a new key manually.

C#

```
using System;
using System.IO;
```

```
using System.Threading;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.Extensions.DependencyInjection;

public class Program
{
    public static void Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddDataProtection()
            // point at a specific folder and use DPAPI to encrypt keys
            .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys"))
            .ProtectKeysWithDpapi();
        var services = serviceCollection.BuildServiceProvider();

        // perform a protect operation to force the system to put at least
        // one key in the key ring

services.GetDataProtector("Sample.KeyManager.v1").Protect("payload");
        Console.WriteLine("Performed a protect operation.");
        Thread.Sleep(2000);

        // get a reference to the key manager
        var keyManager = services.GetService<IKeyManager>();

        // list all keys in the key ring
        var allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created =
{key.CreationDate:u}, IsRevoked = {key.IsRevoked}");
        }

        // revoke all keys in the key ring
        keyManager.RevokeAllKeys(DateTimeOffset.Now, reason: "Revocation
reason here.");
        Console.WriteLine("Revoked all existing keys.");

        // add a new key to the key ring with immediate activation and a 1-
month expiration
        keyManager.CreateNewKey(
            activationDate: DateTimeOffset.Now,
            expirationDate: DateTimeOffset.Now.AddMonths(1));
        Console.WriteLine("Added a new key.");

        // list all keys in the key ring
        allKeys = keyManager.GetAllKeys();
        Console.WriteLine($"The key ring contains {allKeys.Count} key(s).");
        foreach (var key in allKeys)
        {
            Console.WriteLine($"Key {key.KeyId:B}: Created =
{key.CreationDate:u}, IsRevoked = {key.IsRevoked}");
        }
```

```
    }
}

/*
 * SAMPLE OUTPUT
 *
 * Performed a protect operation.
 * The key ring contains 1 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18
22:20:49Z, IsRevoked = False
 * Revoked all existing keys.
 * Added a new key.
 * The key ring contains 2 key(s).
 * Key {1b948618-be1f-440b-b204-64ff5a152552}: Created = 2015-03-18
22:20:49Z, IsRevoked = True
 * Key {2266fc40-e2fb-48c6-8ce2-5fde6b1493f7}: Created = 2015-03-18
22:20:51Z, IsRevoked = False
 */
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

Key storage

The data protection system has a heuristic whereby it attempts to deduce an appropriate key storage location and encryption-at-rest mechanism automatically. The key persistence mechanism is also configurable by the app developer. The following documents discuss the in-box implementations of these mechanisms:

- Key storage providers in ASP.NET Core
- Key encryption at rest in Windows and Azure using ASP.NET Core

Key storage providers in ASP.NET Core

Article • 10/30/2024

The data protection system employs a discovery mechanism by default to determine where cryptographic keys should be persisted. The developer can override the default discovery mechanism and manually specify the location.

> ⚠ **Warning**
>
> If you specify an explicit key persistence location, the data protection system deregisters the default key encryption at rest mechanism, so keys are no longer encrypted at rest. It's recommended that you additionally specify an explicit key encryption mechanism for production deployments.

File system

To configure a file system-based key repository, call the PersistKeysToFileSystem configuration routine as shown below. Provide a DirectoryInfo pointing to the repository where keys should be stored:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo(@"c:\temp-keys\"));
}
```

The `DirectoryInfo` can point to a directory on the local machine, or it can point to a folder on a network share. If pointing to a directory on the local machine (and the scenario is that only apps on the local machine require access to use this repository), consider using Windows DPAPI (on Windows) to encrypt the keys at rest. Otherwise, consider using an X.509 certificate to encrypt keys at rest.

Azure Storage

The Azure.Extensions.AspNetCore.DataProtection.Blobs package allows storing data protection keys in Azure Blob Storage. Keys can be shared across several instances of a web app. Apps can share authentication cookies or CSRF protection across multiple servers.

To configure the Azure Blob Storage provider, call one of the PersistKeysToAzureBlobStorage overloads.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blob URI including SAS
token>"));
}
```

If the web app is running as an Azure service, connection string can be used to authenticate to Azure storage by using Azure.Storage.Blobs.

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

C#

```
string connectionString = "<connection_string>";
string containerName = "my-key-container";
string blobName = "keys.xml";
BlobContainerClient container = new BlobContainerClient(connectionString,
containerName);

// optional - provision the container automatically
await container.CreateIfNotExistsAsync();

BlobClient blobClient = container.GetBlobClient(blobName);

services.AddDataProtection()
    .PersistKeysToAzureBlobStorage(blobClient);
```

> ⓘ **Note**
>
> The connection string to your storage account can be found in the Azure Portal under the "Access Keys" section or by running the following CLI command:

Bash

```
az storage account show-connection-string --name <account_name> --
resource-group <resource_group>
```

Redis

The Microsoft.AspNetCore.DataProtection.StackExchangeRedis package allows storing data protection keys in a Redis cache. Keys can be shared across several instances of a web app. Apps can share authentication cookies or CSRF protection across multiple servers.

To configure on Redis, call one of the PersistKeysToStackExchangeRedis overloads:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    var redis = ConnectionMultiplexer.Connect("<URI>");
    services.AddDataProtection()
        .PersistKeysToStackExchangeRedis(redis, "DataProtection-Keys");
}
```

For more information, see the following topics:

- StackExchange.Redis ConnectionMultiplexer
- Azure Redis Cache
- ASP.NET Core DataProtection samples

Registry

Only applies to Windows deployments.

Sometimes the app might not have write access to the file system. Consider a scenario where an app is running as a virtual service account (such as *w3wp.exe*'s app pool identity). In these cases, the administrator can provision a registry key that's accessible by the service account identity. Call the PersistKeysToRegistry extension method as shown below. Provide a RegistryKey pointing to the location where cryptographic keys should be stored:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()

.PersistKeysToRegistry(Registry.CurrentUser.OpenSubKey(@"SOFTWARE\Sample\keys", true));
}
```

> **ⓘ Important**
>
> We recommend using **Windows DPAPI** to encrypt the keys at rest.

Entity Framework Core

The Microsoft.AspNetCore.DataProtection.EntityFrameworkCore package provides a mechanism for storing data protection keys to a database using Entity Framework Core. The `Microsoft.AspNetCore.DataProtection.EntityFrameworkCore` NuGet package must be added to the project file, it's not part of the Microsoft.AspNetCore.App metapackage.

With this package, keys can be shared across multiple instances of a web app.

To configure the EF Core provider, call the PersistKeysToDbContext method:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    // Add a DbContext to store your Database Keys
    services.AddDbContext<MyKeysContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("MyKeysConnection")));

    // using Microsoft.AspNetCore.DataProtection;
    services.AddDataProtection()
        .PersistKeysToDbContext<MyKeysContext>();

    services.AddDefaultIdentity<IdentityUser>()
```

```
        .AddDefaultUI(UIFramework.Bootstrap4)
        .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

The generic parameter, `TContext`, must inherit from DbContext and implement IDataProtectionKeyContext:

C#

```
using Microsoft.AspNetCore.DataProtection.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
using WebApp1.Data;

namespace WebApp1
{
    class MyKeysContext : DbContext, IDataProtectionKeyContext
    {
        // A recommended constructor overload when using EF Core
        // with dependency injection.
        public MyKeysContext(DbContextOptions<MyKeysContext> options)
            : base(options) { }

        // This maps to the table that stores keys.
        public DbSet<DataProtectionKey> DataProtectionKeys { get; set; }
    }
}
```

Create the `DataProtectionKeys` table.

Visual Studio

Execute the following commands in the **Package Manager Console** (PMC) window:

PowerShell

```
Add-Migration AddDataProtectionKeys -Context MyKeysContext
Update-Database -Context MyKeysContext
```

`MyKeysContext` is the `DbContext` defined in the preceding code sample. If you're using a `DbContext` with a different name, substitute your `DbContext` name for `MyKeysContext`.

The `DataProtectionKeys` class/entity adopts the structure shown in the following table.

Expand table

Property/Field	CLR Type	SQL Type
`Id`	`int`	`int`, PK, `IDENTITY(1,1)`, not null
`FriendlyName`	`string`	`nvarchar(MAX)`, null
`Xml`	`string`	`nvarchar(MAX)`, null

Custom key repository

If the in-box mechanisms aren't appropriate, the developer can specify their own key persistence mechanism by providing a custom IXmlRepository.

Key encryption at rest in Windows and Azure using ASP.NET Core

Article • 03/17/2023

The data protection system employs a discovery mechanism by default to determine how cryptographic keys should be encrypted at rest. The developer can override the discovery mechanism and manually specify how keys should be encrypted at rest.

> ⚠ **Warning**
>
> If you specify an explicit **key persistence location**, the data protection system deregisters the default key encryption at rest mechanism. Consequently, keys are no longer encrypted at rest. We recommend that you **specify an explicit key encryption mechanism** for production deployments. The encryption-at-rest mechanism options are described in this topic.

Azure Key Vault

To store keys in Azure Key Vault, configure the system with ProtectKeysWithAzureKeyVault in the `Startup` class:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()
        .PersistKeysToAzureBlobStorage(new Uri("<blobUriWithSasToken>"))
        .ProtectKeysWithAzureKeyVault("<keyIdentifier>", "<clientId>", "
<clientSecret>");
}
```

For more information, see Configure ASP.NET Core Data Protection: ProtectKeysWithAzureKeyVault.

Windows DPAPI

Only applies to Windows deployments.

When Windows DPAPI is used, key material is encrypted with CryptProtectData before being persisted to storage. DPAPI is an appropriate encryption mechanism for data

that's never read outside of the current machine (though it's possible to back these keys up to Active Directory). To configure DPAPI key-at-rest encryption, call one of the ProtectKeysWithDpapi) extension methods:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    // Only the local user account can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi();
}
```

If `ProtectKeysWithDpapi` is called with no parameters, only the current Windows user account can decipher the persisted key ring. You can optionally specify that any user account on the machine (not just the current user account) be able to decipher the key ring:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    // All user accounts on the machine can decrypt the keys
    services.AddDataProtection()
        .ProtectKeysWithDpapi(protectToLocalMachine: true);
}
```

X.509 certificate

If the app is spread across multiple machines, it may be convenient to distribute a shared X.509 certificate across the machines and configure the hosted apps to use the certificate for encryption of keys at rest:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()

.ProtectKeysWithCertificate("3BCE558E2AD3E0E34A7743EAB5AEA2A9BD2575A0");
}
```

Due to .NET Framework limitations, only certificates with CAPI private keys are supported. See the content below for possible workarounds to these limitations.

Windows DPAPI-NG

This mechanism is available only on Windows 8/Windows Server 2012 or later.

Beginning with Windows 8, Windows OS supports DPAPI-NG (also called CNG DPAPI). For more information, see About CNG DPAPI.

The principal is encoded as a protection descriptor rule. In the following example that calls ProtectKeysWithDpapiNG, only the domain-joined user with the specified SID can decrypt the key ring:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Uses the descriptor rule "SID=S-1-5-21-..."
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG("SID=S-1-5-21-...",
        flags: DpapiNGProtectionDescriptorFlags.None);
}
```

There's also a parameterless overload of `ProtectKeysWithDpapiNG`. Use this convenience method to specify the rule "SID={CURRENT_ACCOUNT_SID}", where *CURRENT_ACCOUNT_SID* is the SID of the current Windows user account:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Use the descriptor rule "SID={current account SID}"
    services.AddDataProtection()
        .ProtectKeysWithDpapiNG();
}
```

In this scenario, the AD domain controller is responsible for distributing the encryption keys used by the DPAPI-NG operations. The target user can decipher the encrypted payload from any domain-joined machine (provided that the process is running under their identity).

Certificate-based encryption with Windows DPAPI-NG

If the app is running on Windows 8.1/Windows Server 2012 R2 or later, you can use Windows DPAPI-NG to perform certificate-based encryption. Use the rule descriptor

string "CERTIFICATE=HashId:THUMBPRINT", where *THUMBPRINT* is the hex-encoded SHA1 thumbprint of the certificate:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDataProtection()

.ProtectKeysWithDpapiNG("CERTIFICATE=HashId:3BCE558E2...B5AEA2A9BD2575A0",
            flags: DpapiNGProtectionDescriptorFlags.None);
}
```

Any app pointed at this repository must be running on Windows 8.1/Windows Server 2012 R2 or later to decipher the keys.

Custom key encryption

If the in-box mechanisms aren't appropriate, the developer can specify their own key encryption mechanism by providing a custom IXmlEncryptor.

Key immutability and key settings in ASP.NET Core

Article • 06/03/2022

Once an object is persisted to the backing store, its representation is forever fixed. New data can be added to the backing store, but existing data can never be mutated. The primary purpose of this behavior is to prevent data corruption.

One consequence of this behavior is that once a key is written to the backing store, it's immutable. Its creation, activation, and expiration dates can never be changed, though it can revoked by using `IKeyManager`. Additionally, its underlying algorithmic information, master keying material, and encryption at rest properties are also immutable.

If the developer changes any setting that affects key persistence, those changes won't go into effect until the next time a key is generated, either via an explicit call to `IKeyManager.CreateNewKey` or via the data protection system's own automatic key generation behavior. The settings that affect key persistence are as follows:

- The default key lifetime
- The key encryption at rest mechanism
- The algorithmic information contained within the key

If you need these settings to kick in earlier than the next automatic key rolling time, consider making an explicit call to `IKeyManager.CreateNewKey` to force the creation of a new key. Remember to provide an explicit activation date ({ now + 2 days } is a good rule of thumb to allow time for the change to propagate) and expiration date in the call.

> **Tip**
>
> All applications touching the repository should specify the same settings with the `IDataProtectionBuilder` extension methods. Otherwise, the properties of the persisted key will be dependent on the particular application that invoked the key generation routines.

Key storage format in ASP.NET Core

Article • 06/03/2022

Objects are stored at rest in XML representation. The default directory for key storage is:

- Windows: *%LOCALAPPDATA%\ASP.NET\DataProtection-Keys*
- macOS / Linux: *$HOME/.aspnet/DataProtection-Keys*

The <key> element

Keys exist as top-level objects in the key repository. By convention keys have the filename `key-{guid}.xml`, where {guid} is the id of the key. Each such file contains a single key. The format of the file is as follows.

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<key id="aaaaaaaa-0b0b-1c1c-2d2d-333333333333" version="1">
  <creationDate>2015-03-19T23:32:02.3949887Z</creationDate>
  <activationDate>2015-03-19T23:32:02.3839429Z</activationDate>
  <expirationDate>2015-06-17T23:32:02.3839429Z</expirationDate>
  <descriptor deserializerType="{deserializerType}">
    <descriptor>
      <encryption algorithm="AES_256_CBC" />
      <validation algorithm="HMACSHA256" />
      <enc:encryptedSecret decryptorType="{decryptorType}" xmlns:enc="...">
        <encryptedKey>
          <!-- This key is encrypted with Windows DPAPI. -->
          <value>AQAAANCM...8/zeP8lcwAg==</value>
        </encryptedKey>
      </enc:encryptedSecret>
    </descriptor>
  </descriptor>
</key>
```

The <key> element contains the following attributes and child elements:

- The key id. This value is treated as authoritative; the filename is simply a nicety for human readability.
- The version of the <key> element, currently fixed at 1.
- The key's creation, activation, and expiration dates.
- A <descriptor> element, which contains information on the authenticated encryption implementation contained within this key.

In the above example, the key's id is {aaaaaaaa-0b0b-1c1c-2d2d-333333333333}, it was created and activated on March 19, 2015, and it has a lifetime of 90 days. (Occasionally the activation date might be slightly before the creation date as in this example. This is due to a nit in how the APIs work and is harmless in practice.)

The <descriptor> element

The outer <descriptor> element contains an attribute deserializerType, which is the assembly-qualified name of a type which implements IAuthenticatedEncryptorDescriptorDeserializer. This type is responsible for reading the inner <descriptor> element and for parsing the information contained within.

The particular format of the <descriptor> element depends on the authenticated encryptor implementation encapsulated by the key, and each deserializer type expects a slightly different format for this. In general, though, this element will contain algorithmic information (names, types, OIDs, or similar) and secret key material. In the above example, the descriptor specifies that this key wraps AES-256-CBC encryption + HMACSHA256 validation.

The <encryptedSecret> element

An **<encryptedSecret>** element which contains the encrypted form of the secret key material may be present if encryption of secrets at rest is enabled. The attribute `decryptorType` is the assembly-qualified name of a type which implements IXmlDecryptor. This type is responsible for reading the inner **<encryptedKey>** element and decrypting it to recover the original plaintext.

As with `<descriptor>`, the particular format of the `<encryptedSecret>` element depends on the at-rest encryption mechanism in use. In the above example, the master key is encrypted using Windows DPAPI per the comment.

The <revocation> element

Revocations exist as top-level objects in the key repository. By convention revocations have the filename *`revocation-{timestamp}.xml`* (for revoking all keys before a specific date) or *`revocation-{guid}.xml`* (for revoking a specific key). Each file contains a single <revocation> element.

For revocations of individual keys, the file contents will be as below.

```
XML
```

```xml
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T22:45:30.2616742Z</revocationDate>
  <key id="bbbbbbbb-1c1c-2d2d-3e3e-444444444444" />
  <reason>human-readable reason</reason>
</revocation>
```

In this case, only the specified key is revoked. If the key id is "*", however, as in the below example, all keys whose creation date is prior to the specified revocation date are revoked.

XML

```xml
<?xml version="1.0" encoding="utf-8"?>
<revocation version="1">
  <revocationDate>2015-03-20T15:45:45.7366491-07:00</revocationDate>
  <!-- All keys created before the revocation date are revoked. -->
  <key id="*" />
  <reason>human-readable reason</reason>
</revocation>
```

The <reason> element is never read by the system. It's simply a convenient place to store a human-readable reason for revocation.

Ephemeral data protection providers in ASP.NET Core

Article • 02/28/2023

There are scenarios where an application needs a throwaway `IDataProtectionProvider`. For example, the developer might just be experimenting in a one-off console application, or the application itself is transient (it's scripted or a unit test project). To support these scenarios the Microsoft.AspNetCore.DataProtection package includes a type `EphemeralDataProtectionProvider`. This type provides a basic implementation of `IDataProtectionProvider` whose key repository is held solely in-memory and isn't written out to any backing store.

Each instance of `EphemeralDataProtectionProvider` uses its own unique primary key. Therefore, if an `IDataProtector` rooted at an `EphemeralDataProtectionProvider` generates a protected payload, that payload can only be unprotected by an equivalent `IDataProtector` (given the same purpose chain) rooted at the same `EphemeralDataProtectionProvider` instance.

The following sample demonstrates instantiating an `EphemeralDataProtectionProvider` and using it to protect and unprotect data.

C#

```
using System;
using Microsoft.AspNetCore.DataProtection;

public class Program
{
    public static void Main(string[] args)
    {
        const string purpose = "Ephemeral.App.v1";

        // create an ephemeral provider and demonstrate that it can round-
trip a payload
        var provider = new EphemeralDataProtectionProvider();
        var protector = provider.CreateProtector(purpose);
        Console.Write("Enter input: ");
        string input = Console.ReadLine();

        // protect the payload
        string protectedPayload = protector.Protect(input);
        Console.WriteLine($"Protect returned: {protectedPayload}");

        // unprotect the payload
        string unprotectedPayload = protector.Unprotect(protectedPayload);
```

```
        Console.WriteLine($"Unprotect returned: {unprotectedPayload}");

        // if I create a new ephemeral provider, it won't be able to
unprotect existing
        // payloads, even if I specify the same purpose
        provider = new EphemeralDataProtectionProvider();
        protector = provider.CreateProtector(purpose);
        unprotectedPayload = protector.Unprotect(protectedPayload); //
THROWS
    }
}

/*
* SAMPLE OUTPUT
*
* Enter input: Hello!
* Protect returned: CfDJ8AAAAAAAAAAAAAAAAAAA...uGoxWLjGKtm1SkNACQ
* Unprotect returned: Hello!
* << throws CryptographicException >>
*/
```

Compatibility in ASP.NET Core

Article • 06/03/2022

- Replacing ASP.NET <machineKey> in ASP.NET Core

Replace the ASP.NET machineKey in ASP.NET Core

Article • 06/03/2022

The implementation of the `<machineKey>` element in ASP.NET is replaceable. This allows most calls to ASP.NET cryptographic routines to be routed through a replacement data protection mechanism, including the new data protection system.

Package installation

> **Note**
>
> The new data protection system can only be installed into an existing ASP.NET application targeting .NET 4.5.1 or later. Installation will fail if the application targets .NET 4.5 or lower.

To install the new data protection system into an existing ASP.NET 4.5.1+ project, install the package Microsoft.AspNetCore.DataProtection.SystemWeb. This will instantiate the data protection system using the default configuration settings.

When you install the package, it inserts a line into *Web.config* that tells ASP.NET to use it for most cryptographic operations, including forms authentication, view state, and calls to MachineKey.Protect. It does not use the data protection API. The line that's inserted reads as follows.

XML

```xml
<machineKey compatibilityMode="Framework45" dataProtectorType="..." />
```

> **Tip**
>
> You can tell if the new data protection system is active by inspecting fields like `__VIEWSTATE`, which should begin with "CfDJ8" as in the example below. "CfDJ8" is the base64 representation of the magic "09 F0 C9 F0" header that identifies a payload protected by the data protection system.

HTML

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="CfDJ8AWPr2EQPTBGs3L2GCZOpk...">
```

Package configuration

The data protection system is instantiated with a default zero-setup configuration. However, since by default keys are persisted to the local file system, this won't work for applications which are deployed in a farm. To resolve this, you can provide configuration by creating a type which subclasses DataProtectionStartup and overrides its ConfigureServices method.

Below is an example of a custom data protection startup type which configured both where keys are persisted and how they're encrypted at rest. It also overrides the default app isolation policy by providing its own application name.

C#

```
using System;
using System.IO;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.SystemWeb;
using Microsoft.Extensions.DependencyInjection;

namespace DataProtectionDemo
{
    public class MyDataProtectionStartup : DataProtectionStartup
    {
        public override void ConfigureServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("my-app")
                .PersistKeysToFileSystem(new
DirectoryInfo(@"\\server\share\myapp-keys\"))
                .ProtectKeysWithCertificate("thumbprint");
        }
    }
}
```

Tip

You can also use `<machineKey applicationName="my-app" ... />` in place of an explicit call to SetApplicationName. This is a convenience mechanism to avoid forcing the developer to create a DataProtectionStartup-derived type if all they wanted to configure was setting the application name.

To enable this custom configuration, go back to Web.config and look for the `<appSettings>` element that the package install added to the config file. It will look like the following markup:

XML

```
<appSettings>
  <!--
  If you want to customize the behavior of the ASP.NET Core Data Protection
stack, set the
  "aspnet:dataProtectionStartupType" switch below to be the fully-qualified
name of a
  type which subclasses
Microsoft.AspNetCore.DataProtection.SystemWeb.DataProtectionStartup.
  -->
  <add key="aspnet:dataProtectionStartupType" value="" />
</appSettings>
```

Fill in the blank value with the assembly-qualified name of the DataProtectionStartup-derived type you just created. If the name of the application is DataProtectionDemo, this would look like the below.

XML

```
<add key="aspnet:dataProtectionStartupType"
     value="DataProtectionDemo.MyDataProtectionStartup, DataProtectionDemo"
/>
```

The newly-configured data protection system is now ready for use inside the application.

Safe storage of app secrets in development in ASP.NET Core

Article • 10/30/2024

By Rick Anderson and Kirk Larkin

View or download sample code (how to download)

This article explains how to manage sensitive data for an ASP.NET Core app on a development machine. Never store passwords or other sensitive data in source code or configuration files. Production secrets shouldn't be used for development or test. Secrets shouldn't be deployed with the app. Production secrets should be accessed through a controlled means like Azure Key Vault. Azure test and production secrets can be stored and protected with the Azure Key Vault configuration provider.

For more information on authentication for deployed test and production apps, see Secure authentication flows.

To use user secrets in a .NET console app, see this GitHub issue.

Environment variables

Environment variables are used to avoid storage of app secrets in code or in local configuration files. Environment variables override configuration values for all previously specified configuration sources.

Consider an ASP.NET Core web app in which **Individual User Accounts** security is enabled. A default database connection string is included in the project's `appsettings.json` file with the key `DefaultConnection`. The default connection string is for LocalDB, which runs in user mode and doesn't require a password. During app deployment, the `DefaultConnection` key value can be overridden with an environment variable's value. The environment variable may store the complete connection string with sensitive credentials.

> ⚠ **Warning**
>
> Environment variables are generally stored in plain, unencrypted text. If the machine or process is compromised, environment variables can be accessed by untrusted parties. Additional measures to prevent disclosure of user secrets may be required.

The `:` separator doesn't work with environment variable hierarchical keys on all platforms. For example, the `:` separator is not supported by Bash. The double underscore, `__`, is:

- Supported by all platforms.
- Automatically replaced by a colon, `:`.

Secret Manager

The Secret Manager tool stores sensitive data during application development. In this context, a piece of sensitive data is an app secret. App secrets are stored in a separate location from the project tree. The app secrets are associated with a specific project or shared across several projects. The app secrets aren't checked into source control.

> ⚠ **Warning**
>
> The Secret Manager tool doesn't encrypt the stored secrets and shouldn't be treated as a trusted store. It's for development purposes only. The keys and values are stored in a JSON configuration file in the user profile directory.

How the Secret Manager tool works

The Secret Manager tool hides implementation details, such as where and how the values are stored. You can use the tool without knowing these implementation details. The values are stored in a JSON file in the local machine's user profile folder:

Windows

File system path:

```
%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json
```

In the preceding file paths, replace `<user_secrets_id>` with the `UserSecretsId` value specified in the project file.

Don't write code that depends on the location or format of data saved with the Secret Manager tool. These implementation details may change. For example, the secret values aren't encrypted.

Enable secret storage

The Secret Manager tool operates on project-specific configuration settings stored in your user profile.

Use the CLI

The Secret Manager tool includes an `init` command. To use user secrets, run the following command in the project directory:

.NET CLI

```
dotnet user-secrets init
```

The preceding command adds a `UserSecretsId` element within a `PropertyGroup` of the project file. By default, the inner text of `UserSecretsId` is a GUID. The inner text is arbitrary, but is unique to the project.

XML

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

Use Visual Studio

In Visual Studio, right-click the project in Solution Explorer, and select **Manage User Secrets** from the context menu. This gesture adds a `UserSecretsId` element, populated with a GUID, to the project file.

If `GenerateAssemblyInfo` is `false`

If the generation of assembly info attributes is disabled, manually add the UserSecretsIdAttribute in `AssemblyInfo.cs`. For example:

C#

```
[assembly: UserSecretsId("your_user_secrets_id")]
```

When manually adding the `UserSecretsId` attribute to `AssemblyInfo.cs`, the `UserSecretsId` value must match the value in the project file.

Set a secret

Define an app secret consisting of a key and its value. The secret is associated with the project's `UserSecretsId` value. For example, run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

In the preceding example, the colon denotes that `Movies` is an object literal with a `ServiceApiKey` property.

The Secret Manager tool can be used from other directories too. Use the `--project` option to supply the file system path at which the project file exists. For example:

.NET CLI

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345" --project
"C:\apps\WebApp1\src\WebApp1"
```

JSON structure flattening in Visual Studio

Visual Studio's **Manage User Secrets** gesture opens a `secrets.json` file in the text editor. Replace the contents of `secrets.json` with the key-value pairs to be stored. For example:

JSON

```
{
  "Movies": {
    "ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
    "ServiceApiKey": "12345"
  }
}
```

The JSON structure is flattened after modifications via `dotnet user-secrets remove` or `dotnet user-secrets set`. For example, running `dotnet user-secrets remove`

`"Movies:ConnectionString"` collapses the `Movies` object literal. The modified file resembles the following JSON:

JSON

```
{
  "Movies:ServiceApiKey": "12345"
}
```

Set multiple secrets

A batch of secrets can be set by piping JSON to the `set` command. In the following example, the `input.json` file's contents are piped to the `set` command.

Windows

Open a command shell, and execute the following command:

.NET CLI

```
type .\input.json | dotnet user-secrets set
```

Access a secret

To access a secret, complete the following steps:

1. Register the user secrets configuration source
2. Read the secret via the Configuration API

Register the user secrets configuration source

The user secrets configuration provider registers the appropriate configuration source with the .NET Configuration API.

ASP.NET Core web apps created with dotnet new or Visual Studio generate the following code:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
```

```
app.MapGet("/", () => "Hello World!");

app.Run();
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. The initialized `WebApplicationBuilder` (`builder`) provides default configuration and calls AddUserSecrets when the EnvironmentName is Development:

Read the secret via the Configuration API

Consider the following examples of reading the `Movies:ServiceApiKey` key:

Program.cs file:

C#

```
var builder = WebApplication.CreateBuilder(args);
var movieApiKey = builder.Configuration["Movies:ServiceApiKey"];

var app = builder.Build();

app.MapGet("/", () => movieApiKey);

app.Run();
```

Razor Pages page model:

C#

```
public class IndexModel : PageModel
{
    private readonly IConfiguration _config;

    public IndexModel(IConfiguration config)
    {
        _config = config;
    }

    public void OnGet()
    {
        var moviesApiKey = _config["Movies:ServiceApiKey"];

        // call Movies service with the API key
    }
}
```

For more information, see Configuration in ASP.NET Core.

Map secrets to a POCO

Mapping an entire object literal to a POCO (a simple .NET class with properties) is useful for aggregating related properties.

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

To map the preceding secrets to a POCO, use the .NET Configuration API's object graph binding feature. The following code binds to a custom `MovieSettings` POCO and accesses the `ServiceApiKey` property value:

C#

```
var moviesConfig =
    Configuration.GetSection("Movies").Get<MovieSettings>();
_moviesApiKey = moviesConfig.ServiceApiKey;
```

The `Movies:ConnectionString` and `Movies:ServiceApiKey` secrets are mapped to the respective properties in `MovieSettings`:

C#

```
public class MovieSettings
{
    public string ConnectionString { get; set; }

    public string ServiceApiKey { get; set; }
}
```

String replacement with secrets

Storing passwords in plain text is insecure. Never store secrets in a configuration file such as `appsettings.json`, which might get checked in to a source code repository.

For example, a database connection string stored in `appsettings.json` should not include a password. Instead, store the password as a secret, and include the password in

the connection string at runtime. For example:

.NET CLI

```
dotnet user-secrets set "DbPassword" "`<secret value>`"
```

Replace the `<secret value>` placeholder in the preceding example with the password value. Set the secret's value on a SqlConnectionStringBuilder object's Password property to include it as the password value in the connection string:

C#

```
using System.Data.SqlClient;

var builder = WebApplication.CreateBuilder(args);

var conStrBuilder = new SqlConnectionStringBuilder(
        builder.Configuration.GetConnectionString("Movies"));
conStrBuilder.Password = builder.Configuration["DbPassword"];
var connection = conStrBuilder.ConnectionString;

var app = builder.Build();

app.MapGet("/", () => connection);

app.Run();
```

List the secrets

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets list
```

The following output appears:

Console

```
Movies:ConnectionString = Server=(localdb)\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true
Movies:ServiceApiKey = 12345
```

In the preceding example, a colon in the key names denotes the object hierarchy within `secrets.json`.

Remove a single secret

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets remove "Movies:ConnectionString"
```

The app's `secrets.json` file was modified to remove the key-value pair associated with the `Movies:ConnectionString` key:

JSON

```
{
  "Movies": {
    "ServiceApiKey": "12345"
  }
}
```

`dotnet user-secrets list` displays the following message:

Console

```
Movies:ServiceApiKey = 12345
```

Remove all secrets

Assume the app's `secrets.json` file contains the following two secrets:

JSON

```
{
  "Movies:ConnectionString": "Server=(localdb)\\mssqllocaldb;Database=Movie-
1;Trusted_Connection=True;MultipleActiveResultSets=true",
  "Movies:ServiceApiKey": "12345"
}
```

Run the following command from the directory in which the project file exists:

.NET CLI

```
dotnet user-secrets clear
```

All user secrets for the app have been deleted from the `secrets.json` file:

JSON

```
{}
```

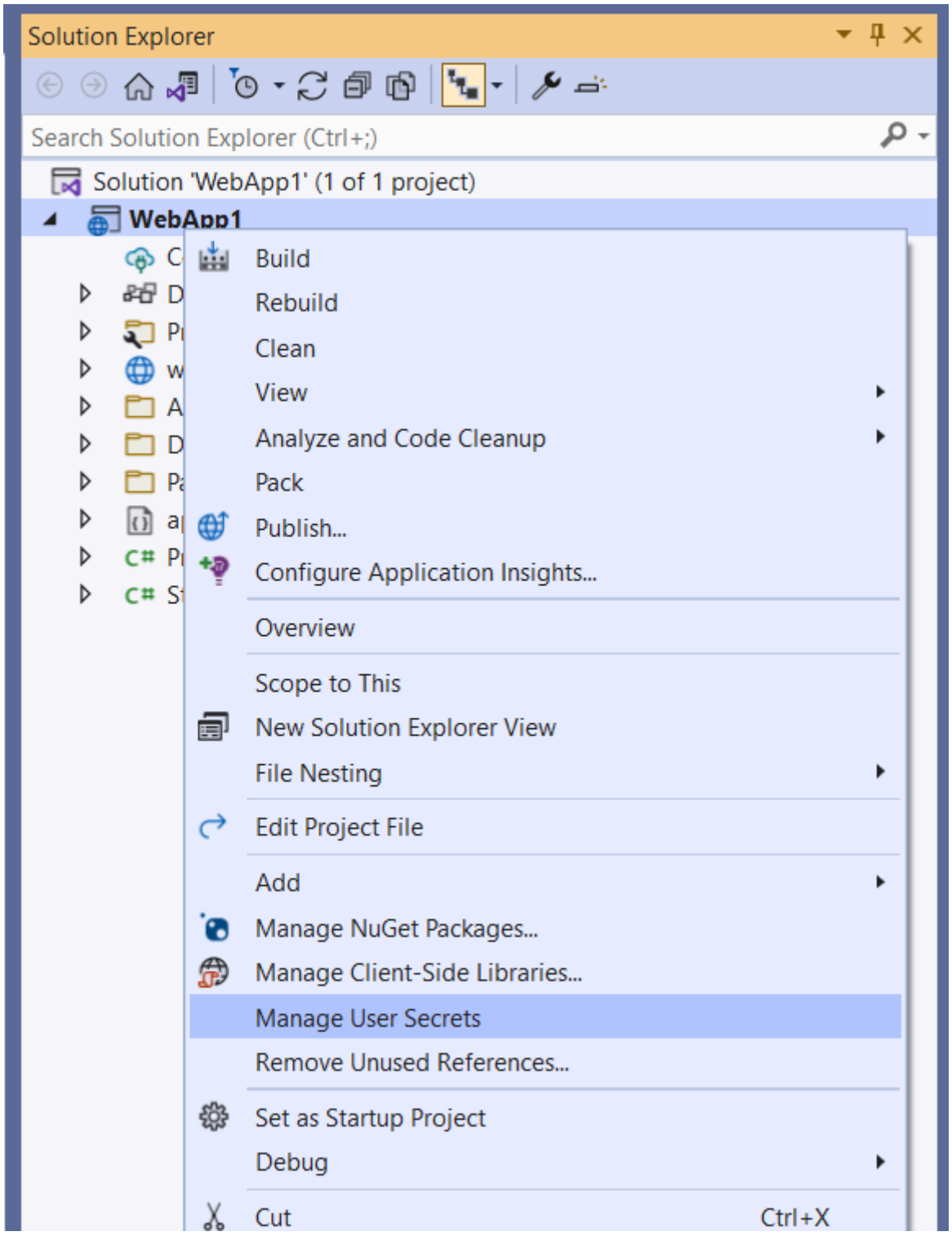
Running `dotnet user-secrets list` displays the following message:

Console

```
No secrets configured for this application.
```

Manage user secrets with Visual Studio

To manage user secrets in Visual Studio, right click the project in solution explorer and select **Manage User Secrets**:



Migrating User Secrets from ASP.NET Framework to ASP.NET Core

See this GitHub issue.

User secrets in non-web applications

Projects that target `Microsoft.NET.Sdk.Web` automatically include support for user secrets. For projects that target `Microsoft.NET.Sdk`, such as console applications, install

the configuration extension and user secrets NuGet packages explicitly.

Using PowerShell:

PowerShell

```
Install-Package Microsoft.Extensions.Configuration
Install-Package Microsoft.Extensions.Configuration.UserSecrets
```

Using the .NET CLI:

.NET CLI

```
dotnet add package Microsoft.Extensions.Configuration
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

Once the packages are installed, initialize the project and set secrets the same way as for a web app. The following example shows a console application that retrieves the value of a secret that was set with the key "AppSecret":

C#

```
using Microsoft.Extensions.Configuration;

namespace ConsoleApp;

class Program
{
    static void Main(string[] args)
    {
        IConfigurationRoot config = new ConfigurationBuilder()
            .AddUserSecrets<Program>()
            .Build();

        Console.WriteLine(config["AppSecret"]);
    }
}
```

Additional resources

- See this issue and this issue for information on accessing user secrets from IIS.
- Configuration in ASP.NET Core
- Azure Key Vault configuration provider in ASP.NET Core

Manage JSON Web Tokens in development with dotnet user-jwts

Article • 04/11/2023

By Rick Anderson

The `dotnet user-jwts` command line tool can create and manage app specific local JSON Web Tokens (JWTs).

Synopsis

.NET CLI

```
dotnet user-jwts [<PROJECT>] [command]
dotnet user-jwts [command] -h|--help
```

Description

Creates and manages project specific local JSON Web Tokens.

Arguments

`PROJECT | SOLUTION`

The MSBuild project to apply a command on. If a project is not specified, MSBuild searches the current working directory for a file that has a file extension that ends in *proj* and uses that file.

Commands

Expand table

Command	Description
clear	Delete all issued JWTs for a project.
create	Issue a new JSON Web Token.
remove	Delete a given JWT.

Command	Description
key	Display or reset the signing key used to issue JWTs.
list	Lists the JWTs issued for the project.
print	Display the details of a given JWT.

Create

Usage: `dotnet user-jwts create [options]`

Expand table

Option	Description
-p | --project	The path of the project to operate on. Defaults to the project in the current directory.
--scheme	The scheme name to use for the generated token. Defaults to 'Bearer'.
-n | --name	The name of the user to create the JWT for. Defaults to the current environment user.
--audience	The audiences to create the JWT for. Defaults to the URLs configured in the project's launchSettings.json.
--issuer	The issuer of the JWT. Defaults to 'dotnet-user-jwts'.
--scope	A scope claim to add to the JWT. Specify once for each scope.
--role	A role claim to add to the JWT. Specify once for each role.
--claim	Claims to add to the JWT. Specify once for each claim in the format "name=value".
--not-before	The UTC date & time the JWT should not be valid before in the format 'yyyy-MM-dd [[HH:mm[[:ss]]]]'. Defaults to the date & time the JWT is created.
--expires-on	The UTC date & time the JWT should expire in the format 'yyyy-MM-dd [[[[HH:mm]]:ss]]'. Defaults to 6 months after the --not-before date. Do not use this option in conjunction with the --valid-for option.
--valid-for	The period the JWT should expire after. Specify using a number followed by duration type like 'd' for days, 'h' for hours, 'm' for minutes, and 's' for seconds, for example 365d'. Do not use this option in conjunction with the --expires-on option.
-o | --output	The format to use for displaying output from the command. Can be one of 'default', 'token', or 'json'.
-h | --help	Show help information

Examples

Run the following commands to create an empty web project and add the Microsoft.AspNetCore.Authentication.JwtBearer NuGet package:

.NET CLI

```
dotnet new web -o MyJWT
cd MyJWT
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Replace the contents of `Program.cs` with the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.UseAuthorization();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();

app.Run();
```

In the preceding code, a GET request to `/secret` returns an `401 Unauthorized` error. A production app might get the JWT from a Security token service (STS), perhaps in response to logging in via a set of credentials. For the purpose of working with the API during local development, the `dotnet user-jwts` command line tool can be used to create and manage app-specific local JWTs.

The `user-jwts` tool is similar in concept to the user-secrets tool, it can be used to manage values for the app that are only valid for the developer on the local machine. In fact, the user-jwts tool utilizes the `user-secrets` infrastructure to manage the key that the JWTs are signed with, ensuring it's stored safely in the user profile.

The `user-jwts` tool hides implementation details, such as where and how the values are stored. The tool can be used without knowing the implementation details. The values

are stored in a JSON file in the local machine's user profile folder:

Windows

File system path:

`%APPDATA%\Microsoft\UserSecrets\<secrets_GUID>\user-jwts.json`

Create a JWT

The following command creates a local JWT:

.NET CLI

```
dotnet user-jwts create
```

The preceding command creates a JWT and updates the project's `appsettings.Development.json` file with JSON similar to the following:

C#

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Authentication": {
    "Schemes": {
      "Bearer": {
        "ValidAudiences": [
          "http://localhost:8401",
          "https://localhost:44308",
          "http://localhost:5182",
          "https://localhost:7076"
        ],
        "ValidIssuer": "dotnet-user-jwts"
      }
    }
  }
}
```

Copy the JWT and the `ID` created in the preceding command. Use a tool like Curl to test `/secret`:

.NET CLI

```
curl -i -H "Authorization: Bearer {token}" https://localhost:{port}/secret
```

Where `{token}` is the previously generated JWT.

Display JWT security information

The following command displays the JWT security information, including expiration, scopes, roles, token header and payload, and the compact token:

.NET CLI

```
dotnet user-jwts print {ID} --show-all
```

Create a token for a specific user and scope

See Create in this topic for supported create options.

The following command creates a JWT for a user named `MyTestUser`:

.NET CLI

```
dotnet user-jwts create --name MyTestUser --scope "myapi:secrets"
```

The preceding command has output similar to the following:

.NET CLI

```
New JWT saved with ID '43e0b748'.
Name: MyTestUser
Scopes: myapi:secrets

Token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.{Remaining token deleted}
```

The preceding token can be used to test the `/secret2` endpoint in the following code:

C#

```
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
```

```
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var app = builder.Build();

app.MapGet("/", () => "Hello, World!");
app.MapGet("/secret", (ClaimsPrincipal user) => $"Hello
{user.Identity?.Name}. My secret")
    .RequireAuthorization();
app.MapGet("/secret2", () => "This is a different secret!")
    .RequireAuthorization(p => p.RequireClaim("scope", "myapi:secrets"));

app.Run();
```

Azure Key Vault configuration provider in ASP.NET Core

Article • 10/29/2024

This article explains how to use the Azure Key Vault configuration provider to load app configuration values from Azure Key Vault secrets. Azure Key Vault is a cloud-based service that helps safeguard cryptographic keys and secrets used by apps and services. Common scenarios for using Azure Key Vault with ASP.NET Core apps include:

- Controlling access to sensitive configuration data.
- Meeting the requirement for FIPS 140-2 Level 2 validated Hardware Security Modules (HSMs) when storing configuration data.

Packages

Add package references for the following packages:

- Azure.Extensions.AspNetCore.Configuration.Secrets
- Azure.Identity

Sample app

The sample app runs in either of two modes determined by the `#define` preprocessor directive at the top of `Program.cs`:

- `Certificate`: Demonstrates using an Azure Key Vault Client ID and X.509 certificate to access secrets stored in Azure Key Vault. This sample can be run from any location, whether deployed to Azure App Service or any host that can serve an ASP.NET Core app.
- `Managed`: Demonstrates how to use Managed identities for Azure resources. The managed identity authenticates the app to Azure Key Vault with Managed identities for Azure resources without storing credentials in the app's code or configuration. The `Managed` version of the sample must be deployed to Azure. Follow the guidance in the Use the managed identities for Azure resources section.

For more information configuring a sample app using preprocessor directives (`#define`), see Overview of ASP.NET Core.

View or download sample code (how to download)

Secret storage in the Development environment

Set secrets locally using Secret Manager. When the sample app runs on the local machine in the Development environment, secrets are loaded from the local user secrets store.

Secret Manager requires a `<UserSecretsId>` property in the app's project file. Set the property value (`{GUID}`) to any unique GUID:

XML

```
<PropertyGroup>
  <UserSecretsId>{GUID}</UserSecretsId>
</PropertyGroup>
```

Secrets are created as name-value pairs. Hierarchical values (configuration sections) use a `:` (colon) as a separator in ASP.NET Core configuration key names.

Secret Manager is used from a command shell opened to the project's content root, where `{SECRET NAME}` is the name and `{SECRET VALUE}` is the value:

.NET CLI

```
dotnet user-secrets set "{SECRET NAME}" "{SECRET VALUE}"
```

Execute the following commands in a command shell from the project's content root to set the secrets for the sample app:

.NET CLI

```
dotnet user-secrets set "SecretName" "secret_value_1_dev"
dotnet user-secrets set "Section:SecretName" "secret_value_2_dev"
```

When these secrets are stored in Azure Key Vault in the Secret storage in the Production environment with Azure Key Vault section, the `_dev` suffix is changed to `_prod`. The suffix provides a visual cue in the app's output indicating the source of the configuration values.

Secret storage in the Production environment with Azure Key Vault

Complete the following steps to create an Azure Key Vault and store the sample app's secrets in it. For more information, see Quickstart: Set and retrieve a secret from Azure Key Vault using Azure CLI.

1. Open Azure Cloud Shell using any one of the following methods in the Azure portal:

   - Select **Try It** in the upper-right corner of a code block. Use the search string "Azure CLI" in the text box.
   - Open Cloud Shell in your browser with the **Launch Cloud Shell** button.
   - Select the **Cloud Shell** button on the menu in the upper-right corner of the Azure portal.

   For more information, see Azure CLI and Overview of Azure Cloud Shell.

2. If you aren't already authenticated, sign in with the `az login` command.

3. Create a resource group with the following command, where `{RESOURCE GROUP NAME}` is the new resource group's name and `{LOCATION}` is the Azure region:

   Azure CLI

   ```
   az group create --name "{RESOURCE GROUP NAME}" --location {LOCATION}
   ```

4. Create a Key Vault in the resource group with the following command, where `{KEY VAULT NAME}` is the new vault's name and `{LOCATION}` is the Azure region:

   Azure CLI

   ```
   az keyvault create --name {KEY VAULT NAME} --resource-group "{RESOURCE GROUP NAME}" --location {LOCATION}
   ```

5. Create secrets in the vault as name-value pairs.

   Azure Key Vault secret names are limited to alphanumeric characters and dashes. Hierarchical values (configuration sections) use `--` (two dashes) as a delimiter, as colons aren't allowed in Key Vault secret names. Colons delimit a section from a subkey in ASP.NET Core configuration. The two-dash sequence is replaced with a colon when the secrets are loaded into the app's configuration.

   The following secrets are for use with the sample app. The values include a `_prod` suffix to distinguish them from the `_dev` suffix values loaded in the Development

environment from Secret Manager. Replace `{KEY VAULT NAME}` with the name of the Key Vault you created in the prior step:

Azure CLI

```
az keyvault secret set --vault-name {KEY VAULT NAME} --name
"SecretName" --value "secret_value_1_prod"
az keyvault secret set --vault-name {KEY VAULT NAME} --name "Section--
SecretName" --value "secret_value_2_prod"
```

Use Application ID and X.509 certificate for non-Azure-hosted apps

Configure Azure Key Vault and the app to use an Microsoft Entra ID Application ID and X.509 certificate to authenticate to a vault **when the app is hosted outside of Azure**. For more information, see About keys, secrets, and certificates.

> **Note**
>
> Although using an Application ID and X.509 certificate is supported for apps hosted in Azure, it's not recommended. Instead, use Managed identities for Azure resources when hosting an app in Azure. Managed identities don't require storing a certificate in the app or in the development environment.

The sample app uses an Application ID and X.509 certificate when the `#define` preprocessor directive at the top of `Program.cs` is set to `Certificate`.

1. Create a PKCS#12 archive (*.pfx*) certificate. Options for creating certificates include New-SelfSignedCertificate on Windows and OpenSSL.
2. Install the certificate into the current user's personal certificate store. Marking the key as exportable is optional. Note the certificate's thumbprint, which is used later in this process.
3. Export the PKCS#12 archive (*.pfx*) certificate as a DER-encoded certificate (*.cer*).
4. Register the app with Microsoft Entra ID (**App registrations**).
5. Upload the DER-encoded certificate (*.cer*) to Microsoft Entra ID:
   a. Select the app in Microsoft Entra ID.
   b. Navigate to **Certificates & secrets**.
   c. Select **Upload certificate** to upload the certificate, which contains the public key. A *.cer*, *.pem*, or *.crt* certificate is acceptable.

6. Store the Key Vault name, Application ID, and certificate thumbprint in the app's `appsettings.json` file.
7. Navigate to **Key Vaults** in the Azure portal.
8. Select the Key Vault you created in the Secret storage in the Production environment with Azure Key Vault section.
9. Select **Access policies**.
10. Select **Add Access Policy**.
11. Open **Secret permissions** and provide the app with **Get** and **List** permissions.
12. Select **Select principal** and select the registered app by name. Select the **Select** button.
13. Select **OK**.
14. Select **Save**.
15. Deploy the app.

The `Certificate` sample app obtains its configuration values from IConfigurationRoot with the same name as the secret name:

- Non-hierarchical values: The value for `SecretName` is obtained with `config["SecretName"]`.
- Hierarchical values (sections): Use `:` (colon) notation or the GetSection method. Use either of these approaches to obtain the configuration value:
  - `config["Section:SecretName"]`
  - `config.GetSection("Section")["SecretName"]`

The X.509 certificate is managed by the OS. The app calls AddAzureKeyVault with values supplied by the `appsettings.json` file:

C#

```
using System.Security.Cryptography.X509Certificates;
using Azure.Identity;

var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsProduction())
{
    using var x509Store = new X509Store(StoreLocation.CurrentUser);

    x509Store.Open(OpenFlags.ReadOnly);

    var x509Certificate = x509Store.Certificates
        .Find(
            X509FindType.FindByThumbprint,
            builder.Configuration["AzureADCertThumbprint"],
            validOnly: false)
```

```
        .OfType<X509Certificate2>()
        .Single();

    builder.Configuration.AddAzureKeyVault(
        new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
        new ClientCertificateCredential(
            builder.Configuration["AzureADDirectoryId"],
            builder.Configuration["AzureADApplicationId"],
            x509Certificate));
}

var app = builder.Build();
```

Example values:

- Key Vault name: `contosovault`
- Application ID: `00001111-aaaa-2222-bbbb-3333cccc4444`
- Certificate thumbprint: `fe14593dd66b2406c5269d742d04b6e1ab03adb1`

`appsettings.json`:

JSON

```
{
  "KeyVaultName": "Key Vault Name",
  "AzureADApplicationId": "Azure AD Application ID",
  "AzureADCertThumbprint": "Azure AD Certificate Thumbprint",
  "AzureADDirectoryId": "Azure AD Directory ID"
}
```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values load with the `_dev` suffix. In the Production environment, the values load with the `_prod` suffix.

Use managed identities for Azure resources

An app deployed to Azure can take advantage of Managed identities for Azure resources. A managed identity allows the app to authenticate with Azure Key Vault using Microsoft Entra ID authentication without storing credentials in the app's code or configuration.

The sample app uses a system-assigned managed identity when the `#define` preprocessor directive at the top of `Program.cs` is set to `Managed`. To create a managed identity for an Azure App Service app, see How to use managed identities for App

Service and Azure Functions. Once the managed identity has been created, note the app's Object ID shown in the Azure portal on the **Identity** panel of the App Service.

Enter the vault name into the app's `appsettings.json` file. The sample app doesn't require an Application ID and Password (Client Secret) when set to the `Managed` version, so you can ignore those configuration entries. The app is deployed to Azure, and Azure authenticates the app to access Azure Key Vault only using the vault name stored in the `appsettings.json` file.

Deploy the sample app to Azure App Service.

Using Azure CLI and the app's Object ID, provide the app with `list` and `get` permissions to access the vault:

Azure CLI

```
az keyvault set-policy --name {KEY VAULT NAME} --object-id {OBJECT ID} --
secret-permissions get list
```

Restart the app using Azure CLI, PowerShell, or the Azure portal.

The sample app creates an instance of the DefaultAzureCredential class. The credential attempts to obtain an access token from environment for Azure resources:

C#

```
using Azure.Identity;

var builder = WebApplication.CreateBuilder(args);

if (builder.Environment.IsProduction())
{
    builder.Configuration.AddAzureKeyVault(
        new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
        new DefaultAzureCredential());
}
```

Key Vault name example value: `contosovault`

`appsettings.json`:

JSON

```
{
  "KeyVaultName": "Key Vault Name"
```

```
}
```

For apps that use a user-assigned managed identity, configure the managed identity's Client ID using one of the following approaches:

1. Set the `AZURE_CLIENT_ID` environment variable.
2. Set the DefaultAzureCredentialOptions.ManagedIdentityClientId property when calling `AddAzureKeyVault`:

   C#

   ```
   builder.Configuration.AddAzureKeyVault(
       new
   Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"
   ),
       new DefaultAzureCredential(new DefaultAzureCredentialOptions
       {
           ManagedIdentityClientId =
   builder.Configuration["AzureADManagedIdentityClientId"]
       }));
   ```

When you run the app, a webpage shows the loaded secret values. In the Development environment, secret values have the `_dev` suffix because they're provided by Secret Manager. In the Production environment, the values load with the `_prod` suffix because they're provided by Azure Key Vault.

If you receive an `Access denied` error, confirm that the app is registered with Microsoft Entra ID and provided access to the vault. Confirm that you've restarted the service in Azure.

For information on using the provider with a managed identity and Azure Pipelines, see Create an Azure Resource Manager service connection to a VM with a managed service identity.

Configuration options

`AddAzureKeyVault` can accept an AzureKeyVaultConfigurationOptions object:

C#

```
// using Azure.Extensions.AspNetCore.Configuration.Secrets;

builder.Configuration.AddAzureKeyVault(
    new
```

```
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
    new DefaultAzureCredential(),
    new AzureKeyVaultConfigurationOptions
    {
        // ...
    });
```

The `AzureKeyVaultConfigurationOptions` object contains the following properties:

Expand table

Property	Description
Manager	KeyVaultSecretManager instance used to control secret loading.
ReloadInterval	`TimeSpan` to wait between attempts at polling the vault for changes. The default value is `null` (configuration isn't reloaded).

Use a key name prefix

`AddAzureKeyVault` provides an overload that accepts an implementation of KeyVaultSecretManager, which allows you to control how Key Vault secrets are converted into configuration keys. For example, you can implement the interface to load secret values based on a prefix value you provide at app startup. This technique allows you, for example, to load secrets based on the version of the app.

> ⚠ **Warning**
>
> Don't use prefixes on Key Vault secrets to:
>
> - Place secrets for multiple apps into the same vault.
> - Place environmental secrets (for example, *development* versus *production* secrets) into the same vault.
>
> Different apps and development/production environments should use separate Key Vaults to isolate app environments for the highest level of security.

In the following example, a secret is established in Key Vault (and using Secret Manager for the Development environment) for `5000-AppSecret` (periods aren't allowed in Key Vault secret names). This secret represents an app secret for version 5.0.0.0 of the app. For another version of the app, 5.1.0.0, a secret is added to the vault (and using Secret

Manager) for `5100-AppSecret`. Each app version loads its versioned secret value into its configuration as `AppSecret`, removing the version as it loads the secret.

`AddAzureKeyVault` is called with a custom `KeyVaultSecretManager` implementation:

C#

```
// using Azure.Extensions.AspNetCore.Configuration.Secrets;

builder.Configuration.AddAzureKeyVault(
    new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
    new DefaultAzureCredential(),
    new SamplePrefixKeyVaultSecretManager("5000"));
```

The implementation reacts to the version prefixes of secrets to load the proper secret into configuration:

- `Load` loads a secret when its name starts with the prefix. Other secrets aren't loaded.
- `GetKey`:
  - Removes the prefix from the secret name.
  - Replaces two dashes in any name with the `KeyDelimiter`, which is the delimiter used in configuration (usually a colon). Azure Key Vault doesn't allow a colon in secret names.

C#

```
public class SamplePrefixKeyVaultSecretManager : KeyVaultSecretManager
{
    private readonly string _prefix;

    public SamplePrefixKeyVaultSecretManager(string prefix)
        => _prefix = $"{prefix}-";

    public override bool Load(SecretProperties properties)
        => properties.Name.StartsWith(_prefix);

    public override string GetKey(KeyVaultSecret secret)
        => secret.Name[_prefix.Length..].Replace("--",
ConfigurationPath.KeyDelimiter);
}
```

The `Load` method is called by a provider algorithm that iterates through the vault secrets to find the version-prefixed secrets. When a version prefix is found with `Load`, the algorithm uses the `GetKey` method to return the configuration name of the secret

name. It removes the version prefix from the secret's name. The rest of the secret name is returned for loading into the app's configuration name-value pairs.

When this approach is implemented:

1. The app's version specified in the app's project file. In the following example, the app's version is set to `5.0.0.0`:

   XML

   ```xml
   <PropertyGroup>
     <Version>5.0.0.0</Version>
   </PropertyGroup>
   ```

2. Confirm that a `<UserSecretsId>` property is present in the app's project file, where `{GUID}` is a user-supplied GUID:

   XML

   ```xml
   <PropertyGroup>
     <UserSecretsId>{GUID}</UserSecretsId>
   </PropertyGroup>
   ```

   Save the following secrets locally with Secret Manager:

   .NET CLI

   ```
   dotnet user-secrets set "5000-AppSecret" "5.0.0.0_secret_value_dev"
   dotnet user-secrets set "5100-AppSecret" "5.1.0.0_secret_value_dev"
   ```

3. Secrets are saved in Azure Key Vault using the following Azure CLI commands:

   Azure CLI

   ```
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5000-
   AppSecret" --value "5.0.0.0_secret_value_prod"
   az keyvault secret set --vault-name {KEY VAULT NAME} --name "5100-
   AppSecret" --value "5.1.0.0_secret_value_prod"
   ```

4. When the app is run, the Key Vault secrets are loaded. The string secret for `5000-AppSecret` is matched to the app's version specified in the app's project file (`5.0.0.0`).

5. The version, `5000` (with the dash), is stripped from the key name. Throughout the app, reading configuration with the key `AppSecret` loads the secret value.

6. If the app's version is changed in the project file to `5.1.0.0` and the app is run again, the secret value returned is `5.1.0.0_secret_value_dev` in the Development environment and `5.1.0.0_secret_value_prod` in Production.

> **Note**
>
> You can also provide your own SecretClient implementation to `AddAzureKeyVault`. A custom client permits sharing a single instance of the client across the app.

Bind an array to a class

The provider can read configuration values into an array for binding to a POCO array.

When reading from a configuration source that allows keys to contain colon (`:`) separators, a numeric key segment is used to distinguish the keys that make up an array (`:0:`, `:1:`, ... `:{n}:`). For more information, see Configuration: Bind an array to a class.

Azure Key Vault keys can't use a colon as a separator. The approach described in this article uses double dashes (`--`) as a separator for hierarchical values (sections). Array keys are stored in Azure Key Vault with double dashes and numeric key segments (`--0--`, `--1--`, ... `--{n}--`).

Examine the following Serilog logging provider configuration provided by a JSON file. There are two object literals defined in the `WriteTo` array that reflect two Serilog *sinks*, which describe destinations for logging output:

JSON

```
"Serilog": {
  "WriteTo": [
    {
      "Name": "AzureTableStorage",
      "Args": {
        "storageTableName": "logs",
        "connectionString": "DefaultEnd...ountKey=Eby8...GMGw=="
      }
    },
    {
      "Name": "AzureDocumentDB",
      "Args": {
        "endpointUrl": "https://contoso.documents.azure.com:443",
        "authorizationKey": "Eby8...GMGw=="
      }
    }
```

```
  ]
}
```

The configuration shown in the preceding JSON file is stored in Azure Key Vault using double dash (`--`) notation and numeric segments:

Expand table

Key	Value
`Serilog--WriteTo--0--Name`	`AzureTableStorage`
`Serilog--WriteTo--0--Args--storageTableName`	`logs`
`Serilog--WriteTo--0--Args--connectionString`	`DefaultEnd...ountKey=Eby8...GMGw==`
`Serilog--WriteTo--1--Name`	`AzureDocumentDB`
`Serilog--WriteTo--1--Args--endpointUrl`	`https://contoso.documents.azure.com:443`
`Serilog--WriteTo--1--Args--authorizationKey`	`Eby8...GMGw==`

Reload secrets

By default, secrets are cached by the configuration provider for the app's lifetime. Secrets that have been subsequently disabled or updated in the vault are ignored by the app.

To reload secrets, call IConfigurationRoot.Reload:

C#

```
config.Reload();
```

To reload secrets periodically, at a specified interval, set the AzureKeyVaultConfigurationOptions.ReloadInterval property. For more information, see Configuration options.

Disabled and expired secrets

Expired secrets are included by default in the configuration provider. To exclude values for these secrets in app configuration, update the expired secret or provide the configuration using a custom configuration provider:

C#

```
class SampleKeyVaultSecretManager : KeyVaultSecretManager
{
  public override bool Load(SecretProperties properties) =>
    properties.ExpiresOn.HasValue &&
    properties.ExpiresOn.Value > DateTimeOffset.Now;
}
```

Pass this custom `KeyVaultSecretManager` to `AddAzureKeyVault`:

C#

```
// using Azure.Extensions.AspNetCore.Configuration.Secrets;

builder.Configuration.AddAzureKeyVault(
    new
Uri($"https://{builder.Configuration["KeyVaultName"]}.vault.azure.net/"),
    new DefaultAzureCredential(),
    new SampleKeyVaultSecretManager());
```

Disabled secrets cannot be retrieved from Key Vault and are never included.

Troubleshoot

When the app fails to load configuration using the provider, an error message is written to the ASP.NET Core Logging infrastructure. The following conditions will prevent configuration from loading:

- The app or certificate isn't configured correctly in Microsoft Entra ID.
- The vault doesn't exist in Azure Key Vault.
- The app isn't authorized to access the vault.
- The access policy doesn't include `Get` and `List` permissions.
- In the vault, the configuration data (name-value pair) is incorrectly named, missing, or disabled.
- The app has the wrong Key Vault name (`KeyVaultName`), Microsoft Entra ID Application ID (`AzureADApplicationId`), or Microsoft Entra ID certificate thumbprint (`AzureADCertThumbprint`), or Microsoft Entra ID Directory ID (`AzureADDirectoryId`).
- When adding the Key Vault access policy for the app, the policy was created, but the **Save** button wasn't selected in the **Access policies** UI.

Additional resources

- View or download sample code (how to download)
- Configuration in ASP.NET Core
- Microsoft Azure: Key Vault Documentation
- How to generate and transfer HSM-protected keys for Azure Key Vault
- Quickstart: Set and retrieve a secret from Azure Key Vault by using a .NET web app
- Tutorial: How to use Azure Key Vault with Azure Windows Virtual Machine in .NET

Enforce HTTPS in ASP.NET Core

Article • 10/24/2024

By David Galvan and Rick Anderson

this article shows how to:

- Require HTTPS for all requests.
- Redirect all HTTP requests to HTTPS.

No API can prevent a client from sending sensitive data on the first request.

> **Note**
>
> If you're using .NET 9 SDK or later, see the updated Linux procedures in the .NET 9 version of this article.

> **Warning**

API projects

Do **not** use RequireHttpsAttribute on Web APIs that receive sensitive information. `RequireHttpsAttribute` uses HTTP status codes to redirect browsers from HTTP to HTTPS. API clients may not understand or obey redirects from HTTP to HTTPS. Such clients may send information over HTTP. Web APIs should either:

- Not listen on HTTP.
- Close the connection with status code 400 (Bad Request) and not serve the request.

To disable HTTP redirection in an API, set the `ASPNETCORE_URLS` environment variable or use the `--urls` command line flag. For more information, see Use multiple environments in ASP.NET Core and 8 ways to set the URLs for an ASP.NET Core app by Andrew Lock.

HSTS and API projects

The default API projects don't include HSTS because HSTS is generally a browser only instruction. Other callers, such as phone or desktop apps, do **not** obey the

instruction. Even within browsers, a single authenticated call to an API over HTTP has risks on insecure networks. The secure approach is to configure API projects to only listen to and respond over HTTPS.

HTTP redirection to HTTPS causes ERR_INVALID_REDIRECT on the CORS preflight request

Requests to an endpoint using HTTP that are redirected to HTTPS by UseHttpsRedirection fail with `ERR_INVALID_REDIRECT` on the CORS preflight request.

API projects can reject HTTP requests rather than use `UseHttpsRedirection` to redirect requests to HTTPS.

Require HTTPS

We recommend that production ASP.NET Core web apps use:

- HTTPS Redirection Middleware (UseHttpsRedirection) to redirect HTTP requests to HTTPS.
- HSTS Middleware (UseHsts) to send HTTP Strict Transport Security Protocol (HSTS) headers to clients.

> **Note**
>
> Apps deployed in a reverse proxy configuration allow the proxy to handle connection security (HTTPS). If the proxy also handles HTTPS redirection, there's no need to use HTTPS Redirection Middleware. If the proxy server also handles writing HSTS headers (for example, native HSTS support in IIS 10.0 (1709) or later), HSTS Middleware isn't required by the app. For more information, see Opt-out of HTTPS/HSTS on project creation.

UseHttpsRedirection

The following code calls UseHttpsRedirection in the `Program.cs` file:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding highlighted code:

- Uses the default HttpsRedirectionOptions.RedirectStatusCode (Status307TemporaryRedirect).
- Uses the default HttpsRedirectionOptions.HttpsPort (null) unless overridden by the `ASPNETCORE_HTTPS_PORT` environment variable or IServerAddressesFeature.

We recommend using temporary redirects rather than permanent redirects. Link caching can cause unstable behavior in development environments. If you prefer to send a permanent redirect status code when the app is in a non-Development environment, see the Configure permanent redirects in production section. We recommend using HSTS to signal to clients that only secure resource requests should be sent to the app (only in production).

Port configuration

A port must be available for the middleware to redirect an insecure request to HTTPS. If no port is available:

- Redirection to HTTPS doesn't occur.
- The middleware logs the warning "Failed to determine the https port for redirect."

Specify the HTTPS port using any of the following approaches:

- Set HttpsRedirectionOptions.HttpsPort.
- Set the `https_port` host setting:

- In host configuration.
- By setting the `ASPNETCORE_HTTPS_PORT` environment variable.
- By adding a top-level entry in `appsettings.json`:

```
JSON

{
  "https_port": 443,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

- Indicate a port with the secure scheme using the ASPNETCORE_URLS environment variable. The environment variable configures the server. The middleware indirectly discovers the HTTPS port via IServerAddressesFeature. This approach doesn't work in reverse proxy deployments.
- The ASP.NET Core web templates set an HTTPS URL in `Properties/launchsettings.json` for both Kestrel and IIS Express. `launchsettings.json` is only used on the local machine.
- Configure an HTTPS URL endpoint for a public-facing edge deployment of Kestrel server or HTTP.sys server. Only **one HTTPS port** is used by the app. The middleware discovers the port via IServerAddressesFeature.

> ⓘ **Note**
>
> When an app is run in a reverse proxy configuration, IServerAddressesFeature isn't available. Set the port using one of the other approaches described in this section.

Edge deployments

When Kestrel or HTTP.sys is used as a public-facing edge server, Kestrel or HTTP.sys must be configured to listen on both:

- The secure port where the client is redirected (typically, 443 in production and 5001 in development).

- The insecure port (typically, 80 in production and 5000 in development).

The insecure port must be accessible by the client in order for the app to receive an insecure request and redirect the client to the secure port.

For more information, see Kestrel endpoint configuration or HTTP.sys web server implementation in ASP.NET Core.

Deployment scenarios

Any firewall between the client and server must also have communication ports open for traffic.

If requests are forwarded in a reverse proxy configuration, use Forwarded Headers Middleware before calling HTTPS Redirection Middleware. Forwarded Headers Middleware updates the `Request.Scheme`, using the `X-Forwarded-Proto` header. The middleware permits redirect URIs and other security policies to work correctly. When Forwarded Headers Middleware isn't used, the backend app might not receive the correct scheme and end up in a redirect loop. A common end user error message is that too many redirects have occurred.

When deploying to Azure App Service, follow the guidance in Tutorial: Bind an existing custom SSL certificate to Azure Web Apps.

Options

The following highlighted code calls AddHttpsRedirection to configure middleware options:

C#

```
using static Microsoft.AspNetCore.Http.StatusCodes;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddHsts(options =>
{
    options.Preload = true;
    options.IncludeSubDomains = true;
    options.MaxAge = TimeSpan.FromDays(60);
    options.ExcludedHosts.Add("example.com");
    options.ExcludedHosts.Add("www.example.com");
});
```

```
builder.Services.AddHttpsRedirection(options =>
{
    options.RedirectStatusCode = Status307TemporaryRedirect;
    options.HttpsPort = 5001;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

Calling `AddHttpsRedirection` is only necessary to change the values of `HttpsPort` or `RedirectStatusCode`.

The preceding highlighted code:

- Sets HttpsRedirectionOptions.RedirectStatusCode to Status307TemporaryRedirect, which is the default value. Use the fields of the StatusCodes class for assignments to `RedirectStatusCode`.
- Sets the HTTPS port to 5001.

Configure permanent redirects in production

The middleware defaults to sending a Status307TemporaryRedirect with all redirects. If you prefer to send a permanent redirect status code when the app is in a non-Development environment, wrap the middleware options configuration in a conditional check for a non-Development environment.

When configuring services in `Program.cs`:

C#

```
using static Microsoft.AspNetCore.Http.StatusCodes;
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

if (!builder.Environment.IsDevelopment())
{
    builder.Services.AddHttpsRedirection(options =>
    {
        options.RedirectStatusCode = Status308PermanentRedirect;
        options.HttpsPort = 443;
    });
}

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");

    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

HTTPS Redirection Middleware alternative approach

An alternative to using HTTPS Redirection Middleware (`UseHttpsRedirection`) is to use URL Rewriting Middleware (`AddRedirectToHttps`). `AddRedirectToHttps` can also set the status code and port when the redirect is executed. For more information, see URL Rewriting Middleware.

When redirecting to HTTPS without the requirement for additional redirect rules, we recommend using HTTPS Redirection Middleware (`UseHttpsRedirection`) described in this article.

HTTP Strict Transport Security Protocol (HSTS)

Per OWASP, HTTP Strict Transport Security (HSTS) is an opt-in security enhancement that's specified by a web app through the use of a response header. When a browser that supports HSTS receives this header:

- The browser stores configuration for the domain that prevents sending any communication over HTTP. The browser forces all communication over HTTPS.
- The browser prevents the user from using untrusted or invalid certificates. The browser disables prompts that allow a user to temporarily trust such a certificate.

Because HSTS is enforced by the client, it has some limitations:

- The client must support HSTS.
- HSTS requires at least one successful HTTPS request to establish the HSTS policy.
- The application must check every HTTP request and redirect or reject the HTTP request.

ASP.NET Core implements HSTS with the UseHsts extension method. The following code calls `UseHsts` when the app isn't in development mode:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

`UseHsts` isn't recommended in development because the HSTS settings are highly cacheable by browsers. By default, `UseHsts` excludes the local loopback address.

For production environments that are implementing HTTPS for the first time, set the initial HstsOptions.MaxAge to a small value using one of the TimeSpan methods. Set the value from hours to no more than a single day in case you need to revert the HTTPS infrastructure to HTTP. After you're confident in the sustainability of the HTTPS configuration, increase the HSTS `max-age` value; a commonly used value is one year.

The following highlighted code:

C#

```
using static Microsoft.AspNetCore.Http.StatusCodes;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddHsts(options =>
{
    options.Preload = true;
    options.IncludeSubDomains = true;
    options.MaxAge = TimeSpan.FromDays(60);
    options.ExcludedHosts.Add("example.com");
    options.ExcludedHosts.Add("www.example.com");
});

builder.Services.AddHttpsRedirection(options =>
{
    options.RedirectStatusCode = Status307TemporaryRedirect;
    options.HttpsPort = 5001;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

- Sets the preload parameter of the `Strict-Transport-Security` header. Preload isn't part of the RFC HSTS specification, but is supported by web browsers to preload HSTS sites on fresh install. For more information, see https://hstspreload.org/.
- Enables includeSubDomain, which applies the HSTS policy to Host subdomains.
- Explicitly sets the `max-age` parameter of the `Strict-Transport-Security` header to 60 days. If not set, defaults to 30 days. For more information, see the max-age directive.
- Adds `example.com` to the list of hosts to exclude.

`UseHsts` excludes the following loopback hosts:

- `localhost` : The IPv4 loopback address.
- `127.0.0.1` : The IPv4 loopback address.
- `[::1]` : The IPv6 loopback address.

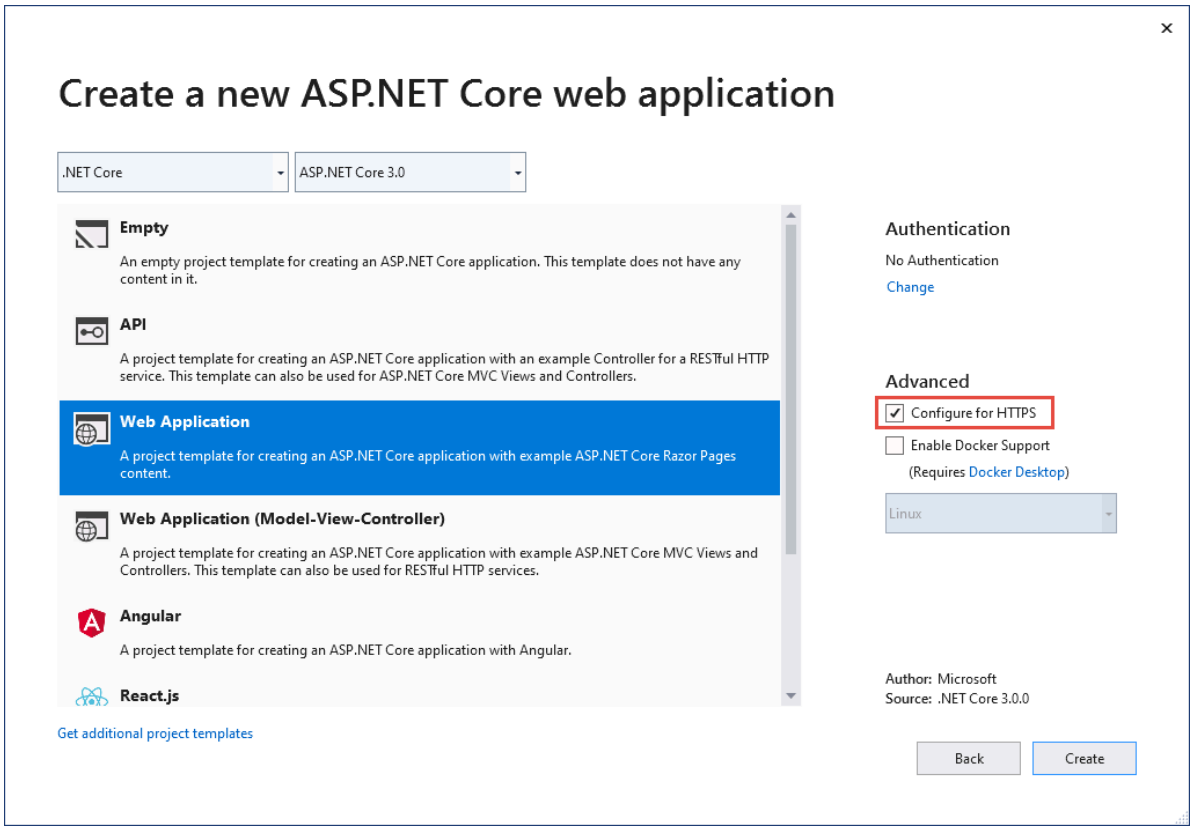
Opt-out of HTTPS/HSTS on project creation

In some backend service scenarios where connection security is handled at the public-facing edge of the network, configuring connection security at each node isn't required. Web apps that are generated from the templates in Visual Studio or from the dotnet new command enable HTTPS redirection and HSTS. For deployments that don't require these scenarios, you can opt-out of HTTPS/HSTS when the app is created from the template.

To opt-out of HTTPS/HSTS:

Visual Studio

Uncheck the **Configure for HTTPS** checkbox.



Trust the ASP.NET Core HTTPS development certificate on Windows and macOS

For the Firefox browser, see the next section.

The .NET Core SDK includes an HTTPS development certificate. The certificate is installed as part of the first-run experience. For example, `dotnet --info` produces a variation of the following output:

cli

```
ASP.NET Core
------------
Successfully installed the ASP.NET Core HTTPS Development Certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and
macOS only).
For establishing trust on other platforms refer to the platform specific
documentation.
For more information on configuring HTTPS see
https://go.microsoft.com/fwlink/?linkid=848054.
```

Installing the .NET Core SDK installs the ASP.NET Core HTTPS development certificate to the local user certificate store. The certificate has been installed, but it's not trusted. To trust the certificate, perform the one-time step to run the `dotnet dev-certs` tool:

.NET CLI

```
dotnet dev-certs https --trust
```

The following command provides help on the `dotnet dev-certs` tool:

.NET CLI

```
dotnet dev-certs https --help
```

> ⚠ **Warning**
>
> Do not create a development certificate in an environment that will be redistributed, such as a container image or virtual machine. Doing so can lead to spoofing and elevation of privilege. To help prevent this, set the `DOTNET_GENERATE_ASPNET_CERTIFICATE` environment variable to `false` prior to calling the .NET CLI for the first time. This will skip the automatic generation of the ASP.NET Core development certificate during the CLI's first-run experience.

Trust the HTTPS certificate with Firefox to prevent SEC_ERROR_INADEQUATE_KEY_USAGE error

The Firefox browser uses its own certificate store, and therefore doesn't trust the IIS Express or Kestrel developer certificates.

There are two approaches to trusting the HTTPS certificate with Firefox, create a policy file or configure with the FireFox browser. Configuring with the browser creates the policy file, so the two approaches are equivalent.

Create a policy file to trust HTTPS certificate with Firefox

Create a policy file (`policies.json`) at:

- Windows: `%PROGRAMFILES%\Mozilla Firefox\distribution\`
- MacOS: `Firefox.app/Contents/Resources/distribution`
- Linux: See Trust the certificate with Firefox on Linux in this article.

Add the following JSON to the Firefox policy file:

JSON

```
{
  "policies": {
    "Certificates": {
      "ImportEnterpriseRoots": true
    }
  }
}
```

The preceding policy file makes Firefox trust certificates from the trusted certificates in the Windows certificate store. The next section provides an alternative approach to create the preceding policy file by using the Firefox browser.

Configure trust of HTTPS certificate using Firefox browser

Set `security.enterprise_roots.enabled` = `true` using the following instructions:

1. Enter `about:config` in the FireFox browser.
2. Select **Accept the Risk and Continue** if you accept the risk.
3. Select **Show All**
4. Set `security.enterprise_roots.enabled` = `true`
5. Exit and restart Firefox

For more information, see Setting Up Certificate Authorities (CAs) in Firefox and the mozilla/policy-templates/README file.

How to set up a developer certificate for Docker

See this GitHub issue.

Trust HTTPS certificate on Linux

Establishing trust is distribution and browser specific. The following sections provide instructions for some popular distributions and the Chromium browsers (Edge and Chrome) and for Firefox.

Trust HTTPS certificate on Linux with linux-dev-certs

linux-dev-certs is an open-source, community-supported, .NET global tool that provides a convenient way to create and trust a developer certificate on Linux. The tool is not maintained or supported by Microsoft.

The following commands install the tool and create a trusted developer certificate:

cli

```
dotnet tool update -g linux-dev-certs
dotnet linux-dev-certs install
```

For more information or to report issues, see the linux-dev-certs GitHub repository.

Ubuntu trust the certificate for service-to-service communication

The following instructions don't work for some Ubuntu versions, such as 20.04. For more information, see GitHub issue dotnet/AspNetCore.Docs #23686.

1. Install OpenSSL 1.1.1h or later. See your distribution for instructions on how to update OpenSSL.
2. Run the following commands:

   cli

   ```
   dotnet dev-certs https
   sudo -E dotnet dev-certs https -ep /usr/local/share/ca-
   certificates/aspnet/https.crt --format PEM
   sudo update-ca-certificates
   ```

The preceding commands:

- Ensure the current user's developer certificate is created.
- Exports the certificate with elevated permissions needed for the `ca-certificates` folder, using the current user's environment.
- Removing the `-E` flag exports the root user certificate, generating it if necessary. Each newly generated certificate has a different thumbprint. When running as root, `sudo` and `-E` are not needed.

The path in the preceding command is specific for Ubuntu. For other distributions, select an appropriate path or use the path for the Certificate Authorities (CAs).

Trust HTTPS certificate on Linux using Edge or Chrome

SUSE Linux Enterprise Server

See this GitHub issue

Trust the certificate with Fedora 34

See:

- This GitHub comment
- Fedora: Using Shared System Certificates
- Set up a .NET development environment on Fedora.

Trust the certificate with other distros

See this GitHub issue.

Trust HTTPS certificate from Windows Subsystem for Linux

The following instructions don't work for some Linux distributions, such as Ubuntu 20.04. For more information, see GitHub issue dotnet/AspNetCore.Docs #23686.

The Windows Subsystem for Linux (WSL) generates an HTTPS self-signed development certificate, which by default isn't trusted in Windows. The easiest way to have Windows trust the WSL certificate, is to configure WSL to use the same certificate as Windows:

- On ***Windows***, export the developer certificate to a file:

  ```
  dotnet dev-certs https -ep https.pfx -p $CREDENTIAL_PLACEHOLDER$ --
  trust
  ```

  Where `$CREDENTIAL_PLACEHOLDER$` is a password.

- In a WSL window, import the exported certificate on the WSL instance:

  ```
  dotnet dev-certs https --clean --import <<path-to-pfx>> --password
  $CREDENTIAL_PLACEHOLDER$
  ```

The preceding approach is a one time operation per certificate and per WSL distribution. It's easier than exporting the certificate over and over. If you update or regenerate the certificate on windows, you might need to run the preceding commands again.

Troubleshoot certificate problems such as certificate not trusted

This section provides help when the ASP.NET Core HTTPS development certificate has been installed and trusted, but you still have browser warnings that the certificate is not trusted. The ASP.NET Core HTTPS development certificate is used by Kestrel.

To repair the IIS Express certificate, see this Stackoverflow issue.

All platforms - certificate not trusted

Run the following commands:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app. Certificate trust is cached by browsers.

dotnet dev-certs https --clean Fails

The preceding commands solve most browser trust issues. If the browser is still not trusting the certificate, follow the platform-specific suggestions that follow.

Docker - certificate not trusted

- Delete the *C:\Users{USER}\AppData\Roaming\ASP.NET\Https* folder.
- Clean the solution. Delete the *bin* and *obj* folders.
- Restart the development tool. For example, Visual Studio or Visual Studio Code.

Windows - certificate not trusted

- Check the certificates in the certificate store. There should be a `localhost` certificate with the `ASP.NET Core HTTPS development certificate` friendly name

both under `Current User > Personal > Certificates` and `Current User > Trusted root certification authorities > Certificates`

- Remove all the found certificates from both Personal and Trusted root certification authorities. Do **not** remove the IIS Express localhost certificate.
- Run the following commands:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app.

OS X - certificate not trusted

- Open KeyChain Access.
- Select the System keychain.
- Check for the presence of a localhost certificate.
- Check that it contains a `+` symbol on the icon to indicate it's trusted for all users.
- Remove the certificate from the system keychain.
- Run the following commands:

.NET CLI

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Close any browser instances open. Open a new browser window to app.

See HTTPS Error using IIS Express (dotnet/AspNetCore #16892) for troubleshooting certificate issues with Visual Studio.

Linux certificate not trusted

Check that the certificate being configured for trust is the user HTTPS developer certificate that will be used by the Kestrel server.

Check the current user default HTTPS developer Kestrel certificate at the following location:

```
```

```
ls -la ~/.dotnet/corefx/cryptography/x509stores/my
```

The HTTPS developer Kestrel certificate file is the SHA1 thumbprint. When the file is deleted via `dotnet dev-certs https --clean`, it's regenerated when needed with a different thumbprint. Check the thumbprint of the exported certificate matches with the following command:

```
openssl x509 -noout -fingerprint -sha1 -inform pem -in /usr/local/share/ca-certificates/aspnet/https.crt
```

If the certificate doesn't match, it could be one of the following:

- An old certificate.
- An exported a developer certificate for the root user. For this case, export the certificate.

The root user certificate can be checked at:

```
ls -la /root/.dotnet/corefx/cryptography/x509stores/my
```

IIS Express SSL certificate used with Visual Studio

To fix problems with the IIS Express certificate, select **Repair** from the Visual Studio installer. For more information, see this GitHub issue.

Group policy prevents self-signed certificates from being trusted

In some cases, group policy may prevent self-signed certificates from being trusted. For more information, see this GitHub issue.

Additional information

- Configure ASP.NET Core to work with proxy servers and load balancers
- Host ASP.NET Core on Linux with Nginx: HTTPS configuration
- How to Set Up SSL on IIS

- Configure endpoints for the ASP.NET Core Kestrel web server
- OWASP HSTS browser support

Hosting ASP.NET Core images with Docker over HTTPS

Article • 09/07/2024

By Rick Anderson

ASP.NET Core uses HTTPS by default. HTTPS relies on certificates for trust, identity, and encryption.

This document explains how to run pre-built container images with HTTPS using the .NET command-line interface (CLI). For instructions on how to run Docker in development with Visual Studio, see Developing ASP.NET Core Applications with Docker over HTTPS.

This sample requires Docker 17.06 or later of the Docker client.

Prerequisites

The current .NET SDK.

Certificates

A certificate from a certificate authority is required for production hosting for a domain. Let's Encrypt is a certificate authority that offers free certificates.

This document uses self-signed development certificates for hosting pre-built images over `localhost`. The instructions are similar to using production certificates. The certificate generated by `dotnet dev-certs` is for use with `localhost` only and should ***not*** be used in an environment like Kubernetes. To support HTTPS within a Kubernetes cluster, use the tools provided by Manage TLS Certificates in a Cluster to setup TLS within pods.

Use dotnet dev-certs to create self-signed certificates for development and testing.

For production certs:

- The `dotnet dev-certs` tool is not required.
- Certificates do not need to be stored in the location used in the instructions. Any location should work, although storing certs within your site directory is not recommended.

The instructions contained in the following section volume mount certificates into containers using Docker's `-v` command-line option. You could add certificates into container images with a `COPY` command in a *Dockerfile*, but it's not recommended. Copying certificates into an image isn't recommended for the following reasons:

- It's difficult to use the same image for testing with developer certificates.
- It's difficult to use the same image for Hosting with production certificates.
- There is significant risk of certificate disclosure.

Running pre-built container images with HTTPS

Use the following instructions for your operating system configuration.

Windows using Linux containers

Generate a certificate and configure the local machine:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
<CREDENTIAL_PLACEHOLDER>
dotnet dev-certs https --trust
```

In the preceding commands, replace `<CREDENTIAL_PLACEHOLDER>` with a password.

Run the container image with ASP.NET Core configured for HTTPS in a command shell:

Console

```
docker pull mcr.microsoft.com/dotnet/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e
ASPNETCORE_URLS="https://+;http://+" -e ASPNETCORE_HTTPS_PORTS=8001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="
<CREDENTIAL_PLACEHOLDER>" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v
%USERPROFILE%\.aspnet\https:/https/
mcr.microsoft.com/dotnet/samples:aspnetapp
```

In the preceding code, replace `<CREDENTIAL_PLACEHOLDER>` with the password. The password must match the password used for the certificate.

When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Note: The certificate in this case must be a `.pfx` file. Utilizing a `.crt` or `.key` file with or without the password isn't supported with the sample container. For example, when specifying a `.crt` file, the container may return error messages such as 'The server mode SSL must use a certificate with the associated private key.'. When using WSL, validate the mount path to ensure that the certificate loads correctly.

macOS or Linux

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep ${HOME}/.aspnet/https/aspnetapp.pfx -p
<CREDENTIAL_PLACEHOLDER>
dotnet dev-certs https --trust
```

On Linux, `dotnet dev-certs https --trust` requires .NET 9 SDK or later. For Linux on .NET 8.0.401 SDK and earlier, see your Linux distribution's documentation for trusting a certificate.

In the preceding commands, replace `<CREDENTIAL_PLACEHOLDER>` with a password.

Run the container image with ASP.NET Core configured for HTTPS:

Console

```
docker pull mcr.microsoft.com/dotnet/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e
ASPNETCORE_URLS="https://+;http://+" -e ASPNETCORE_HTTPS_PORTS=8001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="
<CREDENTIAL_PLACEHOLDER>" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx -v
${HOME}/.aspnet/https:/https/ mcr.microsoft.com/dotnet/samples:aspnetapp
```

In the preceding code, replace `<CREDENTIAL_PLACEHOLDER>` with the password. The password must match the password used for the certificate.

Windows using Windows containers

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
<CREDENTIAL_PLACEHOLDER>
```

```
dotnet dev-certs https --trust
```

In the preceding commands, replace `<CREDENTIAL_PLACEHOLDER>` with a password. When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Run the container image with ASP.NET Core configured for HTTPS:

Console

```
docker pull mcr.microsoft.com/dotnet/samples:aspnetapp
docker run --rm -it -p 8000:80 -p 8001:443 -e
ASPNETCORE_URLS="https://+;http://+" -e ASPNETCORE_HTTPS_PORTS=8001 -e
ASPNETCORE_Kestrel__Certificates__Default__Password="
<CREDENTIAL_PLACEHOLDER>" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=c:\https\aspnetapp.pfx -v
%USERPROFILE%\.aspnet\https:C:\https\ --user ContainerAdministrator
mcr.microsoft.com/dotnet/samples:aspnetapp
```

NOTE: `<CREDENTIAL_PLACEHOLDER>` is a placeholder for the Kestrel certificates default password.

The password must match the password used for the certificate. When using PowerShell, replace `%USERPROFILE%` with `$env:USERPROFILE`.

Developing ASP.NET Core Applications with Docker over HTTPS

See Developing ASP.NET Core Applications with Docker over HTTPS for information and samples on how to develop ASP.NET Core applications with HTTPS in Docker containers.

See also

- Developing ASP.NET Core Applications with Docker over HTTPS
- dotnet dev-certs

Hosting ASP.NET Core images with Docker Compose over HTTPS

Article • 09/07/2024

ASP.NET Core uses HTTPS by default. HTTPS relies on certificates for trust, identity, and encryption.

This document explains how to run pre-built container images with HTTPS.

See Developing ASP.NET Core Applications with Docker over HTTPS for development scenarios.

This sample requires Docker 17.06 or later of the Docker client.

Prerequisites

The .NET Core 2.2 SDK or later is required for some of the instructions in this document.

Certificates

A certificate from a certificate authority is required for production hosting for a domain. Let's Encrypt is a certificate authority that offers free certificates.

This document uses self-signed development certificates for hosting pre-built images over `localhost`. The instructions are similar to using production certificates.

For production certificates:

- The `dotnet dev-certs` tool is not required.
- Certificates don't need to be stored in the location used in the instructions. Store the certificates in any location outside the site directory.

The instructions contained in the following section volume mount certificates into containers using the `volumes` property in *docker-compose.yml*. You could add certificates into container images with a `COPY` command in a *Dockerfile*, but it's not recommended. Copying certificates into an image isn't recommended for the following reasons:

- It makes it difficult to use the same image for testing with developer certificates.
- It makes it difficult to use the same image for Hosting with production certificates.
- There is significant risk of certificate disclosure.

Starting a container with https support using docker compose

Use the following instructions for your operating system configuration.

Windows using Linux containers

Generate certificate and configure local machine:

PowerShell

```
dotnet dev-certs https -ep "$env:USERPROFILE\.aspnet\https\aspnetapp.pfx"  -
p $CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

The previous command using the .NET CLI:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
$CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

In the preceding commands, replace `$CREDENTIAL_PLACEHOLDER$` with a password.

Create a *docker-compose.debug.yml* file with the following content:

YAML

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
    volumes:
      - ~/.aspnet/https:/https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

Console

```
docker-compose -f "docker-compose.debug.yml" up -d
```

macOS or Linux

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep ${HOME}/.aspnet/https/aspnetapp.pfx -p
$CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

On Linux, `dotnet dev-certs https --trust` requires .NET 9 SDK or later. For Linux on .NET 8.0.401 SDK and earlier, see your Linux distribution's documentation for trusting a certificate.

In the preceding commands, replace `$CREDENTIAL_PLACEHOLDER$` with a password.

Create a *docker-compose.debug.yml* file with the following content:

YAML

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.pfx
    volumes:
      - ~/.aspnet/https:/https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

Console

```
docker-compose -f "docker-compose.debug.yml" up -d
```

Windows using Windows containers

Generate certificate and configure local machine:

.NET CLI

```
dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx -p
$CREDENTIAL_PLACEHOLDER$
dotnet dev-certs https --trust
```

In the preceding commands, replace `$CREDENTIAL_PLACEHOLDER$` with a password.

Create a *docker-compose.debug.yml* file with the following content:

YAML

```
version: '3.4'

services:
  webapp:
    image: mcr.microsoft.com/dotnet/samples:aspnetapp
    ports:
      - 80
      - 443
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password=password
      -
ASPNETCORE_Kestrel__Certificates__Default__Path=C:\https\aspnetapp.pfx
    volumes:
      - ${USERPROFILE}\.aspnet\https:C:\https:ro
```

The password specified in the docker compose file must match the password used for the certificate.

Start the container with ASP.NET Core configured for HTTPS:

Console

```
docker-compose -f "docker-compose.debug.yml" up -d
```

See also

- dotnet dev-certs

EU General Data Protection Regulation (GDPR) support in ASP.NET Core

Article • 04/04/2023

By Rick Anderson

ASP.NET Core provides APIs and templates to help meet some of the EU General Data Protection Regulation (GDPR) requirements:

- The project templates include extension points and stubbed markup that you can replace with your privacy and cookie use policy.
- The `Pages/Privacy.cshtml` page or `Views/Home/Privacy.cshtml` view provides a page to detail your site's privacy policy.

To enable the default cookie consent feature like that found in the ASP.NET Core 2.2 templates in a current ASP.NET Core template generated app, add the following highlighted code to `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<CookiePolicyOptions>(options =>
{
    // This lambda determines whether user consent for non-essential
    // cookies is needed for a given request.
    options.CheckConsentNeeded = context => true;

    options.MinimumSameSitePolicy = SameSiteMode.None;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseCookiePolicy();

app.UseRouting();

app.UseAuthorization();
```

```
app.MapRazorPages();

app.Run();
```

In the preceding code, CookiePolicyOptions and UseCookiePolicy are used.

- Add the cookie consent partial to the `_Layout.cshtml` file:

CSHTML

```
            @*Previous markup removed for brevity*@
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2022 - WebGDPR - <a asp-area="" asp-
page="/Privacy">Privacy</a>
        </div>
    </footer>

    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js">
</script>
    <script src="~/js/site.js" asp-append-version="true"></script>

    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

- Add the `_CookieConsentPartial.cshtml` file to the project:

CSHTML

```
@using Microsoft.AspNetCore.Http.Features

@{
    var consentFeature = Context.Features.Get<ITrackingConsentFeature>
();
    var showBanner = !consentFeature?.CanTrack ?? false;
    var cookieString = consentFeature?.CreateConsentCookie();
}

@if (showBanner)
{
```

```
    <div id="cookieConsent" class="alert alert-info alert-dismissible
fade show" role="alert">
        Use this space to summarize your privacy and cookie use policy.
<a asp-page="/Privacy">Learn More</a>.
        <button type="button" class="accept-policy close" data-bs-
dismiss="alert" aria-label="Close" data-cookie-string="@cookieString">
            <span aria-hidden="true">Accept</span>
        </button>
    </div>
    <script>
        (function () {
            var button = document.querySelector("#cookieConsent
button[data-cookie-string]");
            button.addEventListener("click", function (event) {
                document.cookie = button.dataset.cookieString;
            }, false);
        })();
    </script>
}
```

- Select the ASP.NET Core 2.2 version of this article to read about the cookie consent feature.

Customize the cookie consent value

Specify the value used to track if the user consented to the cookie use policy using the CookiePolicyOptions.ConsentCookieValue property:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.CheckConsentNeeded = context => true;
    options.MinimumSameSitePolicy = SameSiteMode.None;
    options.ConsentCookieValue = "true";
});

var app = builder.Build();
```

Encryption at rest

Some databases and storage mechanisms allow for encryption at rest. Encryption at rest:

- Encrypts stored data automatically.

- Encrypts without configuration, programming, or other work for the software that accesses the data.
- Is the easiest and safest option.
- Allows the database to manage keys and encryption.

For example:

- Microsoft SQL and Azure SQL provide Transparent Data Encryption (TDE).
- SQL Azure encrypts the database by default.
- Azure Blobs, Files, Table, and Queue Storage are encrypted by default.

For databases that don't provide built-in encryption at rest, you may be able to use disk encryption to provide the same protection. For example:

- BitLocker for Windows Server
- Linux:
  - eCryptfs
  - EncFS.

Additional resources

- Microsoft.com/GDPR

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core

Article • 10/21/2024

By Fiyaz Hasan and Rick Anderson

Cross-site request forgery is an attack against web-hosted apps whereby a malicious web app can influence the interaction between a client browser and a web app that trusts that browser. These attacks are possible because web browsers send some types of authentication tokens automatically with every request to a website. This form of exploit is also known as a *one-click attack* or *session riding* because the attack takes advantage of the user's previously authenticated session. Cross-site request forgery is also known as XSRF or CSRF.

An example of a CSRF attack:

1. A user signs into `www.good-banking-site.example.com` using forms authentication. The server authenticates the user and issues a response that includes an authentication cookie. The site is vulnerable to attack because it trusts any request that it receives with a valid authentication cookie.

2. The user visits a malicious site, `www.bad-crook-site.example.com`.

   The malicious site, `www.bad-crook-site.example.com`, contains an HTML form similar to the following example:

   HTML

   ```
   <h1>Congratulations! You're a Winner!</h1>
   <form action="https://good-banking-site.com/api/account" method="post">
       <input type="hidden" name="Transaction" value="withdraw" />
       <input type="hidden" name="Amount" value="1000000" />
       <input type="submit" value="Click to collect your prize!" />
   </form>
   ```

   Notice that the form's `action` posts to the vulnerable site, not to the malicious site. This is the "cross-site" part of CSRF.

3. The user selects the submit button. The browser makes the request and automatically includes the authentication cookie for the requested domain, `www.good-banking-site.example.com`.

4. The request runs on the `www.good-banking-site.example.com` server with the user's authentication context and can perform any action that an authenticated user is allowed to perform.

In addition to the scenario where the user selects the button to submit the form, the malicious site could:

- Run a script that automatically submits the form.
- Send the form submission as an AJAX request.
- Hide the form using CSS.

These alternative scenarios don't require any action or input from the user other than initially visiting the malicious site.

Using HTTPS doesn't prevent a CSRF attack. The malicious site can send an `https://www.good-banking-site.com/` request as easily as it can send an insecure request.

Some attacks target endpoints that respond to GET requests, in which case an image tag can be used to perform the action. This form of attack is common on forum sites that permit images but block JavaScript. Apps that change state on GET requests, where variables or resources are altered, are vulnerable to malicious attacks. **GET requests that change state are insecure. A best practice is to never change state on a GET request.**

CSRF attacks are possible against web apps that use cookies for authentication because:

- Browsers store cookies issued by a web app.
- Stored cookies include session cookies for authenticated users.
- Browsers send all of the cookies associated with a domain to the web app every request regardless of how the request to app was generated within the browser.

However, CSRF attacks aren't limited to exploiting cookies. For example, Basic and Digest authentication are also vulnerable. After a user signs in with Basic or Digest authentication, the browser automatically sends the credentials until the session ends.

In this context, *session* refers to the client-side session during which the user is authenticated. It's unrelated to server-side sessions or ASP.NET Core Session Middleware.

Users can protect against CSRF vulnerabilities by taking precautions:

- Sign out of web apps when finished using them.
- Clear browser cookies periodically.

However, CSRF vulnerabilities are fundamentally a problem with the web app, not the end user.

Authentication fundamentals

Cookie-based authentication is a popular form of authentication. Token-based authentication systems are growing in popularity, especially for Single Page Applications (SPAs).

Cookie-based authentication

When a user authenticates using their username and password they're issued a token containing an authentication ticket. The token can be used for authentication and authorization. The token is stored as a cookie that's sent with every request the client makes. Generating and validating this cookie is performed with the Cookie Authentication Middleware. The middleware serializes a user principal into an encrypted cookie. On subsequent requests, the middleware validates the cookie, recreates the principal, and assigns the principal to the HttpContext.User property.

Token-based authentication

When a user is authenticated, they're issued a token (not an antiforgery token). The token contains user information in the form of claims or a reference token that points the app to user state maintained in the app. When a user attempts to access a resource that requires authentication, the token is sent to the app with an extra authorization header in the form of a Bearer token. This approach makes the app stateless. In each subsequent request, the token is passed in the request for server-side validation. This token isn't *encrypted*; it's *encoded*. On the server, the token is decoded to access its information. To send the token on subsequent requests, store the token in the browser's local storage. Placing a token in the browser local storage and retrieving it and using it as a bearer token provides protection against CSRF attacks. However, should the app be vulnerable to script injection via XSS or a compromised external JavaScript file, a cyberattacker could retrieve any value from local storage and send it to themselves. ASP.NET Core encodes all server side output from variables by default, reducing the risk of XSS. If you override this behavior by using Html.Raw or custom code with untrusted input then you may increase the risk of XSS.

Don't be concerned about CSRF vulnerability if the token is stored in the browser's local storage. CSRF is a concern when the token is stored in a cookie. For more information, see the GitHub issue SPA code sample adds two cookies.

Multiple apps hosted at one domain

Shared hosting environments are vulnerable to session hijacking, sign-in CSRF, and other attacks.

Although `example1.contoso.net` and `example2.contoso.net` are different hosts, there's an implicit trust relationship between hosts under the `*.contoso.net` domain. This implicit trust relationship allows potentially untrusted hosts to affect each other's cookies (the same-origin policies that govern AJAX requests don't necessarily apply to HTTP cookies).

Attacks that exploit trusted cookies between apps hosted on the same domain can be prevented by not sharing domains. When each app is hosted on its own domain, there's no implicit cookie trust relationship to exploit.

Antiforgery in ASP.NET Core

> ⚠ **Warning**
>
> ASP.NET Core implements antiforgery using ASP.NET Core Data Protection. The data protection stack must be configured to work in a server farm. For more information, see Configuring data protection.

Antiforgery middleware is added to the Dependency injection container when one of the following APIs is called in `Program.cs`:

- AddMvc
- MapRazorPages
- MapControllerRoute
- AddRazorComponents

For more information, see Antiforgery with Minimal APIs.

The FormTagHelper injects antiforgery tokens into HTML form elements. The following markup in a Razor file automatically generates antiforgery tokens:

CSHTML

```
<form method="post">
    <!-- ... -->
</form>
```

Similarly, IHtmlHelper.BeginForm generates antiforgery tokens by default if the form's method isn't GET.

The automatic generation of antiforgery tokens for HTML form elements happens when the `<form>` tag contains the `method="post"` attribute and either of the following are true:

- The action attribute is empty (`action=""`).
- The action attribute isn't supplied (`<form method="post">`).

Automatic generation of antiforgery tokens for HTML form elements can be disabled:

- Explicitly disable antiforgery tokens with the `asp-antiforgery` attribute:

  CSHTML

  ```
  <form method="post" asp-antiforgery="false">
      <!-- ... -->
  </form>
  ```

- The form element is opted-out of Tag Helpers by using the Tag Helper ! opt-out symbol:

  CSHTML

  ```
  <!form method="post">
      <!-- ... -->
  </!form>
  ```

- Remove the `FormTagHelper` from the view. The `FormTagHelper` can be removed from a view by adding the following directive to the Razor view:

  CSHTML

  ```
  @removeTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.FormTagHelper,
  Microsoft.AspNetCore.Mvc.TagHelpers
  ```

> ⓘ Note
>
> Razor Pages are automatically protected from XSRF/CSRF. For more information, see XSRF/CSRF and Razor Pages.

The most common approach to protecting against CSRF attacks is to use the *Synchronizer Token Pattern* (STP). STP is used when the user requests a page with form data:

1. The server sends a token associated with the current user's identity to the client.
2. The client sends back the token to the server for verification.
3. If the server receives a token that doesn't match the authenticated user's identity, the request is rejected.

The token is unique and unpredictable. The token can also be used to ensure proper sequencing of a series of requests (for example, ensuring the request sequence of: page 1 > page 2 > page 3). All of the forms in ASP.NET Core MVC and Razor Pages templates generate antiforgery tokens. The following pair of view examples generates antiforgery tokens:

CSHTML

```
<form asp-action="Index" asp-controller="Home" method="post">
    <!-- ... -->
</form>

@using (Html.BeginForm("Index", "Home"))
{
    <!-- ... -->
}
```

Explicitly add an antiforgery token to a `<form>` element without using Tag Helpers with the HTML helper @Html.AntiForgeryToken:

CSHTML

```
<form asp-action="Index" asp-controller="Home" method="post">
    @Html.AntiForgeryToken()

    <!-- ... -->
</form>
```

In each of the preceding cases, ASP.NET Core adds a hidden form field similar to the following example:

HTML

```
<input name="__RequestVerificationToken" type="hidden" value="CfDJ8NrAkS ...
s2-m9Yw">
```

ASP.NET Core includes three filters for working with antiforgery tokens:

- ValidateAntiForgeryToken
- AutoValidateAntiforgeryToken

- IgnoreAntiforgeryToken

Antiforgery with `AddControllers`

Calling AddControllers does ***not*** enable antiforgery tokens. AddControllersWithViews must be called to have built-in antiforgery token support.

Multiple browser tabs and the Synchronizer Token Pattern

With the Synchronizer Token Pattern, only the most recently loaded page is guaranteed to contain a valid antiforgery token. Apps that wish to support multiple tabs should test supported browsers and log failures. ***Using multiple tabs can be problematic***. For example, if a user opens multiple tabs, requests made from previously loaded tabs might fail with an error: `Antiforgery token validation failed. The antiforgery cookie token and request token do not match`

Consider alternative CSRF protection patterns if this poses an issue.

Configure antiforgery with `AntiforgeryOptions`

Customize AntiforgeryOptions in `Program.cs`:

C#

```
builder.Services.AddAntiforgery(options =>
{
    // Set Cookie properties using CookieBuilder properties†.
    options.FormFieldName = "AntiforgeryFieldname";
    options.HeaderName = "X-CSRF-TOKEN-HEADERNAME";
    options.SuppressXFrameOptionsHeader = false;
});
```

Set the antiforgery `Cookie` properties using the properties of the CookieBuilder class, as shown in the following table.

Expand table

Option	Description
Cookie	Determines the settings used to create the antiforgery cookies.

Option	Description
FormFieldName	The name of the hidden form field used by the antiforgery system to render antiforgery tokens in views.
HeaderName	The name of the header used by the antiforgery system. If `null`, the system considers only form data.
SuppressXFrameOptionsHeader	Specifies whether to suppress generation of the `X-Frame-Options` header. By default, the header is generated with a value of "SAMEORIGIN". Defaults to `false`.

For more information, see CookieAuthenticationOptions.

Generate antiforgery tokens with `IAntiforgery`

IAntiforgery provides the API to configure antiforgery features. `IAntiforgery` can be requested in `Program.cs` using WebApplication.Services. The following example uses middleware from the app's home page to generate an antiforgery token and send it in the response as a cookie:

C#

```
app.UseRouting();

app.UseAuthorization();

var antiforgery = app.Services.GetRequiredService<IAntiforgery>();

app.Use((context, next) =>
{
    var requestPath = context.Request.Path.Value;

    if (string.Equals(requestPath, "/", StringComparison.OrdinalIgnoreCase)
        || string.Equals(requestPath, "/index.html",
StringComparison.OrdinalIgnoreCase))
    {
        var tokenSet = antiforgery.GetAndStoreTokens(context);
        context.Response.Cookies.Append("XSRF-TOKEN",
tokenSet.RequestToken!,
            new CookieOptions { HttpOnly = false });
    }

    return next(context);
});
```

The preceding example sets a cookie named `XSRF-TOKEN`. The client can read this cookie and provide its value as a header attached to AJAX requests. For example, Angular

includes built-in XSRF protection that reads a cookie named `XSRF-TOKEN` by default.

Require antiforgery validation

The ValidateAntiForgeryToken action filter can be applied to an individual action, a controller, or globally. Requests made to actions that have this filter applied are blocked unless the request includes a valid antiforgery token:

```
C#

[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Index()
{
    // ...

    return RedirectToAction();
}
```

The `ValidateAntiForgeryToken` attribute requires a token for requests to the action methods it marks, including HTTP GET requests. If the `ValidateAntiForgeryToken` attribute is applied across the app's controllers, it can be overridden with the `IgnoreAntiforgeryToken` attribute.

Automatically validate antiforgery tokens for unsafe HTTP methods only

Instead of broadly applying the `ValidateAntiForgeryToken` attribute and then overriding it with `IgnoreAntiforgeryToken` attributes, the AutoValidateAntiforgeryToken attribute can be used. This attribute works identically to the `ValidateAntiForgeryToken` attribute, except that it doesn't require tokens for requests made using the following HTTP methods:

- GET
- HEAD
- OPTIONS
- TRACE

We recommend use of `AutoValidateAntiforgeryToken` broadly for non-API scenarios. This attribute ensures POST actions are protected by default. The alternative is to ignore antiforgery tokens by default, unless `ValidateAntiForgeryToken` is applied to individual action methods. It's more likely in this scenario for a POST action method to be left

unprotected by mistake, leaving the app vulnerable to CSRF attacks. All POSTs should send the antiforgery token.

APIs don't have an automatic mechanism for sending the non-cookie part of the token. The implementation probably depends on the client code implementation. Some examples are shown below:

Class-level example:

```
C#

[AutoValidateAntiforgeryToken]
public class HomeController : Controller
```

Global example:

```
C#

builder.Services.AddControllersWithViews(options =>
{
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
});
```

Override global or controller antiforgery attributes

The IgnoreAntiforgeryToken filter is used to eliminate the need for an antiforgery token for a given action (or controller). When applied, this filter overrides `ValidateAntiForgeryToken` and `AutoValidateAntiforgeryToken` filters specified at a higher level (globally or on a controller).

```
C#

[IgnoreAntiforgeryToken]
public IActionResult IndexOverride()
{
    // ...

    return RedirectToAction();
}
```

Refresh tokens after authentication

Tokens should be refreshed after the user is authenticated by redirecting the user to a view or Razor Pages page.

JavaScript, AJAX, and SPAs

In traditional HTML-based apps, antiforgery tokens are passed to the server using hidden form fields. In modern JavaScript-based apps and SPAs, many requests are made programmatically. These AJAX requests may use other techniques, such as request headers or cookies, to send the token.

If cookies are used to store authentication tokens and to authenticate API requests on the server, CSRF is a potential problem. If local storage is used to store the token, CSRF vulnerability might be mitigated because values from local storage aren't sent automatically to the server with every request. Using local storage to store the antiforgery token on the client and sending the token as a request header is a recommended approach.

Blazor

For more information, see ASP.NET Core Blazor authentication and authorization.

JavaScript

Using JavaScript with views, the token can be created using a service from within the view. Inject the IAntiforgery service into the view and call GetAndStoreTokens:

CSHTML

```
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery Antiforgery

@{
    ViewData["Title"] = "JavaScript";

    var requestToken = Antiforgery.GetAndStoreTokens(Context).RequestToken;
}

<input id="RequestVerificationToken" type="hidden" value="@requestToken" />

<button id="button" class="btn btn-primary">Submit with Token</button>
<div id="result" class="mt-2"></div>

@section Scripts {
<script>
    document.addEventListener("DOMContentLoaded", () => {
        const resultElement = document.getElementById("result");

        document.getElementById("button").addEventListener("click", async ()
=> {

            const response = await fetch("@Url.Action("FetchEndpoint")", {
```

```
                method: "POST",
                headers: {
                    RequestVerificationToken:

document.getElementById("RequestVerificationToken").value
                }
            });

            if (response.ok) {
                resultElement.innerText = await response.text();
            } else {
                resultElement.innerText = `Request Failed:
${response.status}`
            }
        });
    });
</script>
}
```

The preceding example uses JavaScript to read the hidden field value for the AJAX POST header.

This approach eliminates the need to deal directly with setting cookies from the server or reading them from the client. However, when injecting the IAntiforgery service isn't possible, use JavaScript to access tokens in cookies:

- Access tokens in an additional request to the server, typically usually `same-origin`.
- Use the cookie's contents to create a header with the token's value.

Assuming the script sends the token in a request header called `X-XSRF-TOKEN`, configure the antiforgery service to look for the `X-XSRF-TOKEN` header:

C#

```
builder.Services.AddAntiforgery(options => options.HeaderName = "X-XSRF-
TOKEN");
```

The following example adds a protected endpoint that writes the request token to a JavaScript-readable cookie:

C#

```
app.UseAuthorization();
app.MapGet("antiforgery/token", (IAntiforgery forgeryService, HttpContext
context) =>
{
    var tokens = forgeryService.GetAndStoreTokens(context);
    context.Response.Cookies.Append("XSRF-TOKEN", tokens.RequestToken!,
            new CookieOptions { HttpOnly = false });
```

```
    return Results.Ok();
}).RequireAuthorization();
```

The following example uses JavaScript to make an AJAX request to obtain the token and make another request with the appropriate header:

JavaScript

```
var response = await fetch("/antiforgery/token", {
    method: "GET",
    headers: { "Authorization": authorizationToken }
});

if (response.ok) {
    // https://developer.mozilla.org/docs/web/api/document/cookie
    const xsrfToken = document.cookie
        .split("; ")
        .find(row => row.startsWith("XSRF-TOKEN="))
        .split("=")[1];

    response = await fetch("/JavaScript/FetchEndpoint", {
        method: "POST",
        headers: { "X-XSRF-TOKEN": xsrfToken, "Authorization":
authorizationToken }
    });

    if (response.ok) {
        resultElement.innerText = await response.text();
    } else {
        resultElement.innerText = `Request Failed: ${response.status}`
    }
} else {
    resultElement.innerText = `Request Failed: ${response.status}`
}
```

> **Note**
>
> When the antiforgery token is provided in both the request header and in the form payload, only the token in the header is validated.

Antiforgery with Minimal APIs

Call AddAntiforgery and UseAntiforgery(IApplicationBuilder) to register antiforgery services in DI. Antiforgery tokens are used to mitigate cross-site request forgery attacks.

C#

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The antiforgery middleware:

- Does ***not*** short-circuit the execution of the rest of the request pipeline.
- Sets the IAntiforgeryValidationFeature in the HttpContext.Features of the current request.

The antiforgery token is only validated if:

- The endpoint contains metadata implementing IAntiforgeryMetadata where `RequiresValidation=true`.
- The HTTP method associated with the endpoint is a relevant HTTP method. The relevant methods are all HTTP methods except for TRACE, OPTIONS, HEAD, and GET.
- The request is associated with a valid endpoint.

Note: When enabled manually, the antiforgery middleware must run after the authentication and authorization middleware to prevent reading form data when the user is unauthenticated.

By default, minimal APIs that accept form data require antiforgery token validation.

Consider the following `GenerateForm` method:

C#

```
public static string GenerateForm(string action,
    AntiforgeryTokenSet token, bool UseToken=true)
{
    string tokenInput = "";
    if (UseToken)
    {
        tokenInput = $@"<input name=""{token.FormFieldName}""
                        type=""hidden"" value=""{token.RequestToken}"" />";
    }

    return $@"
```

```
    <html><body>
        <form action=""{action}"" method=""POST"" enctype=""multipart/form-
data"">
            {tokenInput}
            <input type=""text"" name=""name"" />
            <input type=""date"" name=""dueDate"" />
            <input type=""checkbox"" name=""isCompleted"" />
            <input type=""submit"" />
        </form>
    </body></html>
";
}
```

The preceding code has three arguments, the action, the antiforgery token, and a `bool` indicating whether the token should be used.

Consider the following sample:

C#

```
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder();

builder.Services.AddAntiforgery();

var app = builder.Build();

app.UseAntiforgery();

// Pass token
app.MapGet("/", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    return Results.Content(MyHtml.GenerateForm("/todo", token),
"text/html");
});

// Don't pass a token, fails
app.MapGet("/SkipToken", (HttpContext context, IAntiforgery antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    return Results.Content(MyHtml.GenerateForm("/todo",token, false ),
"text/html");
});

// Post to /todo2. DisableAntiforgery on that endpoint so no token needed.
app.MapGet("/DisableAntiforgery", (HttpContext context, IAntiforgery
antiforgery) =>
{
    var token = antiforgery.GetAndStoreTokens(context);
    return Results.Content(MyHtml.GenerateForm("/todo2", token, false),
```

```
"text/html");
});

app.MapPost("/todo", ([FromForm] Todo todo) => Results.Ok(todo));

app.MapPost("/todo2", ([FromForm] Todo todo) => Results.Ok(todo))
                                                    .DisableAntiforgery();

app.Run();

class Todo
{
    public required string Name { get; set; }
    public bool IsCompleted { get; set; }
    public DateTime DueDate { get; set; }
}

public static class MyHtml
{
    public static string GenerateForm(string action,
        AntiforgeryTokenSet token, bool UseToken=true)
    {
        string tokenInput = "";
        if (UseToken)
        {
            tokenInput = $@"<input name=""{token.FormFieldName}""
                                type=""hidden"" value=""{token.RequestToken}""
/>";
        }

        return $@"
        <html><body>
            <form action=""{action}"" method=""POST""
enctype=""multipart/form-data"">
                {tokenInput}
                <input type=""text"" name=""name"" />
                <input type=""date"" name=""dueDate"" />
                <input type=""checkbox"" name=""isCompleted"" />
                <input type=""submit"" />
            </form>
        </body></html>
    ";
    }
}
```

In the preceding code, posts to:

- `/todo` require a valid antiforgery token.
- `/todo2` do ***not*** require a valid antiforgery token because DisableAntiforgery is called.

C#

```
app.MapPost("/todo", ([FromForm] Todo todo) => Results.Ok(todo));

app.MapPost("/todo2", ([FromForm] Todo todo) => Results.Ok(todo))
                                                .DisableAntiforgery();
```

A POST to:

- `/todo` from the form generated by the `/` endpoint succeeds because the antiforgery token is valid.
- `/todo` from the form generated by the `/SkipToken` fails because the antiforgery is not included.
- `/todo2` from the form generated by the `/DisableAntiforgery` endpoint succeeds because the antiforgery is not required.

C#

```
app.MapPost("/todo", ([FromForm] Todo todo) => Results.Ok(todo));

app.MapPost("/todo2", ([FromForm] Todo todo) => Results.Ok(todo))
                                                .DisableAntiforgery();
```

When a form is submitted without a valid antiforgery token:

- In the development environment, an exception is thrown.
- In the production environment, a message is logged.

Windows authentication and antiforgery cookies

When using Windows Authentication, application endpoints must be protected against CSRF attacks in the same way as done for cookies. The browser implicitly sends the authentication context to the server and endpoints need to be protected against CSRF attacks.

Extend antiforgery

The IAntiforgeryAdditionalDataProvider type allows developers to extend the behavior of the anti-CSRF system by round-tripping additional data in each token. The GetAdditionalData method is called each time a field token is generated, and the return value is embedded within the generated token. An implementer could return a timestamp, a nonce, or any other value and then call ValidateAdditionalData to validate

this data when the token is validated. The client's username is already embedded in the generated tokens, so there's no need to include this information. If a token includes supplemental data but no `IAntiForgeryAdditionalDataProvider` is configured, the supplemental data isn't validated.

Additional resources

- Host ASP.NET Core in a web farm

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Prevent open redirect attacks in ASP.NET Core

Article • 06/03/2022

A web app that redirects to a URL that's specified via the request such as the querystring or form data can potentially be tampered with to redirect users to an external, malicious URL. This tampering is called an open redirection attack.

Whenever your application logic redirects to a specified URL, you must verify that the redirection URL hasn't been tampered with. ASP.NET Core has built-in functionality to help protect apps from open redirect (also known as open redirection) attacks.

What is an open redirect attack?

Web applications frequently redirect users to a login page when they access resources that require authentication. The redirection typically includes a `returnUrl` querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in. After the user authenticates, they're redirected to the URL they had originally requested.

Because the destination URL is specified in the querystring of the request, a malicious user could tamper with the querystring. A tampered querystring could allow the site to redirect the user to an external, malicious site. This technique is called an open redirect (or redirection) attack.

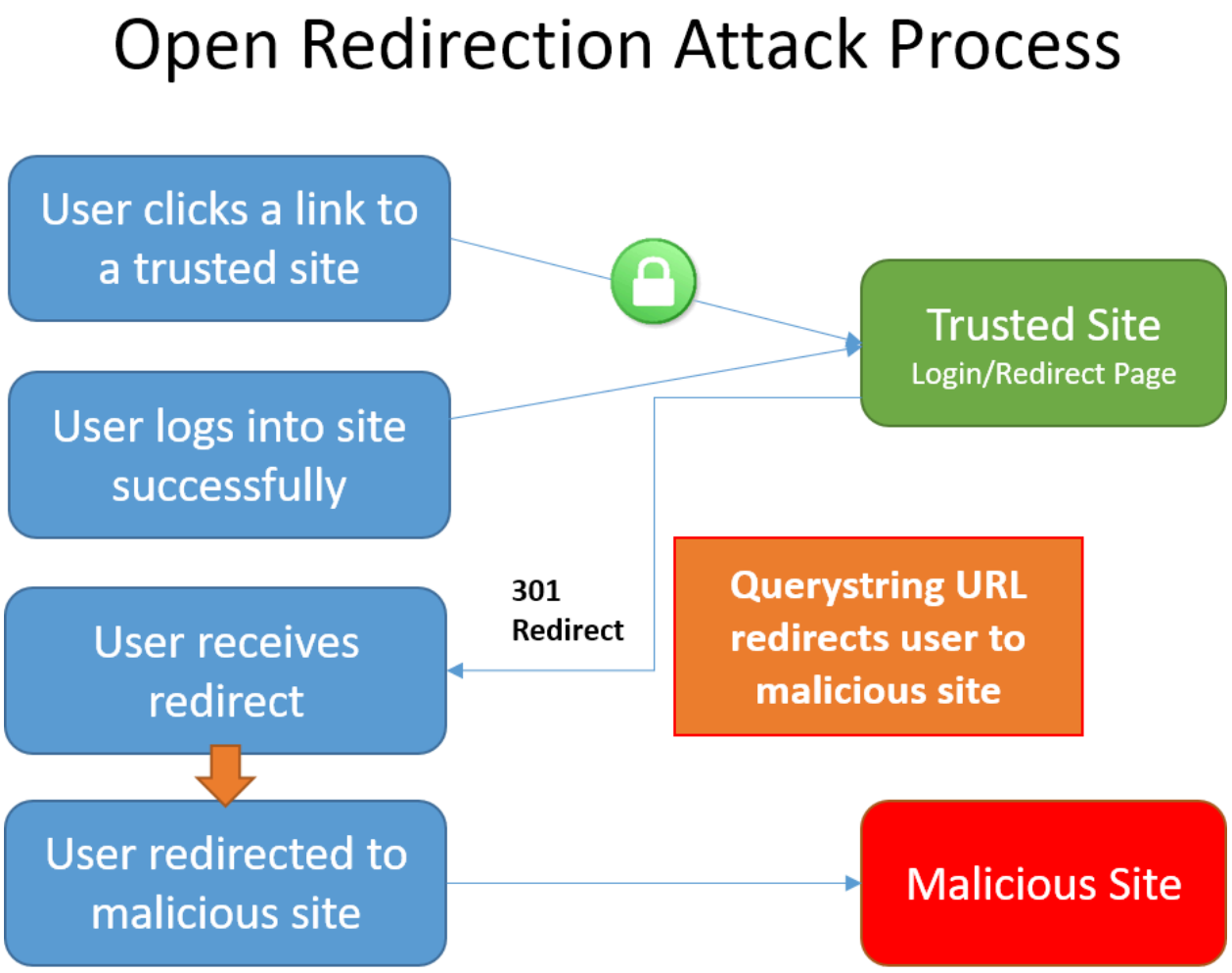
An example attack

A malicious user can develop an attack intended to allow the malicious user access to a user's credentials or sensitive information. To begin the attack, the malicious user convinces the user to click a link to your site's login page with a `returnUrl` querystring value added to the URL. For example, consider an app at `contoso.com` that includes a login page at `http://contoso.com/Account/LogOn?returnUrl=/Home/About`. The attack follows these steps:

1. The user clicks a malicious link to `http://contoso.com/Account/LogOn?returnUrl=http://contoso1.com/Account/LogOn` (the second URL is "contoso**1**.com", not "contoso.com").
2. The user logs in successfully.

3. The user is redirected (by the site) to `http://contoso1.com/Account/LogOn` (a malicious site that looks exactly like real site).
4. The user logs in again (giving malicious site their credentials) and is redirected back to the real site.

The user likely believes that their first attempt to log in failed and that their second attempt is successful. The user most likely remains unaware that their credentials are compromised.



In addition to login pages, some sites provide redirect pages or endpoints. Imagine your app has a page with an open redirect, `/Home/Redirect`. A cyberattacker could create, for example, a link in an email that goes to `[yoursite]/Home/Redirect?url=http://phishingsite.com/Home/Login`. A typical user will look at the URL and see it begins with your site name. Trusting that, they will click the link. The open redirect would then send the user to the phishing site, which looks identical to yours, and the user would likely login to what they believe is your site.

Protecting against open redirect attacks

When developing web applications, treat all user-provided data as untrustworthy. If your application has functionality that redirects the user based on the contents of the URL, ensure that such redirects are only done locally within your app (or to a known URL, not any URL that may be supplied in the querystring).

LocalRedirect

Use the `LocalRedirect` helper method from the base `Controller` class:

C#

```
public IActionResult SomeAction(string redirectUrl)
{
    return LocalRedirect(redirectUrl);
}
```

`LocalRedirect` will throw an exception if a non-local URL is specified. Otherwise, it behaves just like the `Redirect` method.

IsLocalUrl

Use the IsLocalUrl method to test URLs before redirecting:

The following example shows how to check whether a URL is local before redirecting.

C#

```
private IActionResult RedirectToLocal(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction(nameof(HomeController.Index), "Home");
    }
}
```

The `IsLocalUrl` method protects users from being inadvertently redirected to a malicious site. You can log the details of the URL that was provided when a non-local URL is supplied in a situation where you expected a local URL. Logging redirect URLs may help in diagnosing redirection attacks.

Prevent Cross-Site Scripting (XSS) in ASP.NET Core

Article • 09/27/2024

By Rick Anderson

Cross-Site Scripting (XSS) is a security vulnerability that enables a cyberattacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages, the cyberattacker's scripts run, enabling the cyberattacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation, or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it to a page without validating, encoding or escaping it.

This article applies primarily to ASP.NET Core MVC with views, Razor Pages, and other apps that return HTML that may be vulnerable to XSS. Web APIs that return data in the form of HTML, XML, or JSON can trigger XSS attacks in their client apps if they don't properly sanitize user input, depending on how much trust the client app places in the API. For example, if an API accepts user-generated content and returns it in an HTML response, a cyberattacker could inject malicious scripts into the content that executes when the response is rendered in the user's browser.

To prevent XSS attacks, web APIs should implement input validation and output encoding. Input validation ensures that user input meets expected criteria and doesn't include malicious code. Output encoding ensures that any data returned by the API is properly sanitized so that it can't be executed as code by the user's browser. For more information, see this GitHub issue.

Protecting your application against XSS

At a basic level, XSS works by tricking your application into inserting a `<script>` tag into your rendered page, or by inserting an `On*` event into an element. Developers should use the following prevention steps to avoid introducing XSS into their applications:

1. Never put untrusted data into your HTML input, unless you follow the rest of the steps below. Untrusted data is any data that may be controlled by a cyberattacker, such as HTML form inputs, query strings, HTTP headers, or even data sourced from a database, as a cyberattacker may be able to breach your database even if they can't breach your application.

2. Before putting untrusted data inside an HTML element, ensure it's HTML encoded. HTML encoding takes characters such as < and changes them into a safe form like <.
3. Before putting untrusted data into an HTML attribute, ensure it's HTML encoded. HTML attribute encoding is a subset of HTML encoding and encodes double quote ("), single quote ('), ampersand (&), and less-than (<) characters.
4. Before putting untrusted data into JavaScript, place the data in an HTML element whose contents you retrieve at runtime. If this isn't possible, then ensure the data is JavaScript encoded. JavaScript encoding takes dangerous characters for JavaScript and replaces them with their hex, for example, < would be encoded as `\u003C`.
5. Before putting untrusted data into a URL query string ensure it's URL encoded.

HTML Encoding using Razor

The Razor engine used in MVC automatically encodes all output sourced from variables, unless you work really hard to prevent it doing so. It uses HTML attribute encoding rules whenever you use the @ directive. As HTML attribute encoding is a superset of HTML encoding this means you don't have to concern yourself with whether you should use HTML encoding or HTML attribute encoding. You must ensure that you only use @ in an HTML context, not when attempting to insert untrusted input directly into JavaScript. Tag helpers will also encode input you use in tag parameters.

Take the following Razor view:

CSHTML

```
@{
    var untrustedInput = "<\"123\">";
}

@untrustedInput
```

This view outputs the contents of the *untrustedInput* variable. This variable includes some characters which are used in XSS attacks, namely <, " and >. Examining the source shows the rendered output encoded as:

HTML

```
&lt;&quot;123&quot;&gt;
```

> ⚠ **Warning**

ASP.NET Core MVC provides an `HtmlString` class which isn't automatically encoded upon output. This should never be used in combination with untrusted input as this will expose an XSS vulnerability.

JavaScript Encoding using Razor

There may be times you want to insert a value into JavaScript to process in your view. There are two ways to do this. The safest way to insert values is to place the value in a data attribute of a tag and retrieve it in your JavaScript. For example:

CSHTML

```
@{
    var untrustedInput = "<script>alert(1)</script>";
}

<div id="injectedData"
     data-untrustedinput="@untrustedInput" />

<div id="scriptedWrite" />
<div id="scriptedWrite-html5" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    // Put the injected, untrusted data into the scriptedWrite div tag.
    // Do NOT use document.write() on dynamically generated data as it
    // can lead to XSS.

    document.getElementById("scriptedWrite").innerText +=
clientSideUntrustedInputOldStyle;

    // Or you can use createElement() to dynamically create document
elements
    // This time we're using textContent to ensure the data is properly
encoded.
    var x = document.createElement("div");
    x.textContent = clientSideUntrustedInputHtml5;
    document.body.appendChild(x);

    // You can also use createTextNode on an element to ensure data is
properly encoded.
```

```
    var y = document.createElement("div");
    y.appendChild(document.createTextNode(clientSideUntrustedInputHtml5));
    document.body.appendChild(y);

</script>
```

The preceding markup generates the following HTML:

HTML

```
<div id="injectedData"
     data-untrustedinput="&lt;script&gt;alert(1)&lt;/script&gt;" />

<div id="scriptedWrite" />
<div id="scriptedWrite-html5" />

<script>
    var injectedData = document.getElementById("injectedData");

    // All clients
    var clientSideUntrustedInputOldStyle =
        injectedData.getAttribute("data-untrustedinput");

    // HTML 5 clients only
    var clientSideUntrustedInputHtml5 =
        injectedData.dataset.untrustedinput;

    // Put the injected, untrusted data into the scriptedWrite div tag.
    // Do NOT use document.write() on dynamically generated data as it can
    // lead to XSS.

    document.getElementById("scriptedWrite").innerText +=
clientSideUntrustedInputOldStyle;

    // Or you can use createElement() to dynamically create document
elements
    // This time we're using textContent to ensure the data is properly
encoded.
    var x = document.createElement("div");
    x.textContent = clientSideUntrustedInputHtml5;
    document.body.appendChild(x);

    // You can also use createTextNode on an element to ensure data is
properly encoded.
    var y = document.createElement("div");
    y.appendChild(document.createTextNode(clientSideUntrustedInputHtml5));
    document.body.appendChild(y);

</script>
```

The preceding code generates the following output:

```
<script>alert(1)</script>
<script>alert(1)</script>
<script>alert(1)</script>
```

> ⚠ **Warning**
>
> Do ***NOT*** concatenate untrusted input in JavaScript to create DOM elements or use `document.write()` on dynamically generated content.
>
> Use one of the following approaches to prevent code from being exposed to DOM-based XSS:
>
> - `createElement()` and assign property values with appropriate methods or properties such as `node.textContent=` or `node.InnerText=`.
> - `document.CreateTextNode()` and append it in the appropriate DOM location.
> - `element.SetAttribute()`
> - `element[attribute]=`

Accessing encoders in code

The HTML, JavaScript and URL encoders are available to your code in two ways:

- Inject them via dependency injection.
- Use the default encoders contained in the `System.Text.Encodings.Web` namespace.

When using the default encoders, then any customizations applied to character ranges to be treated as safe won't take effect. The default encoders use the safest encoding rules possible.

To use the configurable encoders via DI your constructors should take an *HtmlEncoder*, *JavaScriptEncoder* and *UrlEncoder* parameter as appropriate. For example;

C#

```
public class HomeController : Controller
{
    HtmlEncoder _htmlEncoder;
    JavaScriptEncoder _javaScriptEncoder;
    UrlEncoder _urlEncoder;

    public HomeController(HtmlEncoder htmlEncoder,
                          JavaScriptEncoder javascriptEncoder,
```

```
                            UrlEncoder urlEncoder)
    {
        _htmlEncoder = htmlEncoder;
        _javaScriptEncoder = javascriptEncoder;
        _urlEncoder = urlEncoder;
    }
}
```

Encoding URL Parameters

If you want to build a URL query string with untrusted input as a value use the `UrlEncoder` to encode the value. For example,

C#

```
var example = "\"Quoted Value with spaces and &\"";
var encodedValue = _urlEncoder.Encode(example);
```

After encoding the encodedValue variable contains `%22Quoted%20Value%20with%20spaces%20and%20%26%22`. Spaces, quotes, punctuation and other unsafe characters are percent encoded to their hexadecimal value, for example a space character will become %20.

> ⚠ **Warning**
>
> Don't use untrusted input as part of a URL path. Always pass untrusted input as a query string value.

Customizing the Encoders

By default encoders use a safe list limited to the Basic Latin Unicode range and encode all characters outside of that range as their character code equivalents. This behavior also affects Razor TagHelper and HtmlHelper rendering as it uses the encoders to output your strings.

The reasoning behind this is to protect against unknown or future browser bugs (previous browser bugs have tripped up parsing based on the processing of non-English characters). If your web site makes heavy use of non-Latin characters, such as Chinese, Cyrillic or others this is probably not the behavior you want.

The encoder safe lists can be customized to include Unicode ranges appropriate to the app during startup, in `Program.cs`:

For example, using the default configuration using a Razor HtmlHelper similar to the following:

```HTML
<p>This link text is in Chinese: @Html.ActionLink("汉语/漢語", "Index")</p>
```

The preceding markup is rendered with Chinese text encoded:

```HTML
<p>This link text is in Chinese: <a
href="/">&#x6C49;&#x8BED;/&#x6F22;&#x8A9E;</a></p>
```

To widen the characters treated as safe by the encoder, insert the following line into `Program.cs`.:

```C#
builder.Services.AddSingleton<HtmlEncoder>(
     HtmlEncoder.Create(allowedRanges: new[] { UnicodeRanges.BasicLatin,

UnicodeRanges.CjkUnifiedIdeographs }));
```

This example widens the safe list to include the Unicode Range CjkUnifiedIdeographs. The rendered output would now become

```HTML
<p>This link text is in Chinese: <a href="/">汉语/漢語</a></p>
```

Safe list ranges are specified as Unicode code charts, not languages. The Unicode standard has a list of code charts you can use to find the chart containing your characters. Each encoder, Html, JavaScript and Url, must be configured separately.

> **Note**
>
> Customization of the safe list only affects encoders sourced via DI. If you directly access an encoder via `System.Text.Encodings.Web.*Encoder.Default` then the default, Basic Latin only safelist will be used.

Where should encoding take place?

The general accepted practice is that encoding takes place at the point of output and encoded values should never be stored in a database. Encoding at the point of output allows you to change the use of data, for example, from HTML to a query string value. It also enables you to easily search your data without having to encode values before searching and allows you to take advantage of any changes or bug fixes made to encoders.

Validation as an XSS prevention technique

Validation can be a useful tool in limiting XSS attacks. For example, a numeric string containing only the characters 0-9 won't trigger an XSS attack. Validation becomes more complicated when accepting HTML in user input. Parsing HTML input is difficult, if not impossible. Markdown, coupled with a parser that strips embedded HTML, is a safer option for accepting rich input. Never rely on validation alone. Always encode untrusted input before output, no matter what validation or sanitization has been performed.

Enable Cross-Origin Requests (CORS) in ASP.NET Core

Article • 09/02/2024

By Rick Anderson and Kirk Larkin

This article shows how **C**ross-**O**rigin **R**esource **S**haring (CORS) is enabled in an ASP.NET Core app.

Browser security prevents a web page from making requests to a different domain than the one that served the web page. This restriction is called the *same-origin policy*. The same-origin policy prevents a malicious site from reading sensitive data from another site. Sometimes, you might want to allow other sites to make cross-origin requests to your app. For more information, see the Mozilla CORS article.

Cross Origin Resource Sharing (CORS):

- Is a W3C standard that allows a server to relax the same-origin policy.
- Is **not** a security feature, CORS relaxes security. An API is not safer by allowing CORS. For more information, see How CORS works.
- Allows a server to explicitly allow some cross-origin requests while rejecting others.
- Is safer and more flexible than earlier techniques, such as JSONP.

View or download sample code (how to download)

Same origin

Two URLs have the same origin if they have identical schemes, hosts, and ports (RFC 6454).

These two URLs have the same origin:

- `https://example.com/foo.html`
- `https://example.com/bar.html`

These URLs have different origins than the previous two URLs:

- `https://example.net`: Different domain
- `https://contoso.example.com/foo.html`: Different subdomain
- `http://example.com/foo.html`: Different scheme
- `https://example.com:9000/foo.html`: Different port

Enable CORS

There are three ways to enable CORS:

- In middleware using a named policy or default policy.
- Using endpoint routing.
- With the [EnableCors] attribute.

Using the [EnableCors] attribute with a named policy provides the finest control in limiting endpoints that support CORS.

> ⚠ **Warning**
>
> UseCors must be called in the correct order. For more information, see Middleware order. For example, `UseCors` must be called before UseResponseCaching when using `UseResponseCaching`.

Each approach is detailed in the following sections.

CORS with named policy and middleware

CORS Middleware handles cross-origin requests. The following code applies a CORS policy to all the app's endpoints with the specified origins:

C#

```
var  MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      policy  =>
                      {
                          policy.WithOrigins("http://example.com",
                                             "http://www.contoso.com");
                      });
});

// services.AddResponseCaching();

builder.Services.AddControllers();

var app = builder.Build();
app.UseHttpsRedirection();
```

```
app.UseStaticFiles();
app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code:

- Sets the policy name to `_myAllowSpecificOrigins`. The policy name is arbitrary.
- Calls the UseCors extension method and specifies the `_myAllowSpecificOrigins` CORS policy. `UseCors` adds the CORS middleware. The call to `UseCors` must be placed after `UseRouting`, but before `UseAuthorization`. For more information, see Middleware order.
- Calls AddCors with a lambda expression. The lambda takes a CorsPolicyBuilder object. Configuration options, such as `WithOrigins`, are described later in this article.
- Enables the `_myAllowSpecificOrigins` CORS policy for all controller endpoints. See endpoint routing to apply a CORS policy to specific endpoints.
- When using Response Caching Middleware, call UseCors before UseResponseCaching.

With endpoint routing, the CORS middleware **must** be configured to execute between the calls to `UseRouting` and `UseEndpoints`.

The AddCors method call adds CORS services to the app's service container:

C#

```
var  MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      policy  =>
                      {
                          policy.WithOrigins("http://example.com",
                                             "http://www.contoso.com");
                      });
});

// services.AddResponseCaching();
```

```
builder.Services.AddControllers();

var app = builder.Build();
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

app.UseAuthorization();

app.MapControllers();

app.Run();
```

For more information, see CORS policy options in this document.

The CorsPolicyBuilder methods can be chained, as shown in the following code:

C#

```
var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(MyAllowSpecificOrigins,
                      policy =>
                      {
                          policy.WithOrigins("http://example.com",
                                             "http://www.contoso.com")
                                             .AllowAnyHeader()
                                             .AllowAnyMethod();
                      });
});

builder.Services.AddControllers();

var app = builder.Build();
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors(MyAllowSpecificOrigins);

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Note: The specified URL must **not** contain a trailing slash (`/`). If the URL terminates with `/`, the comparison returns `false` and no header is returned.

UseCors and UseStaticFiles order

Typically, `UseStaticFiles` is called before `UseCors`. Apps that use JavaScript to retrieve static files cross site must call `UseCors` before `UseStaticFiles`.

CORS with default policy and middleware

The following highlighted code enables the default CORS policy:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(
        policy =>
        {
            policy.WithOrigins("http://example.com",
                               "http://www.contoso.com");
        });
});

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The preceding code applies the default CORS policy to all controller endpoints.

Enable Cors with endpoint routing

With endpoint routing, CORS can be enabled on a per-endpoint basis using the RequireCors set of extension methods:

C#

```
var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
                      policy =>
                      {
                          policy.WithOrigins("http://example.com",
                                             "http://www.contoso.com");
                      });
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/echo",
        context => context.Response.WriteAsync("echo"))
        .RequireCors(MyAllowSpecificOrigins);

    endpoints.MapControllers()
             .RequireCors(MyAllowSpecificOrigins);

    endpoints.MapGet("/echo2",
        context => context.Response.WriteAsync("echo2"));

    endpoints.MapRazorPages();
});

app.Run();
```

In the preceding code:

- `app.UseCors` enables the CORS middleware. Because a default policy hasn't been configured, `app.UseCors()` alone doesn't enable CORS.
- The `/echo` and controller endpoints allow cross-origin requests using the specified policy.
- The `/echo2` and Razor Pages endpoints do **not** allow cross-origin requests because no default policy was specified.

The [DisableCors] attribute does **not** disable CORS that has been enabled by endpoint routing with `RequireCors`.

See Test CORS with [EnableCors] attribute and RequireCors method for instructions on testing code similar to the preceding.

Enable CORS with attributes

Enabling CORS with the [EnableCors] attribute and applying a named policy to only those endpoints that require CORS provides the finest control.

The [EnableCors] attribute provides an alternative to applying CORS globally. The `[EnableCors]` attribute enables CORS for selected endpoints, rather than all endpoints:

- `[EnableCors]` specifies the default policy.
- `[EnableCors("{Policy String}")]` specifies a named policy.

The `[EnableCors]` attribute can be applied to:

- Razor Page `PageModel`
- Controller
- Controller action method

Different policies can be applied to controllers, page models, or action methods with the `[EnableCors]` attribute. When the `[EnableCors]` attribute is applied to a controller, page model, or action method, and CORS is enabled in middleware, **both** policies are applied. **We recommend against combining policies. Use the `[EnableCors]` attribute or middleware, not both in the same app.**

The following code applies a different policy to each method:

C#

```
[Route("api/[controller]")]
[ApiController]
public class WidgetController : ControllerBase
{
```

```
    // GET api/values
    [EnableCors("AnotherPolicy")]
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "green widget", "red widget" };
    }

    // GET api/values/5
    [EnableCors("Policy1")]
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    {
        return id switch
        {
            1 => "green widget",
            2 => "red widget",
            _ => NotFound(),
        };
    }
}
```

The following code creates two CORS policies:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("Policy1",
        policy =>
        {
            policy.WithOrigins("http://example.com",
                               "http://www.contoso.com");
        });

    options.AddPolicy("AnotherPolicy",
        policy =>
        {
            policy.WithOrigins("http://www.contoso.com")
                               .AllowAnyHeader()
                               .AllowAnyMethod();
        });
});

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseRouting();
```

```
app.UseCors();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

For the finest control of limiting CORS requests:

- Use `[EnableCors("MyPolicy")]` with a named policy.
- Don't define a default policy.
- Don't use endpoint routing.

The code in the next section meets the preceding list.

Disable CORS

The [DisableCors] attribute does **not** disable CORS that has been enabled by endpoint routing.

The following code defines the CORS policy `"MyPolicy"`:

C#

```
var MyAllowSpecificOrigins = "_myAllowSpecificOrigins";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "MyPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com",
                               "http://www.contoso.com")
                  .WithMethods("PUT", "DELETE", "GET");
        });
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
```

```
app.UseCors();

app.UseAuthorization();

app.UseEndpoints(endpoints => {
    endpoints.MapControllers();
    endpoints.MapRazorPages();
});

app.Run();
```

The following code disables CORS for the `GetValues2` action:

C#

```
[EnableCors("MyPolicy")]
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public IActionResult Get() =>
        ControllerContext.MyDisplayRouteInfo();

    // GET api/values/5
    [HttpGet("{id}")]
    public IActionResult Get(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // PUT api/values/5
    [HttpPut("{id}")]
    public IActionResult Put(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);


    // GET: api/values/GetValues2
    [DisableCors]
    [HttpGet("{action}")]
    public IActionResult GetValues2() =>
        ControllerContext.MyDisplayRouteInfo();

}
```

The preceding code:

- Doesn't enable CORS with endpoint routing.
- Doesn't define a default CORS policy.
- Uses [EnableCors("MyPolicy")] to enable the `"MyPolicy"` CORS policy for the controller.

- Disables CORS for the `GetValues2` method.

See Test CORS for instructions on testing the preceding code.

CORS policy options

This section describes the various options that can be set in a CORS policy:

- Set the allowed origins
- Set the allowed HTTP methods
- Set the allowed request headers
- Set the exposed response headers
- Credentials in cross-origin requests
- Set the preflight expiration time

AddPolicy is called in `Program.cs`. For some options, it may be helpful to read the How CORS works section first.

Set the allowed origins

AllowAnyOrigin: Allows CORS requests from all origins with any scheme (`http` or `https`). `AllowAnyOrigin` is insecure because *any website* can make cross-origin requests to the app.

> **Note**
>
> Specifying `AllowAnyOrigin` and `AllowCredentials` is an insecure configuration and can result in cross-site request forgery. The CORS service returns an invalid CORS response when an app is configured with both methods.

`AllowAnyOrigin` affects preflight requests and the `Access-Control-Allow-Origin` header. For more information, see the Preflight requests section.

SetIsOriginAllowedToAllowWildcardSubdomains: Sets the IsOriginAllowed property of the policy to be a function that allows origins to match a configured wildcard domain when evaluating if the origin is allowed.

C#

```
var MyAllowSpecificOrigins = "_MyAllowSubdomainPolicy";

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                .SetIsOriginAllowedToAllowWildcardSubdomains();
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

Set the allowed HTTP methods

AllowAnyMethod:

- Allows any HTTP method:
- Affects preflight requests and the `Access-Control-Allow-Methods` header. For more information, see the Preflight requests section.

Set the allowed request headers

To allow specific headers to be sent in a CORS request, called *author request headers*, call WithHeaders and specify the allowed headers:

C#

```
using Microsoft.Net.Http.Headers;

var MyAllowSpecificOrigins = "_MyAllowSubdomainPolicy";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
       policy =>
       {
           policy.WithOrigins("http://example.com")
                  .WithHeaders(HeaderNames.ContentType, "x-custom-header");
       });
});

builder.Services.AddControllers();
```

```
var app = builder.Build();
```

To allow all author request headers, call AllowAnyHeader:

C#

```
var MyAllowSpecificOrigins = "_MyAllowSubdomainPolicy";

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: MyAllowSpecificOrigins,
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                  .AllowAnyHeader();
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

`AllowAnyHeader` affects preflight requests and the Access-Control-Request-Headers header. For more information, see the Preflight requests section.

A CORS Middleware policy match to specific headers specified by `WithHeaders` is only possible when the headers sent in `Access-Control-Request-Headers` exactly match the headers stated in `WithHeaders`.

For instance, consider an app configured as follows:

C#

```
app.UseCors(policy => policy.WithHeaders(HeaderNames.CacheControl));
```

CORS Middleware declines a preflight request with the following request header because `Content-Language` (HeaderNames.ContentLanguage) isn't listed in `WithHeaders`:

```
Access-Control-Request-Headers: Cache-Control, Content-Language
```

The app returns a *200 OK* response but doesn't send the CORS headers back. Therefore, the browser doesn't attempt the cross-origin request.

Set the exposed response headers

By default, the browser doesn't expose all of the response headers to the app. For more information, see W3C Cross-Origin Resource Sharing (Terminology): Simple Response Header.

The response headers that are available by default are:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

The CORS specification calls these headers *simple response headers*. To make other headers available to the app, call WithExposedHeaders:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyExposeResponseHeadersPolicy",
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                  .WithExposedHeaders("x-custom-header");
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

Credentials in cross-origin requests

Credentials require special handling in a CORS request. By default, the browser doesn't send credentials with a cross-origin request. Credentials include cookies and HTTP authentication schemes. To send credentials with a cross-origin request, the client must set `XMLHttpRequest.withCredentials` to `true`.

Using `XMLHttpRequest` directly:

JavaScript

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'https://www.example.com/api/test');
xhr.withCredentials = true;
```

Using jQuery:

JavaScript

```
$.ajax({
  type: 'get',
  url: 'https://www.example.com/api/test',
  xhrFields: {
    withCredentials: true
  }
});
```

Using the Fetch API:

JavaScript

```
fetch('https://www.example.com/api/test', {
    credentials: 'include'
});
```

The server must allow the credentials. To allow cross-origin credentials, call AllowCredentials:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyMyAllowCredentialsPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com")
                  .AllowCredentials();
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

The HTTP response includes an `Access-Control-Allow-Credentials` header, which tells the browser that the server allows credentials for a cross-origin request.

If the browser sends credentials but the response doesn't include a valid `Access-Control-Allow-Credentials` header, the browser doesn't expose the response to the app, and the cross-origin request fails.

Allowing cross-origin credentials is a security risk. A website at another domain can send a signed-in user's credentials to the app on the user's behalf without the user's knowledge.

The CORS specification also states that setting origins to `"*"` (all origins) is invalid if the `Access-Control-Allow-Credentials` header is present.

Preflight requests

For some CORS requests, the browser sends an additional OPTIONS request before making the actual request. This request is called a preflight request. The browser can skip the preflight request if **all** the following conditions are true:

- The request method is GET, HEAD, or POST.
- The app doesn't set request headers other than `Accept`, `Accept-Language`, `Content-Language`, `Content-Type`, or `Last-Event-ID`.
- The `Content-Type` header, if set, has one of the following values:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`

The rule on request headers set for the client request applies to headers that the app sets by calling `setRequestHeader` on the `XMLHttpRequest` object. The CORS specification calls these headers *author request headers*. The rule doesn't apply to headers the browser can set, such as `User-Agent`, `Host`, or `Content-Length`.

> **Note**
>
> This article contains URLs created by deploying the sample code to two Azure web sites, `https://cors3.azurewebsites.net` and `https://cors.azurewebsites.net`.

The following is an example response similar to the preflight request made from the **[Put test]** button in the Test CORS section of this document.

```
General:
Request URL: https://cors3.azurewebsites.net/api/values/5
Request Method: OPTIONS
Status Code: 204 No Content

Response Headers:
Access-Control-Allow-Methods: PUT,DELETE,GET
Access-Control-Allow-Origin: https://cors1.azurewebsites.net
Server: Microsoft-IIS/10.0
Set-Cookie:
ARRAffinity=8f8...8;Path=/;HttpOnly;Domain=cors1.azurewebsites.net
Vary: Origin

Request Headers:
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Access-Control-Request-Method: PUT
Connection: keep-alive
Host: cors3.azurewebsites.net
Origin: https://cors1.azurewebsites.net
Referer: https://cors1.azurewebsites.net/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0
```

The preflight request uses the HTTP OPTIONS method. It may include the following headers:

- Access-Control-Request-Method: The HTTP method that will be used for the actual request.
- Access-Control-Request-Headers: A list of request headers that the app sets on the actual request. As stated earlier, this doesn't include headers that the browser sets, such as `User-Agent`.

If the preflight request is denied, the app returns a `200 OK` response but doesn't set the CORS headers. Therefore, the browser doesn't attempt the cross-origin request. For an example of a denied preflight request, see the Test CORS section of this document.

Using the F12 tools, the console app shows an error similar to one of the following, depending on the browser:

- Firefox: Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at `https://cors1.azurewebsites.net/api/TodoItems1/MyDelete2/5`. (Reason: CORS request did not succeed). Learn More

- Chromium based: Access to fetch at 'https://cors1.azurewebsites.net/api/TodoItems1/MyDelete2/5' from origin 'https://cors3.azurewebsites.net' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

To allow specific headers, call WithHeaders:

C#

```
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyAllowHeadersPolicy",
        policy =>
        {
        policy.WithOrigins("http://example.com")
                    .WithHeaders(HeaderNames.ContentType, "x-custom-header");
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

To allow all author request headers, call AllowAnyHeader:

C#

```
using Microsoft.Net.Http.Headers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MyAllowAllHeadersPolicy",
        policy =>
        {
            policy.WithOrigins("https://*.example.com")
                    .AllowAnyHeader();
        });
});

builder.Services.AddControllers();
```

```
var app = builder.Build();
```

Browsers aren't consistent in how they set `Access-Control-Request-Headers`. If either:

- Headers are set to anything other than `"*"`
- AllowAnyHeader is called: Include at least `Accept`, `Content-Type`, and `Origin`, plus any custom headers that you want to support.

Automatic preflight request code

When the CORS policy is applied either:

- Globally by calling `app.UseCors` in `Program.cs`.
- Using the `[EnableCors]` attribute.

ASP.NET Core responds to the preflight OPTIONS request.

The Test CORS section of this document demonstrates this behavior.

[HttpOptions] attribute for preflight requests

When CORS is enabled with the appropriate policy, ASP.NET Core generally responds to CORS preflight requests automatically.

The following code uses the [HttpOptions] attribute to create endpoints for OPTIONS requests:

C#

```
[Route("api/[controller]")]
[ApiController]
public class TodoItems2Controller : ControllerBase
{
    // OPTIONS: api/TodoItems2/5
    [HttpOptions("{id}")]
    public IActionResult PreflightRoute(int id)
    {
        return NoContent();
    }

    // OPTIONS: api/TodoItems2
    [HttpOptions]
    public IActionResult PreflightRoute()
    {
        return NoContent();
    }
```

```
    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id)
    {
        if (id < 1)
        {
            return BadRequest();
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }
```

See Test CORS with [EnableCors] attribute and RequireCors method for instructions on testing the preceding code.

Set the preflight expiration time

The `Access-Control-Max-Age` header specifies how long the response to the preflight request can be cached. To set this header, call SetPreflightMaxAge:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy("MySetPreflightExpirationPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com")
                  .SetPreflightMaxAge(TimeSpan.FromSeconds(2520));
        });
});

builder.Services.AddControllers();

var app = builder.Build();
```

Enable CORS on an endpoint

How CORS works

This section describes what happens in a CORS request at the level of the HTTP messages.

- CORS is **not** a security feature. CORS is a W3C standard that allows a server to relax the same-origin policy.
  - For example, a malicious actor could use Cross-Site Scripting (XSS) against your site and execute a cross-site request to their CORS enabled site to steal information.
- An API isn't safer by allowing CORS.
  - It's up to the client (browser) to enforce CORS. The server executes the request and returns the response, it's the client that returns an error and blocks the response. For example, any of the following tools will display the server response:
    - Fiddler
    - .NET HttpClient
    - A web browser by entering the URL in the address bar.
- It's a way for a server to allow browsers to execute a cross-origin XHR or Fetch API request that otherwise would be forbidden.
  - Browsers without CORS can't do cross-origin requests. Before CORS, JSONP was used to circumvent this restriction. JSONP doesn't use XHR, it uses the `<script>` tag to receive the response. Scripts are allowed to be loaded cross-origin.

The CORS specification introduced several new HTTP headers that enable cross-origin requests. If a browser supports CORS, it sets these headers automatically for cross-origin requests. Custom JavaScript code isn't required to enable CORS.

The following is an example of a cross-origin request from the **Values** test button to `https://cors1.azurewebsites.net/api/values`. The `Origin` header:

- Provides the domain of the site that's making the request.
- Is required and must be different from the host.

General headers

```
Request URL: https://cors1.azurewebsites.net/api/values
Request Method: GET
Status Code: 200 OK
```

Response headers

```
Content-Encoding: gzip
Content-Type: text/plain; charset=utf-8
```

```
Server: Microsoft-IIS/10.0
Set-Cookie: ARRAffinity=8f...;Path=/;HttpOnly;Domain=cors1.azurewebsites.net
Transfer-Encoding: chunked
Vary: Accept-Encoding
X-Powered-By: ASP.NET
```

Request headers

```

Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
Host: cors1.azurewebsites.net
Origin: https://cors3.azurewebsites.net
Referer: https://cors3.azurewebsites.net/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0 ...
```

In `OPTIONS` requests, the server sets the **Response headers** `Access-Control-Allow-Origin: {allowed origin}` header in the response. For example, in the sample code, the `Delete [EnableCors]` button `OPTIONS` request contains the following headers:

General headers

```

Request URL: https://cors3.azurewebsites.net/api/TodoItems2/MyDelete2/5
Request Method: OPTIONS
Status Code: 204 No Content
```

Response headers

```

Access-Control-Allow-Headers: Content-Type,x-custom-header
Access-Control-Allow-Methods: PUT,DELETE,GET,OPTIONS
Access-Control-Allow-Origin: https://cors1.azurewebsites.net
Server: Microsoft-IIS/10.0
Set-Cookie: ARRAffinity=8f...;Path=/;HttpOnly;Domain=cors3.azurewebsites.net
Vary: Origin
X-Powered-By: ASP.NET
```

Request headers

```
```

```
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Access-Control-Request-Headers: content-type
Access-Control-Request-Method: DELETE
Connection: keep-alive
Host: cors3.azurewebsites.net
Origin: https://cors1.azurewebsites.net
Referer: https://cors1.azurewebsites.net/test?number=2
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
User-Agent: Mozilla/5.0
```

In the preceding **Response headers**, the server sets the Access-Control-Allow-Origin header in the response. The `https://cors1.azurewebsites.net` value of this header matches the `Origin` header from the request.

If AllowAnyOrigin is called, the `Access-Control-Allow-Origin: *`, the wildcard value, is returned. `AllowAnyOrigin` allows any origin.

If the response doesn't include the `Access-Control-Allow-Origin` header, the cross-origin request fails. Specifically, the browser disallows the request. Even if the server returns a successful response, the browser doesn't make the response available to the client app.

HTTP redirection to HTTPS causes ERR_INVALID_REDIRECT on the CORS preflight request

Requests to an endpoint using HTTP that are redirected to HTTPS by UseHttpsRedirection fail with `ERR_INVALID_REDIRECT on the CORS preflight request`.

API projects can reject HTTP requests rather than use `UseHttpsRedirection` to redirect requests to HTTPS.

CORS in IIS

When deploying to IIS, CORS has to run before Windows Authentication if the server isn't configured to allow anonymous access. To support this scenario, the IIS CORS module needs to be installed and configured for the app.

Test CORS

The sample download has code to test CORS. See how to download. The sample is an API project with Razor Pages added:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "MyPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com",
                    "http://www.contoso.com",
                    "https://cors1.azurewebsites.net",
                    "https://cors3.azurewebsites.net",
                    "https://localhost:44398",
                    "https://localhost:5001")
                .WithMethods("PUT", "DELETE", "GET");
        });
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.MapControllers();
app.MapRazorPages();

app.Run();
```

> ⚠ **Warning**
>
> `WithOrigins("https://localhost:<port>");` should only be used for testing a sample app similar to the download sample code.

The following `ValuesController` provides the endpoints for testing:

C#

```
[EnableCors("MyPolicy")]
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public IActionResult Get() =>
        ControllerContext.MyDisplayRouteInfo();

    // GET api/values/5
    [HttpGet("{id}")]
    public IActionResult Get(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // PUT api/values/5
    [HttpPut("{id}")]
    public IActionResult Put(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);


    // GET: api/values/GetValues2
    [DisableCors]
    [HttpGet("{action}")]
    public IActionResult GetValues2() =>
        ControllerContext.MyDisplayRouteInfo();

}
```

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

Test the preceding sample code by using one of the following approaches:

- Run the sample with `dotnet run` using the default URL of `https://localhost:5001`.
- Run the sample from Visual Studio with the port set to 44398 for a URL of `https://localhost:44398`.

Using a browser with the F12 tools:

- Select the **Values** button and review the headers in the **Network** tab.
- Select the **PUT test** button. See Display OPTIONS requests for instructions on displaying the OPTIONS request. The **PUT test** creates two requests, an OPTIONS preflight request and the PUT request.
- Select the **`GetValues2 [DisableCors]`** button to trigger a failed CORS request. As mentioned in the document, the response returns 200 success, but the CORS

request is not made. Select the **Console** tab to see the CORS error. Depending on the browser, an error similar to the following is displayed:

Access to fetch at `'https://cors1.azurewebsites.net/api/values/GetValues2'` from origin `'https://cors3.azurewebsites.net'` has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

CORS-enabled endpoints can be tested with a tool, such as curl or Fiddler. When using a tool, the origin of the request specified by the `Origin` header must differ from the host receiving the request. If the request isn't *cross-origin* based on the value of the `Origin` header:

- There's no need for CORS Middleware to process the request.
- CORS headers aren't returned in the response.

The following command uses `curl` to issue an OPTIONS request with information:

Bash

```
curl -X OPTIONS https://cors3.azurewebsites.net/api/TodoItems2/5 -i
```

Test CORS with [EnableCors] attribute and RequireCors method

Consider the following code which uses endpoint routing to enable CORS on a per-endpoint basis using `RequireCors`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCors(options =>
{
    options.AddPolicy(name: "MyPolicy",
        policy =>
        {
            policy.WithOrigins("http://example.com",
                    "http://www.contoso.com",
                    "https://cors1.azurewebsites.net",
                    "https://cors3.azurewebsites.net",
                    "https://localhost:44398",
                    "https://localhost:5001")
                .WithMethods("PUT", "DELETE", "GET");
        });
```

```
});

builder.Services.AddControllers();
builder.Services.AddRazorPages();

var app = builder.Build();

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseCors();

app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/echo",
        context => context.Response.WriteAsync("echo"))
        .RequireCors("MyPolicy");

    endpoints.MapControllers();
    endpoints.MapRazorPages();
});

app.Run();
```

Notice that only the `/echo` endpoint is using the `RequireCors` to allow cross-origin requests using the specified policy. The controllers below enable CORS using [EnableCors] attribute.

The following `TodoItems1Controller` provides endpoints for testing:

C#

```
[Route("api/[controller]")]
[ApiController]
public class TodoItems1Controller : ControllerBase
{
    // PUT: api/TodoItems1/5
    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id) {
        if (id < 1) {
            return Content($"ID = {id}");
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }

    // Delete: api/TodoItems1/5
    [HttpDelete("{id}")]
    public IActionResult MyDelete(int id) =>
```

```
        ControllerContext.MyDisplayRouteInfo(id);

    // GET: api/TodoItems1
    [HttpGet]
    public IActionResult GetTodoItems() =>
        ControllerContext.MyDisplayRouteInfo();

    [EnableCors("MyPolicy")]
    [HttpGet("{action}")]
    public IActionResult GetTodoItems2() =>
        ControllerContext.MyDisplayRouteInfo();

    // Delete: api/TodoItems1/MyDelete2/5
    [EnableCors("MyPolicy")]
    [HttpDelete("{action}/{id}")]
    public IActionResult MyDelete2(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);
}
```

The **Delete [EnableCors]** and **GET [EnableCors]** buttons succeed, because the endpoints have `[EnableCors]` and respond to preflight requests. The other endpoints fails. The **GET** button fails, because the JavaScript sends:

JavaScript

```
 headers: {
      "Content-Type": "x-custom-header"
 },
```

The following `TodoItems2Controller` provides similar endpoints, but includes explicit code to respond to OPTIONS requests:

C#

```
[Route("api/[controller]")]
[ApiController]
public class TodoItems2Controller : ControllerBase
{
    // OPTIONS: api/TodoItems2/5
    [HttpOptions("{id}")]
    public IActionResult PreflightRoute(int id)
    {
        return NoContent();
    }

    // OPTIONS: api/TodoItems2
    [HttpOptions]
    public IActionResult PreflightRoute()
    {
        return NoContent();
```

```
    }

    [HttpPut("{id}")]
    public IActionResult PutTodoItem(int id)
    {
        if (id < 1)
        {
            return BadRequest();
        }

        return ControllerContext.MyDisplayRouteInfo(id);
    }

    // [EnableCors] // Not needed as OPTIONS path provided.
    [HttpDelete("{id}")]
    public IActionResult MyDelete(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);

    // [EnableCors] //  Warning ASP0023 Route '{id}' conflicts with another
action route.
    //                  An HTTP request that matches multiple routes results
in an ambiguous
    //                  match error.
    [EnableCors("MyPolicy")] // Required for this path.
    [HttpGet]
    public IActionResult GetTodoItems() =>
        ControllerContext.MyDisplayRouteInfo();

    [HttpGet("{action}")]
    public IActionResult GetTodoItems2() =>
        ControllerContext.MyDisplayRouteInfo();

    [EnableCors("MyPolicy")]  // Required for this path.
    [HttpDelete("{action}/{id}")]
    public IActionResult MyDelete2(int id) =>
        ControllerContext.MyDisplayRouteInfo(id);
}
```

The preceding code can be tested by deploying the sample to Azure. In the **Controller** drop down list, select **Preflight** and then **Set Controller**. All the CORS calls to the `TodoItems2Controller` endpoints succeed.

Additional resources

- Cross-Origin Resource Sharing (CORS)
- IIS CORS module Configuration Reference

Share authentication cookies among ASP.NET apps

Article • 01/12/2023

By Rick Anderson

Websites often consist of individual web apps working together. To provide a single sign-on (SSO) experience, web apps within a site must share authentication cookies. To support this scenario, the data protection stack allows sharing Katana cookie authentication and ASP.NET Core cookie authentication tickets.

In the examples that follow:

- The authentication cookie name is set to a common value of `.AspNet.SharedCookie`.
- The `AuthenticationType` is set to `Identity.Application` either explicitly or by default.
- A common app name, `SharedCookieApp`, is used to enable the data protection system to share data protection keys.
- `Identity.Application` is used as the authentication scheme. Whatever scheme is used, it must be used consistently *within and across* the shared cookie apps either as the default scheme or by explicitly setting it. The scheme is used when encrypting and decrypting cookies, so a consistent scheme must be used across apps.
- A common data protection key storage location is used.
  - In ASP.NET Core apps, PersistKeysToFileSystem is used to set the key storage location.
  - In .NET Framework apps, Cookie Authentication Middleware uses an implementation of DataProtectionProvider. `DataProtectionProvider` provides data protection services for the encryption and decryption of authentication cookie payload data. The `DataProtectionProvider` instance is isolated from the data protection system used by other parts of the app. DataProtectionProvider.Create(System.IO.DirectoryInfo, Action<IDataProtectionBuilder>) accepts a DirectoryInfo to specify the location for data protection key storage.
- `DataProtectionProvider` requires the Microsoft.AspNetCore.DataProtection.Extensions NuGet package:
  - In .NET Framework apps, add a package reference to Microsoft.AspNetCore.DataProtection.Extensions.

- SetApplicationName sets the common app name.

Share authentication cookies with ASP.NET Core Identity

When using ASP.NET Core Identity:

- Data protection keys and the app name must be shared among apps. A common key storage location is provided to the PersistKeysToFileSystem method in the following examples. Use SetApplicationName to configure a common shared app name (`SharedCookieApp` in the following examples). For more information, see Configure ASP.NET Core Data Protection.
- Use the ConfigureApplicationCookie extension method to set up the data protection service for cookies.
- The default authentication type is `Identity.Application`.

In `Program.cs`:

C#

```
using Microsoft.AspNetCore.DataProtection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\PATH TO COMMON KEY RING
FOLDER"))
    .SetApplicationName("SharedCookieApp");

builder.Services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = ".AspNet.SharedCookie";
});

var app = builder.Build();
```

Note: The preceding instructions don't work with `ITicketStore` (`CookieAuthenticationOptions.SessionStore`). For more information, see this GitHub issue.

For security reasons, authentication cookies are not compressed in ASP.NET Core. When using authentication cookies, developers should minimize the number of claim information included to just that necessary for their needs.

Share authentication cookies without ASP.NET Core Identity

When using cookies directly without ASP.NET Core Identity, configure data protection and authentication. In the following example, the authentication type is set to `Identity.Application`:

C#

```
using Microsoft.AspNetCore.DataProtection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\PATH TO COMMON KEY RING
FOLDER"))
    .SetApplicationName("SharedCookieApp");

builder.Services.AddAuthentication("Identity.Application")
    .AddCookie("Identity.Application", options =>
    {
        options.Cookie.Name = ".AspNet.SharedCookie";
    });

var app = builder.Build();
```

For security reasons, authentication cookies are not compressed in ASP.NET Core. When using authentication cookies, developers should minimize the number of claim information included to just that necessary for their needs.

Share cookies across different base paths

An authentication cookie uses the HttpRequest.PathBase as its default Cookie.Path. If the app's cookie must be shared across different base paths, `Path` must be overridden:

C#

```
using Microsoft.AspNetCore.DataProtection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
```

```
builder.Services.AddDataProtection()
    .PersistKeysToFileSystem(new DirectoryInfo(@"c:\PATH TO COMMON KEY RING
FOLDER"))
    .SetApplicationName("SharedCookieApp");

builder.Services.ConfigureApplicationCookie(options => {
    options.Cookie.Name = ".AspNet.SharedCookie";
    options.Cookie.Path = "/";
});
var app = builder.Build();
```

Share cookies across subdomains

When hosting apps that share cookies across subdomains, specify a common domain in the Cookie.Domain property. To share cookies across apps at `contoso.com`, such as `first_subdomain.contoso.com` and `second_subdomain.contoso.com`, specify the `Cookie.Domain` as `.contoso.com`:

C#

```
options.Cookie.Domain = ".contoso.com";
```

Encrypt data protection keys at rest

For production deployments, configure the `DataProtectionProvider` to encrypt keys at rest with DPAPI or an X509Certificate. For more information, see Key encryption at rest in Windows and Azure using ASP.NET Core. In the following example, a certificate thumbprint is provided to ProtectKeysWithCertificate:

C#

```
using Microsoft.AspNetCore.DataProtection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

builder.Services.AddDataProtection()
    .ProtectKeysWithCertificate("{CERTIFICATE THUMBPRINT}");
```

Use a common user database

When apps use the same Identity schema (same version of Identity), confirm that the Identity system for each app is pointed at the same user database. Otherwise, the identity system produces failures at runtime when it attempts to match the information in the authentication cookie against the information in its database.

When the Identity schema is different among apps, usually because apps are using different Identity versions, sharing a common database based on the latest version of Identity isn't possible without remapping and adding columns in other app's Identity schemas. It's often more efficient to upgrade the other apps to use the latest Identity version so that a common database can be shared by the apps.

Application name change

In .NET 6, WebApplicationBuilder normalizes the content root path to end with a DirectorySeparatorChar. Most apps migrating from HostBuilder or WebHostBuilder won't have the same app name because they aren't normalized. For more information, see SetApplicationName

Share authentication cookies between ASP.NET 4.x and ASP.NET Core apps

ASP.NET 4.x apps that use Microsoft.Owin Cookie Authentication Middleware can be configured to generate authentication cookies that are compatible with the ASP.NET Core Cookie Authentication Middleware. This can be useful if a web application consists of both ASP.NET 4.x apps and ASP.NET Core apps that must share a single sign-on experience. A specific example of such a scenario is incrementally migrating a web app from ASP.NET to ASP.NET Core. In such scenarios, it's common for some parts of an app to be served by the original ASP.NET app while others are served by the new ASP.NET Core app. Users should only have to sign in once, though. This can be accomplished by either of the following approaches:

- Using the System.Web adapters' remote authentication feature, which uses the ASP.NET app to sign users in.
- Configuring the ASP.NET app to use Microsoft.Owin Cookie Authentication Middleware so that authentication cookies are shared with the ASP.NET Core app.

To configure ASP.NET Microsoft.Owin Cookie Authentication Middleware to share cookies with an ASP.NET Core app, follow the preceding instructions to configure the ASP.NET Core app to use a specific cookie name, app name, and to persist data protection keys to a well-known location. See Configure ASP.NET Core Data Protection for more information on persisting data protection keys.

In the ASP.NET app, install the Microsoft.Owin.Security.Interop package.

Update the `UseCookieAuthentication` call in Startup.Auth.cs to configure an AspNetTicketDataFormat to match the ASP.NET Core app's settings:

C#

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
{
    LoginPath = new PathString("/Account/Login"),
    Provider = new CookieAuthenticationProvider
    {
        OnValidateIdentity =
SecurityStampValidator.OnValidateIdentity<ApplicationUserManager,
ApplicationUser>(
            validateInterval: TimeSpan.FromMinutes(30),
            regenerateIdentity: (manager, user) =>
user.GenerateUserIdentityAsync(manager))
    },

    // Settings to configure shared cookie with ASP.NET Core app
    CookieName = ".AspNet.ApplicationCookie",
    AuthenticationType = "Identity.Application",
    TicketDataFormat = new AspNetTicketDataFormat(
        new DataProtectorShim(
            DataProtectionProvider.Create(new DirectoryInfo(@"c:\PATH TO
COMMON KEY RING FOLDER"),
            builder => builder.SetApplicationName("SharedCookieApp"))
            .CreateProtector(

"Microsoft.AspNetCore.Authentication.Cookies.CookieAuthenticationMiddleware"
,
                // Must match the Scheme name used in the ASP.NET Core app,
i.e. IdentityConstants.ApplicationScheme
                "Identity.Application",
                "v2"))),
    CookieManager = new ChunkingCookieManager()
});
```

Important items configured here include:

- The cookie name is set to the same name as in the ASP.NET Core app.
- A data protection provider is created using the same key ring path. Note that in these examples, data protection keys are stored on disk but other data protection providers can be used. For example, Redis or Azure Blob Storage can be used for data protection providers as long as the configuration matches between the apps. See Configure ASP.NET Core Data Protection for more information on persisting data protection keys.

- The app name is set to be the same as the app name used in the ASP.NET Core app.
- The authentication type is set to the name of the authentication scheme in the ASP.NET Core app.
- `System.Web.Helpers.AntiForgeryConfig.UniqueClaimTypeIdentifier` is set to a claim from the ASP.NET Core identity that will be unique to a user.

Because the authentication type was changed to match the authentication scheme of the ASP.NET Core app, it's also necessary to update how the ASP.NET app generates new identities to use that same name. This is typically done in `Models/IdentityModels.cs`:

C#

```
public class ApplicationUser : IdentityUser
{
    public async Task<ClaimsIdentity>
GenerateUserIdentityAsync(UserManager<ApplicationUser> manager)
    {
        // Note the authenticationType must match the one defined in
CookieAuthenticationOptions.AuthenticationType
        var userIdentity = await manager.CreateIdentityAsync(this,
"Identity.Application");

        // Add custom user claims here
        return userIdentity;
    }
}
```

With these changes, the ASP.NET and ASP.NET Core apps are able to use the same authentication cookies so that users signing in or out of one app are reflected in the other app.

Note that because there are differences between ASP.NET Identity and ASP.NET Core Identity's database schemas, it is recommended that users only *sign-in* using one of the apps - either the ASP.NET or ASP.NET Core app. Once users are signed in, the steps documented in this section will allow the authentication cookie to be *used* by either app and both apps should be able to log users out.

Additional resources

- Host ASP.NET Core in a web farm
- A primer on OWIN cookie authentication middleware for the ASP.NET developer by Brock Allen
- OWIN Authentication Middleware Architecture by Brock Allen

- Posts from the 'OWIN / Katana' Category by Brock Allen

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Work with SameSite cookies in ASP.NET Core

Article • 06/21/2023

By Rick Anderson

SameSite is an IETF draft standard designed to provide some protection against cross-site request forgery (CSRF) attacks. Originally drafted in 2016, the draft standard was updated in 2019. The updated standard is not backward compatible with the previous standard, with the following being the most noticeable differences:

- Cookies without SameSite header are treated as `SameSite=Lax` by default.
- `SameSite=None` must be used to allow cross-site cookie use.
- Cookies that assert `SameSite=None` must also be marked as `Secure`.
- Applications that use <iframe> may experience issues with `sameSite=Lax` or `sameSite=Strict` cookies because `<iframe>` is treated as cross-site scenarios.
- The value `SameSite=None` is not allowed by the 2016 standard and causes some implementations to treat such cookies as `SameSite=Strict`. See Supporting older browsers in this document.

The `SameSite=Lax` setting works for most application cookies. Some forms of authentication like OpenID Connect (OIDC) and WS-Federation default to POST based redirects. The POST based redirects trigger the SameSite browser protections, so SameSite is disabled for these components. Most OAuth logins are not affected due to differences in how the request flows.

Each ASP.NET Core component that emits cookies needs to decide if SameSite is appropriate.

SameSite and Identity

ASP.NET Core Identity is largely unaffected by SameSite cookies except for advanced scenarios like `IFrames` or `OpenIdConnect` integration.

When using `Identity`, do ***not*** add any cookie providers or call `services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)`, `Identity` takes care of that.

SameSite test sample code

The following sample can be downloaded and tested:

Expand table

Sample	Document
.NET Core Razor Pages	ASP.NET Core 3.1 Razor Pages SameSite cookie sample

.NET Core support for the sameSite attribute

.NET Core supports the 2019 draft standard for SameSite. Developers are able to programmatically control the value of the sameSite attribute using the `HttpCookie.SameSite` property. Setting the `SameSite` property to `Strict`, `Lax`, or `None` results in those values being written on the network with the cookie. Setting to `SameSiteMode.Unspecified` indicates no sameSite should be sent with the cookie.

C#

```
    var cookieOptions = new CookieOptions
    {
        // Set the secure flag, which Chrome's changes will require for
SameSite none.
        // Note this will also require you to be running on HTTPS.
        Secure = true,

        // Set the cookie to HTTP only which is good practice unless you
really do need
        // to access it client side in scripts.
        HttpOnly = true,

        // Add the SameSite attribute, this will emit the attribute with a
value of none.
        SameSite = SameSiteMode.None

        // The client should follow its default cookie policy.
        // SameSite = SameSiteMode.Unspecified
    };

    // Add the cookie to the response cookie collection
    Response.Cookies.Append("MyCookie", "cookieValue", cookieOptions);
}
```

API usage with SameSite

HttpContext.Response.Cookies.Append defaults to `Unspecified`, meaning no SameSite attribute added to the cookie and the client will use its default behavior (Lax for new

browsers, None for old ones). The following code shows how to change the cookie SameSite value to `SameSiteMode.Lax`:

```
C#

HttpContext.Response.Cookies.Append(
                    "name", "value",
                    new CookieOptions() { SameSite = SameSiteMode.Lax });
```

All ASP.NET Core components that emit cookies override the preceding defaults with settings appropriate for their scenarios. The overridden preceding default values haven't changed.

Expand table

Component	cookie	Default
CookieBuilder	SameSite	`Unspecified`
Session	SessionOptions.Cookie	`Lax`
CookieTempDataProvider	CookieTempDataProviderOptions.Cookie	`Lax`
IAntiforgery	AntiforgeryOptions.Cookie	`Strict`
Cookie Authentication	CookieAuthenticationOptions.Cookie	`Lax`
AddTwitter	TwitterOptions.StateCookie	`Lax`
RemoteAuthenticationHandler<TOptions>	RemoteAuthenticationOptions.CorrelationCookie	`None`
AddOpenIdConnect	OpenIdConnectOptions.NonceCookie	`None`
HttpContext.Response.Cookies.Append	CookieOptions	`Unspecified`

ASP.NET Core 3.1 and later provides the following SameSite support:

- Redefines the behavior of `SameSiteMode.None` to emit `SameSite=None`
- Adds a new value `SameSiteMode.Unspecified` to omit the SameSite attribute.
- All cookies APIs default to `Unspecified`. Some components that use cookies set values more specific to their scenarios. See the table above for examples.

In ASP.NET Core 3.0 and later the SameSite defaults were changed to avoid conflicting with inconsistent client defaults. The following APIs have changed the default from `SameSiteMode.Lax` to `-1` to avoid emitting a SameSite attribute for these cookies:

- CookieOptions used with HttpContext.Response.Cookies.Append
- CookieBuilder used as a factory for `CookieOptions`

- CookiePolicyOptions.MinimumSameSitePolicy

History and changes

SameSite support was first implemented in ASP.NET Core in 2.0 using the 2016 draft standard. The 2016 standard was opt-in. ASP.NET Core opted-in by setting several cookies to `Lax` by default. After encountering several issues with authentication, most SameSite usage was disabled.

Patches were issued in November 2019 to update from the 2016 standard to the 2019 standard. The 2019 draft of the SameSite specification:

- Is **not** backwards compatible with the 2016 draft. For more information, see Supporting older browsers in this document.
- Specifies cookies are treated as `SameSite=Lax` by default.
- Specifies cookies that explicitly assert `SameSite=None` in order to enable cross-site delivery should be marked as `Secure`. `None` is a new entry to opt out.
- Is supported by patches issued for ASP.NET Core 2.1, 2.2, and 3.0. ASP.NET Core 3.1 and later has additional SameSite support.
- Is scheduled to be enabled by Chrome by default in Feb 2020. Browsers started moving to this standard in 2019.

APIs impacted by the change from the 2016 SameSite draft standard to the 2019 draft standard

- Http.SameSiteMode
- CookieOptions.SameSite
- CookieBuilder.SameSite
- CookiePolicyOptions.MinimumSameSitePolicy
- Microsoft.Net.Http.Headers.SameSiteMode
- Microsoft.Net.Http.Headers.SetCookieHeaderValue.SameSite

Supporting older browsers

The 2016 SameSite standard mandated that unknown values must be treated as `SameSite=Strict` values. Apps accessed from older browsers which support the 2016 SameSite standard may break when they get a SameSite property with a value of `None`. Web apps must implement browser detection if they intend to support older browsers. ASP.NET Core doesn't implement browser detection because User-Agents values are highly

volatile and change frequently. An extension point in Microsoft.AspNetCore.CookiePolicy allows plugging in User-Agent specific logic.

In `Program.cs`, add code that calls UseCookiePolicy before calling UseAuthentication or *any* method that writes cookies:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
    options.OnAppendCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    options.OnDeleteCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
});

void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}

    builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseCookiePolicy();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In `Program.cs`, add code similar to the following highlighted code:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<CookiePolicyOptions>(options =>
{
    options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
    options.OnAppendCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    options.OnDeleteCookie = cookieContext =>
        CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
});

void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}

    builder.Services.AddRazorPages();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseCookiePolicy();
app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

In the preceding sample, `MyUserAgentDetectionLib.DisallowsSameSiteNone` is a user supplied library that detects if the user agent doesn't support SameSite `None`:

C#

```
if (MyUserAgentDetectionLib.DisallowsSameSiteNone(userAgent))
{
    options.SameSite = SameSiteMode.Unspecified;
}
```

The following code shows a sample `DisallowsSameSiteNone` method:

> ⚠ **Warning**
>
> The following code is for demonstration only:
>
> - It should not be considered complete.
> - It is not maintained or supported.

C#

```
public static bool DisallowsSameSiteNone(string userAgent)
{
    // Check if a null or empty string has been passed in, since this
    // will cause further interrogation of the useragent to fail.
     if (String.IsNullOrWhiteSpace(userAgent))
        return false;

    // Cover all iOS based browsers here. This includes:
    // - Safari on iOS 12 for iPhone, iPod Touch, iPad
    // - WkWebview on iOS 12 for iPhone, iPod Touch, iPad
    // - Chrome on iOS 12 for iPhone, iPod Touch, iPad
    // All of which are broken by SameSite=None, because they use the iOS
networking
    // stack.
    if (userAgent.Contains("CPU iPhone OS 12") ||
        userAgent.Contains("iPad; CPU OS 12"))
    {
        return true;
    }

    // Cover Mac OS X based browsers that use the Mac OS networking stack.
    // This includes:
    // - Safari on Mac OS X.
    // This does not include:
    // - Chrome on Mac OS X
    // Because they do not use the Mac OS networking stack.
    if (userAgent.Contains("Macintosh; Intel Mac OS X 10_14") &&
        userAgent.Contains("Version/") && userAgent.Contains("Safari"))
    {
        return true;
    }

    // Cover Chrome 50-69, because some versions are broken by SameSite=None,
```

```
    // and none in this range require it.
    // Note: this covers some pre-Chromium Edge versions,
    // but pre-Chromium Edge does not require SameSite=None.
    if (userAgent.Contains("Chrome/5") || userAgent.Contains("Chrome/6"))
    {
        return true;
    }

    return false;
}
```

Test apps for SameSite problems

Apps that interact with remote sites such as through third-party login need to:

- Test the interaction on multiple browsers.
- Apply the CookiePolicy browser detection and mitigation discussed in this document.

Test web apps using a client version that can opt-in to the new SameSite behavior. Chrome, Firefox, and Chromium Edge all have new opt-in feature flags that can be used for testing. After your app applies the SameSite patches, test it with older client versions, especially Safari. For more information, see Supporting older browsers in this document.

Test with Chrome

Chrome 78+ gives misleading results because it has a temporary mitigation in place. The Chrome 78+ temporary mitigation allows cookies less than two minutes old. Chrome 76 or 77 with the appropriate test flags enabled provides more accurate results. To test the new SameSite behavior toggle `chrome://flags/#same-site-by-default-cookies` to **Enabled**. Older versions of Chrome (75 and below) are reported to fail with the new `None` setting. See Supporting older browsers in this document.

Google does not make older chrome versions available. Follow the instructions at Download Chromium to test older versions of Chrome. Do **not** download Chrome from links provided by searching for older versions of chrome.

- Chromium 76 Win64
- Chromium 74 Win64

Starting in Canary version `80.0.3975.0`, the Lax+POST temporary mitigation can be disabled for testing purposes using the new flag `--enable-features=SameSiteDefaultChecksMethodRigorously` to allow testing of sites and services in the eventual end state of the feature where the mitigation has been removed. For more information, see The Chromium Projects SameSite Updates

Test with Safari

Safari 12 strictly implemented the prior draft and fails when the new `None` value is in a cookie. `None` is avoided via the browser detection code Supporting older browsers in this document. Test Safari 12, Safari 13, and WebKit based OS style logins using MSAL, ADAL or whatever library you are using. The problem is dependent on the underlying OS version. OSX Mojave (10.14) and iOS 12 are known to have compatibility problems with the new SameSite behavior. Upgrading the OS to OSX Catalina (10.15) or iOS 13 fixes the problem. Safari does not currently have an opt-in flag for testing the new spec behavior.

Test with Firefox

Firefox support for the new standard can be tested on version 68+ by opting in on the `about:config` page with the feature flag `network.cookie.sameSite.laxByDefault`. There haven't been reports of compatibility issues with older versions of Firefox.

Test with Edge browser

Edge supports the old SameSite standard. Edge version 44 doesn't have any known compatibility problems with the new standard.

Test with Edge (Chromium)

SameSite flags are set on the `edge://flags/#same-site-by-default-cookies` page. No compatibility issues were discovered with Edge Chromium.

Test with Electron

Versions of Electron include older versions of Chromium. For example, the version of Electron used by Teams is Chromium 66, which exhibits the older behavior. You must perform your own compatibility testing with the version of Electron your product uses. See Supporting older browsers in the following section.

Additional resources

- Chromium Blog:Developers: Get Ready for New SameSite=None; Secure Cookie Settings
- SameSite cookies explained
- November 2019 Patches

Expand table

Sample	Document
.NET Core Razor Pages	ASP.NET Core 3.1 Razor Pages SameSite cookie sample

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Client IP safelist for ASP.NET Core

Article • 06/03/2022

By Damien Bowden and Tom Dykstra

This article shows three ways to implement an IP address safelist (also known as an allow list) in an ASP.NET Core app. An accompanying sample app demonstrates all three approaches. You can use:

- Middleware to check the remote IP address of every request.
- MVC action filters to check the remote IP address of requests for specific controllers or action methods.
- Razor Pages filters to check the remote IP address of requests for Razor pages.

In each case, a string containing approved client IP addresses is stored in an app setting. The middleware or filter:

- Parses the string into an array.
- Checks if the remote IP address exists in the array.

Access is allowed if the array contains the IP address. Otherwise, an HTTP 403 Forbidden status code is returned.

View or download sample code (how to download)

IP address safelist

In the sample app, the IP address safelist is:

- Defined by the `AdminSafeList` property in the `appsettings.json` file.
- A semicolon-delimited string that may contain both Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) addresses.

JSON

```
{
  "AdminSafeList": "127.0.0.1;192.168.1.5;::1",
  "Logging": {
```

In the preceding example, the IPv4 addresses of `127.0.0.1` and `192.168.1.5` and the IPv6 loopback address of `::1` (compressed format for `0:0:0:0:0:0:0:1`) are allowed.

Middleware

The `Startup.Configure` method adds the custom `AdminSafeListMiddleware` middleware type to the app's request pipeline. The safelist is retrieved with the .NET Core configuration provider and is passed as a constructor parameter.

C#

```
app.UseMiddleware<AdminSafeListMiddleware>(Configuration["AdminSafeList"]);
```

The middleware parses the string into an array and searches for the remote IP address in the array. If the remote IP address isn't found, the middleware returns HTTP 403 Forbidden. This validation process is bypassed for HTTP GET requests.

C#

```
public class AdminSafeListMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<AdminSafeListMiddleware> _logger;
    private readonly byte[][] _safelist;

    public AdminSafeListMiddleware(
        RequestDelegate next,
        ILogger<AdminSafeListMiddleware> logger,
        string safelist)
    {
        var ips = safelist.Split(';');
        _safelist = new byte[ips.Length][];
        for (var i = 0; i < ips.Length; i++)
        {
            _safelist[i] = IPAddress.Parse(ips[i]).GetAddressBytes();
        }

        _next = next;
        _logger = logger;
    }

    public async Task Invoke(HttpContext context)
    {
        if (context.Request.Method != HttpMethod.Get.Method)
        {
            var remoteIp = context.Connection.RemoteIpAddress;
            _logger.LogDebug("Request from Remote IP address: {RemoteIp}",
remoteIp);

            var bytes = remoteIp.GetAddressBytes();
            var badIp = true;
            foreach (var address in _safelist)
            {
```

```
                if (address.SequenceEqual(bytes))
                {
                    badIp = false;
                    break;
                }
            }

            if (badIp)
            {
                _logger.LogWarning(
                    "Forbidden Request from Remote IP address: {RemoteIp}",
remoteIp);
                context.Response.StatusCode = (int)
HttpStatusCode.Forbidden;
                return;
            }
        }

        await _next.Invoke(context);
    }
}
```

Action filter

If you want safelist-driven access control for specific MVC controllers or action methods, use an action filter. For example:

C#

```
public class ClientIpCheckActionFilter : ActionFilterAttribute
{
    private readonly ILogger _logger;
    private readonly string _safelist;

    public ClientIpCheckActionFilter(string safelist, ILogger logger)
    {
        _safelist = safelist;
        _logger = logger;
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
        _logger.LogDebug("Remote IpAddress: {RemoteIp}", remoteIp);
        var ip = _safelist.Split(';');
        var badIp = true;

        if (remoteIp.IsIPv4MappedToIPv6)
        {
            remoteIp = remoteIp.MapToIPv4();
        }
```

```
        foreach (var address in ip)
        {
            var testIp = IPAddress.Parse(address);

            if (testIp.Equals(remoteIp))
            {
                badIp = false;
                break;
            }
        }

        if (badIp)
        {
            _logger.LogWarning("Forbidden Request from IP: {RemoteIp}",
remoteIp);
            context.Result = new
StatusCodeResult(StatusCodes.Status403Forbidden);
            return;
        }

        base.OnActionExecuting(context);
    }
}
```

In `Startup.ConfigureServices`, add the action filter to the MVC filters collection. In the following example, a `ClientIpCheckActionFilter` action filter is added. A safelist and a console logger instance are passed as constructor parameters.

C#

```
services.AddScoped<ClientIpCheckActionFilter>(container =>
{
    var loggerFactory = container.GetRequiredService<ILoggerFactory>();
    var logger = loggerFactory.CreateLogger<ClientIpCheckActionFilter>();

    return new ClientIpCheckActionFilter(
        Configuration["AdminSafeList"], logger);
});
```

The action filter can then be applied to a controller or action method with the [ServiceFilter] attribute:

C#

```
[ServiceFilter(typeof(ClientIpCheckActionFilter))]
[HttpGet]
public IEnumerable<string> Get()
```

In the sample app, the action filter is applied to the controller's `Get` action method. When you test the app by sending:

- An HTTP GET request, the `[ServiceFilter]` attribute validates the client IP address. If access is allowed to the `Get` action method, a variation of the following console output is produced by the action filter and action method:

```
dbug: ClientIpSafelistComponents.Filters.ClientIpCheckActionFilter[0]
      Remote IpAddress: ::1
dbug: ClientIpAspNetCore.Controllers.ValuesController[0]
      successful HTTP GET
```

- An HTTP request verb other than GET, the `AdminSafeListMiddleware` middleware validates the client IP address.

Razor Pages filter

If you want safelist-driven access control for a Razor Pages app, use a Razor Pages filter. For example:

C#

```
public class ClientIpCheckPageFilter : IPageFilter
{
    private readonly ILogger _logger;
    private readonly IPAddress[] _safelist;

    public ClientIpCheckPageFilter(
        string safelist,
        ILogger logger)
    {
        var ips = safelist.Split(';');
        _safelist = new IPAddress[ips.Length];
        for (var i = 0; i < ips.Length; i++)
        {
            _safelist[i] = IPAddress.Parse(ips[i]);
        }

        _logger = logger;
    }

    public void OnPageHandlerExecuting(PageHandlerExecutingContext context)
    {
        var remoteIp = context.HttpContext.Connection.RemoteIpAddress;
        if (remoteIp.IsIPv4MappedToIPv6)
        {
```

```
            remoteIp = remoteIp.MapToIPv4();
        }
        _logger.LogDebug(
            "Remote IpAddress: {RemoteIp}", remoteIp);

        var badIp = true;
        foreach (var testIp in _safelist)
        {
            if (testIp.Equals(remoteIp))
            {
                badIp = false;
                break;
            }
        }

        if (badIp)
        {
            _logger.LogWarning(
                "Forbidden Request from Remote IP address: {RemoteIp}",
remoteIp);
            context.Result = new
StatusCodeResult(StatusCodes.Status403Forbidden);
            return;
        }
    }

    public void OnPageHandlerExecuted(PageHandlerExecutedContext context)
    {
    }

    public void OnPageHandlerSelected(PageHandlerSelectedContext context)
    {
    }
}
```

In `Startup.ConfigureServices`, enable the Razor Pages filter by adding it to the MVC filters collection. In the following example, a `ClientIpCheckPageFilter` Razor Pages filter is added. A safelist and a console logger instance are passed as constructor parameters.

C#

```
services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        var logger = LoggerFactory.Create(builder => builder.AddConsole())
                        .CreateLogger<ClientIpCheckPageFilter>();
        var filter = new ClientIpCheckPageFilter(
            Configuration["AdminSafeList"], logger);

        options.Filters.Add(filter);
    });
```

When the sample app's *Index* Razor page is requested, the Razor Pages filter validates the client IP address. The filter produces a variation of the following console output:

```
dbug: ClientIpSafelistComponents.Filters.ClientIpCheckPageFilter[0]
      Remote IpAddress: ::1
```

Additional resources

- ASP.NET Core Middleware
- Action filters
- Filter methods for Razor Pages in ASP.NET Core

ASP.NET Core performance

Article • 09/18/2024

The following articles provide information about how to optimize the performance of ASP.NET Core apps:

- ASP.NET Core Blazor performance best practices
- ASP.NET Core Best Practices
- Overview of caching in ASP.NET Core
- Rate limiting middleware in ASP.NET Core
- Memory management and patterns in ASP.NET Core
- Scaling ASP.NET Core Apps on Azure
- Object reuse with ObjectPool in ASP.NET Core
- Response compression in ASP.NET Core
- Performance Diagnostics Tools
- ASP.NET Core load/stress testing
- Request timeouts middleware in ASP.NET Core
- Short-circuit middleware after routing

Overview of caching in ASP.NET Core

Article • 05/21/2024

By Rick Anderson and Tom Dykstra

In-memory caching

In-memory caching uses server memory to store cached data. This type of caching is suitable for a single server or multiple servers using session affinity. Session affinity is also known as *sticky sessions*. Session affinity means that the requests from a client are always routed to the same server for processing.

For more information, see Cache in-memory in ASP.NET Core and Troubleshoot Azure Application Gateway session affinity issues.

Distributed Cache

Use a distributed cache to store data when the app is hosted in a cloud or server farm. The cache is shared across the servers that process requests. A client can submit a request that's handled by any server in the group if cached data for the client is available. ASP.NET Core works with SQL Server, Redis, and NCache distributed caches.

For more information, see Distributed caching in ASP.NET Core.

HybridCache

The HybridCache API bridges some gaps in the IDistributedCache and IMemoryCache APIs. `HybridCache` is an abstract class wth a default implementation that handles most aspects of saving to cache and retrieving from cache.

Features

`HybridCache` has the following features that the other APIs don't have:

- A unified API for both in-process and out-of-process caching.

  `HybridCache` is designed to be a drop-in replacement for existing `IDistributedCache` and `IMemoryCache` usage, and it provides a simple API for adding new caching code. If the app has an `IDistributedCache` implementation,

the `HybridCache` service uses it for secondary caching. This two-level caching strategy allows `HybridCache` to provide the speed of an in-memory cache and the durability of a distributed or persistent cache.

- Stampede protection.

  Cache stampede happens when a frequently used cache entry is revoked, and too many requests try to repopulate the same cache entry at the same time. `HybridCache` combines concurrent operations, ensuring that all requests for a given response wait for the first request to populate the cache.

- Configurable serialization.

  Serialization is configured as part of registering the service, with support for type-specific and generalized serializers via the `WithSerializer` and `WithSerializerFactory` methods, chained from the `AddHybridCache` call. By default, the service handles `string` and `byte[]` internally, and uses `System.Text.Json` for everything else. It can be configured for other types of serializers, such as protobuf or XML.

To see the relative simplicity of the `HybridCache` API, compare code that uses it to code that uses `IDistributedCache`. Here's an example of what using `IDistributedCache` looks like:

C#

```
public class SomeService(IDistributedCache cache)
{
    public async Task<SomeInformation> GetSomeInformationAsync
        (string name, int id, CancellationToken token = default)
    {
        var key = $"someinfo:{name}:{id}"; // Unique key for this
combination.
        var bytes = await cache.GetAsync(key, token); // Try to get from
cache.
        SomeInformation info;
        if (bytes is null)
        {
            // Cache miss; get the data from the real source.
            info = await SomeExpensiveOperationAsync(name, id, token);

            // Serialize and cache it.
            bytes = SomeSerializer.Serialize(info);
            await cache.SetAsync(key, bytes, token);
        }
        else
        {
            // Cache hit; deserialize it.
```

```
            info = SomeSerializer.Deserialize<SomeInformation>(bytes);
        }
        return info;
    }

    // This is the work we're trying to cache.
    private async Task<SomeInformation> SomeExpensiveOperationAsync(string
name, int id,
        CancellationToken token = default)
    { /* ... */ }
}
```

That's a lot of work to get right each time, including things like serialization. And in the "cache miss" scenario, you could end up with multiple concurrent threads, all getting a cache miss, all fetching the underlying data, all serializing it, and all sending that data to the cache.

Here's equivalent code using `HybridCache`:

C#

```
public class SomeService(HybridCache cache)
{
    public async Task<SomeInformation> GetSomeInformationAsync
        (string name, int id, CancellationToken token = default)
    {
        return await cache.GetOrCreateAsync(
            $"someinfo:{name}:{id}", // Unique key for this entry.
            async cancel => await SomeExpensiveOperationAsync(name, id,
cancel),
            token: token
        );
    }
}
```

The code is simpler and the library provides stampede protection and other features that `IDistributedCache` doesn't.

Compatibility

The library supports older .NET runtimes, down to .NET Framework 4.7.2 and .NET Standard 2.0.

Additional resources

For more information, see the following resources:

- HybridCache library in ASP.NET Core
- GitHub issue dotnet/aspnetcore #54647
- HybridCache source code

Cache Tag Helper

Cache the content from an MVC view or Razor Page with the Cache Tag Helper. The Cache Tag Helper uses in-memory caching to store data.

For more information, see Cache Tag Helper in ASP.NET Core MVC.

Distributed Cache Tag Helper

Cache the content from an MVC view or Razor Page in distributed cloud or web farm scenarios with the Distributed Cache Tag Helper. The Distributed Cache Tag Helper uses SQL Server, Redis, or NCache to store data.

For more information, see Distributed Cache Tag Helper in ASP.NET Core.

Response caching

The Response caching middleware:

- Enables caching server responses based on HTTP cache headers. Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. Output caching, which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the Conditions for caching are met.

To test response caching, use Fiddler, or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see Troubleshooting.

Output caching

The output caching middleware enables caching of HTTP responses. Output caching differs from response caching in the following ways:

- The caching behavior is configurable on the server.

  Response caching behavior is defined by HTTP headers. For example, when you visit a website with Chrome or Edge, the browser automatically sends a `Cache-control: max-age=0` header. This header effectively disables response caching, since the server follows the directions provided by the client. A new response is returned for every request, even if the server has a fresh cached response. With output caching the client doesn't override the caching behavior that you configure on the server.

- The cache storage medium is extensible.

  Memory is used by default. Response caching is limited to memory.

- You can programmatically invalidate selected cache entries.

  Response caching's dependence on HTTP headers leaves you with few options for invalidating cache entries.

- Resource locking mitigates the risk of cache stampede and thundering herd.

  Cache stampede happens when a frequently used cache entry is revoked, and too many requests try to repopulate the same cache entry at the same time. *Thundering herd* is similar: a burst of requests for the same response that isn't already in a cache entry. Resource locking ensures that all requests for a given response wait for the first request to populate the cache. Response caching doesn't have a resource locking feature.

- Cache revalidation minimizes bandwidth usage.

  Cache revalidation means the server can return a `304 Not Modified` HTTP status code instead of a cached response body. This status code informs the client that the response to the request is unchanged from what was previously received. Response caching doesn't do cache revalidation.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Cache in-memory in ASP.NET Core

Article • 11/08/2023

ⓘ **AI-assisted content.** This article was partially created with the help of AI. An author reviewed and revised the content as needed. Learn more

By Rick Anderson, John Luo, and Steve Smith

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently **and** is expensive to generate. Caching makes a copy of data that can be returned much faster than from the source. Apps should be written and tested to **never** depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the IMemoryCache. `IMemoryCache` represents a cache stored in the memory of the web server. Apps running on a server farm (multiple servers) should ensure sessions are sticky when using the in-memory cache. Sticky sessions ensure that requests from a client all go to the same server. For example, Azure Web apps use Application Request Routing (ARR) to route all requests to the same server.

Non-sticky sessions in a web farm require a distributed cache to avoid cache consistency problems. For some apps, a distributed cache can support higher scale-out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

The in-memory cache can store any object. The distributed cache interface is limited to `byte[]`. The in-memory and distributed cache store cache items as key-value pairs.

System.Runtime.Caching/MemoryCache

System.Runtime.Caching/MemoryCache (NuGet package) can be used with:

- .NET Standard 2.0 or later.
- Any .NET implementation that targets .NET Standard 2.0 or later. For example, ASP.NET Core 3.1 or later.
- .NET Framework 4.5 or later.

Microsoft.Extensions.Caching.Memory / `IMemoryCache` (described in this article) is recommended over `System.Runtime.Caching` / `MemoryCache` because it's better integrated into ASP.NET Core. For example, `IMemoryCache` works natively with ASP.NET Core dependency injection.

Use `System.Runtime.Caching`/`MemoryCache` as a compatibility bridge when porting code from ASP.NET 4.x to ASP.NET Core.

Cache guidelines

- Code should always have a fallback option to fetch data and **not** depend on a cached value being available.
- The cache uses a scarce resource, memory. Limit cache growth:
  - Do **not** insert external input into the cache. As an example, using arbitrary user-provided input as a cache key is not recommended since the input might consume an unpredictable amount of memory.
  - Use expirations to limit cache growth.
  - Use SetSize, Size, and SizeLimit to limit cache size. The ASP.NET Core runtime does **not** limit cache size based on memory pressure. It's up to the developer to limit cache size.

Use IMemoryCache

> ⚠ **Warning**
>
> Using a *shared* memory cache from Dependency Injection and calling `SetSize`, `Size`, or `SizeLimit` to limit cache size can cause the app to fail. When a size limit is set on a cache, all entries must specify a size when being added. This can lead to issues since developers may not have full control on what uses the shared cache. When using `SetSize`, `Size`, or `SizeLimit` to limit cache, create a cache singleton for caching. For more information and an example, see Use SetSize, Size, and SizeLimit to limit cache size. A shared cache is one shared by other frameworks or libraries.

In-memory caching is a *service* that's referenced from an app using Dependency Injection. Request the `IMemoryCache` instance in the constructor:

C#

```
public class IndexModel : PageModel
{
    private readonly IMemoryCache _memoryCache;

    public IndexModel(IMemoryCache memoryCache) =>
        _memoryCache = memoryCache;
```

```
    // ...
```

The following code uses TryGetValue to check if a time is in the cache. If a time isn't cached, a new entry is created and added to the cache with Set:

C#

```
public void OnGet()
{
    CurrentDateTime = DateTime.Now;

    if (!_memoryCache.TryGetValue(CacheKeys.Entry, out DateTime cacheValue))
    {
        cacheValue = CurrentDateTime;

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        _memoryCache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);
    }

    CacheCurrentDateTime = cacheValue;
}
```

In the preceding code, the cache entry is configured with a sliding expiration of three seconds. If the cache entry isn't accessed for more than three seconds, it gets evicted from the cache. Each time the cache entry is accessed, it remains in the cache for a further 3 seconds. The `CacheKeys` class is part of the download sample.

The current time and the cached time are displayed:

CSHTML

```
<ul>
    <li>Current Time: @Model.CurrentDateTime</li>
    <li>Cached Time: @Model.CacheCurrentDateTime</li>
</ul>
```

The following code uses the `Set` extension method to cache data for a relative time without `MemoryCacheEntryOptions`:

C#

```
_memoryCache.Set(CacheKeys.Entry, DateTime.Now, TimeSpan.FromDays(1));
```

In the preceding code, the cache entry is configured with a relative expiration of one day. The cache entry gets evicted from the cache after one day, even if it's accessed within this timeout period.

The following code uses GetOrCreate and GetOrCreateAsync to cache data.

C#

```
public void OnGetCacheGetOrCreate()
{
    var cachedValue = _memoryCache.GetOrCreate(
        CacheKeys.Entry,
        cacheEntry =>
        {
            cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return DateTime.Now;
        });

    // ...
}

public async Task OnGetCacheGetOrCreateAsync()
{
    var cachedValue = await _memoryCache.GetOrCreateAsync(
        CacheKeys.Entry,
        cacheEntry =>
        {
            cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return Task.FromResult(DateTime.Now);
        });

    // ...
}
```

The following code calls Get to fetch the cached time:

C#

```
var cacheEntry = _memoryCache.Get<DateTime?>(CacheKeys.Entry);
```

The following code gets or creates a cached item with absolute expiration:

C#

```
var cachedValue = _memoryCache.GetOrCreate(
    CacheKeys.Entry,
    cacheEntry =>
    {
        cacheEntry.AbsoluteExpirationRelativeToNow =
TimeSpan.FromSeconds(20);
```

```
        return DateTime.Now;
    });
```

A cached item set with only a sliding expiration is at risk of never expiring. If the cached item is repeatedly accessed within the sliding expiration interval, the item never expires. Combine a sliding expiration with an absolute expiration to guarantee the item expires. The absolute expiration sets an upper bound on how long the item can be cached while still allowing the item to expire earlier if it isn't requested within the sliding expiration interval. If either the sliding expiration interval *or* the absolute expiration time pass, the item is evicted from the cache.

The following code gets or creates a cached item with both sliding *and* absolute expiration:

C#

```
var cachedValue = _memoryCache.GetOrCreate(
    CacheKeys.CallbackEntry,
    cacheEntry =>
    {
        cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
        cacheEntry.AbsoluteExpirationRelativeToNow =
TimeSpan.FromSeconds(20);
        return DateTime.Now;
    });
```

The preceding code guarantees the data won't be cached longer than the absolute time.

GetOrCreate, GetOrCreateAsync, and Get are extension methods in the CacheExtensions class. These methods extend the capability of IMemoryCache.

`MemoryCacheEntryOptions`

The following example:

- Sets the cache priority to CacheItemPriority.NeverRemove.
- Sets a PostEvictionDelegate that gets called after the entry is evicted from the cache. The callback is run on a different thread from the code that removes the item from the cache.

C#

```
public void OnGetCacheRegisterPostEvictionCallback()
{
    var memoryCacheEntryOptions = new MemoryCacheEntryOptions()
```

```
            .SetPriority(CacheItemPriority.NeverRemove)
            .RegisterPostEvictionCallback(PostEvictionCallback, _memoryCache);

        _memoryCache.Set(CacheKeys.CallbackEntry, DateTime.Now,
    memoryCacheEntryOptions);
    }

    private static void PostEvictionCallback(
        object cacheKey, object cacheValue, EvictionReason evictionReason,
    object state)
    {
        var memoryCache = (IMemoryCache)state;

        memoryCache.Set(
            CacheKeys.CallbackMessage,
            $"Entry {cacheKey} was evicted: {evictionReason}.");
    }
```

Use SetSize, Size, and SizeLimit to limit cache size

A `MemoryCache` instance may optionally specify and enforce a size limit. The cache size limit doesn't have a defined unit of measure because the cache has no mechanism to measure the size of entries. If the cache size limit is set, all entries must specify size. The ASP.NET Core runtime doesn't limit cache size based on memory pressure. It's up to the developer to limit cache size. The size specified is in units the developer chooses.

For example:

- If the web app was primarily caching strings, each cache entry size could be the string length.
- The app could specify the size of all entries as 1, and the size limit is the count of entries.

If SizeLimit isn't set, the cache grows without bound. The ASP.NET Core runtime doesn't trim the cache when system memory is low. Apps must be architected to:

- Limit cache growth.
- Call Compact or Remove when available memory is limited.

The following code creates a unitless fixed size MemoryCache accessible by dependency injection:

```
C#
```

```
public class MyMemoryCache
{
    public MemoryCache Cache { get; } = new MemoryCache(
        new MemoryCacheOptions
        {
            SizeLimit = 1024
        });
}
```

`SizeLimit` doesn't have units. Cached entries must specify size in whatever units they consider most appropriate if the cache size limit has been set. All users of a cache instance should use the same unit system. An entry won't be cached if the sum of the cached entry sizes exceeds the value specified by `SizeLimit`. If no cache size limit is set, the cache size set on the entry is ignored.

The following code registers `MyMemoryCache` with the dependency injection container:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddSingleton<MyMemoryCache>();
```

`MyMemoryCache` is created as an independent memory cache for components that are aware of this size limited cache and know how to set cache entry size appropriately.

The size of the cache entry can be set using the SetSize extension method or the Size property:

C#

```
if (!_myMemoryCache.Cache.TryGetValue(CacheKeys.Entry, out DateTime
cacheValue))
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        .SetSize(1);

    // cacheEntryOptions.Size = 1;

    _myMemoryCache.Cache.Set(CacheKeys.Entry, cacheValue,
cacheEntryOptions);
}
```

In the preceding code, the two highlighted lines achieve the same result of setting the size of the cache entry. `SetSize` is provided for convenience when chaining calls onto

`new MemoryCacheOptions()`.

MemoryCache.Compact

`MemoryCache.Compact` attempts to remove the specified percentage of the cache in the following order:

- All expired items.
- Items by priority. Lowest priority items are removed first.
- Least recently used objects.
- Items with the earliest absolute expiration.
- Items with the earliest sliding expiration.

Pinned items with priority NeverRemove are never removed. The following code removes a cache item and calls `Compact` to remove 25% of cached entries:

C#

```
_myMemoryCache.Cache.Remove(CacheKeys.Entry);
_myMemoryCache.Cache.Compact(.25);
```

For more information, see the Compact source on GitHub.

Cache dependencies

The following sample shows how to expire a cache entry if a dependent entry expires. A CancellationChangeToken is added to the cached item. When `Cancel` is called on the `CancellationTokenSource`, both cache entries are evicted:

C#

```
public void OnGetCacheCreateDependent()
{
    var cancellationTokenSource = new CancellationTokenSource();

    _memoryCache.Set(
        CacheKeys.DependentCancellationTokenSource,
        cancellationTokenSource);

    using var parentCacheEntry = _memoryCache.CreateEntry(CacheKeys.Parent);

    parentCacheEntry.Value = DateTime.Now;

    _memoryCache.Set(
        CacheKeys.Child,
```

```
            DateTime.Now,
            new CancellationChangeToken(cancellationTokenSource.Token));
    }

    public void OnGetCacheRemoveDependent()
    {
        var cancellationTokenSource = _memoryCache.Get<CancellationTokenSource>(
            CacheKeys.DependentCancellationTokenSource);

        cancellationTokenSource.Cancel();
    }
```

Using a CancellationTokenSource allows multiple cache entries to be evicted as a group. With the `using` pattern in the code above, cache entries created inside the `using` scope inherit triggers and expiration settings.

Additional notes

- Expiration doesn't happen in the background. There's no timer that actively scans the cache for expired items. Any activity on the cache (`Get`, `Set`, `Remove`) can trigger a background scan for expired items. A timer on the `CancellationTokenSource` (CancelAfter) also removes the entry and triggers a scan for expired items. The following example uses CancellationTokenSource(TimeSpan) for the registered token. When this token fires, it removes the entry immediately and fires the eviction callbacks:

  C#

  ```
  if (!_memoryCache.TryGetValue(CacheKeys.Entry, out DateTime
  cacheValue))
  {
      cacheValue = DateTime.Now;

      var cancellationTokenSource = new CancellationTokenSource(
          TimeSpan.FromSeconds(10));

      var cacheEntryOptions = new MemoryCacheEntryOptions()
          .AddExpirationToken(
              new CancellationChangeToken(cancellationTokenSource.Token))
          .RegisterPostEvictionCallback((key, value, reason, state) =>
          {
              ((CancellationTokenSource)state).Dispose();
          }, cancellationTokenSource);

      _memoryCache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);
  }
  ```

- When using a callback to repopulate a cache item:
  - Multiple requests can find the cached key value empty because the callback hasn't completed.
  - This can result in several threads repopulating the cached item.
- When one cache entry is used to create another, the child copies the parent entry's expiration tokens and time-based expiration settings. The child isn't expired by manual removal or updating of the parent entry.
- Use PostEvictionCallbacks to set the callbacks that will be fired after the cache entry is evicted from the cache.
- For most apps, `IMemoryCache` is enabled. For example, calling `AddMvc`, `AddControllersWithViews`, `AddRazorPages`, `AddMvcCore().AddRazorViewEngine`, and many other `Add{Service}` methods in `Program.cs`, enables `IMemoryCache`. For apps that don't call one of the preceding `Add{Service}` methods, it may be necessary to call AddMemoryCache in `Program.cs`.

Background cache update

Use a background service such as IHostedService to update the cache. The background service can recompute the entries and then assign them to the cache only when they're ready.

Additional resources

- View or download sample code (how to download)
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Distributed caching in ASP.NET Core

Article • 01/29/2024

By Mohsin Nasir and smandia

A distributed cache is a cache shared by multiple app servers, typically maintained as an external service to the app servers that access it. A distributed cache can improve the performance and scalability of an ASP.NET Core app, especially when the app is hosted by a cloud service or a server farm.

A distributed cache has several advantages over other caching scenarios where cached data is stored on individual app servers.

When cached data is distributed, the data:

- Is *coherent* (consistent) across requests to multiple servers.
- Survives server restarts and app deployments.
- Doesn't use local memory.

Distributed cache configuration is implementation specific. This article describes how to configure SQL Server and Redis distributed caches. Third party implementations are also available, such as NCache (NCache on GitHub). Regardless of which implementation is selected, the app interacts with the cache using the IDistributedCache interface.

View or download sample code (how to download)

> ⚠ **Warning**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Prerequisites

Add a package reference for the distributed cache provider used:

- For a Redis distributed cache, Microsoft.Extensions.Caching.StackExchangeRedis.
- For SQL Server, Microsoft.Extensions.Caching.SqlServer.
- For the NCache distributed cache, NCache.Microsoft.Extensions.Caching.OpenSource.

IDistributedCache interface

The IDistributedCache interface provides the following methods to manipulate items in the distributed cache implementation:

- Get, GetAsync: Accepts a string key and retrieves a cached item as a `byte[]` array if found in the cache.
- Set, SetAsync: Adds an item (as `byte[]` array) to the cache using a string key.
- Refresh, RefreshAsync: Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).
- Remove, RemoveAsync: Removes a cache item based on its string key.

Establish distributed caching services

Register an implementation of IDistributedCache in `Program.cs`. Framework-provided implementations described in this topic include:

- Distributed Redis cache
- Distributed Memory Cache
- Distributed SQL Server cache
- Distributed NCache cache
- Distributed Azure CosmosDB cache

Distributed Redis Cache

We recommend production apps use the Distributed Redis Cache because it's the most performant. For more information see Recommendations.

Redis is an open source in-memory data store, which is often used as a distributed cache. You can configure an Azure Cache for Redis for an Azure-hosted ASP.NET Core app, and use an Azure Cache for Redis for local development.

An app configures the cache implementation using a RedisCache instance, by calling AddStackExchangeRedisCache. For output caching, use `AddStackExchangeRedisOutputCache`.

1. Create an Azure Cache for Redis.
2. Copy the Primary connection string (StackExchange.Redis) to Configuration.
    - Local development: Save the connection string with Secret Manager.
    - Azure: Save the connection string in a secure store such as Azure Key Vault

The following code enables the Azure Cache for Redis:

```
C#
```

```
builder.Services.AddStackExchangeRedisCache(options =>
 {
     options.Configuration =
builder.Configuration.GetConnectionString("MyRedisConStr");
     options.InstanceName = "SampleInstance";
 });
```

The preceding code assumes the Primary connection string (StackExchange.Redis) was saved in configuration with the key name `MyRedisConStr`.

For more information, see Azure Cache for Redis.

See this GitHub issue for a discussion on alternative approaches to a local Redis cache.

Distributed Memory Cache

The Distributed Memory Cache (AddDistributedMemoryCache) is a framework-provided implementation of IDistributedCache that stores items in memory. The Distributed Memory Cache isn't an actual distributed cache. Cached items are stored by the app instance on the server where the app is running.

The Distributed Memory Cache is a useful implementation:

- In development and testing scenarios.
- When a single server is used in production and memory consumption isn't an issue. Implementing the Distributed Memory Cache abstracts cached data storage. It allows for implementing a true distributed caching solution in the future if multiple nodes or fault tolerance become necessary.

The sample app makes use of the Distributed Memory Cache when the app is run in the Development environment in `Program.cs`:

```
C#
```

```
builder.Services.AddDistributedMemoryCache();
```

Distributed SQL Server Cache

The Distributed SQL Server Cache implementation (AddDistributedSqlServerCache) allows the distributed cache to use a SQL Server database as its backing store. To create a SQL Server cached item table in a SQL Server instance, you can use the `sql-cache` tool. The tool creates a table with the name and schema that you specify.

Create a table in SQL Server by running the `sql-cache create` command. Provide the SQL Server instance (`Data Source`), database (`Initial Catalog`), schema (for example, `dbo`), and table name (for example, `TestCache`):

.NET CLI

```
dotnet sql-cache create "Data Source=(localdb)/MSSQLLocalDB;Initial
Catalog=DistCache;Integrated Security=True;" dbo TestCache
```

A message is logged to indicate that the tool was successful:

Console

```
Table and index were created successfully.
```

The table created by the `sql-cache` tool has the following schema:

	Name	Data Type	Allow Nulls
🔑	Id	nvarchar(449)	☐
	Value	varbinary(MAX)	☐
	ExpiresAtTime	datetimeoffset(7)	☐
	SlidingExpirationInSeconds	bigint	☑
	AbsoluteExpiration	datetimeoffset(7)	☑

> **Note**
>
> An app should manipulate cache values using an instance of IDistributedCache, not a SqlServerCache.

The sample app implements SqlServerCache in a non-Development environment in `Program.cs`:

C#

```
builder.Services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString = builder.Configuration.GetConnectionString(
        "DistCache_ConnectionString");
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});
```

> **Note**
>
> A ConnectionString (and optionally, SchemaName and TableName) are typically stored outside of source control (for example, stored by the Secret Manager or in `appsettings.json` / `appsettings.{Environment}.json` files). The connection string may contain credentials that should be kept out of source control systems.

Distributed NCache Cache

NCache is an open source in-memory distributed cache developed natively in .NET and .NET Core. NCache works both locally and configured as a distributed cache cluster for an ASP.NET Core app running in Azure or on other hosting platforms.

To install and configure NCache on your local machine, see Getting Started Guide for Windows (.NET and .NET Core).

To configure NCache:

1. Install NCache open source NuGet.
2. Configure the cache cluster in client.ncconf.
3. Add the following code to `Program.cs`:

C#

```
builder.Services.AddNCacheDistributedCache(configuration =>
{
    configuration.CacheName = "democache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});
```

Distributed Azure CosmosDB Cache

Azure Cosmos DB can be used in ASP.NET Core as a session state provider by using the `IDistributedCache` interface. Azure Cosmos DB is a fully managed NoSQL and relational database for modern app development that offers high availability, scalability, and low-latency access to data for mission-critical applications.

After installing the Microsoft.Extensions.Caching.Cosmos NuGet package, configure an Azure Cosmos DB distributed cache as follows:

Reuse an existing client

The easiest way to configure distributed cache is by reusing an existing Azure Cosmos DB client. In this case, the `CosmosClient` instance won't be disposed when the provider is disposed.

C#

```
services.AddCosmosCache((CosmosCacheOptions cacheOptions) =>
{
    cacheOptions.ContainerName = Configuration["CosmosCacheContainer"];
    cacheOptions.DatabaseName = Configuration["CosmosCacheDatabase"];
    cacheOptions.CosmosClient = existingCosmosClient;
    cacheOptions.CreateIfNotExists = true;
});
```

Create a new client

Alternatively, instantiate a new client. In this case, the `CosmosClient` instance will get disposed when the provider is disposed.

C#

```
services.AddCosmosCache((CosmosCacheOptions cacheOptions) =>
{
    cacheOptions.ContainerName = Configuration["CosmosCacheContainer"];
    cacheOptions.DatabaseName = Configuration["CosmosCacheDatabase"];
    cacheOptions.ClientBuilder = new
CosmosClientBuilder(Configuration["CosmosConnectionString"]);
    cacheOptions.CreateIfNotExists = true;
});
```

Use the distributed cache

To use the IDistributedCache interface, request an instance of IDistributedCache in the app. The instance is provided by dependency injection (DI).

When the sample app starts, IDistributedCache is injected into `Program.cs`. The current time is cached using IHostApplicationLifetime (for more information, see Generic Host: IHostApplicationLifetime):

C#

```
app.Lifetime.ApplicationStarted.Register(() =>
{
    var currentTimeUTC = DateTime.UtcNow.ToString();
    byte[] encodedCurrentTimeUTC =
System.Text.Encoding.UTF8.GetBytes(currentTimeUTC);
    var options = new DistributedCacheEntryOptions()
        .SetSlidingExpiration(TimeSpan.FromSeconds(20));
    app.Services.GetService<IDistributedCache>()
                                .Set("cachedTimeUTC", encodedCurrentTimeUTC,
options);
});
```

The sample app injects IDistributedCache into the `IndexModel` for use by the Index page.

Each time the Index page is loaded, the cache is checked for the cached time in `OnGetAsync`. If the cached time hasn't expired, the time is displayed. If 20 seconds have elapsed since the last time the cached time was accessed (the last time this page was loaded), the page displays *Cached Time Expired*.

Immediately update the cached time to the current time by selecting the **Reset Cached Time** button. The button triggers the `OnPostResetCachedTime` handler method.

C#

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public string? CachedTimeUTC { get; set; }
    public string? ASP_Environment { get; set; }

    public async Task OnGetAsync()
    {
        CachedTimeUTC = "Cached Time Expired";
        var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");

        if (encodedCachedTimeUTC != null)
        {
            CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
```

```
        }

        ASP_Environment =
Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT");
        if (String.IsNullOrEmpty(ASP_Environment))
        {
            ASP_Environment = "Null, so Production";
        }
    }
    public async Task<IActionResult> OnPostResetCachedTime()
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC =
Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        await _cache.SetAsync("cachedTimeUTC", encodedCurrentTimeUTC,
options);

        return RedirectToPage();
    }
}
```

> There's ***no*** need to use a Singleton or Scoped lifetime for IDistributedCache instances with the built-in implementations.
>
> You can also create an IDistributedCache instance wherever you might need one instead of using DI, but creating an instance in code can make your code harder to test and violates the Explicit Dependencies Principle.

Recommendations

When deciding which implementation of IDistributedCache is best for your app, consider the following:

- Existing infrastructure
- Performance requirements
- Cost
- Team experience

Caching solutions usually rely on in-memory storage to provide fast retrieval of cached data, but memory is a limited resource and costly to expand. Only store commonly used data in a cache.

For most apps, a Redis cache provides higher throughput and lower latency than a SQL Server cache. However, benchmarking is recommended to determine the performance characteristics of caching strategies.

When SQL Server is used as a distributed cache backing store, use of the same database for the cache and the app's ordinary data storage and retrieval can negatively impact the performance of both. We recommend using a dedicated SQL Server instance for the distributed cache backing store.

Additional resources

- Redis Cache on Azure
- SQL Database on Azure
- ASP.NET Core IDistributedCache Provider for NCache in Web Farms (NCache on GitHub)
- Repository README file for Microsoft.Extensions.Caching.Cosmos
- Cache in-memory in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core
- Host ASP.NET Core in a web farm

Output caching middleware in ASP.NET Core

Article • 07/11/2024

By Tom Dykstra

This article explains how to configure output caching middleware in an ASP.NET Core app. For an introduction to output caching, see Output caching.

The output caching middleware can be used in all types of ASP.NET Core apps: Minimal API, Web API with controllers, MVC, and Razor Pages. Code examples are provided for minimal APIs and controller-based APIs. The controller-based API examples show how to use attributes to configure caching. These attributes can also be used in MVC and Razor Pages apps.

The code examples refer to a Gravatar class that generates an image and provides a "generated at" date and time. The class is defined and used only in the sample app. Its purpose is to make it easy to see when cached output is being used. For more information, see How to download a sample and Preprocessor directives in sample code.

Add the middleware to the app

Add the output caching middleware to the service collection by calling AddOutputCache.

Add the middleware to the request processing pipeline by calling UseOutputCache.

For example:

C#

```
builder.Services.AddOutputCache();
```

C#

```
var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseHttpsRedirection();
app.UseOutputCache();
app.UseAuthorization();
```

Calling `AddOutputCache` and `UseOutputCache` doesn't start caching behavior, it makes caching available. To make the app cache responses, caching must be configured as shown in the following sections.

> ⓘ Note
>
> - In apps that use CORS middleware, `UseOutputCache` must be called after UseCors.
> - In Razor Pages apps and apps with controllers, `UseOutputCache` must be called after `UseRouting`.

Configure one endpoint or page

For minimal API apps, configure an endpoint to do caching by calling CacheOutput, or by applying the [OutputCache] attribute, as shown in the following examples:

C#

```
app.MapGet("/cached", Gravatar.WriteGravatar).CacheOutput();
app.MapGet("/attribute", [OutputCache] (context) =>
    Gravatar.WriteGravatar(context));
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method as shown here:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache]
public class CachedController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

For Razor Pages apps, apply the attribute to the Razor page class.

Configure multiple endpoints or pages

Create *policies* when calling `AddOutputCache` to specify caching configuration that applies to multiple endpoints. A policy can be selected for specific endpoints, while a base policy provides default caching configuration for a collection of endpoints.

The following highlighted code configures caching for all of the app's endpoints, with expiration time of 10 seconds. If an expiration time isn't specified, it defaults to one minute.

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder =>
        builder.Expire(TimeSpan.FromSeconds(10)));
    options.AddPolicy("Expire20", builder =>
        builder.Expire(TimeSpan.FromSeconds(20)));
    options.AddPolicy("Expire30", builder =>
        builder.Expire(TimeSpan.FromSeconds(30)));
});
```

The following highlighted code creates two policies, each specifying a different expiration time. Selected endpoints can use the 20-second expiration, and others can use the 30-second expiration.

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder =>
        builder.Expire(TimeSpan.FromSeconds(10)));
    options.AddPolicy("Expire20", builder =>
        builder.Expire(TimeSpan.FromSeconds(20)));
    options.AddPolicy("Expire30", builder =>
        builder.Expire(TimeSpan.FromSeconds(30)));
});
```

You can select a policy for an endpoint when calling the `CacheOutput` method or using the `[OutputCache]` attribute.

In a minimal API app, the following code configures one endpoint with a 20-second expiration and one with a 30-second expiration:

C#

```
app.MapGet("/20", Gravatar.WriteGravatar).CacheOutput("Expire20");
app.MapGet("/30", [OutputCache(PolicyName = "Expire30")] (context) =>
```

```
    Gravatar.WriteGravatar(context));
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method to select a policy:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "Expire20")]
public class Expire20Controller : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

For Razor Pages apps, apply the attribute to the Razor page class.

Default output caching policy

By default, output caching follows these rules:

- Only HTTP 200 responses are cached.
- Only HTTP GET or HEAD requests are cached.
- Responses that set cookies aren't cached.
- Responses to authenticated requests aren't cached.

The following code applies all of the default caching rules to all of an app's endpoints:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder.Cache());
});
```

Override the default policy

The following code shows how to override the default rules. The highlighted lines in the following custom policy code enable caching for HTTP POST methods and HTTP 301 responses:

```
```

C#

```
using Microsoft.AspNetCore.OutputCaching;
using Microsoft.Extensions.Primitives;

namespace OCMinimal;

public sealed class MyCustomPolicy : IOutputCachePolicy
{
    public static readonly MyCustomPolicy Instance = new();

    private MyCustomPolicy()
    {
    }

    ValueTask IOutputCachePolicy.CacheRequestAsync(
        OutputCacheContext context,
        CancellationToken cancellationToken)
    {
        var attemptOutputCaching = AttemptOutputCaching(context);
        context.EnableOutputCaching = true;
        context.AllowCacheLookup = attemptOutputCaching;
        context.AllowCacheStorage = attemptOutputCaching;
        context.AllowLocking = true;

        // Vary by any query by default
        context.CacheVaryByRules.QueryKeys = "*";

        return ValueTask.CompletedTask;
    }

    ValueTask IOutputCachePolicy.ServeFromCacheAsync
        (OutputCacheContext context, CancellationToken cancellationToken)
    {
        return ValueTask.CompletedTask;
    }

    ValueTask IOutputCachePolicy.ServeResponseAsync
        (OutputCacheContext context, CancellationToken cancellationToken)
    {
        var response = context.HttpContext.Response;

        // Verify existence of cookie headers
        if (!StringValues.IsNullOrEmpty(response.Headers.SetCookie))
        {
            context.AllowCacheStorage = false;
            return ValueTask.CompletedTask;
        }

        // Check response code
        if (response.StatusCode != StatusCodes.Status200OK &&
            response.StatusCode != StatusCodes.Status301MovedPermanently)
        {
            context.AllowCacheStorage = false;
```

```
            return ValueTask.CompletedTask;
        }

        return ValueTask.CompletedTask;
    }

    private static bool AttemptOutputCaching(OutputCacheContext context)
    {
        // Check if the current request fulfills the requirements
        // to be cached
        var request = context.HttpContext.Request;

        // Verify the method
        if (!HttpMethods.IsGet(request.Method) &&
            !HttpMethods.IsHead(request.Method) &&
            !HttpMethods.IsPost(request.Method))
        {
            return false;
        }

        // Verify existence of authorization headers
        if (!StringValues.IsNullOrEmpty(request.Headers.Authorization) ||
            request.HttpContext.User?.Identity?.IsAuthenticated == true)
        {
            return false;
        }

        return true;
    }
}
```

To use this custom policy, create a named policy:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddPolicy("CachePost", MyCustomPolicy.Instance);
});
```

And select the named policy for an endpoint. The following code selects the custom policy for an endpoint in a minimal API app:

C#

```
app.MapPost("/cachedpost", Gravatar.WriteGravatar)
    .CacheOutput("CachePost");
```

The following code does the same for a controller action:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "CachePost")]
public class PostController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Alternative default policy override

Alternatively, use Dependency Injection (DI) to initialize an instance, with the following changes to the custom policy class:

- A public constructor instead of a private constructor.
- Eliminate the `Instance` property in the custom policy class.

For example:

C#

```
public sealed class MyCustomPolicy2 : IOutputCachePolicy
{

    public MyCustomPolicy2()
    {
    }
```

The remainder of the class is the same as shown previously. Add the custom policy as shown in the following example:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder =>
        builder.AddPolicy<MyCustomPolicy2>(), true);
});
```

The preceding code uses DI to create the instance of the custom policy class. Any public arguments in the constructor are resolved.

When using a custom policy as a base policy, don't call `OutputCache()` (with no arguments) or use the `[OutputCache]` attribute on any endpoint that the base policy should apply to. Calling `OutputCache()` or using the attribute adds the default policy to the endpoint.

Specify the cache key

By default, every part of the URL is included as the key to a cache entry, that is, the scheme, host, port, path, and query string. However, you might want to explicitly control the cache key. For example, suppose you have an endpoint that returns a unique response only for each unique value of the `culture` query string. Variation in other parts of the URL, such as other query strings, shouldn't result in different cache entries. You can specify such rules in a policy, as shown in the following highlighted code:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});
```

You can then select the `VaryByQuery` policy for an endpoint. In a minimal API app, the following code selects the `VaryByQuery` policy for an endpoint that returns a unique response only for each unique value of the `culture` query string:

C#

```
app.MapGet("/query", Gravatar.WriteGravatar).CacheOutput("Query");
```

The following code does the same for a controller action:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "Query")]
public class QueryController : ControllerBase
{
```

```
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Here are some of the options for controlling the cache key:

- SetVaryByQuery - Specify one or more query string names to add to the cache key.
- SetVaryByHeader - Specify one or more HTTP headers to add to the cache key.
- VaryByValue- Specify a value to add to the cache key. The following example uses a value that indicates whether the current server time in seconds is odd or even. A new response is generated only when the number of seconds goes from odd to even or even to odd.

  C#

  ```
  builder.Services.AddOutputCache(options =>
  {
      options.AddBasePolicy(builder => builder
          .With(c =>
  c.HttpContext.Request.Path.StartsWithSegments("/blog"))
          .Tag("tag-blog"));
      options.AddBasePolicy(builder => builder.Tag("tag-all"));
      options.AddPolicy("Query", builder =>
  builder.SetVaryByQuery("culture"));
      options.AddPolicy("NoCache", builder => builder.NoCache());
      options.AddPolicy("NoLock", builder => builder.SetLocking(false));
      options.AddPolicy("VaryByValue", builder =>
          builder.VaryByValue((context) =>
              new KeyValuePair<string, string>(
              "time", (DateTime.Now.Second % 2)
                  .ToString(CultureInfo.InvariantCulture))));
  });
  ```

Use OutputCacheOptions.UseCaseSensitivePaths to specify that the path part of the key is case sensitive. The default is case insensitive.

For more options, see the OutputCachePolicyBuilder class.

Cache revalidation

Cache revalidation means the server can return a `304 Not Modified` HTTP status code instead of the full response body. This status code informs the client that the response to the request is unchanged from what the client previously received.

The following code illustrates the use of an Etag header to enable cache revalidation. If the client sends an If-None-Match header with the etag value of an earlier response, and the cache entry is fresh, the server returns 304 Not Modified instead of the full response. Here's how to set the etag value in a policy, in a Minimal API app:

C#

```
app.MapGet("/etag", async (context) =>
{
    var etag = $"\"{Guid.NewGuid():n}\"";
    context.Response.Headers.ETag = etag;
    await Gravatar.WriteGravatar(context);

}).CacheOutput();
```

And here's how to set the etag value in a controller-based API:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache]
public class EtagController : ControllerBase
{
    public async Task GetAsync()
    {
        var etag = $"\"{Guid.NewGuid():n}\"";
        HttpContext.Response.Headers.ETag = etag;
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Another way to do cache revalidation is to check the date of the cache entry creation compared to the date requested by the client. When the request header `If-Modified-Since` is provided, output caching returns 304 if the cached entry is older and isn't expired.

Cache revalidation is automatic in response to these headers sent from the client. No special configuration is required on the server to enable this behavior, aside from enabling output caching.

Use tags to evict cache entries

You can use tags to identify a group of endpoints and evict all cache entries for the group. For example, the following minimal API code creates a pair of endpoints whose URLs begin with "blog", and tags them "tag-blog":

C#

```
app.MapGet("/blog", Gravatar.WriteGravatar)
    .CacheOutput(builder => builder.Tag("tag-blog"));
app.MapGet("/blog/post/{id}", Gravatar.WriteGravatar)
    .CacheOutput(builder => builder.Tag("tag-blog"));
```

The following code shows how to assign tags to an endpoint in a controller-based API:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(Tags = new[] { "tag-blog", "tag-all" })]
public class TagEndpointController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

An alternative way to assign tags for endpoints with routes that begin with `blog` is to define a base policy that applies to all endpoints with that route. The following code shows how to do that:

C#

```
builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});
```

Another alternative for minimal API apps is to call `MapGroup`:

C#

```
var blog = app.MapGroup("blog")
    .CacheOutput(builder => builder.Tag("tag-blog"));
blog.MapGet("/", Gravatar.WriteGravatar);
blog.MapGet("/post/{id}", Gravatar.WriteGravatar);
```

In the preceding tag assignment examples, both endpoints are identified by the `tag-blog` tag. You can then evict the cache entries for those endpoints with a single statement that references that tag:

```
C#

app.MapPost("/purge/{tag}", async (IOutputCacheStore cache, string tag) =>
{
    await cache.EvictByTagAsync(tag, default);
});
```

With this code, an HTTP POST request sent to `https://localhost:<port>/purge/tag-blog` evicts cache entries for these endpoints.

You might want a way to evict all cache entries for all endpoints. To do that, create a base policy for all endpoints as the following code does:

```
C#

builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});
```

This base policy enables you to use the "tag-all" tag to evict everything in cache.

Disable resource locking

By default, resource locking is enabled to mitigate the risk of cache stampede and thundering herd. For more information, see Output Caching.

To disable resource locking, call SetLocking(false) while creating a policy, as shown in the following example:

```
C#

builder.Services.AddOutputCache(options =>
{
    options.AddBasePolicy(builder => builder
```

```
        .With(c => c.HttpContext.Request.Path.StartsWithSegments("/blog"))
        .Tag("tag-blog"));
    options.AddBasePolicy(builder => builder.Tag("tag-all"));
    options.AddPolicy("Query", builder =>
builder.SetVaryByQuery("culture"));
    options.AddPolicy("NoCache", builder => builder.NoCache());
    options.AddPolicy("NoLock", builder => builder.SetLocking(false));
});
```

The following example selects the no-locking policy for an endpoint in a minimal API app:

C#

```
app.MapGet("/nolock", Gravatar.WriteGravatar)
    .CacheOutput("NoLock");
```

In a controller-based API, use the attribute to select the policy:

C#

```
[ApiController]
[Route("/[controller]")]
[OutputCache(PolicyName = "NoLock")]
public class NoLockController : ControllerBase
{
    public async Task GetAsync()
    {
        await Gravatar.WriteGravatar(HttpContext);
    }
}
```

Limits

The following properties of OutputCacheOptions let you configure limits that apply to all endpoints:

- SizeLimit - Maximum size of cache storage. When this limit is reached, no new responses are cached until older entries are evicted. Default value is 100 MB.
- MaximumBodySize - If the response body exceeds this limit, it isn't cached. Default value is 64 MB.
- DefaultExpirationTimeSpan - The expiration time duration that applies when not specified by a policy. Default value is 60 seconds.

Cache storage

IOutputCacheStore is used for storage. By default it's used with MemoryCache. Cached responses are stored in-process, so each server has a separate cache that is lost whenever the server process is restarted.

Redis cache

An alternative is to use Redis cache. Redis cache provides consistency between server nodes via a shared cache that outlives individual server processes. To use Redis for output caching:

- Install the Microsoft.AspNetCore.OutputCaching.StackExchangeRedis NuGet package.
- Call `builder.Services.AddStackExchangeRedisOutputCache` (not `AddStackExchangeRedisCache`), and provide a connection string that points to a Redis server.

  For example:

  ```
  C#

  builder.Services.AddStackExchangeRedisOutputCache(options =>
  {
      options.Configuration =
          builder.Configuration.GetConnectionString("MyRedisConStr");
      options.InstanceName = "SampleInstance";
  });

  builder.Services.AddOutputCache(options =>
  {
      options.AddBasePolicy(builder =>
          builder.Expire(TimeSpan.FromSeconds(10)));
  });
  ```

  - options.Configuration - A connection string to an on-premises Redis server or to a hosted offering such as Azure Cache for Redis. For example, `<instance_name>.redis.cache.windows.net:6380,password=<password>,ssl=True,abortConnect=False` for Azure cache for Redis.
  - options.InstanceName - Optional, specifies a logical partition for the cache.

  The configuration options are identical to the Redis-based distributed caching options.

`IDistributedCache` not recommended

We don't recommend IDistributedCache for use with output caching. `IDistributedCache` doesn't have atomic features, which are required for tagging. We recommend that you use the built-in support for Redis or create custom IOutputCacheStore implementations by using direct dependencies on the underlying storage mechanism.

See also

- Overview of caching in ASP.NET Core
- ASP.NET Core Middleware
- OutputCacheOptions
- OutputCachePolicyBuilder

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Response caching in ASP.NET Core

Article • 04/11/2023

By Rick Anderson and Kirk Larkin

View or download sample code (how to download)

Response caching reduces the number of requests a client or proxy makes to a web server. Response caching also reduces the amount of work the web server performs to generate a response. Response caching is set in headers.

The ResponseCache attribute sets response caching headers. Clients and intermediate proxies should honor the headers for caching responses under RFC 9111: HTTP Caching.

For server-side caching that follows the HTTP 1.1 Caching specification, use Response Caching Middleware. The middleware can use the ResponseCacheAttribute properties to influence server-side caching behavior.

The Response caching middleware:

- Enables caching server responses based on HTTP cache headers. Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. Output caching, which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the Conditions for caching are met.

To test response caching, use Fiddler, or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see Troubleshooting.

HTTP-based response caching

RFC 9111: HTTP Caching describes how Internet caches should behave. The primary HTTP header used for caching is Cache-Control, which is used to specify cache directives. The directives control caching behavior as requests make their way from clients to servers and as responses make their way from servers back to clients. Requests

and responses move through proxy servers, and proxy servers must also conform to the HTTP 1.1 Caching specification.

Common `Cache-Control` directives are shown in the following table.

Expand table

Directive	Action
public	A cache may store the response.
private	The response must not be stored by a shared cache. A private cache may store and reuse the response.
max-age	The client doesn't accept a response whose age is greater than the specified number of seconds. Examples: `max-age=60` (60 seconds), `max-age=2592000` (1 month)
no-cache	**On requests**: A cache must not use a stored response to satisfy the request. The origin server regenerates the response for the client, and the middleware updates the stored response in its cache. **On responses**: The response must not be used for a subsequent request without validation on the origin server.
no-store	**On requests**: A cache must not store the request. **On responses**: A cache must not store any part of the response.

Other cache headers that play a role in caching are shown in the following table.

Expand table

Header	Function
Age	An estimate of the amount of time in seconds since the response was generated or successfully validated at the origin server.
Expires	The time after which the response is considered stale.
Pragma	Exists for backwards compatibility with HTTP/1.0 caches for setting `no-cache` behavior. If the `Cache-Control` header is present, the `Pragma` header is ignored.
Vary	Specifies that a cached response must not be sent unless all of the `Vary` header fields match in both the cached response's original request and the new request.

HTTP-based caching respects request Cache-Control directives

RFC 9111: HTTP Caching (Section 5.2. Cache-Control) requires a cache to honor a valid `Cache-Control` header sent by the client. A client can make requests with a `no-cache` header value and force the server to generate a new response for every request.

Always honoring client `Cache-Control` request headers makes sense if you consider the goal of HTTP caching. Under the official specification, caching is meant to reduce the latency and network overhead of satisfying requests across a network of clients, proxies, and servers. It isn't necessarily a way to control the load on an origin server.

There's no developer control over this caching behavior when using the Response Caching Middleware because the middleware adheres to the official caching specification. Support for *output caching* to better control server load was added in .NET 7. For more information, see Output caching.

ResponseCache attribute

The ResponseCacheAttribute specifies the parameters necessary for setting appropriate headers in response caching.

> ⚠ **Warning**
>
> Disable caching for content that contains information for authenticated clients. Caching should only be enabled for content that doesn't change based on a user's identity or whether a user is signed in.

VaryByQueryKeys varies the stored response by the values of the given list of query keys. When a single value of `*` is provided, the middleware varies responses by all request query string parameters.

Response Caching Middleware must be enabled to set the VaryByQueryKeys property. Otherwise, a runtime exception is thrown. There isn't a corresponding HTTP header for the VaryByQueryKeys property. The property is an HTTP feature handled by Response Caching Middleware. For the middleware to serve a cached response, the query string and query string value must match a previous request. For example, consider the sequence of requests and results shown in the following table:

Expand table

Request	Returned from
`http://example.com?key1=value1`	Server

Request	Returned from
`http://example.com?key1=value1`	Middleware
`http://example.com?key1=NewValue`	Server

The first request is returned by the server and cached in middleware. The second request is returned by middleware because the query string matches the previous request. The third request isn't in the middleware cache because the query string value doesn't match a previous request.

The ResponseCacheAttribute is used to configure and create (via IFilterFactory) a `Microsoft.AspNetCore.Mvc.Internal.ResponseCacheFilter`. The `ResponseCacheFilter` performs the work of updating the appropriate HTTP headers and features of the response. The filter:

- Removes any existing headers for `Vary`, `Cache-Control`, and `Pragma`.
- Writes out the appropriate headers based on the properties set in the ResponseCacheAttribute.
- Updates the response caching HTTP feature if VaryByQueryKeys is set.

Vary

This header is only written when the VaryByHeader property is set. The property set to the `Vary` property's value. The following sample uses the VaryByHeader property:

C#

```
[ApiController]
public class TimeController : ControllerBase
{
    [Route("api/[controller]")]
    [HttpGet]
    [ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
    public ContentResult GetTime() => Content(
                      DateTime.Now.Millisecond.ToString());
```

View the response headers with Fiddler or another tool. The response headers include:

text

```
Cache-Control: public,max-age=30
Vary: User-Agent
```

The preceding code requires adding the Response Caching Middleware services AddResponseCaching to the service collection and configures the app to use the middleware with the UseResponseCaching extension method.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddResponseCaching();

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

`NoStore` and `Location.None`

NoStore overrides most of the other properties. When this property is set to `true`, the `Cache-Control` header is set to `no-store`. If Location is set to `None`:

- `Cache-Control` is set to `no-store,no-cache`.
- `Pragma` is set to `no-cache`.

If NoStore is `false` and Location is `None`, `Cache-Control`, and `Pragma` are set to `no-cache`.

NoStore is typically set to `true` for error pages. The following produces response headers that instruct the client not to store the response.

C#

```
[Route("api/[controller]/ticks")]
[HttpGet]
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
public ContentResult GetTimeTicks() => Content(
                  DateTime.Now.Ticks.ToString());
```

The preceding code includes the following headers in the response:

```
Cache-Control: no-store,no-cache
Pragma: no-cache
```

To apply the ResponseCacheAttribute to all of the app's MVC controller or Razor Pages page responses, add it with an MVC filter or Razor Pages filter.

In an MVC app:

C#

```
builder.Services.AddControllersWithViews().AddMvcOptions(options =>
    options.Filters.Add(
        new ResponseCacheAttribute
        {
            NoStore = true,
            Location = ResponseCacheLocation.None
        }));
```

For an approach that applies to Razor Pages apps, see Adding ResponseCacheAttribute to MVC global filter list does not apply to Razor Pages (dotnet/aspnetcore #18890). The example provided in the issue comment was written for apps targeting ASP.NET Core prior to the release of Minimal APIs at 6.0. For 6.0 or later apps, change the service registration in the example to `builder.Services.AddSingleton...` for `Program.cs`.

Location and Duration

To enable caching, Duration must be set to a positive value and Location must be either `Any` (the default) or `Client`. The framework sets the `Cache-Control` header to the location value followed by the `max-age` of the response.

Location's options of `Any` and `Client` translate into `Cache-Control` header values of `public` and `private`, respectively. As noted in the NoStore and Location.None section, setting Location to `None` sets both `Cache-Control` and `Pragma` headers to `no-cache`.

`Location.Any` (`Cache-Control` set to `public`) indicates that the *client or any intermediate proxy* may cache the value, including Response Caching Middleware.

`Location.Client` (`Cache-Control` set to `private`) indicates that *only the client* may cache the value. No intermediate cache should cache the value, including Response Caching Middleware.

Cache control headers provide guidance to clients and intermediary proxies when and how to cache responses. There's no guarantee that clients and proxies will honor RFC 9111: HTTP Caching. Response Caching Middleware always follows the caching rules laid out by the specification.

The following example shows the headers produced by setting Duration and leaving the default Location value:

C#

```
[Route("api/[controller]/ms")]
[HttpGet]
[ResponseCache(Duration = 10, Location = ResponseCacheLocation.Any, NoStore
= false)]
public ContentResult GetTimeMS() => Content(
                  DateTime.Now.Millisecond.ToString());
```

The preceding code includes the following headers in the response:

```
Cache-Control: public,max-age=10
```

Cache profiles

Instead of duplicating response cache settings on many controller action attributes, cache profiles can be configured as options when setting up MVC/Razor Pages. Values found in a referenced cache profile are used as the defaults by the ResponseCacheAttribute and are overridden by any properties specified on the attribute.

The following example shows a 30 second cache profile:

C#

```
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddResponseCaching();
builder.Services.AddControllers(options =>
{
    options.CacheProfiles.Add("Default30",
        new CacheProfile()
        {
            Duration = 30
        });
});

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The following code references the `Default30` cache profile:

C#

```
[ApiController]
[ResponseCache(CacheProfileName = "Default30")]
public class Time2Controller : ControllerBase
{
    [Route("api/[controller]")]
    [HttpGet]
    public ContentResult GetTime() => Content(
                    DateTime.Now.Millisecond.ToString());

    [Route("api/[controller]/ticks")]
    [HttpGet]
    public ContentResult GetTimeTicks() => Content(
                    DateTime.Now.Ticks.ToString());
}
```

The resulting header response by the `Default30` cache profile includes:

```
Cache-Control: public,max-age=30
```

The [ResponseCache] attribute can be applied to:

- Razor Pages: Attributes can't be applied to handler methods. Browsers used with UI apps prevent response caching.
- MVC controllers.
- MVC action methods: Method-level attributes override the settings specified in class-level attributes.

The following code applies the `[ResponseCache]` attribute at the controller level and method level:

C#

```
[ApiController]
[ResponseCache(VaryByHeader = "User-Agent", Duration = 30)]
public class Time4Controller : ControllerBase
{
    [Route("api/[controller]")]
    [HttpGet]
    public ContentResult GetTime() => Content(
                      DateTime.Now.Millisecond.ToString());

    [Route("api/[controller]/ticks")]
    [HttpGet]
    public ContentResult GetTimeTicks() => Content(
                  DateTime.Now.Ticks.ToString());

    [Route("api/[controller]/ms")]
    [HttpGet]
    [ResponseCache(Duration = 10, Location = ResponseCacheLocation.Any,
NoStore = false)]
    public ContentResult GetTimeMS() => Content(
                      DateTime.Now.Millisecond.ToString());
}
```

Additional resources

- Storing Responses in Caches
- Cache-Control
- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Response Caching Middleware in ASP.NET Core

Article • 04/11/2023

By John Luo and Rick Anderson

This article explains how to configure Response Caching Middleware in an ASP.NET Core app. The middleware determines when responses are cacheable, stores responses, and serves responses from cache. For an introduction to HTTP caching and the [ResponseCache] attribute, see Response Caching.

The Response caching middleware:

- Enables caching server responses based on HTTP cache headers. Implements the standard HTTP caching semantics. Caches based on HTTP cache headers like proxies do.
- Is typically not beneficial for UI apps such as Razor Pages because browsers generally set request headers that prevent caching. Output caching, which is available in ASP.NET Core 7.0 and later, benefits UI apps. With output caching, configuration decides what should be cached independently of HTTP headers.
- May be beneficial for public GET or HEAD API requests from clients where the Conditions for caching are met.

To test response caching, use Fiddler, or another tool that can explicitly set request headers. Setting headers explicitly is preferred for testing caching. For more information, see Troubleshooting.

Configuration

In `Program.cs`, add the Response Caching Middleware services AddResponseCaching to the service collection and configure the app to use the middleware with the UseResponseCaching extension method. `UseResponseCaching` adds the middleware to the request processing pipeline:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching();

var app = builder.Build();
```

```
app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();
```

> ⚠ **Warning**
>
> UseCors must be called before UseResponseCaching when using CORS middleware.

The sample app adds headers to control caching on subsequent requests:

- Cache-Control: Caches cacheable responses for up to 10 seconds.
- Vary: Configures the middleware to serve a cached response only if the Accept-Encoding header of subsequent requests matches that of the original request.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching();

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next();
});

app.MapGet("/", () => DateTime.Now.Millisecond);
```

```
app.Run();
```

The preceding headers are not written to the response and are overridden when a controller, action, or Razor Page:

- Has a [ResponseCache] attribute. This applies even if a property isn't set. For example, omitting the VaryByHeader property will cause the corresponding header to be removed from the response.

Response Caching Middleware only caches server responses that result in a 200 (OK) status code. Any other responses, including error pages, are ignored by the middleware.

> ⚠ **Warning**
>
> Responses containing content for authenticated clients must be marked as not cacheable to prevent the middleware from storing and serving those responses. See **Conditions for caching** for details on how the middleware determines if a response is cacheable.

The preceding code typically doesn't return a cached value to a browser. Use Fiddler or another tool that can explicitly set request headers and is preferred for testing caching. For more information, see Troubleshooting in this article.

Options

Response caching options are shown in the following table.

Expand table

Option	Description
MaximumBodySize	The largest cacheable size for the response body in bytes. The default value is `64 * 1024 * 1024` (64 MB).
SizeLimit	The size limit for the response cache middleware in bytes. The default value is `100 * 1024 * 1024` (100 MB).
UseCaseSensitivePaths	Determines if responses are cached on case-sensitive paths. The default value is `false`.

The following example configures the middleware to:

- Cache responses with a body size smaller than or equal to 1,024 bytes.

- Store the responses by case-sensitive paths. For example, `/page1` and `/Page1` are stored separately.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCaching(options =>
{
    options.MaximumBodySize = 1024;
    options.UseCaseSensitivePaths = true;
});

var app = builder.Build();

app.UseHttpsRedirection();

// UseCors must be called before UseResponseCaching
//app.UseCors();

app.UseResponseCaching();

app.Use(async (context, next) =>
{
    context.Response.GetTypedHeaders().CacheControl =
        new Microsoft.Net.Http.Headers.CacheControlHeaderValue()
        {
            Public = true,
            MaxAge = TimeSpan.FromSeconds(10)
        };
    context.Response.Headers[Microsoft.Net.Http.Headers.HeaderNames.Vary] =
        new string[] { "Accept-Encoding" };

    await next(context);
});

app.MapGet("/", () => DateTime.Now.Millisecond);

app.Run();
```

VaryByQueryKeys

When using MVC, web API controllers, or Razor Pages page models, the [ResponseCache] attribute specifies the parameters necessary for setting the appropriate headers for response caching. The only parameter of the `[ResponseCache]` attribute that strictly requires the middleware is VaryByQueryKeys, which doesn't correspond to an actual HTTP header. For more information, see Response caching in ASP.NET Core.

When not using the `[ResponseCache]` attribute, response caching can be varied with `VaryByQueryKeys`. Use the ResponseCachingFeature directly from the HttpContext.Features:

```
C#

var responseCachingFeature =
context.HttpContext.Features.Get<IResponseCachingFeature>();

if (responseCachingFeature != null)
{
    responseCachingFeature.VaryByQueryKeys = new[] { "MyKey" };
}
```

Using a single value equal to `*` in `VaryByQueryKeys` varies the cache by all request query parameters.

HTTP headers used by Response Caching Middleware

The following table provides information on HTTP headers that affect response caching.

Expand table

Header	Details
`Authorization`	The response isn't cached if the header exists.
`Cache-Control`	The middleware only considers caching responses marked with the `public` cache directive. Control caching with the following parameters: • max-age • max-stale† • min-fresh • must-revalidate • no-cache • no-store • only-if-cached • private • public • s-maxage • proxy-revalidate‡ †If no limit is specified to `max-stale`, the middleware takes no action. ‡`proxy-revalidate` has the same effect as `must-revalidate`.

Header	Details
	For more information, see RFC 9111: Request Directives.
`Pragma`	A `Pragma: no-cache` header in the request produces the same effect as `Cache-Control: no-cache`. This header is overridden by the relevant directives in the `Cache-Control` header, if present. Considered for backward compatibility with HTTP/1.0.
`Set-Cookie`	The response isn't cached if the header exists. Any middleware in the request processing pipeline that sets one or more cookies prevents the Response Caching Middleware from caching the response (for example, the cookie-based TempData provider).
`Vary`	The `Vary` header is used to vary the cached response by another header. For example, cache responses by encoding by including the `Vary: Accept-Encoding` header, which caches responses for requests with headers `Accept-Encoding: gzip` and `Accept-Encoding: text/plain` separately. A response with a header value of `*` is never stored.
`Expires`	A response deemed stale by this header isn't stored or retrieved unless overridden by other `Cache-Control` headers.
`If-None-Match`	The full response is served from cache if the value isn't `*` and the `ETag` of the response doesn't match any of the values provided. Otherwise, a 304 (Not Modified) response is served.
`If-Modified-Since`	If the `If-None-Match` header isn't present, a full response is served from cache if the cached response date is newer than the value provided. Otherwise, a *304 - Not Modified* response is served.
`Date`	When serving from cache, the `Date` header is set by the middleware if it wasn't provided on the original response.
`Content-Length`	When serving from cache, the `Content-Length` header is set by the middleware if it wasn't provided on the original response.
`Age`	The `Age` header sent in the original response is ignored. The middleware computes a new value when serving a cached response.

Caching respects request Cache-Control directives

The middleware respects the rules of RFC 9111: HTTP Caching (Section 5.2. Cache-Control). The rules require a cache to honor a valid `Cache-Control` header sent by the client. Under the specification, a client can make requests with a `no-cache` header value

and force the server to generate a new response for every request. Currently, there's no developer control over this caching behavior when using the middleware because the middleware adheres to the official caching specification.

For more control over caching behavior, explore other caching features of ASP.NET Core. See the following topics:

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Troubleshooting

The Response Caching Middleware uses IMemoryCache, which has a limited capacity. When the capacity is exceeded, the memory cache is compacted.

If caching behavior isn't as expected, confirm that responses are cacheable and capable of being served from the cache. Examine the request's incoming headers and the response's outgoing headers. Enable logging to help with debugging.

When testing and troubleshooting caching behavior, a browser typically sets request headers that prevent caching. For example, a browser may set the `Cache-Control` header to `no-cache` or `max-age=0` when refreshing a page. Fiddler and other tools can explicitly set request headers and are preferred for testing caching.

Conditions for caching

- The request must result in a server response with a 200 (OK) status code.
- The request method must be GET or HEAD.
- Response Caching Middleware must be placed before middleware that require caching. For more information, see ASP.NET Core Middleware.
- The `Authorization` header must not be present.
- `Cache-Control` header parameters must be valid, and the response must be marked `public` and not marked `private`.
- The `Pragma: no-cache` header must not be present if the `Cache-Control` header isn't present, as the `Cache-Control` header overrides the `Pragma` header when present.
- The `Set-Cookie` header must not be present.
- `Vary` header parameters must be valid and not equal to `*`.

- The `Content-Length` header value (if set) must match the size of the response body.
- The IHttpSendFileFeature isn't used.
- The response must not be stale as specified by the `Expires` header and the `max-age` and `s-maxage` cache directives.
- Response buffering must be successful. The size of the response must be smaller than the configured or default SizeLimit. The body size of the response must be smaller than the configured or default MaximumBodySize.
- The response must be cacheable according to RFC 9111: HTTP Caching. For example, the `no-store` directive must not exist in request or response header fields. See RFC 9111: HTTP Caching (Section 3: *Storing Responses in Caches*) for details.

> **Note**
>
> The Antiforgery system for generating secure tokens to prevent Cross-Site Request Forgery (CSRF) attacks sets the `Cache-Control` and `Pragma` headers to `no-cache` so that responses aren't cached. For information on how to disable antiforgery tokens for HTML form elements, see **Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core**.

Additional resources

- View or download sample code (how to download)
- GitHub source for IResponseCachingPolicyProvider
- GitHub source for IResponseCachingPolicyProvider
- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Collaborate with us on GitHub

.NET ASP.NET Core feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Rate limiting middleware in ASP.NET Core

Article • 10/03/2023

By Arvin Kahbazi, Maarten Balliauw, and Rick Anderson

The `Microsoft.AspNetCore.RateLimiting` middleware provides rate limiting middleware. Apps configure rate limiting policies and then attach the policies to endpoints. Apps using rate limiting should be carefully load tested and reviewed before deploying. See Testing endpoints with rate limiting in this article for more information.

For an introduction to rate limiting, see Rate limiting middleware.

Rate limiter algorithms

The RateLimiterOptionsExtensions class provides the following extension methods for rate limiting:

- Fixed window
- Sliding window
- Token bucket
- Concurrency

Fixed window limiter

The AddFixedWindowLimiter method uses a fixed time window to limit requests. When the time window expires, a new time window starts and the request limit is reset.

Consider the following code:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: "fixed", options =>
    {
        options.PermitLimit = 4;
        options.Window = TimeSpan.FromSeconds(12);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
```

```
        options.QueueLimit = 2;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"))
                          .RequireRateLimiting("fixed");

app.Run();
```

The preceding code:

- Calls AddRateLimiter to add a rate limiting service to the service collection.
- Calls `AddFixedWindowLimiter` to create a fixed window limiter with a policy name of `"fixed"` and sets:
- PermitLimit to 4 and the time Window to 12. A maximum of 4 requests per each 12-second window are allowed.
- QueueProcessingOrder to OldestFirst.
- QueueLimit to 2.
- Calls UseRateLimiter to enable rate limiting.

Apps should use Configuration to set limiter options. The following code updates the preceding code using MyRateLimitOptions for configuration:

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);
builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
```

```
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Fixed Window Limiter {GetTicks()}"))
                            .RequireRateLimiting(fixedPolicy);

app.Run();
```

UseRateLimiter must be called after `UseRouting` when rate limiting endpoint specific APIs are used. For example, if the [EnableRateLimiting] attribute is used, `UseRateLimiter` must be called after `UseRouting`. When calling only global limiters, `UseRateLimiter` can be called before `UseRouting`.

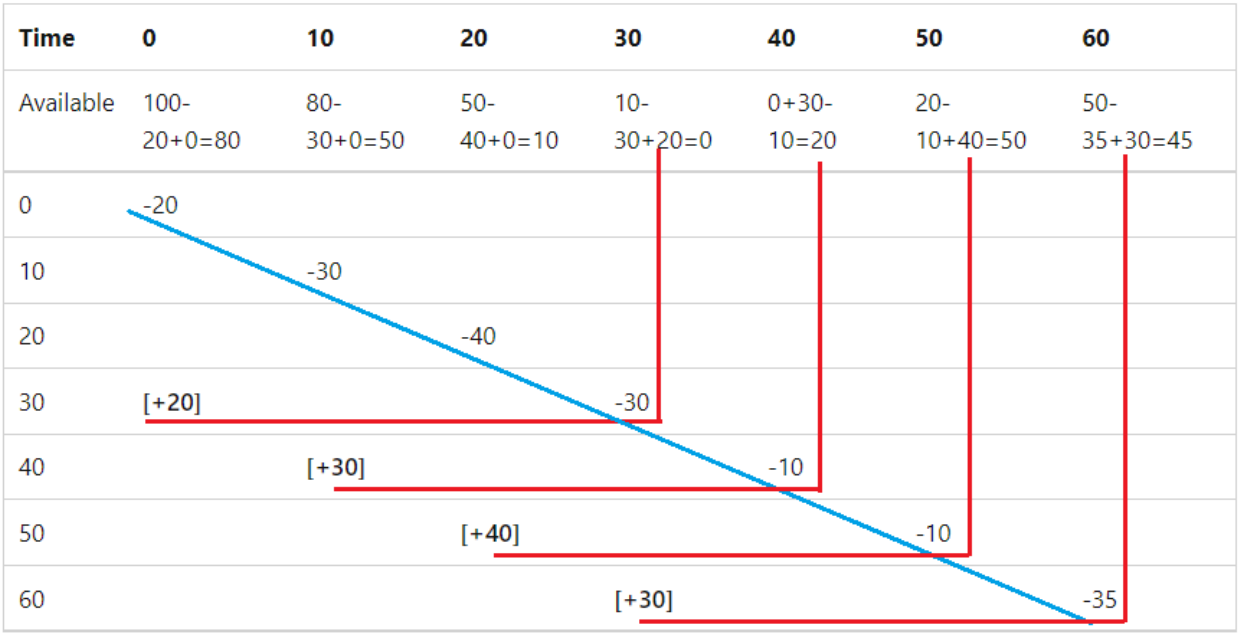
Sliding window limiter

A sliding window algorithm:

- Is similar to the fixed window limiter but adds segments per window. The window slides one segment each segment interval. The segment interval is (window time)/(segments per window).
- Limits the requests for a window to `permitLimit` requests.
- Each time window is divided in `n` segments per window.
- Requests taken from the expired time segment one window back (`n` segments prior to the current segment) are added to the current segment. We refer to the most expired time segment one window back as the expired segment.

Consider the following table that shows a sliding window limiter with a 30-second window, three segments per window, and a limit of 100 requests:

- The top row and first column shows the time segment.
- The second row shows the remaining requests available. The remaining requests are calculated as the available requests minus the processed requests plus the recycled requests.
- Requests at each time moves along the diagonal blue line.
- From time 30 on, the request taken from the expired time segment are added back to the request limit, as shown in the red lines.



The following table shows the data in the previous graph in a different format. The **Available** column shows the requests available from the previous segment (The **Carry over** from the previous row). The first row shows 100 available requests because there's no previous segment.

Expand table

Time	Available	Taken	Recycled from expired	Carry over
0	100	20	0	80
10	80	30	0	50
20	50	40	0	10
30	10	30	20	0
40	0	10	30	20
50	20	10	40	50
60	50	35	30	45

The following code uses the sliding window rate limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);
```

```
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Sliding Window Limiter {GetTicks()}"))
                          .RequireRateLimiting(slidingPolicy);

app.Run();
```

Token bucket limiter

The token bucket limiter is similar to the sliding window limiter, but rather than adding back the requests taken from the expired segment, a fixed number of tokens are added each replenishment period. The tokens added each segment can't increase the available tokens to a number higher than the token bucket limit. The following table shows a token bucket limiter with a limit of 100 tokens and a 10-second replenishment period.

Expand table

Time	Available	Taken	Added	Carry over
0	100	20	0	80
10	80	10	20	90
20	90	5	15	100
30	100	30	20	90
40	90	6	16	100
50	100	40	20	80

Time	Available	Taken	Added	Carry over
60	80	50	20	50

The following code uses the token bucket limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var tokenPolicy = "token";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddTokenBucketLimiter(policyName: tokenPolicy, options =>
    {
        options.TokenLimit = myOptions.TokenLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
        options.ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod);
        options.TokensPerPeriod = myOptions.TokensPerPeriod;
        options.AutoReplenishment = myOptions.AutoReplenishment;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Token Limiter {GetTicks()}"))
                            .RequireRateLimiting(tokenPolicy);

app.Run();
```

When AutoReplenishment is set to `true`, an internal timer replenishes the tokens every ReplenishmentPeriod; when set to `false`, the app must call TryReplenish on the limiter.

Concurrency limiter

The concurrency limiter limits the number of concurrent requests. Each request reduces the concurrency limit by one. When a request completes, the limit is increased by one.

Unlike the other requests limiters that limit the total number of requests for a specified period, the concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period.

The following code uses the concurrency limiter:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

var concurrencyPolicy = "Concurrency";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: concurrencyPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", async () =>
{
    await Task.Delay(500);
    return Results.Ok($"Concurrency Limiter {GetTicks()}");

}).RequireRateLimiting(concurrencyPolicy);

app.Run();
```

Create chained limiters

The CreateChained API allows passing in multiple PartitionedRateLimiter which are combined into one `PartitionedRateLimiter`. The combined limiter runs all the input limiters in sequence.

The following code uses `CreateChained`:

C#

```
using System.Globalization;
using System.Threading.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRateLimiter(_ =>
{
    _.OnRejected = (context, _) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.Response.WriteAsync("Too many requests. Please
try again later.");

        return new ValueTask();
    };
    _.GlobalLimiter = PartitionedRateLimiter.CreateChained(
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
            (userAgent, _ =>
                new FixedWindowRateLimiterOptions
                {
                    AutoReplenishment = true,
                    PermitLimit = 4,
                    Window = TimeSpan.FromSeconds(2)
                });
        }),
        PartitionedRateLimiter.Create<HttpContext, string>(httpContext =>
        {
            var userAgent =
httpContext.Request.Headers.UserAgent.ToString();

            return RateLimitPartition.GetFixedWindowLimiter
            (userAgent, _ =>
                new FixedWindowRateLimiterOptions
                {
                    AutoReplenishment = true,
                    PermitLimit = 20,
                    Window = TimeSpan.FromSeconds(30)
                });
```

```
        }));
});

var app = builder.Build();
app.UseRateLimiter();

static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/", () => Results.Ok($"Hello {GetTicks()}"));

app.Run();
```

For more information, see the CreateChained source code

`EnableRateLimiting` and `DisableRateLimiting` attributes

The [EnableRateLimiting] and [DisableRateLimiting] attributes can be applied to a Controller, action method, or Razor Page. For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, `[EnableRateLimiting]` can't be applied to `OnGet`, `OnPost`, or any other page handler.

The `[DisableRateLimiting]` attribute ***disables*** rate limiting to the Controller, action method, or Razor Page regardless of named rate limiters or global limiters applied. For example, consider the following code which calls RequireRateLimiting to apply the `fixedPolicy` rate limiting to all controller endpoints:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
```

```
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.SlidingPermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();
app.UseRateLimiter();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.MapRazorPages().RequireRateLimiting(slidingPolicy);
app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy);

app.Run();
```

In the following code, `[DisableRateLimiting]` disables rate limiting and overrides `[EnableRateLimiting("fixed")]` applied to the `Home2Controller` and `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy)` called in `Program.cs`:

C#

```
[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {
        _logger = logger;
```

```
    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
    }
}
```

In the preceding code, the `[EnableRateLimiting("sliding")]` is ***not*** applied to the `Privacy` action method because `Program.cs` called `app.MapDefaultControllerRoute().RequireRateLimiting(fixedPolicy)`.

Consider the following code which doesn't call `RequireRateLimiting` on `MapRazorPages` or `MapDefaultControllerRoute`:

C#

```
using Microsoft.AspNetCore.RateLimiting;
using System.Threading.RateLimiting;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
```

```
ons);
var fixedPolicy = "fixed";

builder.Services.AddRateLimiter(_ => _
    .AddFixedWindowLimiter(policyName: fixedPolicy, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var slidingPolicy = "sliding";

builder.Services.AddRateLimiter(_ => _
    .AddSlidingWindowLimiter(policyName: slidingPolicy, options =>
    {
        options.PermitLimit = myOptions.SlidingPermitLimit;
        options.Window = TimeSpan.FromSeconds(myOptions.Window);
        options.SegmentsPerWindow = myOptions.SegmentsPerWindow;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
    }));

var app = builder.Build();

app.UseRateLimiter();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.MapRazorPages();
app.MapDefaultControllerRoute();  // RequireRateLimiting not called

app.Run();
```

Consider the following controller:

C#

```
[EnableRateLimiting("fixed")]
public class Home2Controller : Controller
{
    private readonly ILogger<Home2Controller> _logger;

    public Home2Controller(ILogger<Home2Controller> logger)
    {
```

```
        _logger = logger;
    }

    public ActionResult Index()
    {
        return View();
    }

    [EnableRateLimiting("sliding")]
    public ActionResult Privacy()
    {
        return View();
    }

    [DisableRateLimiting]
    public ActionResult NoLimit()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ??
HttpContext.TraceIdentifier });
    }
}
```

In the preceding controller:

- The `"fixed"` policy rate limiter is applied to all action methods that don't have `EnableRateLimiting` and `DisableRateLimiting` attributes.
- The `"sliding"` policy rate limiter is applied to the `Privacy` action.
- Rate limiting is disabled on the `NoLimit` action method.

Applying attributes to Razor Pages

For Razor Pages, the attribute must be applied to the Razor Page and not the page handlers. For example, `[EnableRateLimiting]` can't be applied to `OnGet`, `OnPost`, or any other page handler.

The `DisableRateLimiting` attribute disables rate limiting on a Razor Page. `EnableRateLimiting` is only applied to a Razor Page if `MapRazorPages().RequireRateLimiting(Policy)` has ***not*** been called.

Limiter algorithm comparison

The fixed, sliding, and token limiters all limit the maximum number of requests in a time period. The concurrency limiter limits only the number of concurrent requests and doesn't cap the number of requests in a time period. The cost of an endpoint should be considered when selecting a limiter. The cost of an endpoint includes the resources used, for example, time, data access, CPU, and I/O.

Rate limiter samples

The following samples aren't meant for production code but are examples on how to use the limiters.

Limiter with `OnRejected`, `RetryAfter`, and `GlobalLimiter`

The following sample:

- Creates a RateLimiterOptions.OnRejected callback that is called when a request exceeds the specified limit. `retryAfter` can be used with the TokenBucketRateLimiter, FixedWindowLimiter, and SlidingWindowLimiter because these algorithms are able to estimate when more permits will be added. The `ConcurrencyLimiter` has no way of calculating when permits will be available.
- Adds the following limiters:
  - A `SampleRateLimiterPolicy` which implements the `IRateLimiterPolicy<TPartitionKey>` interface. The `SampleRateLimiterPolicy` class is shown later in this article.
  - A `SlidingWindowLimiter`:
    - With a partition for each authenticated user.
    - One shared partition for all anonymous users.
  - A GlobalLimiter that is applied to all requests. The global limiter will be executed first, followed by the endpoint-specific limiter, if one exists. The `GlobalLimiter` creates a partition for each IPAddress.

C#

```
// Preceding code removed for brevity.
using System.Globalization;
using System.Net;
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using WebRateLimitAuth;
using WebRateLimitAuth.Data;
using WebRateLimitAuth.Models;
```

```
var builder = WebApplication.CreateBuilder(args);

var connectionString =
builder.Configuration.GetConnectionString("DefaultConnection") ??
    throw new InvalidOperationException("Connection string
'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));
builder.Services.AddDatabaseDeveloperPageExceptionFilter();

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.Configure<MyRateLimitOptions>(
    builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit));

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

var userPolicyName = "user";
var helloPolicy = "hello";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.OnRejected = (context, cancellationToken) =>
    {
        if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out var
retryAfter))
        {
            context.HttpContext.Response.Headers.RetryAfter =
                ((int)
retryAfter.TotalSeconds).ToString(NumberFormatInfo.InvariantInfo);
        }

        context.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
        context.HttpContext.RequestServices.GetService<ILoggerFactory>()?
            .CreateLogger("Microsoft.AspNetCore.RateLimitingMiddleware")
            .LogWarning("OnRejected: {GetUserEndPoint}",
GetUserEndPoint(context.HttpContext));

        return new ValueTask();
    };

    limiterOptions.AddPolicy<string, SampleRateLimiterPolicy>(helloPolicy);
    limiterOptions.AddPolicy(userPolicyName, context =>
    {
        var username = "anonymous user";
        if (context.User.Identity?.IsAuthenticated is true)
        {
```

```
            username = context.User.ToString()!;
        }

        return RateLimitPartition.GetSlidingWindowLimiter(username,
            _ => new SlidingWindowRateLimiterOptions
            {
                PermitLimit = myOptions.PermitLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                Window = TimeSpan.FromSeconds(myOptions.Window),
                SegmentsPerWindow = myOptions.SegmentsPerWindow
            });

    });

    limiterOptions.GlobalLimiter =
PartitionedRateLimiter.Create<HttpContext, IPAddress>(context =>
    {
        IPAddress? remoteIpAddress = context.Connection.RemoteIpAddress;

        if (!IPAddress.IsLoopback(remoteIpAddress!))
        {
            return RateLimitPartition.GetTokenBucketLimiter
            (remoteIpAddress!, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetNoLimiter(IPAddress.Loopback);
    });
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseMigrationsEndPoint();
}
else
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
```

```
app.UseRouting();
app.UseRateLimiter();

app.UseAuthentication();
app.UseAuthorization();

app.MapRazorPages().RequireRateLimiting(userPolicyName);
app.MapDefaultControllerRoute();

static string GetUserEndPoint(HttpContext context) =>
   $"User {context.User.Identity?.Name ?? "Anonymous"} endpoint:
{context.Request.Path}"
   + $" {context.Connection.RemoteIpAddress}";
static string GetTicks() => (DateTime.Now.Ticks &
0x11111).ToString("00000");

app.MapGet("/a", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(userPolicyName);

app.MapGet("/b", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}")
    .RequireRateLimiting(helloPolicy);

app.MapGet("/c", (HttpContext context) => $"{GetUserEndPoint(context)}
{GetTicks()}");

app.Run();
```

> ⚠ **Warning**
>
> Creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks which employ IP Source Address Spoofing. For more information, see BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing.

See the samples repository for the complete Program.cs file.

The `SampleRateLimiterPolicy` class

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.RateLimiting;
using Microsoft.Extensions.Options;
using WebRateLimitAuth.Models;

namespace WebRateLimitAuth;

public class SampleRateLimiterPolicy : IRateLimiterPolicy<string>
```

```
{
    private Func<OnRejectedContext, CancellationToken, ValueTask>?
_onRejected;
    private readonly MyRateLimitOptions _options;

    public SampleRateLimiterPolicy(ILogger<SampleRateLimiterPolicy> logger,
                                   IOptions<MyRateLimitOptions> options)
    {
        _onRejected = (ctx, token) =>
        {
            ctx.HttpContext.Response.StatusCode =
StatusCodes.Status429TooManyRequests;
            logger.LogWarning($"Request rejected by
{nameof(SampleRateLimiterPolicy)}");
            return ValueTask.CompletedTask;
        };
        _options = options.Value;
    }

    public Func<OnRejectedContext, CancellationToken, ValueTask>? OnRejected
=> _onRejected;

    public RateLimitPartition<string> GetPartition(HttpContext httpContext)
    {
        return RateLimitPartition.GetSlidingWindowLimiter(string.Empty,
            _ => new SlidingWindowRateLimiterOptions
            {
                PermitLimit = _options.PermitLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = _options.QueueLimit,
                Window = TimeSpan.FromSeconds(_options.Window),
                SegmentsPerWindow = _options.SegmentsPerWindow
            });
    }
}
```

In the preceding code, OnRejected uses OnRejectedContext to set the response status to 429 Too Many Requests. The default rejected status is 503 Service Unavailable.

Limiter with authorization

The following sample uses JSON Web Tokens (JWT) and creates a partition with the JWT access token. In a production app, the JWT would typically be provided by a server acting as a Security token service (STS). For local development, the dotnet user-jwts command line tool can be used to create and manage app-specific local JWTs.

C#

```
using System.Threading.RateLimiting;
using Microsoft.AspNetCore.Authentication;
```

```
using Microsoft.Extensions.Primitives;
using WebRateLimitAuth.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthorization();
builder.Services.AddAuthentication("Bearer").AddJwtBearer();

var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);
var jwtPolicyName = "jwt";

builder.Services.AddRateLimiter(limiterOptions =>
{
    limiterOptions.RejectionStatusCode =
StatusCodes.Status429TooManyRequests;
    limiterOptions.AddPolicy(policyName: jwtPolicyName, partitioner:
httpContext =>
    {
        var accessToken =
httpContext.Features.Get<IAuthenticateResultFeature>()?

.AuthenticateResult?.Properties?.GetTokenValue("access_token")?.ToString()
                           ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(accessToken))
        {
            return RateLimitPartition.GetTokenBucketLimiter(accessToken, _
=>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    });
});
```

```
var app = builder.Build();

app.UseAuthorization();
app.UseRateLimiter();

app.MapGet("/", () => "Hello, World!");

app.MapGet("/jwt", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.MapPost("/post", (HttpContext context) => $"Hello
{GetUserEndPointMethod(context)}")
    .RequireRateLimiting(jwtPolicyName)
    .RequireAuthorization();

app.Run();

static string GetUserEndPointMethod(HttpContext context) =>
    $"Hello {context.User.Identity?.Name ?? "Anonymous"} " +
    $"Endpoint:{context.Request.Path} Method: {context.Request.Method}";
```

Limiter with `ConcurrencyLimiter`, `TokenBucketRateLimiter`, and authorization

The following sample:

- Adds a `ConcurrencyLimiter` with a policy name of `"get"` that is used on the Razor Pages.
- Adds a `TokenBucketRateLimiter` with a partition for each authorized user and a partition for all anonymous users.
- Sets RateLimiterOptions.RejectionStatusCode to 429 Too Many Requests.

C#

```
var getPolicyName = "get";
var postPolicyName = "post";
var myOptions = new MyRateLimitOptions();
builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOpti
ons);

builder.Services.AddRateLimiter(_ => _
    .AddConcurrencyLimiter(policyName: getPolicyName, options =>
    {
        options.PermitLimit = myOptions.PermitLimit;
        options.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
        options.QueueLimit = myOptions.QueueLimit;
```

```
    })
    .AddPolicy(policyName: postPolicyName, partitioner: httpContext =>
    {
        string userName = httpContext.User.Identity?.Name ?? string.Empty;

        if (!StringValues.IsNullOrEmpty(userName))
        {
            return RateLimitPartition.GetTokenBucketLimiter(userName, _ =>
                new TokenBucketRateLimiterOptions
                {
                    TokenLimit = myOptions.TokenLimit2,
                    QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                    QueueLimit = myOptions.QueueLimit,
                    ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                    TokensPerPeriod = myOptions.TokensPerPeriod,
                    AutoReplenishment = myOptions.AutoReplenishment
                });
        }

        return RateLimitPartition.GetTokenBucketLimiter("Anon", _ =>
            new TokenBucketRateLimiterOptions
            {
                TokenLimit = myOptions.TokenLimit,
                QueueProcessingOrder = QueueProcessingOrder.OldestFirst,
                QueueLimit = myOptions.QueueLimit,
                ReplenishmentPeriod =
TimeSpan.FromSeconds(myOptions.ReplenishmentPeriod),
                TokensPerPeriod = myOptions.TokensPerPeriod,
                AutoReplenishment = true
            });
    }));
```

See the samples repository for the complete Program.cs file.

Testing endpoints with rate limiting

Before deploying an app using rate limiting to production, stress test the app to validate the rate limiters and options used. For example, create a JMeter script with a tool like BlazeMeter or Apache JMeter HTTP(S) Test Script Recorder and load the script to Azure Load Testing.

Creating partitions with user input makes the app vulnerable to Denial of Service (DoS) Attacks. For example, creating partitions on client IP addresses makes the app vulnerable to Denial of Service Attacks that employ IP Source Address Spoofing. For more information, see BCP 38 RFC 2827 Network Ingress Filtering: Defeating Denial of Service Attacks that employ IP Source Address Spoofing.

Additional resources

- Rate limiting middleware by Maarten Balliauw provides an excellent introduction and overview to rate limiting.
- Rate limit an HTTP handler in .NET

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Request timeouts middleware in ASP.NET Core

Article • 09/26/2023

By Tom Dykstra

Apps can apply timeout limits selectively to requests. ASP.NET Core servers don't do this by default since request processing times vary widely by scenario. For example, WebSockets, static files, and calling expensive APIs would each require a different timeout limit. So ASP.NET Core provides middleware that configures timeouts per endpoint as well as a global timeout.

When a timeout limit is hit, a CancellationToken in HttpContext.RequestAborted has IsCancellationRequested set to `true`. Abort() isn't automatically called on the request, so the application may still produce a success or failure response. The default behavior if the app doesn't handle the exception and produce a response is to return status code 504.

This article explains how to configure the timeout middleware. The timeout middleware can be used in all types of ASP.NET Core apps: Minimal API, Web API with controllers, MVC, and Razor Pages. The sample app is a Minimal API, but every timeout feature it illustrates is also supported in the other app types.

Request timeouts are in the Microsoft.AspNetCore.Http.Timeouts namespace.

Note: When an app is running in debug mode, the timeout middleware doesn't trigger. This behavior is the same as for Kestrel timeouts. To test timeouts, run the app without the debugger attached.

Add the middleware to the app

Add the request timeouts middleware to the service collection by calling AddRequestTimeouts.

Add the middleware to the request processing pipeline by calling UseRequestTimeouts.

> **Note**
>
> - In apps that explicitly call UseRouting, `UseRequestTimeouts` must be called after `UseRouting`.

Adding the middleware to the app doesn't automatically start triggering timeouts. Timeout limits have to be explicitly configured.

Configure one endpoint or page

For minimal API apps, configure an endpoint to timeout by calling WithRequestTimeout, or by applying the [RequestTimeout] attribute, as shown in the following example:

C#

```
using Microsoft.AspNetCore.Http.Timeouts;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRequestTimeouts();

var app = builder.Build();
app.UseRequestTimeouts();

app.MapGet("/", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
}).WithRequestTimeout(TimeSpan.FromSeconds(2));
// Returns "Timeout!"

app.MapGet("/attribute",
    [RequestTimeout(milliseconds: 2000)] async (HttpContext context) => {
        try
        {
            await Task.Delay(TimeSpan.FromSeconds(10),
context.RequestAborted);
        }
        catch (TaskCanceledException)
        {
            return Results.Content("Timeout!", "text/plain");
        }

        return Results.Content("No timeout!", "text/plain");
    });
// Returns "Timeout!"

app.Run();
```

For apps with controllers, apply the `[RequestTimeout]` attribute to the action method or the controller class. For Razor Pages apps, apply the attribute to the Razor page class.

Configure multiple endpoints or pages

Create named *policies* to specify timeout configuration that applies to multiple endpoints. Add a policy by calling AddPolicy:

```
C#

builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy =
        new RequestTimeoutPolicy { Timeout = TimeSpan.FromMilliseconds(1500)
};
    options.AddPolicy("MyPolicy", TimeSpan.FromSeconds(2));
});
```

A timeout can be specified for an endpoint by policy name:

```
C#

app.MapGet("/namedpolicy", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
}).WithRequestTimeout("MyPolicy");
// Returns "Timeout!"
```

The `[RequestTimeout]` attribute can also be used to specify a named policy.

Set global default timeout policy

Specify a policy for the global default timeout configuration:

```
C#

builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy =
        new RequestTimeoutPolicy { Timeout = TimeSpan.FromMilliseconds(1500)
};
```

```
    options.AddPolicy("MyPolicy", TimeSpan.FromSeconds(2));
});
```

The default timeout applies to endpoints that don't have a timeout specified. The following endpoint code checks for a timeout although it doesn't call the extension method or apply the attribute. The global timeout configuration applies, so the code checks for a timeout:

C#

```
app.MapGet("/", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
});
// Returns "Timeout!" due to default policy.
```

Specify the status code in a policy

The RequestTimeoutPolicy class has a property that can automatically set the status code when a timeout is triggered.

C#

```
builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy = new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        TimeoutStatusCode = 503
    };
    options.AddPolicy("MyPolicy2", new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        WriteTimeoutResponse = async (HttpContext context) => {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("Timeout from MyPolicy2!");
        }
    });
});
```

C#

```
app.MapGet("/", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        throw;
    }

    return Results.Content("No timeout!", "text/plain");
});
// Returns status code 503 due to default policy.
```

Use a delegate in a policy

The `RequestTimeoutPolicy` class has a WriteTimeoutResponse property that can be used to customize the response when a timeout is triggered.

C#

```
builder.Services.AddRequestTimeouts(options => {
    options.DefaultPolicy = new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        TimeoutStatusCode = 503
    };
    options.AddPolicy("MyPolicy2", new RequestTimeoutPolicy {
        Timeout = TimeSpan.FromMilliseconds(1000),
        WriteTimeoutResponse = async (HttpContext context) => {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("Timeout from MyPolicy2!");
        }
    });
});
```

C#

```
app.MapGet("/usepolicy2", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        throw;
    }

    return Results.Content("No timeout!", "text/plain");
}).WithRequestTimeout("MyPolicy2");
```

```
// Returns "Timeout from MyPolicy2!" due to WriteTimeoutResponse in
MyPolicy2.
```

Disable timeouts

To disable all timeouts including the default global timeout, use the [DisableRequestTimeout] attribute or the DisableRequestTimeout extension method:

C#

```
app.MapGet("/disablebyattr", [DisableRequestTimeout] async (HttpContext
context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
});
// Returns "No timeout!", ignores default timeout.
```

C#

```
app.MapGet("/disablebyext", async (HttpContext context) => {
    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
}).DisableRequestTimeout();
// Returns "No timeout!", ignores default timeout.
```

Cancel a timeout

To cancel a timeout that has already been started, use the DisableTimeout() method on IHttpRequestTimeoutFeature. Timeouts cannot be canceled after they've expired.

C#

```
app.MapGet("/canceltimeout", async (HttpContext context) => {
    var timeoutFeature = context.Features.Get<IHttpRequestTimeoutFeature>();
    timeoutFeature?.DisableTimeout();

    try
    {
        await Task.Delay(TimeSpan.FromSeconds(10), context.RequestAborted);
    }
    catch (TaskCanceledException)
    {
        return Results.Content("Timeout!", "text/plain");
    }

    return Results.Content("No timeout!", "text/plain");
}).WithRequestTimeout(TimeSpan.FromSeconds(1));
// Returns "No timeout!" since the default timeout is not triggered.
```

See also

- Microsoft.AspNetCore.Http.Timeouts
- ASP.NET Core Middleware

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Memory management and garbage collection (GC) in ASP.NET Core

Article • 06/17/2024

By Sébastien Ros and Rick Anderson

Memory management is complex, even in a managed framework like .NET. Analyzing and understanding memory issues can be challenging. This article:

- Was motivated by many *memory leak* and *GC not working* issues. Most of these issues were caused by not understanding how memory consumption works in .NET Core, or not understanding how it's measured.
- Demonstrates problematic memory use, and suggests alternative approaches.

How garbage collection (GC) works in .NET Core

The GC allocates heap segments where each segment is a contiguous range of memory. Objects placed in the heap are categorized into one of 3 generations: 0, 1, or 2. The generation determines the frequency the GC attempts to release memory on managed objects that are no longer referenced by the app. Lower numbered generations are GC'd more frequently.

Objects are moved from one generation to another based on their lifetime. As objects live longer, they are moved into a higher generation. As mentioned previously, higher generations are GC'd less often. Short term lived objects always remain in generation 0. For example, objects that are referenced during the life of a web request are short lived. Application level singletons generally migrate to generation 2.

When an ASP.NET Core app starts, the GC:

- Reserves some memory for the initial heap segments.
- Commits a small portion of memory when the runtime is loaded.

The preceding memory allocations are done for performance reasons. The performance benefit comes from heap segments in contiguous memory.

GC.Collect caveats

In general, ASP.NET Core apps in production should **not** use GC.Collect explicitly. Inducing garbage collections at sub-optimal times can decrease performance significantly.

GC.Collect is useful when investigating memory leaks. Calling `GC.Collect()` triggers a blocking garbage collection cycle that tries to reclaim all objects inaccessible from managed code. It's a useful way to understand the size of the reachable live objects in the heap, and track growth of memory size over time.

Analyzing the memory usage of an app

Dedicated tools can help analyzing memory usage:

- Counting object references
- Measuring how much impact the GC has on CPU usage
- Measuring memory space used for each generation

Use the following tools to analyze memory usage:

- dotnet-trace: Can be used on production machines.
- Analyze memory usage without the Visual Studio debugger
- Profile memory usage in Visual Studio

Detecting memory issues

Task Manager can be used to get an idea of how much memory an ASP.NET app is using. The Task Manager memory value:

- Represents the amount of memory that is used by the ASP.NET process.
- Includes the app's living objects and other memory consumers such as native memory usage.

If the Task Manager memory value increases indefinitely and never flattens out, the app has a memory leak. The following sections demonstrate and explain several memory usage patterns.

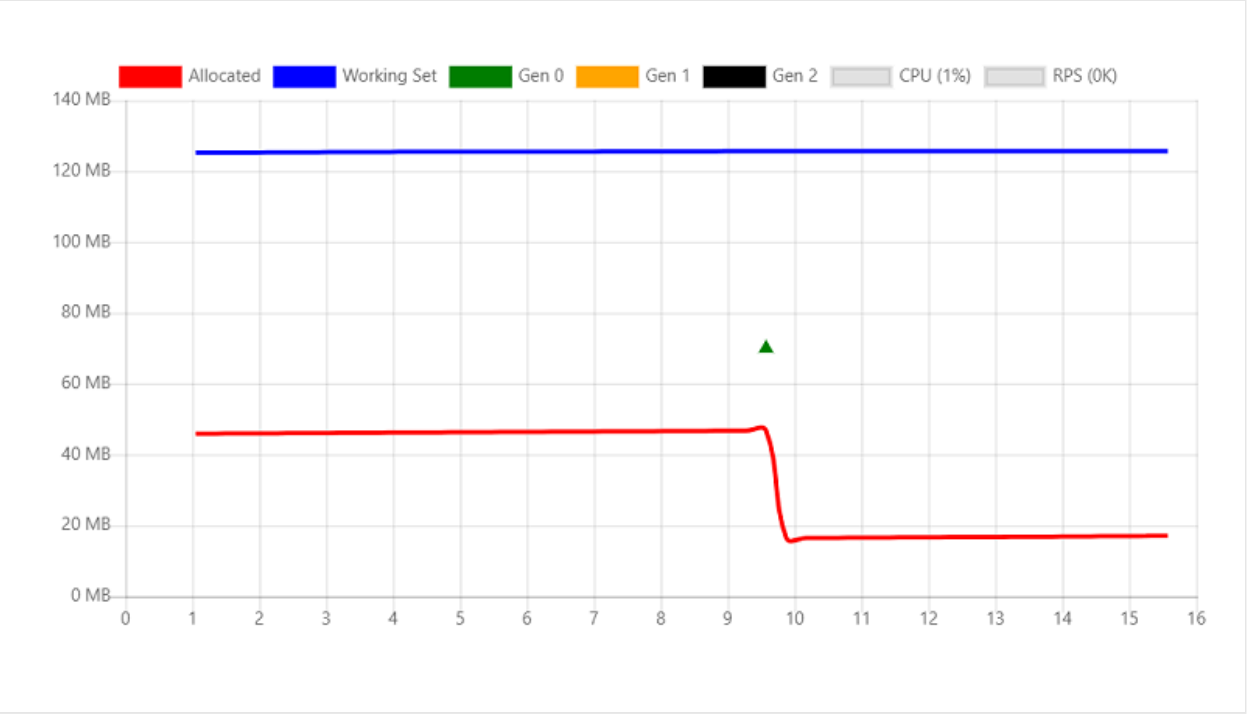
Sample display memory usage app

The MemoryLeak sample app is available on GitHub. The MemoryLeak app:

- Includes a diagnostic controller that gathers real-time memory and GC data for the app.

- Has an Index page that displays the memory and GC data. The Index page is refreshed every second.
- Contains an API controller that provides various memory load patterns.
- Is not a supported tool, however, it can be used to display memory usage patterns of ASP.NET Core apps.

Run MemoryLeak. Allocated memory slowly increases until a GC occurs. Memory increases because the tool allocates custom object to capture data. The following image shows the MemoryLeak Index page when a Gen 0 GC occurs. The chart shows 0 RPS (Requests per second) because no API endpoints from the API controller have been called.



The chart displays two values for the memory usage:

- Allocated: the amount of memory occupied by managed objects
- Working set: The set of pages in the virtual address space of the process that are currently resident in physical memory. The working set shown is the same value Task Manager displays.

Transient objects

The following API creates a 20-KB String instance and returns it to the client. On each request, a new object is allocated in memory and written to the response. Strings are stored as UTF-16 characters in .NET so each character takes 2 bytes in memory.

```
C#
```

```
[HttpGet("bigstring")]
public ActionResult<string> GetBigString()
{
    return new String('x', 10 * 1024);
}
```

The following graph is generated with a relatively small load in to show how memory allocations are impacted by the GC.



The preceding chart shows:

- 4K RPS (Requests per second).
- Generation 0 GC collections occur about every two seconds.
- The working set is constant at approximately 500 MB.
- CPU is 12%.
- The memory consumption and release (through GC) is stable.

The following chart is taken at the max throughput that can be handled by the machine.



The preceding chart shows:

- 22K RPS
- Generation 0 GC collections occur several times per second.
- Generation 1 collections are triggered because the app allocated significantly more memory per second.
- The working set is constant at approximately 500 MB.
- CPU is 33%.
- The memory consumption and release (through GC) is stable.
- The CPU (33%) is not over-utilized, therefore the garbage collection can keep up with a high number of allocations.

Workstation GC vs. Server GC

The .NET Garbage Collector has two different modes:

- **Workstation GC**: Optimized for the desktop.
- **Server GC**. The default GC for ASP.NET Core apps. Optimized for the server.

The GC mode can be set explicitly in the project file or in the `runtimeconfig.json` file of the published app. The following markup shows setting `ServerGarbageCollection` in the project file:

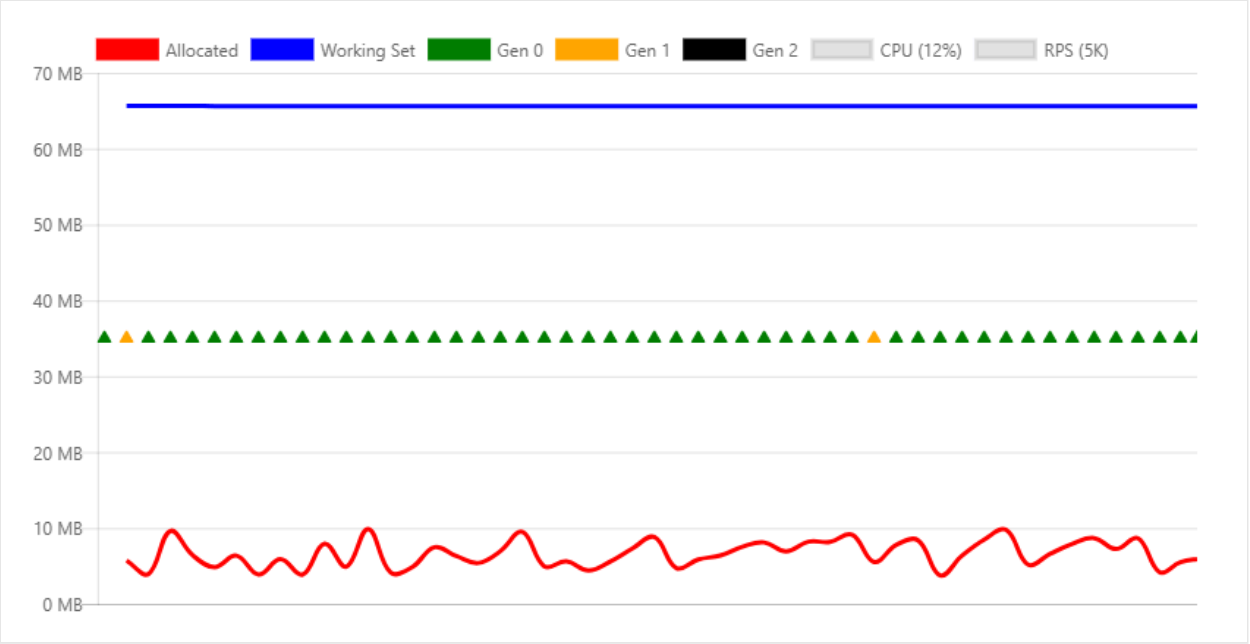
XML

```xml
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Changing `ServerGarbageCollection` in the project file requires the app to be rebuilt.

Note: Server garbage collection is **not** available on machines with a single core. For more information, see IsServerGC.

The following image shows the memory profile under a 5K RPS using the Workstation GC.



The differences between this chart and the server version are significant:

- The working set drops from 500 MB to 70 MB.
- The GC does generation 0 collections multiple times per second instead of every two seconds.
- GC drops from 300 MB to 10 MB.

On a typical web server environment, CPU usage is more important than memory, therefore the Server GC is better. If memory utilization is high and CPU usage is relatively low, the Workstation GC might be more performant. For example, high density hosting several web apps where memory is scarce.

GC using Docker and small containers

When multiple containerized apps are running on one machine, Workstation GC might be more performant than Server GC. For more information, see Running with Server GC in a Small Container and Running with Server GC in a Small Container Scenario Part 1 – Hard Limit for the GC Heap.

Persistent object references

The GC cannot free objects that are referenced. Objects that are referenced but no longer needed result in a memory leak. If the app frequently allocates objects and fails to free them after they are no longer needed, memory usage will increase over time.

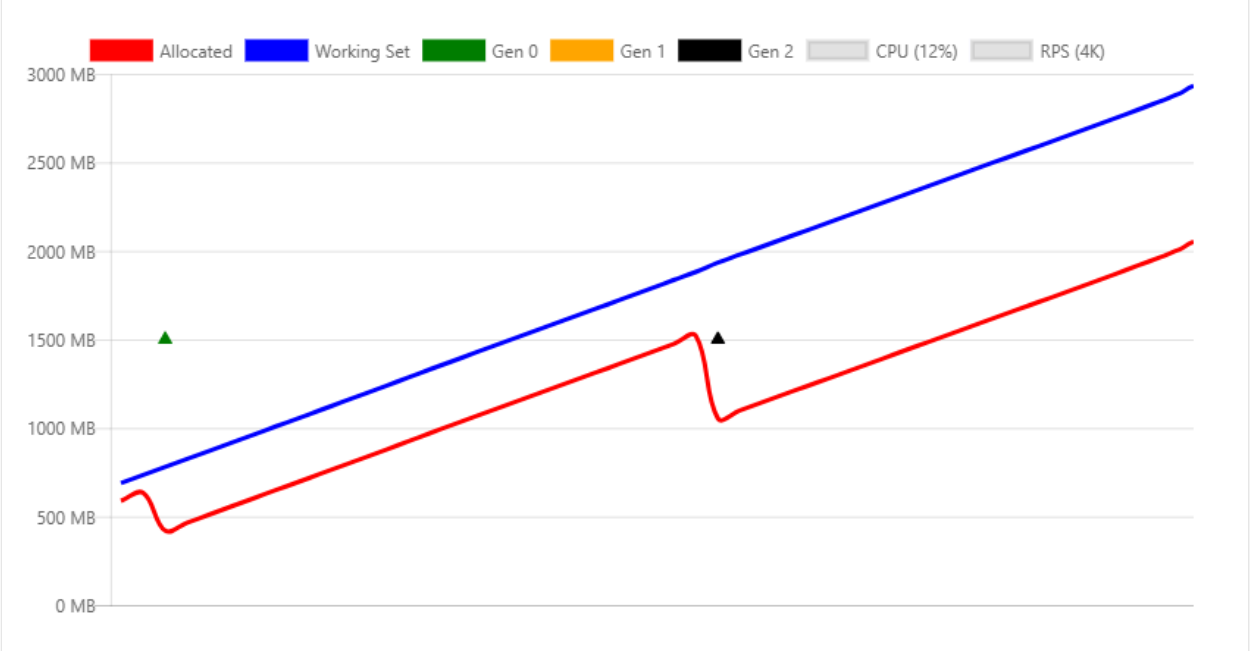
The following API creates a 20-KB String instance and returns it to the client. The difference with the previous example is that this instance is referenced by a static member, which means it's never available for collection.

C#

```
private static ConcurrentBag<string> _staticStrings = new
ConcurrentBag<string>();

[HttpGet("staticstring")]
public ActionResult<string> GetStaticString()
{
    var bigString = new String('x', 10 * 1024);
    _staticStrings.Add(bigString);
    return bigString;
}
```

The preceding code:

- Is an example of a typical memory leak.
- With frequent calls, causes app memory to increase until the process crashes with an `OutOfMemory` exception.



In the preceding image:

- Load testing the `/api/staticstring` endpoint causes a linear increase in memory.

- The GC tries to free memory as the memory pressure grows, by calling a generation 2 collection.
- The GC cannot free the leaked memory. Allocated and working set increase with time.

Some scenarios, such as caching, require object references to be held until memory pressure forces them to be released. The WeakReference class can be used for this type of caching code. A `WeakReference` object is collected under memory pressures. The default implementation of IMemoryCache uses `WeakReference`.

Native memory

Some .NET Core objects rely on native memory. Native memory can **not** be collected by the GC. The .NET object using native memory must free it using native code.

.NET provides the IDisposable interface to let developers release native memory. Even if Dispose is not called, correctly implemented classes call `Dispose` when the finalizer runs.

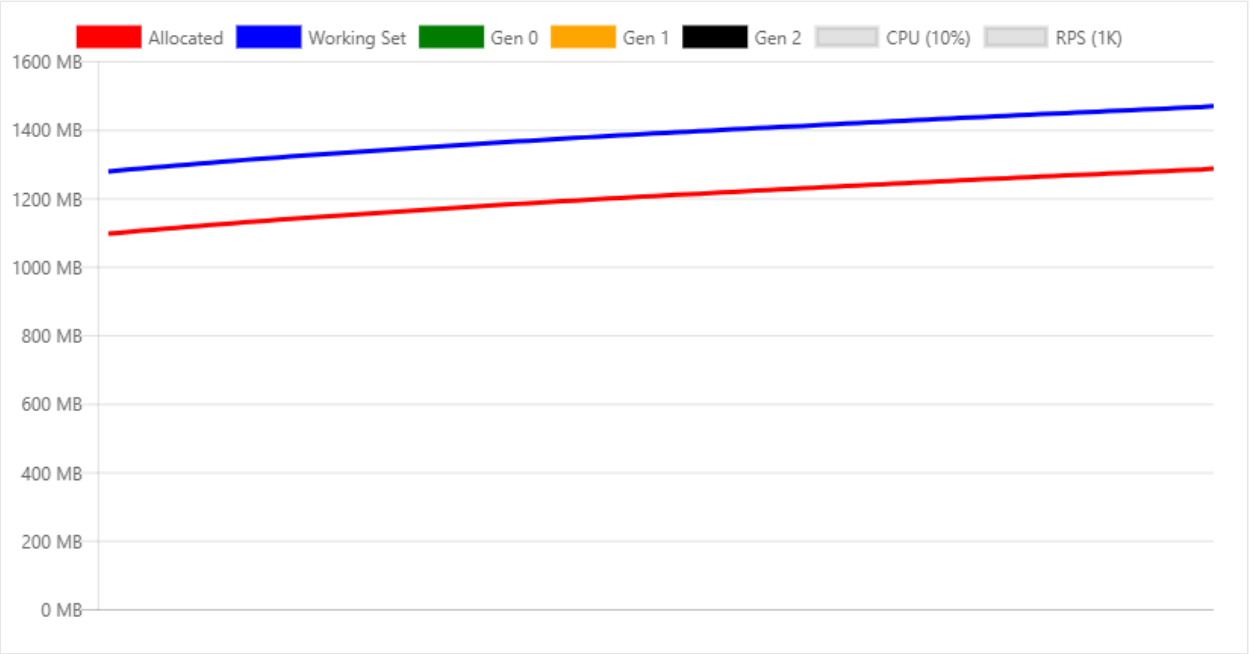
Consider the following code:

C#

```
[HttpGet("fileprovider")]
public void GetFileProvider()
{
    var fp = new PhysicalFileProvider(TempPath);
    fp.Watch("*.*");
}
```

PhysicalFileProvider is a managed class, so any instance will be collected at the end of the request.

The following image shows the memory profile while invoking the `fileprovider` API continuously.



The preceding chart shows an obvious issue with the implementation of this class, as it keeps increasing memory usage. This is a known problem that is being tracked in this issue.

The same leak could happen in user code, by one of the following:

- Not releasing the class correctly.
- Forgetting to invoke the `Dispose` method of the dependent objects that should be disposed.

Large object heap

Frequent memory allocation/free cycles can fragment memory, especially when allocating large chunks of memory. Objects are allocated in contiguous blocks of memory. To mitigate fragmentation, when the GC frees memory, it tries to defragment it. This process is called **compaction**. Compaction involves moving objects. Moving large objects imposes a performance penalty. For this reason the GC creates a special memory zone for *large* objects, called the large object heap (LOH). Objects that are greater than 85,000 bytes (approximately 83 KB) are:

- Placed on the LOH.
- Not compacted.
- Collected during generation 2 GCs.

When the LOH is full, the GC will trigger a generation 2 collection. Generation 2 collections:

- Are inherently slow.
- Additionally incur the cost of triggering a collection on all other generations.

The following code compacts the LOH immediately:

```
C#

GCSettings.LargeObjectHeapCompactionMode =
GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

See LargeObjectHeapCompactionMode for information on compacting the LOH.

In containers using .NET Core 3.0 and later, the LOH is automatically compacted.

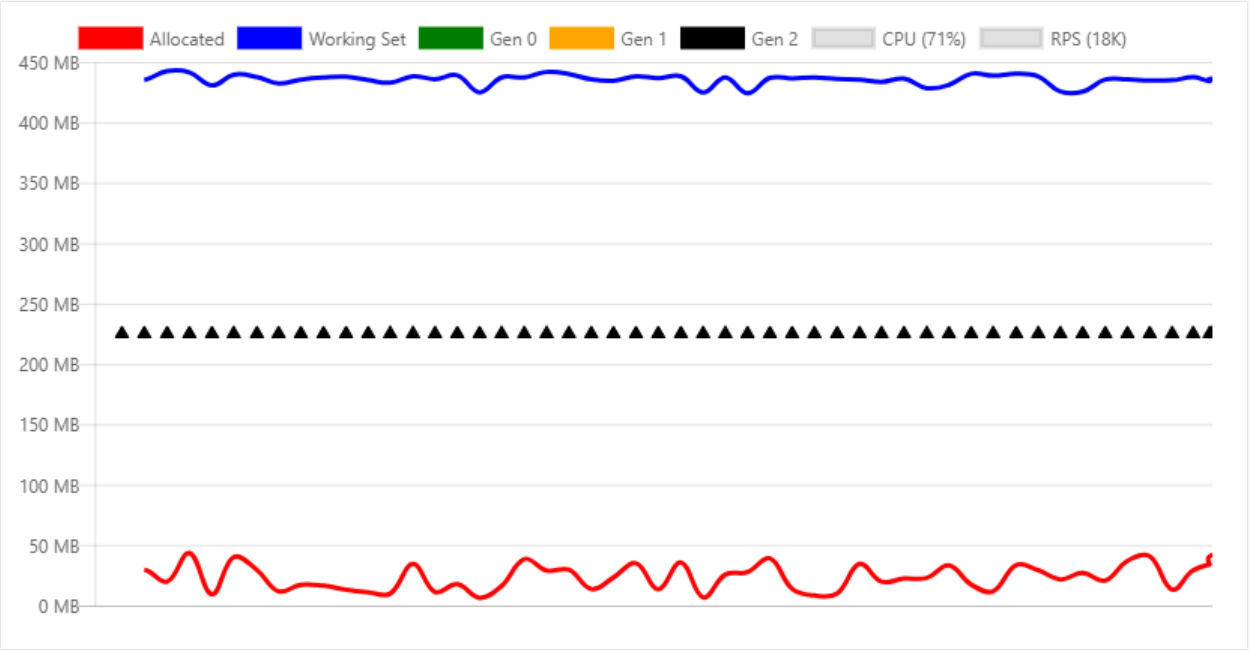
The following API that illustrates this behavior:

```
C#

[HttpGet("loh/{size=85000}")]
public int GetLOH1(int size)
{
   return new byte[size].Length;
}
```

The following chart shows the memory profile of calling the `/api/loh/84975` endpoint, under maximum load:



The following chart shows the memory profile of calling the `/api/loh/84976` endpoint, allocating *just one more byte*:



Note: The `byte[]` structure has overhead bytes. That's why 84,976 bytes triggers the 85,000 limit.

Comparing the two preceding charts:

- The working set is similar for both scenarios, about 450 MB.
- The under LOH requests (84,975 bytes) shows mostly generation 0 collections.
- The over LOH requests generate constant generation 2 collections. Generation 2 collections are expensive. More CPU is required and throughput drops almost 50%.

Temporary large objects are particularly problematic because they cause gen2 GCs.

For maximum performance, large object use should be minimized. If possible, split up large objects. For example, Response Caching middleware in ASP.NET Core split the cache entries into blocks less than 85,000 bytes.

The following links show the ASP.NET Core approach to keeping objects under the LOH limit:

- ResponseCaching/Streams/StreamUtilities.cs
- ResponseCaching/MemoryResponseCache.cs

For more information, see:

- Large Object Heap Uncovered
- Large object heap

HttpClient

Incorrectly using HttpClient can result in a resource leak. System resources, such as database connections, sockets, file handles, etc.:

- Are more scarce than memory.
- Are more problematic when leaked than memory.

Experienced .NET developers know to call Dispose on objects that implement IDisposable. Not disposing objects that implement `IDisposable` typically results in leaked memory or leaked system resources.

`HttpClient` implements `IDisposable`, but should **not** be disposed on every invocation. Rather, `HttpClient` should be reused.

The following endpoint creates and disposes a new `HttpClient` instance on every request:

C#

```
[HttpGet("httpclient1")]
public async Task<int> GetHttpClient1(string url)
{
    using (var httpClient = new HttpClient())
    {
        var result = await httpClient.GetAsync(url);
        return (int)result.StatusCode;
    }
}
```

Under load, the following error messages are logged:

```
fail: Microsoft.AspNetCore.Server.Kestrel[13]
      Connection id "0HLG70PBE1CR1", Request id "0HLG70PBE1CR1:00000031":
      An unhandled exception was thrown by the application.
System.Net.Http.HttpRequestException: Only one usage of each socket address
    (protocol/network address/port) is normally permitted --->
    System.Net.Sockets.SocketException: Only one usage of each socket
address
    (protocol/network address/port) is normally permitted
   at System.Net.Http.ConnectHelper.ConnectAsync(String host, Int32 port,
    CancellationToken cancellationToken)
```

Even though the `HttpClient` instances are disposed, the actual network connection takes some time to be released by the operating system. By continuously creating new connections, *ports exhaustion* occurs. Each client connection requires its own client port.

One way to prevent port exhaustion is to reuse the same `HttpClient` instance:

```
C#
```

```
private static readonly HttpClient _httpClient = new HttpClient();

[HttpGet("httpclient2")]
public async Task<int> GetHttpClient2(string url)
{
    var result = await _httpClient.GetAsync(url);
    return (int)result.StatusCode;
}
```

The `HttpClient` instance is released when the app stops. This example shows that not every disposable resource should be disposed after each use.

See the following for a better way to handle the lifetime of an `HttpClient` instance:

- HttpClient and lifetime management
- HTTPClient factory blog

Object pooling

The previous example showed how the `HttpClient` instance can be made static and reused by all requests. Reuse prevents running out of resources.

Object pooling:

- Uses the reuse pattern.
- Is designed for objects that are expensive to create.

A pool is a collection of pre-initialized objects that can be reserved and released across threads. Pools can define allocation rules such as limits, predefined sizes, or growth rate.

The NuGet package Microsoft.Extensions.ObjectPool contains classes that help to manage such pools.

The following API endpoint instantiates a `byte` buffer that is filled with random numbers on each request:

```
C#
```

```
        [HttpGet("array/{size}")]
        public byte[] GetArray(int size)
        {
            var random = new Random();
            var array = new byte[size];
```

```
            random.NextBytes(array);

            return array;
        }
```

The following chart display calling the preceding API with moderate load:



In the preceding chart, generation 0 collections happen approximately once per second.

The preceding code can be optimized by pooling the `byte` buffer by using ArrayPool<T>. A static instance is reused across requests.

What's different with this approach is that a pooled object is returned from the API. That means:

- The object is out of your control as soon as you return from the method.
- You can't release the object.

To set up disposal of the object:

- Encapsulate the pooled array in a disposable object.
- Register the pooled object with HttpContext.Response.RegisterForDispose.

`RegisterForDispose` will take care of calling `Dispose` on the target object so that it's only released when the HTTP request is complete.

C#

```
private static ArrayPool<byte> _arrayPool = ArrayPool<byte>.Create();

private class PooledArray : IDisposable
{
```

```
    public byte[] Array { get; private set; }

    public PooledArray(int size)
    {
        Array = _arrayPool.Rent(size);
    }

    public void Dispose()
    {
        _arrayPool.Return(Array);
    }
}

[HttpGet("pooledarray/{size}")]
public byte[] GetPooledArray(int size)
{
    var pooledArray = new PooledArray(size);

    var random = new Random();
    random.NextBytes(pooledArray.Array);

    HttpContext.Response.RegisterForDispose(pooledArray);

    return pooledArray.Array;
}
```

Applying the same load as the non-pooled version results in the following chart:



The main difference is allocated bytes, and as a consequence much fewer generation 0 collections.

Additional resources

- Garbage Collection
- Understanding different GC modes with Concurrency Visualizer
- Large Object Heap Uncovered
- Large object heap

Deploying and scaling an ASP.NET Core app on Azure Container Apps

Article • 04/13/2023

Apps deployed to Azure that experience intermittent high demand benefit from scalability to meet demand. Scalable apps can scale out to ensure capacity during workload peaks and then scale down automatically when the peak drops, which can lower costs. Horizontal scaling (scaling out) adds new instances of a resource, such as VMs or database replicas. This article demonstrates how to deploy a horizontally scalable ASP.NET Core app to Azure container apps by completing the following tasks:

1. Set up the sample project
2. Deploy the app to Azure Container Apps
3. Scale and troubleshoot the app
4. Create the Azure Services
5. Connect the Azure services
6. Configure and redeploy the app

This article uses Razor Pages, but most of it applies to other ASP.NET Core apps.

In some cases, basic ASP.NET Core apps are able to scale without special considerations. However, apps that utilize certain framework features or architectural patterns require extra configurations, including the following:

- **Secure form submissions**: Razor Pages, MVC and Web API apps often rely on form submissions. By default these apps use cross site forgery tokens and internal data protection services to secure requests. When deployed to the cloud, these apps must be configured to manage data protection service concerns in a secure, centralized location.

- **SignalR circuits**: Blazor Server apps require the use of a centralized Azure SignalR service in order to securely scale. These services also utilize the data protection services mentioned previously.

- **Centralized caching or state management services**: Scalable apps may use Azure Cache for Redis to provide distributed caching. Azure storage may be needed to store state for frameworks such as Microsoft Orleans, which can help write apps that manage state across many different app instances.

The steps in this article demonstrate how to properly address the preceding concerns by deploying a scalable app to Azure Container Apps. Most of the concepts in this tutorial

also apply when scaling Azure App Service instances.

Set up the sample project

Use the GitHub Explorer sample app to follow along with this tutorial. Clone the app from GitHub using the following command:

.NET CLI

```
git clone "https://github.com/dotnet/AspNetCore.Docs.Samples.git"
```

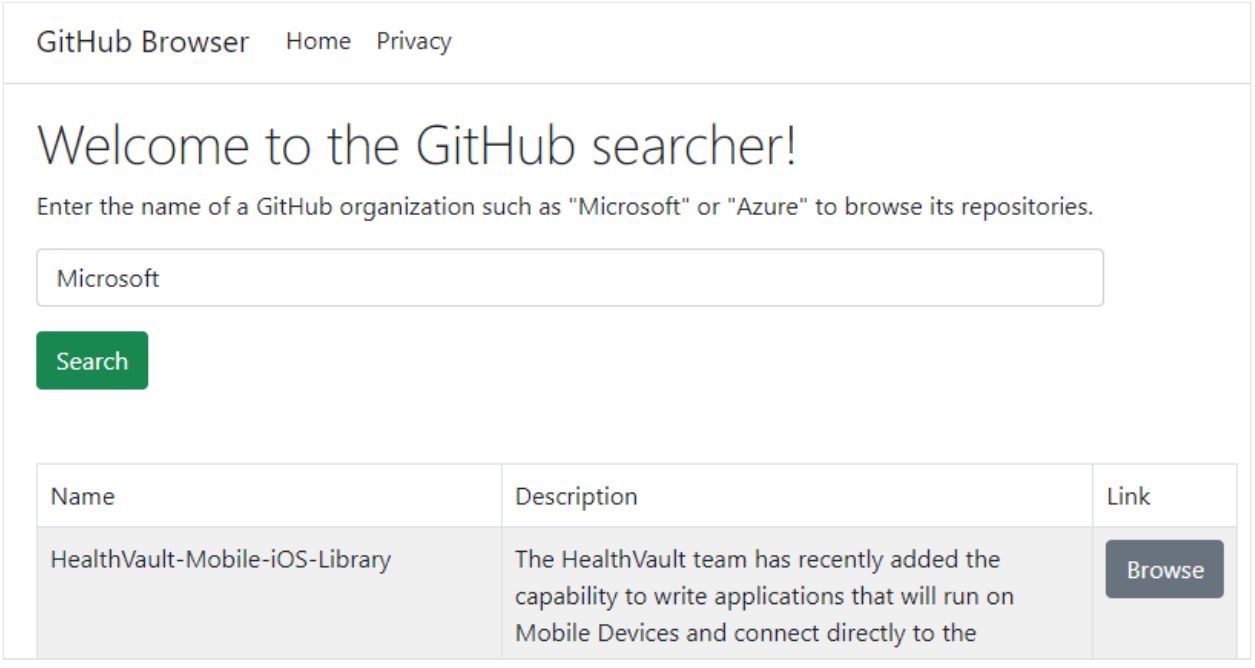
Navigate to the `/tutorials/scalable-razor-apps/start` folder and open the `ScalableRazor.csproj`.

The sample app uses a search form to browse GitHub repositories by name. The form relies on the built-in ASP.NET Core data protection services to handle antiforgery concerns. By default, when the app scales horizontally on Container Apps, the data protection service throws an exception.

Test the app

1. Launch the app in Visual Studio. The project includes a Docker file, which means that the arrow next to the run button can be selected to start the app using either a Docker Desktop setup or the standard ASP.NET Core local web server.

Use the search form to browse for GitHub repositories by name.

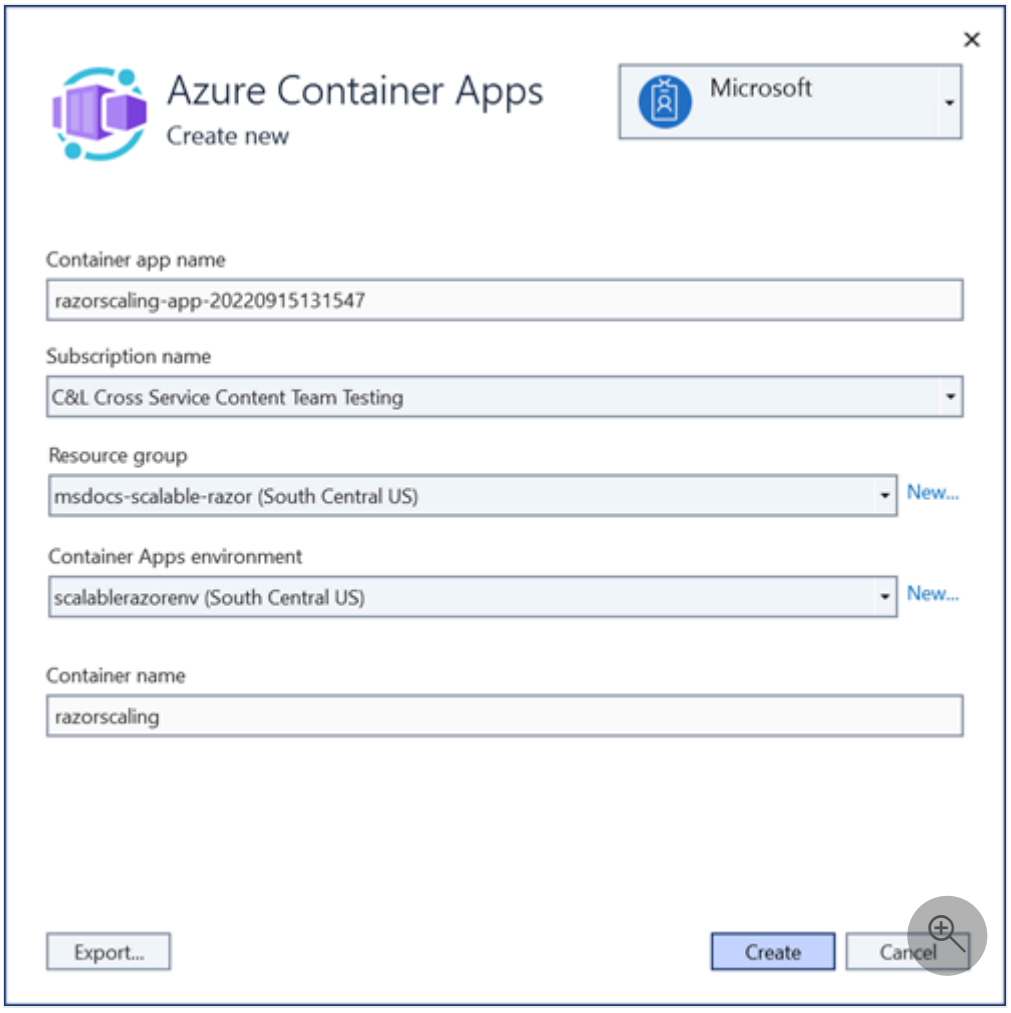


Deploy the app to Azure Container Apps

Visual Studio is used to deploy the app to Azure Container Apps. Container apps provide a managed service designed to simplify hosting containerized apps and microservices.

> **Note**
>
> Many of the resources created for the app require a location. For this app, location isn't important. A real app should select a location closest to the clients. You may want to select a location near you.

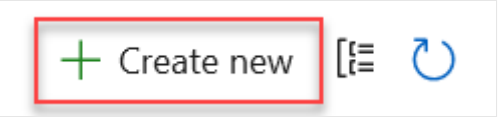
1. In Visual Studio solution explorer, right click on the top level project node and select **Publish**.
2. In the publishing dialog, select **Azure** as the deployment target, and then select **Next**.
3. For the specific target, select **Azure Container Apps (Linux)**, and then select **Next**.
4. Create a new container app to deploy to. Select the green + icon to open a new dialog and enter the following values:



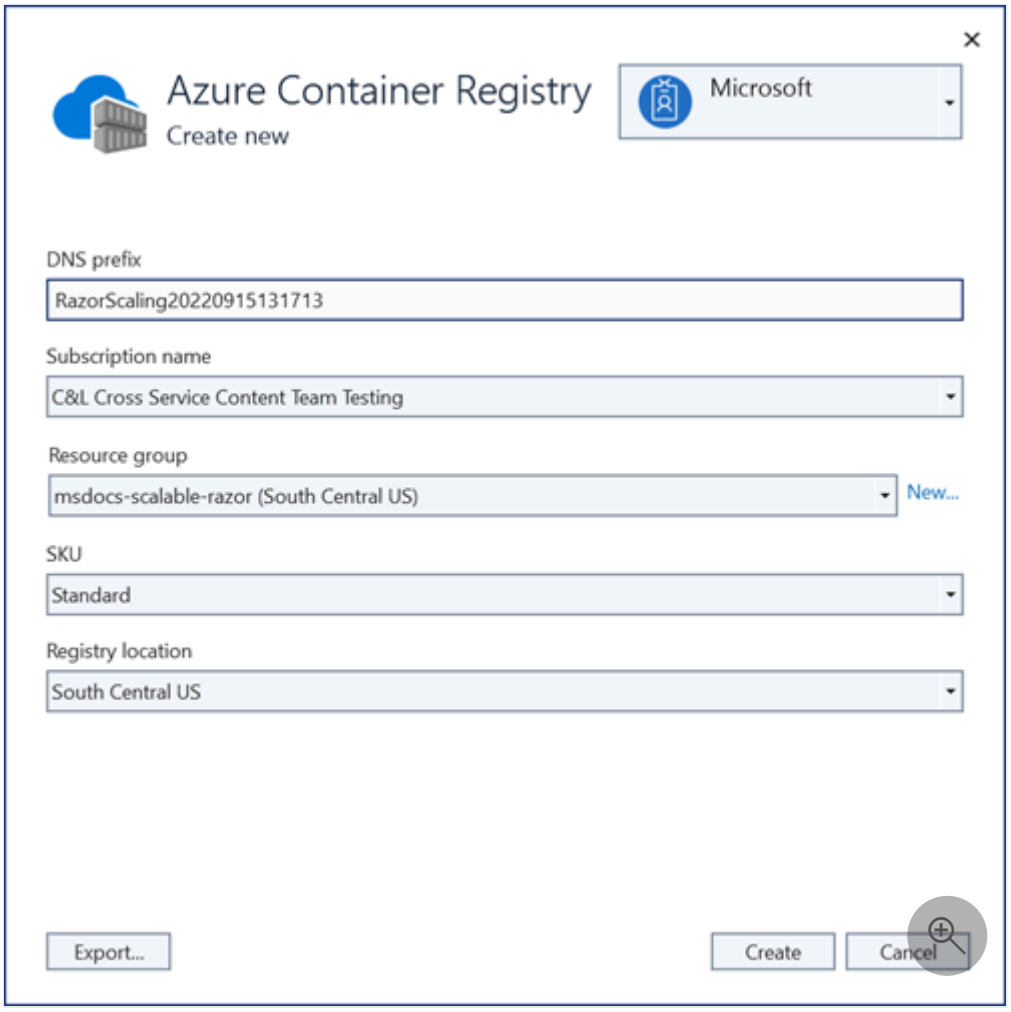
- **Container app name**: Leave the default value or enter a name.
- **Subscription name**: Select the subscription to deploy to.
- **Resource group**: Select **New** and create a new resource group called *msdocs-scalable-razor*.
- **Container apps environment**: Select **New** to open the container apps environment dialog and enter the following values:
  - **Environment name**: Keep the default value.
  - **Location**: Select a location near you.
  - **Azure Log Analytics Workspace**: Select **New** to open the log analytics workspace dialog.
    - **Name**: Leave the default value.
    - **Location**: Select a location near you and then select **Ok** to close the dialog.
  - Select **Ok** to close the container apps environment dialog.
- Select **Create** to close the original container apps dialog. Visual Studio creates the container app resource in Azure.

5. Once the resource is created, make sure it's selected in the list of container apps, and then select **Next**.

6. You'll need to create an Azure Container Registry to store the published image artifact for your app. Select the green + icon on the container registry screen.



7. Leave the default values, and then select **Create**.



8. After the container registry is created, make sure it's selected, and then select finish to close the dialog workflow and display a summary of the publishing profile.

   If Visual Studio prompts you to enable the Admin user to access the published docker container, select **Yes**.

9. Select **Publish** in the upper right of the publishing profile summary to deploy the app to Azure.

When the deployment finishes, Visual Studio launches the browser to display the hosted app. Search for `Microsoft` in the form field, and a list of repositories is displayed.

Scale and troubleshoot the app

The app is currently working without any issues, but we'd like to scale the app across more instances in anticipation of high traffic volumes.

1. In the Azure portal, search for the `razorscaling-app-****` container app in the top level search bar and select it from the results.
2. On the overview page, select **Scale** from the left navigation, and then select **+ Edit and deploy**.
3. On the revisions page, switch to the **Scale** tab.
4. Set both the min and max instances to **4** and then select **Create**. This configuration change guarantees your app is scaled horizontally across four instances.

Navigate back to the app. When the page loads, at first it appears everything is working correctly. However, when a search term is entered and submitted, an error may occur. If an error is not displayed, submit the form several more times.

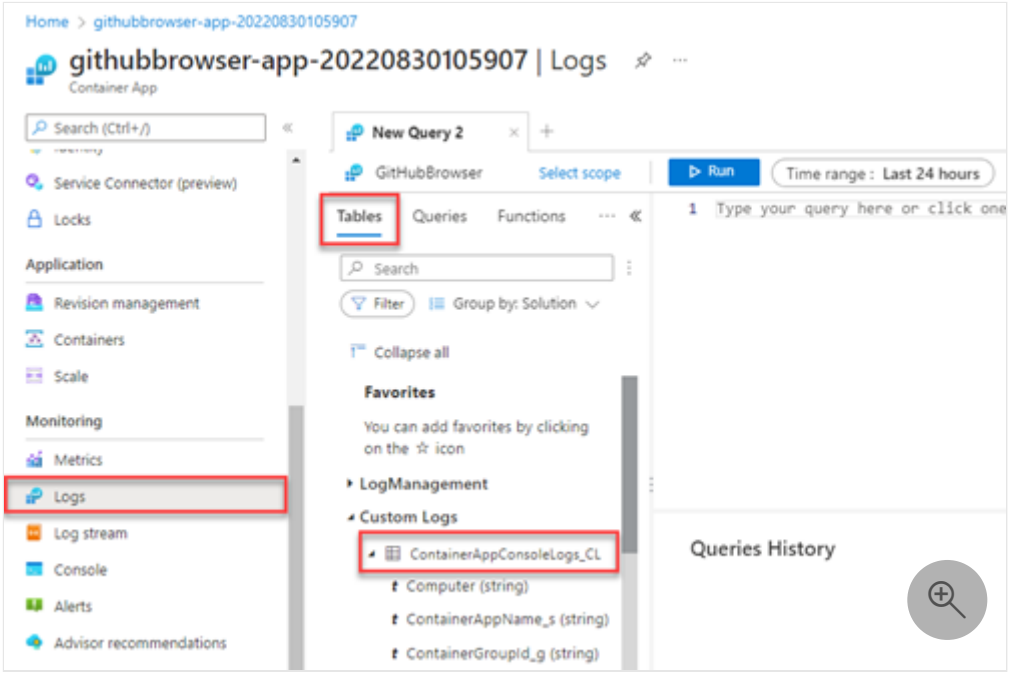
Troubleshooting the error

It's not immediately apparent why the search requests are failing. The browser tools indicate a 400 Bad Request response was sent back. However, you can use the logging features of container apps to diagnose errors occurring in your environment.

1. On the overview page of the container app, select **Logs** from the left navigation.

2. On the **Logs** page, close the pop-up that opens and navigate to the **Tables** tab.

3. Expand the **Custom Logs** item to reveal the **ContainerAppConsoleLogs_CL** node. This table holds various logs for the container app that can be queried to troubleshoot problems.

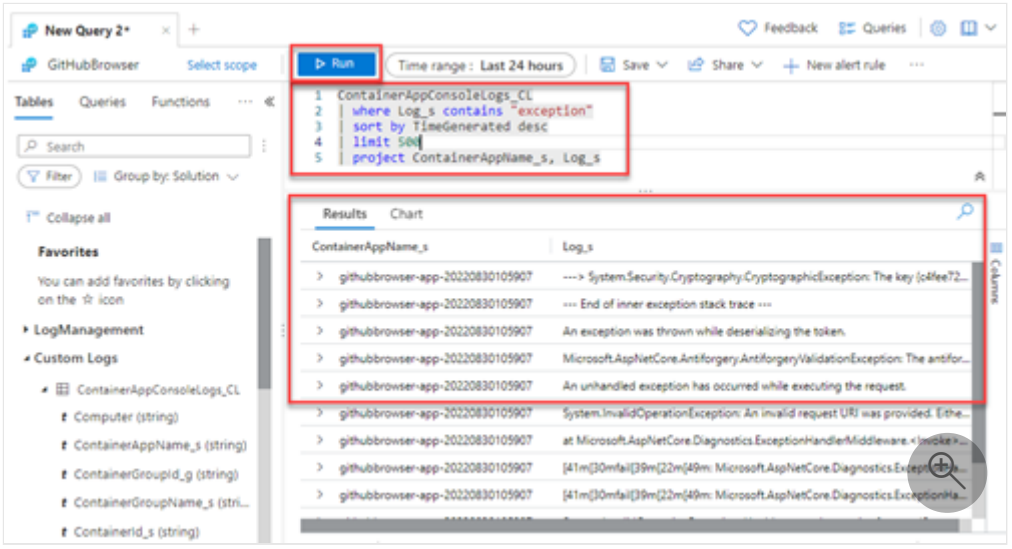


4. In the query editor, compose a basic query to search the **ContainerAppConsoleLogs_CL Logs** table for recent exceptions, such as the following script:

```KQL
ContainerAppConsoleLogs_CL
| where Log_s contains "exception"
| sort by TimeGenerated desc
| limit 500
| project ContainerAppName_s, Log_s
```

   The preceding query searches the **ContainerAppConsoleLogs_CL** table for any rows that contain the word exception. The results are ordered by the time generated, limited to 500 results, and only include the **ContainerAppName_s** and **Log_s** columns to make the results easier to read.

5. Select **Run**, a list of results is displayed. Read through the logs and note that most of them are related to antiforgery tokens and cryptography.



> **ⓘ Important**
>
> The errors in the app are caused by the .NET data protection services. When multiple instances of the app are running, there is no guarantee that the HTTP POST request to submit the form is routed to the same container that initially loaded the page from the HTTP GET request. If the requests are handled by different instances, the antiforgery tokens aren't handled correctly and an exception occurs.

   In the steps ahead, this issue is resolved by centralizing the data protection keys in an Azure storage service and protecting them with Key Vault.

Create the Azure Services

To resolve the preceding errors, the following services are created and connected to the app:

- Azure Storage Account: Handles storing data for the Data Protection Services. Provides a centralized location to store key data as the app scales. Storage accounts can also be used to hold documents, queue data, file shares, and almost any type of blob data.
- Azure KeyVault: This service stores secrets for an app, and is used to help manage encryption concerns for the Data Protection Services.

Create the storage account service

1. In the Azure portal search bar, enter `Storage accounts` and select the matching result.
2. On the storage accounts listing page, select **+ Create**.
3. On the **Basics** tab, enter the following values:

    - **Subscription**: Select the same subscription that you chose for the container app.
    - **Resource Group**: Select the *msdocs-scalable-razor* resource group you created previously.
    - **Storage account name**: Name the account *scalablerazorstorageXXXX* where the X's are random numbers of your choosing. This name must be unique across all of Azure.
    - **Region**: Select the same region you previously selected.

4. Leave the rest of the values at their default and select **Review**. After Azure validates the inputs, select **Create**.

Azure provisions the new storage account. When the task completes, choose **Go to resource** to view the new service.

Create the storage container

Create a Container to store the app's data protection keys.

1. On the overview page for the new storage account, select **Storage browser** on the left navigation.
2. Select **Blob containers**.
3. Select **+ Add container** to open the **New container** flyout menu.

4. Enter a name of *scalablerazorkeys*, leave the rest of the settings at their defaults, and then select **Create**.

The new containers appear on the page list.

Create the key vault service

Create a key vault to hold the keys that protect the data in the blob storage container.

1. In the Azure portal search bar, enter `Key Vault` and select the matching result.
2. On the key vault listing page, select **+ Create**.
3. On the **Basics** tab, enter the following values:

    - **Subscription**: Select the same subscription that was previously selected.
    - **Resource Group**: Select the *msdocs-scalable-razor* resource group previously created.
    - **Key Vault name**: Enter the name *scalablerazorvaultXXXX*.
    - **Region**: Select a region near your location.

4. Leave the rest of the settings at their default, and then select **Review + create**. Wait for Azure to validate your settings, and then select **Create**.

Azure provisions the new key vault. When the task completes, select **Go to resource** to view the new service.

Create the key

Create a secret key to protect the data in the blob storage account.

1. On the main overview page of the key vault, select **Keys** from the left navigation.
2. On the **Create a key** page, select **+ Generate/Import** to open the **Create a key** flyout menu.
3. Enter *razorkey* in the **Name** field. Leave the rest of the settings at their default values and then select **Create**. A new key appears on the key list page.

Connect the Azure Services

The Container App requires a secure connection to the storage account and the key vault services in order to resolve the data protection errors and scale correctly. The new services are connected together using the following steps:

> **ⓘ Important**

> Security role assignments through Service Connector and other tools typically take a minute or two to propagate, and in some rare cases can take up to eight minutes.

Connect the storage account

1. In the Azure portal, navigate to the Container App overview page.
2. On the left navigation, select **Service connector**
3. On the Service Connector page, choose **+ Create** to open the **Creation Connection** flyout panel and enter the following values:

    - **Container**: Select the Container App created previously.
    - **Service type**: Choose **Storage - blob**.
    - **Subscription**: Select the subscription previously used.
    - **Connection name**: Leave the default value.
    - **Storage account**: Select the storage account created previously.
    - **Client type**: Select **.NET**.

4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.
6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector enables a system-assigned managed identity on the container app. It also assigns a role of **Storage Blob Data Contributor** to the identity so it can perform data operations on the storage containers.

Connect the key vault

1. In the Azure portal, navigate to your Container App overview page.
2. On the left navigation, select **Service connector**.
3. On the Service Connector page, choose **+ Create** to open the **Creation Connection** flyout panel and enter the following values:

    - **Container**: Select the container app created previously.
    - **Service type**: Choose **Key Vault**.
    - **Subscription**: Select the subscription previously used.
    - **Connection name**: Leave the default value.
    - **Key vault**: Select the key vault created previously.
    - **Client type**: Select **.NET**.

4. Select **Next: Authentication** to progress to the next step.
5. Select **System assigned managed identity** and choose **Next: Networking**.

6. Leave the default networking options selected, and then select **Review + Create**.
7. After Azure validates the settings, select **Create**.

The service connector assigns a role to the identity so it can perform data operations on the key vault keys.

Configure and redeploy the app

The necessary Azure resources have been created. In this section the app code is configured to use the new resources.

1. Install the following NuGet packages:

    - **Azure.Identity**: Provides classes to work with the Azure identity and access management services.
    - **Microsoft.Extensions.Azure**: Provides helpful extension methods to perform core Azure configurations.
    - **Azure.Extensions.AspNetCore.DataProtection.Blobs**: Allows storing ASP.NET Core DataProtection keys in Azure Blob Storage so that keys can be shared across several instances of a web app.
    - **Azure.Extensions.AspNetCore.DataProtection.Keys**: Enables protecting keys at rest using the Azure Key Vault Key Encryption/Wrapping feature.

    .NET CLI

    ```
    dotnet add package Azure.Identity
    dotnet add package Microsoft.Extensions.Azure
    dotnet add package Azure.Extensions.AspNetCore.DataProtection.Blobs
    dotnet add package Azure.Extensions.AspNetCore.DataProtection.Keys
    ```

2. Update `Program.cs` with the following highlighted code:

    C#

    ```
    using Azure.Identity;
    using Microsoft.AspNetCore.DataProtection;
    using Microsoft.Extensions.Azure;
    var builder = WebApplication.CreateBuilder(args);
    var BlobStorageUri = builder.Configuration["AzureURIs:BlobStorage"];
    var KeyVaultURI = builder.Configuration["AzureURIs:KeyVault"];

    builder.Services.AddRazorPages();
    builder.Services.AddHttpClient();
    builder.Services.AddServerSideBlazor();
    ```

```
builder.Services.AddAzureClientsCore();

builder.Services.AddDataProtection()
                .PersistKeysToAzureBlobStorage(new Uri(BlobStorageUri),
                                               new
DefaultAzureCredential())
                .ProtectKeysWithAzureKeyVault(new Uri(KeyVaultURI),
                                               new
DefaultAzureCredential());
var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding changes allow the app to manage data protection using a centralized, scalable architecture. `DefaultAzureCredential` discovers the managed identity configurations enabled earlier when the app is redeployed.

Update the placeholders in `AzureURIs` section of the `appsettings.json` file to include the following:

1. Replace the `<storage-account-name>` placeholder with the name of the `scalablerazorstorageXXXX` storage account.
2. Replace the `<container-name>` placeholder with the name of the `scalablerazorkeys` storage container.
3. Replace the `<key-vault-name>` placeholder with the name of the `scalablerazorvaultXXXX` key vault.
4. Replace the `<key-name>` placeholder in the key vault URI with the `razorkey` name created previously.

```
C#
```

```
{
  "GitHubURL": "https://api.github.com",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "AzureURIs": {
    "BlobStorage": "https://<storage-account-
name>.blob.core.windows.net/<container-name>/keys.xml",
    "KeyVault": "https://<key-vault-name>.vault.azure.net/keys/<key-
name>/"
  }
}
```

Redeploy the app

The app is now configured correctly to use the Azure services created previously. Redeploy the app for the code changes to be applied.

1. Right click on the project node in the solution explorer and select **Publish**.
2. On the publishing profile summary view, select the **Publish** button in the upper right corner.

Visual Studio redeploys the app to the container apps environment created previously. When the processes finished, the browser launches to the app home page.

Test the app again by searching for *Microsoft* in the search field. The page should now reload with the correct results every time you submit.

Configure roles for local development

The existing code and configuration of the app can also work while running locally during development. The `DefaultAzureCredential` class configured previously is able to pick up local environment credentials to authenticate to Azure Services. You'll need to assign the same roles to your own account that were assigned to your app's managed identity in order for the authentication to work. This should be the same account you use to log into Visual Studio or the Azure CLI.

Sign-in to your local development environment

You'll need to be signed in to the Azure CLI, Visual Studio, or Azure PowerShell for your credentials to be picked up by `DefaultAzureCredential`.

Azure CLI

Azure CLI

```
az login
```

Assign roles to your developer account

1. In the Azure portal, navigate to the `scalablerazor****` storage account created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose **+ Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Storage blob data contributor`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select **+ Select members**.
6. In the **Select members** flyout, search for your own *user@domain* account and select it from the results.
7. Choose **Next** and then select **Review + assign**. After Azure validates the settings, select **Review + assign** again.

As previously stated, role assignment permissions might take a minute or two to propagate, or in rare cases up to eight minutes.

Repeat the previous steps to assign a role to your account so that it can access the key vault service and secret.

1. In the Azure portal, navigate to the `razorscalingkeys` key vault created previously.
2. Select **Access Control (IAM)** from the left navigation.
3. Choose **+ Add** and then **Add role assignment** from the drop-down menu.
4. On the **Add role assignment** page, search for `Key Vault Crypto Service Encryption User`, select the matching result, and then select **Next**.
5. Make sure **User, group, or service principal** is selected, and then select **+ Select members**.
6. In the **Select members** flyout, search for your own *user@domain* account and select it from the results.

7. Choose **Next** and then select **Review + assign**. After Azure validates your settings, select **Review + assign** again.

You may need to wait again for this role assignment to propagate.

You can then return to Visual Studio and run the app locally. The code should continue to function as expected. `DefaultAzureCredential` uses your existing credentials from Visual Studio or the Azure CLI.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Object reuse with ObjectPool in ASP.NET Core

Article • 06/17/2024

By Günther Foidl, Steve Gordon, and Samson Amaugo

Microsoft.Extensions.ObjectPool is part of the ASP.NET Core infrastructure that supports keeping a group of objects in memory for reuse rather than allowing the objects to be garbage collected. All the static and instance methods in `Microsoft.Extensions.ObjectPool` are thread-safe.

Apps might want to use the object pool if the objects that are being managed are:

- Expensive to allocate/initialize.
- Represent a limited resource.
- Used predictably and frequently.

For example, the ASP.NET Core framework uses the object pool in some places to reuse StringBuilder instances. `StringBuilder` allocates and manages its own buffers to hold character data. ASP.NET Core regularly uses `StringBuilder` to implement features, and reusing them provides a performance benefit.

Object pooling doesn't always improve performance:

- Unless the initialization cost of an object is high, it's usually slower to get the object from the pool.
- Objects managed by the pool aren't de-allocated until the pool is de-allocated.

Use object pooling only after collecting performance data using realistic scenarios for your app or library.

NOTE: The ObjectPool doesn't place a limit on the number of objects that it allocates, it places a limit on the number of objects it retains.

ObjectPool concepts

When DefaultObjectPoolProvider is used and `T` implements `IDisposable`:

- Items that are ***not*** returned to the pool will be disposed.
- When the pool gets disposed by DI, all items in the pool are disposed.

NOTE: After the pool is disposed:

- Calling `Get` throws an `ObjectDisposedException`.
- Calling `Return` disposes the given item.

Important `ObjectPool` types and interfaces:

- ObjectPool<T> : The basic object pool abstraction. Used to get and return objects.
- PooledObjectPolicy<T> : Implement this to customize how an object is created and how it's reset when returned to the pool. This can be passed into an object pool that's constructed directly.
- IResettable : Automatically resets the object when returned to an object pool.

The ObjectPool can be used in an app in multiple ways:

- Instantiating a pool.
- Registering a pool in Dependency injection (DI) as an instance.
- Registering the `ObjectPoolProvider<>` in DI and using it as a factory.

How to use ObjectPool

Call Get to get an object and Return to return the object. There's no requirement to return every object. If an object isn't returned, it will be garbage collected.

ObjectPool sample

The following code:

- Adds `ObjectPoolProvider` to the Dependency injection (DI) container.
- Implements the `IResettable` interface to automatically clear the contents of the buffer when returned to the object pool.

C#

```
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.ObjectPool;
using System.Security.Cryptography;

var builder = WebApplication.CreateBuilder(args);

builder.Services.TryAddSingleton<ObjectPoolProvider,
DefaultObjectPoolProvider>();

builder.Services.TryAddSingleton<ObjectPool<ReusableBuffer>>(serviceProvider
=>
{
    var provider = serviceProvider.GetRequiredService<ObjectPoolProvider>();
```

```
    var policy = new DefaultPooledObjectPolicy<ReusableBuffer>();
    return provider.Create(policy);
});

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

// return the SHA256 hash of a word
// https://localhost:7214/hash/SamsonAmaugo
app.MapGet("/hash/{name}", (string name, ObjectPool<ReusableBuffer>
bufferPool) =>
{

    var buffer = bufferPool.Get();
    try
    {
        // Set the buffer data to the ASCII values of a word
        for (var i = 0; i < name.Length; i++)
        {
            buffer.Data[i] = (byte)name[i];
        }

        Span<byte> hash = stackalloc byte[32];
        SHA256.HashData(buffer.Data.AsSpan(0, name.Length), hash);
        return "Hash: " + Convert.ToHexString(hash);
    }
    finally
    {
        // Data is automatically reset because this type implemented
IResettable
        bufferPool.Return(buffer);
    }
});
app.Run();

public class ReusableBuffer : IResettable
{
    public byte[] Data { get; } = new byte[1024 * 1024]; // 1 MB

    public bool TryReset()
    {
        Array.Clear(Data);
        return true;
    }
}
```

NOTE: When the pooled type `T` doesn't implement `IResettable`, then a custom `PooledObjectPolicy<T>` can be used to reset the state of the objects before they are returned to the pool.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Response compression in ASP.NET Core

Article • 06/25/2023

Network bandwidth is a limited resource. Reducing the size of the response usually increases the responsiveness of an app, often dramatically. One way to reduce payload sizes is to compress an app's responses.

Compression with HTTPS

Compressed responses over secure connections can be controlled with the EnableForHttps option, which is disabled by default because of the security risk. Using compression with dynamically generated pages can expose the app to CRIME and BREACH attacks. CRIME and BREACH attacks can be mitigated in ASP.NET Core with antiforgery tokens. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core. For information on mitigating BREACH attacks, see mitigations at http://www.breachattack.com/

Even when `EnableForHttps` is disabled in the app, IIS, IIS Express, and Azure App Service can apply gzip at the IIS web server. When reviewing response headers, take note of the Server value. An unexpected `content-encoding` response header value may be the result of the web server and not the ASP.NET Core app configuration.

When to use Response Compression Middleware

Use server-based response compression technologies in IIS, Apache, or Nginx. The performance of the response compression middleware probably won't match that of the server modules. HTTP.sys server and Kestrel server don't currently offer built-in compression support.

Use Response Compression Middleware when the app is:

- Unable to use the following server-based compression technologies:
  - IIS Dynamic Compression module
  - Apache mod_deflate module
  - Nginx Compression and Decompression
- Hosting directly on:
  - HTTP.sys server
  - Kestrel server

Response compression

Usually, any response not natively compressed can benefit from response compression. Responses not natively compressed typically include CSS, JavaScript, HTML, XML, and JSON. Don't compress natively compressed assets, such as PNG files. When attempting to further compress a natively compressed response, any small extra reduction in size and transmission time will likely be overshadowed by the time it takes to process the compression. Don't compress files smaller than about 150-1000 bytes, depending on the file's content and the efficiency of compression. The overhead of compressing small files may produce a compressed file larger than the uncompressed file.

When a client can process compressed content, the client must inform the server of its capabilities by sending the Accept-Encoding header with the request. When a server sends compressed content, it must include information in the Content-Encoding header on how the compressed response is encoded. Content encoding designations supported by the response compression middleware are shown in the following table.

Expand table

Accept-Encoding header values	**Middleware Supported**	**Description**
`br`	Yes (default)	Brotli compressed data format
`deflate`	No	DEFLATE compressed data format
`exi`	No	W3C Efficient XML Interchange
`gzip`	Yes	Gzip file format
`identity`	Yes	"No encoding" identifier: The response must not be encoded.
`pack200-gzip`	No	Network Transfer Format for Java Archives
`*`	Yes	Any available content encoding not explicitly requested

For more information, see the IANA Official Content Coding List.

The response compression middleware allows adding additional compression providers for custom `Accept-Encoding` header values. For more information, see Custom Providers in this article.

The response compression middleware is capable of reacting to quality value (qvalue, `q`) weighting when sent by the client to prioritize compression schemes. For more

information, see RFC 9110: Accept-Encoding.

Compression algorithms are subject to a tradeoff between compression speed and the effectiveness of the compression. *Effectiveness* in this context refers to the size of the output after compression. The smallest size is achieved by the optimal compression.

The headers involved in requesting, sending, caching, and receiving compressed content are described in the following table.

Expand table

Header	Role
Accept-Encoding	Sent from the client to the server to indicate the content encoding schemes acceptable to the client.
Content-Encoding	Sent from the server to the client to indicate the encoding of the content in the payload.
Content-Length	When compression occurs, the `Content-Length` header is removed, since the body content changes when the response is compressed.
Content-MD5	When compression occurs, the `Content-MD5` header is removed, since the body content has changed and the hash is no longer valid.
Content-Type	Specifies the MIME type of the content. Every response should specify its `Content-Type`. The response compression middleware checks this value to determine if the response should be compressed. The response compression middleware specifies a set of default MIME types that it can encode, and they can bed replaced or added.
Vary	When sent by the server with a value of `Accept-Encoding` to clients and proxies, the `Vary` header indicates to the client or proxy that it should cache (vary) responses based on the value of the `Accept-Encoding` header of the request. The result of returning content with the `Vary: Accept-Encoding` header is that both compressed and uncompressed responses are cached separately.

Explore the features of the Response Compression Middleware with the sample app. The sample illustrates:

- The compression of app responses using Gzip and custom compression providers.
- How to add a MIME type to the default list of MIME types for compression.
- How to add a custom response compression provider.

Configuration

The following code shows how to enable the Response Compression Middleware for default MIME types and compression providers (Brotli and Gzip):

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
});

var app = builder.Build();

app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Notes:

- Setting `EnableForHttps` to `true` is a security risk. See Compression with HTTPS in this article for more information.
- app.UseResponseCompression must be called ***before*** any middleware that compresses responses. For more information, see ASP.NET Core Middleware.
- Use a tool such as Firefox Browser Developer to set the `Accept-Encoding` request header and examine the response headers, size, and body.

Submit a request to the sample app without the `Accept-Encoding` header and observe that the response is uncompressed. The `Content-Encoding` header isn't in the Response Headers collection.

For example, in Firefox Developer:

- Select the network tab.
- Right click the request in the Network request list and select **Edit and resend**
- Change `Accept-Encoding:` from `gzip, deflate, br` to `none`.
- Select **Send**.

Submit a request to the sample app with a browser using the developer tools and observe that the response is compressed. The `Content-Encoding` and `Vary` headers are present on the response.

Providers

Brotli and Gzip compression providers

Use the BrotliCompressionProvider to compress responses with the Brotli compressed data format.

If no compression providers are explicitly added to the CompressionProviderCollection:

- The Brotli compression provider and Gzip compression provider are added by default to the array of compression providers.
- Compression defaults to Brotli compression when the Brotli compressed data format is supported by the client. If Brotli isn't supported by the client, compression defaults to Gzip when the client supports Gzip compression.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

When a compression provider is added, other providers aren't added. For example, if the Gzip compression provider is the only provider explicitly added, no other compression providers are added.

The following code:

- Enables response compression for HTTPS requests.
- Adds the Brotli and Gzip response compression providers.

C#

```
using System.IO.Compression;
using Microsoft.AspNetCore.ResponseCompression;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
});

builder.Services.Configure<BrotliCompressionProviderOptions>(options =>
```

```
{
    options.Level = CompressionLevel.Fastest;
});

builder.Services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.SmallestSize;
});

var app = builder.Build();

app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Set the compression level with BrotliCompressionProviderOptions and GzipCompressionProviderOptions. The Brotli and Gzip compression providers default to the fastest compression level, CompressionLevel.Fastest, which might not produce the most efficient compression. If the most efficient compression is desired, configure the response compression middleware for optimal compression.

See CompressionLevel Enum for values that indicate whether a compression operation emphasizes speed or compression size.

C#

```
using System.IO.Compression;
using Microsoft.AspNetCore.ResponseCompression;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
});

builder.Services.Configure<BrotliCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.Fastest;
});

builder.Services.Configure<GzipCompressionProviderOptions>(options =>
{
    options.Level = CompressionLevel.SmallestSize;
});

var app = builder.Build();
```

```
app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Custom providers

Create custom compression implementations with ICompressionProvider. The EncodingName represents the content encoding that this `ICompressionProvider` produces. The response compression middleware uses this information to choose the provider based on the list specified in the `Accept-Encoding` header of the request.

Requests to the sample app with the `Accept-Encoding: mycustomcompression` header return a response with a `Content-Encoding: mycustomcompression` header. The client must be able to decompress the custom encoding in order for a custom compression implementation to work.

C#

```
using Microsoft.AspNetCore.ResponseCompression;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
});

var app = builder.Build();

app.UseResponseCompression();

app.MapGet("/", () => "Hello World!");

app.Run();
```

C#

```
using Microsoft.AspNetCore.ResponseCompression;

public class CustomCompressionProvider : ICompressionProvider
{
    public string EncodingName => "mycustomcompression";
    public bool SupportsFlush => true;
```

```
    public Stream CreateStream(Stream outputStream)
    {
        // Replace with a custom compression stream wrapper.
        return outputStream;
    }
}
```

With the preceding code, the response body isn't compressed by the sample. However, the sample shows where to implement a custom compression algorithm.

MIME types

The response compression middleware specifies a default set of MIME types for compression. See the source code for a complete list of MIME types supported.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Replace or append MIME types with ResponseCompressionOptions.MimeTypes. Note that wildcard MIME types, such as `text/*` aren't supported. The sample app adds a MIME type for `image/svg+xml` and compresses and serves the ASP.NET Core banner image *banner.svg*.

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using ResponseCompressionSample;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes =
    ResponseCompressionDefaults.MimeTypes.Concat(
        new[] { "image/svg+xml" });
});
```

```
var app = builder.Build();

app.UseResponseCompression();
```

Adding the Vary header

When compressing responses based on the Accept-Encoding request header, there are can be uncompressed and multiple compressed versions of the response. In order to instruct client and proxy caches that multiple versions exist and should be stored, the `Vary` header is added with an `Accept-Encoding` value. The response middleware adds the Vary header automatically when the response is compressed.

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Middleware issue when behind an Nginx reverse proxy

When a request is proxied by Nginx, the `Accept-Encoding` header is removed. Removal of the `Accept-Encoding` header prevents the response compression middleware from compressing the response. For more information, see NGINX: Compression and Decompression. This issue is tracked by Figure out pass-through compression for Nginx (dotnet/aspnetcore#5989).

Disabling IIS dynamic compression

To disable IIS Dynamic Compression Module configured at the server level, see Disabling IIS modules.

Troubleshoot response compression

Use a tool like Firefox Browser Developer, which allows setting the `Accept-Encoding` request header and study the response headers, size, and body. By default, Response Compression Middleware compresses responses that meet the following conditions:

- The `Accept-Encoding` header is present with a value of `br`, `gzip`, `*`, or custom encoding that matches a custom compression provider. The value must not be `identity` or have a quality value (qvalue, `q`) setting of 0 (zero).
- The MIME type (`Content-Type`) must be set and must match a MIME type configured on the ResponseCompressionOptions.
- The request must not include the `Content-Range` header.
- The request must use insecure protocol (http), unless secure protocol (https) is configured in the Response Compression Middleware options. *Note the danger described above when enabling secure content compression.*

Azure deployed sample

The sample app deployed to Azure has the following `Program.cs` file:

C#

```
using Microsoft.AspNetCore.ResponseCompression;
using ResponseCompressionSample;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.Providers.Add<BrotliCompressionProvider>();
    options.Providers.Add<GzipCompressionProvider>();
    options.Providers.Add<CustomCompressionProvider>();
    options.MimeTypes =
    ResponseCompressionDefaults.MimeTypes.Concat(
        new[] { "image/svg+xml" });
});

var app = builder.Build();

app.UseResponseCompression();

app.Map("/trickle", async (HttpResponse httpResponse) =>
{
    httpResponse.ContentType = "text/plain;charset=utf-8";

    for (int i = 0; i < 20; i++)
    {
        await httpResponse.WriteAsync("a");
        await httpResponse.Body.FlushAsync();
```

```
        await Task.Delay(TimeSpan.FromMilliseconds(50));
    }
});

app.Map("/testfile1kb.txt", () => Results.File(

app.Environment.ContentRootFileProvider.GetFileInfo("testfile1kb.txt").Physi
calPath,
    "text/plain;charset=utf-8"));

app.Map("/banner.svg", () => Results.File(

app.Environment.ContentRootFileProvider.GetFileInfo("banner.svg").PhysicalPa
th,
    "image/svg+xml;charset=utf-8"));

app.MapFallback(() => LoremIpsum.Text);

app.Run();
```

Additional resources

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

- View or download sample code (how to download)
- Response compression middleware source
- ASP.NET Core Middleware
- Mozilla Developer Network: Accept-Encoding
- RFC 9110 Section 8.4.1: Content Codings
- RFC 9110 Section 8.4.1.3: Gzip Coding
- GZIP file format specification version 4.3

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

Performance Diagnostic Tools

Article • 11/27/2022

By Mike Rousos

This article lists tools for diagnosing performance issues in ASP.NET Core.

Visual Studio Diagnostic Tools

The profiling and diagnostic tools built into Visual Studio are a good place to start investigating performance issues. These tools are powerful and convenient to use from the Visual Studio development environment. The tooling allows analysis of CPU usage, memory usage, and performance events in ASP.NET Core apps. Being built-in makes profiling easy at development time.

More information is available in Visual Studio documentation.

Application Insights

Application Insights provides in-depth performance data for your app. Application Insights automatically collects data on response rates, failure rates, dependency response times, and more. Application Insights supports logging custom events and metrics specific to your app.

Azure Application Insights provides multiple ways to give insights on monitored apps:

- Application Map – helps spot performance bottlenecks or failure hot-spots across all components of distributed apps.
- Azure Metrics Explorer is a component of the Microsoft Azure portal that allows plotting charts, visually correlating trends, and investigating spikes and dips in metrics' values.
- Performance blade in Application Insights portal:
  - Shows performance details for different operations in the monitored app.
  - Allows drilling into a single operation to check all parts/dependencies that contribute to a long duration.
  - Profiler can be invoked from here to collect performance traces on-demand.
- Azure Application Insights Profiler allows regular and on-demand profiling of .NET apps. Azure portal shows captured performance traces with call stacks and hot

paths. The trace files can also be downloaded for deeper analysis using PerfView.

Application Insights can be used in a variety of environments:

- Optimized to work in Azure.
- Works in production, development, and staging.
- Works locally from Visual Studio or in other hosting environments.

For more information on code-based monitoring, see Application Insights for ASP.NET Core. For more information on codeless monitoring, see Monitor Azure App Service performance.

PerfView

PerfView is a performance analysis tool created by the .NET team specifically for diagnosing .NET performance issues. PerfView allows analysis of CPU usage, memory and GC behavior, performance events, and wall clock time.

For more about PerfView, see the user's guide available in the tool or on GitHub.

Windows Performance Toolkit

Windows Performance Toolkit (WPT) consists of two components: Windows Performance Recorder (WPR) and Windows Performance Analyzer (WPA). The tools produce in-depth performance profiles of Windows operating systems and apps. WPT has richer ways of visualizing data, but its data collecting is less powerful than PerfView's.

PerfCollect

While PerfView is a useful performance analysis tool for .NET scenarios, it only runs on Windows, so you can't use it to collect traces from ASP.NET Core apps running in Linux environments.

PerfCollect is a bash script that uses native Linux profiling tools (Perf and LTTng) to collect traces on Linux that can be analyzed by PerfView. PerfCollect is useful when performance problems show up in Linux environments where PerfView can't be used directly. Instead, PerfCollect can collect traces from .NET Core apps that are then analyzed on a Windows computer using PerfView.

More information about how to install and get started with PerfCollect is available on GitHub.

Other Third-party Performance Tools

The following lists some third-party performance tools that are useful in performance investigation of .NET Core applications.

- MiniProfiler
- dotTrace and dotMemory from JetBrains
- VTune from Intel

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET Core load/stress testing

Article • 08/03/2023

Load testing and stress testing are important to ensure a web app is performant and scalable. Load and stress testing have different goals even though they often share similar tests.

Load tests: Test whether the app can handle a specified load of users for a certain scenario while still satisfying the response goal. The app is run under normal conditions.

Stress tests: Test app stability when running under extreme conditions, often for a long period of time. The tests place high user load, either spikes or gradually increasing load on the app, or they limit the app's computing resources.

Stress tests determine if an app under stress can recover from failure and gracefully return to expected behavior. Under stress, the app is run at abnormally high stress.

Azure Load Testing is a fully managed load-testing service that enables you to generate high-scale load. The service simulates traffic for apps, regardless of where they're hosted. Azure Load Testing Preview enables you to use existing Apache JMeter scripts to generate high-scale load.

Visual Studio 2019 load testing has been deprecated. The corresponding Azure DevOps cloud-based load testing service has been closed.

Third-party tools

The following list contains third-party web performance tools with various feature sets:

- Apache JMeter
- ApacheBench (ab)
- Gatling
- jmeter-dotnet-dsl
- k6
- Locust
- West Wind WebSurge
- Netling
- Vegeta
- NBomber

Load and stress test with release builds

Load and stress tests should be done in release and production mode and not in debug and development mode. Release configurations are fully optimized with minimal logging. Debug configuration is not optimized. Development mode enables more information logging that can impact performance.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Tutorial: Measure performance using EventCounters in .NET Core

Article • 06/24/2023

This article applies to: ✔ .NET Core 3.0 SDK and later versions

In this tutorial, you'll learn how an EventCounter can be used to measure performance with a high frequency of events. You can use the available counters published by various official .NET Core packages, third-party providers, or create your own metrics for monitoring.

In this tutorial, you will:

- ✔ Implement an EventSource.
- ✔ Monitor counters with dotnet-counters.

Prerequisites

The tutorial uses:

- .NET Core 3.1 SDK or a later version.
- dotnet-counters to monitor event counters.
- A sample debug target app to diagnose.

Get the source

The sample application will be used as a basis for monitoring. The sample ASP.NET Core repository is available from the samples browser. You download the zip file, extract it once downloaded, and open it in your favorite IDE. Build and run the application to ensure that it works properly, then stop the application.

Implement an EventSource

For events that happen every few milliseconds, you'll want the overhead per event to be low (less than a millisecond). Otherwise, the impact on performance will be significant. Logging an event means you're going to write something to disk. If the disk is not fast enough, you will lose events. You need a solution other than logging the event itself.

When dealing with a large number of events, knowing the measure per event is not useful either. Most of the time all you need is just some statistics out of it. So you could

get the statistics within the process itself and then write an event once in a while to report the statistics, that's what EventCounter will do.

Below is an example of how to implement an System.Diagnostics.Tracing.EventSource. Create a new file named *MinimalEventCounterSource.cs* and use the code snippet as its source:

C#

```
using System.Diagnostics.Tracing;

[EventSource(Name = "Sample.EventCounter.Minimal")]
public sealed class MinimalEventCounterSource : EventSource
{
    public static readonly MinimalEventCounterSource Log = new
MinimalEventCounterSource();

    private EventCounter _requestCounter;

    private MinimalEventCounterSource() =>
        _requestCounter = new EventCounter("request-time", this)
        {
            DisplayName = "Request Processing Time",
            DisplayUnits = "ms"
        };

    public void Request(string url, long elapsedMilliseconds)
    {
        WriteEvent(1, url, elapsedMilliseconds);
        _requestCounter?.WriteMetric(elapsedMilliseconds);
    }

    protected override void Dispose(bool disposing)
    {
        _requestCounter?.Dispose();
        _requestCounter = null;

        base.Dispose(disposing);
    }
}
```

The EventSource.WriteEvent line is the EventSource part and is not part of EventCounter, it was written to show that you can log a message together with the event counter.

Add an action filter

The sample source code is an ASP.NET Core project. You can add an action filter globally that will log the total request time. Create a new file named *LogRequestTimeFilterAttribute.cs*, and use the following code:

C#

```
using System.Diagnostics;
using Microsoft.AspNetCore.Http.Extensions;
using Microsoft.AspNetCore.Mvc.Filters;

namespace DiagnosticScenarios
{
    public class LogRequestTimeFilterAttribute : ActionFilterAttribute
    {
        readonly Stopwatch _stopwatch = new Stopwatch();

        public override void OnActionExecuting(ActionExecutingContext
context) => _stopwatch.Start();

        public override void OnActionExecuted(ActionExecutedContext context)
        {
            _stopwatch.Stop();

            MinimalEventCounterSource.Log.Request(
                context.HttpContext.Request.GetDisplayUrl(),
_stopwatch.ElapsedMilliseconds);
        }
    }
}
```

The action filter starts a Stopwatch as the request begins, and stops after it has been completed, capturing the elapsed time. The total milliseconds are logged to the `MinimalEventCounterSource` singleton instance. For this filter to be applied, you need to add it to the filter collection. In the *Startup.cs* file, update the `ConfigureServices` method in include this filter.

C#

```
public void ConfigureServices(IServiceCollection services) =>
    services.AddControllers(options =>
options.Filters.Add<LogRequestTimeFilterAttribute>())
            .AddNewtonsoftJson();
```

Monitor event counter

With the implementation on an EventSource and the custom action filter, build and launch the application. You logged the metric to the EventCounter, but unless you access the statistics from of it, it is not useful. To get the statistics, you need to enable the EventCounter by creating a timer that fires as frequently as you want the events, as well as a listener to capture the events. To do that, you can use dotnet-counters.

Use the dotnet-counters ps command to display a list of .NET processes that can be monitored.

Console

```
dotnet-counters ps
```

Using the process identifier from the output of the `dotnet-counters ps` command, you can start monitoring the event counter with the following `dotnet-counters monitor` command:

Console

```
dotnet-counters monitor --process-id 2196 --counters
Sample.EventCounter.Minimal,Microsoft.AspNetCore.Hosting[total-
requests,requests-per-second],System.Runtime[cpu-usage]
```

While the `dotnet-counters monitor` command is running, hold F5 on the browser to start issuing continuous requests to the `https://localhost:5001/api/values` endpoint. After a few seconds stop by pressing q

Console

```
Press p to pause, r to resume, q to quit.
    Status: Running

[Microsoft.AspNetCore.Hosting]
    Request Rate / 1 sec                                 9
    Total Requests                                     134
[System.Runtime]
    CPU Usage (%)                                       13
[Sample.EventCounter.Minimal]
    Request Processing Time (ms)                      34.5
```

The `dotnet-counters monitor` command is great for active monitoring. However, you may want to collect these diagnostic metrics for post processing and analysis. For that, use the `dotnet-counters collect` command. The `collect` switch command is similar to the `monitor` command, but accepts a few additional parameters. You can specify the desired output file name and format. For a JSON file named *diagnostics.json* use the following command:

Console

```
dotnet-counters collect --process-id 2196 --format json -o diagnostics.json
--counters Sample.EventCounter.Minimal,Microsoft.AspNetCore.Hosting[total-
```

```
requests,requests-per-second],System.Runtime[cpu-usage]
```

Again, while the command is running, hold `F5` on the browser to start issuing continuous requests to the `https://localhost:5001/api/values` endpoint. After a few seconds stop by pressing `q`. The *diagnostics.json* file is written. The JSON file that is written is not indented, however; for readability it is indented here.

JSON


```json
{
  "TargetProcess": "DiagnosticScenarios",
  "StartTime": "8/5/2020 3:02:45 PM",
  "Events": [
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "System.Runtime",
      "name": "CPU Usage (%)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Request Rate / 1 sec",
      "counterType": "Rate",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Total Requests",
      "counterType": "Metric",
      "value": 134
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "Sample.EventCounter.Minimal",
      "name": "Request Processing Time (ms)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:47Z",
      "provider": "System.Runtime",
      "name": "CPU Usage (%)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Request Rate / 1 sec",
```

```
      "counterType": "Rate",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Total Requests",
      "counterType": "Metric",
      "value": 134
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "Sample.EventCounter.Minimal",
      "name": "Request Processing Time (ms)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:48Z",
      "provider": "System.Runtime",
      "name": "CPU Usage (%)",
      "counterType": "Metric",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Request Rate / 1 sec",
      "counterType": "Rate",
      "value": 0
    },
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Total Requests",
      "counterType": "Metric",
      "value": 134
    },
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Sample.EventCounter.Minimal",
      "name": "Request Processing Time (ms)",
      "counterType": "Metric",
      "value": 0
    }
  ]
}
```


Next steps

EventCounters

Globalization and localization in ASP.NET Core

Article • 05/09/2023

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

A multilingual website allows a website to reach a wider audience. ASP.NET Core provides services and middleware for localizing into different languages and cultures.

For Blazor localization guidance, which adds to or supersedes the guidance in this article, see ASP.NET Core Blazor globalization and localization.

Terms

- Globalization (G11N): The process of making an app support different languages and regions. The abbreviation comes from the first and last letters and the number of letters between them.
- Localization (L10N): The process of customizing a globalized app for specific languages and regions.
- Internationalization (I18N): Both globalization and localization.
- Culture: A language and, optionally, a region.
- Neutral culture: A culture that has a specified language, but not a region (for example "en", "es").
- Specific culture: A culture that has a specified language and region (for example, "en-US", "en-GB", "es-CL").
- Parent culture: The neutral culture that contains a specific culture (for example, "en" is the parent culture of "en-US" and "en-GB").
- Locale: A locale is the same as a culture.

Language and country/region codes

The RFC 4646 format for the culture name is `<language code>-<country/region code>`, where `<language code>` identifies the language and `<country/region code>` identifies the subculture. For example, `es-CL` for Spanish (Chile), `en-US` for English (United States), and `en-AU` for English (Australia). RFC 4646 is a combination of an ISO 639 two-letter lowercase culture code associated with a language and an ISO 3166 two-letter uppercase subculture code associated with a country or region. For more information, see System.Globalization.CultureInfo.

Tasks to localize an app

Globalizing and localizing an app involves the following tasks:

- Make an ASP.NET Core app's content localizable.
- Provide localized resources for the cultures the app supports
- Implement a strategy to select the culture for each request

View or download sample code (how to download)

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- IStringLocalizer : Uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The interface has an indexer and an `IEnumerable` for returning localized strings.
- IHtmlLocalizer: For resources that contain HTML.
- View and DataAnnotations
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Make an ASP.NET Core app's content localizable

Article • 10/25/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

By Hisham Bin Ateya, Damien Bowden, Bart Calixto and Nadeem Afana

One task for localizing an app is to wrap localizable content with code that facilitates replacing that content for different cultures.

`IStringLocalizer`

IStringLocalizer and IStringLocalizer<T> were architected to improve productivity when developing localized apps. `IStringLocalizer` uses the ResourceManager and ResourceReader to provide culture-specific resources at run time. The interface has an indexer and an `IEnumerable` for returning localized strings. `IStringLocalizer` doesn't require storing the default language strings in a resource file. You can develop an app targeted for localization and not need to create resource files early in development.

The following code example shows how to wrap the string "About Title" for localization.

C#

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace Localization.Controllers;

[Route("api/[controller]")]
public class AboutController : Controller
{
    private readonly IStringLocalizer<AboutController> _localizer;

    public AboutController(IStringLocalizer<AboutController> localizer)
    {
        _localizer = localizer;
    }

    [HttpGet]
```

```
    public string Get()
    {
        return _localizer["About Title"];
    }
}
```

In the preceding code, the `IStringLocalizer<T>` implementation comes from Dependency Injection. If the localized value of "About Title" isn't found, then the indexer key is returned, that is, the string "About Title".

You can leave the default language literal strings in the app and wrap them in the localizer, so that you can focus on developing the app. You develop an app with your default language and prepare it for the localization step without first creating a default resource file.

Alternatively, you can use the traditional approach and provide a key to retrieve the default language string. For many developers, the new workflow of not having a default language *.resx* file and simply wrapping the string literals can reduce the overhead of localizing an app. Other developers prefer the traditional work flow as it can be easier to work with long string literals and easier to update localized strings.

`IHtmlLocalizer`

Use the IHtmlLocalizer<TResource> implementation for resources that contain HTML. IHtmlLocalizer HTML-encodes arguments that are formatted in the resource string, but doesn't HTML-encode the resource string itself. In the following highlighted code, only the value of the `name` parameter is HTML-encoded.

C#

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Localization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Localization;

namespace Localization.Controllers;

public class BookController : Controller
{
    private readonly IHtmlLocalizer<BookController> _localizer;

    public BookController(IHtmlLocalizer<BookController> localizer)
    {
        _localizer = localizer;
    }
```

```
    public IActionResult Hello(string name)
    {
        ViewData["Message"] = _localizer["<b>Hello</b><i> {0}</i>", name];

        return View();
    }
```

NOTE: Generally, only localize text, not HTML.

`IStringLocalizerFactory`

At the lowest level, IStringLocalizerFactory can be retrieved from of Dependency Injection:

C#

```
public class TestController : Controller
{
    private readonly IStringLocalizer _localizer;
    private readonly IStringLocalizer _localizer2;

    public TestController(IStringLocalizerFactory factory)
    {
        var type = typeof(SharedResource);
        var assemblyName = new
AssemblyName(type.GetTypeInfo().Assembly.FullName);
        _localizer = factory.Create(type);
        _localizer2 = factory.Create("SharedResource", assemblyName.Name);
    }

    public IActionResult About()
    {
        ViewData["Message"] = _localizer["Your application description
page."]
            + " loc 2: " + _localizer2["Your application description
page."];

        return View();
    }
```

The preceding code demonstrates each of the two factory create methods.

Shared resources

You can partition your localized strings by controller or area, or have just one container. In the sample app, a marker class named `SharedResource` is used for shared resources.

The marker class is never called:

C#

```
// Dummy class to group shared resources

namespace Localization;

public class SharedResource
{
}
```

In the following sample, the `InfoController` and the `SharedResource` localizers are used:

C#

```
public class InfoController : Controller
{
    private readonly IStringLocalizer<InfoController> _localizer;
    private readonly IStringLocalizer<SharedResource> _sharedLocalizer;

    public InfoController(IStringLocalizer<InfoController> localizer,
                          IStringLocalizer<SharedResource> sharedLocalizer)
    {
        _localizer = localizer;
        _sharedLocalizer = sharedLocalizer;
    }

    public string TestLoc()
    {
        string msg = "Shared resx: " + _sharedLocalizer["Hello!"] +
                     " Info resx " + _localizer["Hello!"];
        return msg;
    }
```

View localization

The IViewLocalizer service provides localized strings for a view. The `ViewLocalizer` class implements this interface and finds the resource location from the view file path. The following code shows how to use the default implementation of `IViewLocalizer`:

CSHTML

```
@using Microsoft.AspNetCore.Mvc.Localization

@inject IViewLocalizer Localizer

@{
```

```
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>

<p>@Localizer["Use this area to provide additional information."]</p>
```

The default implementation of `IViewLocalizer` finds the resource file based on the view's file name. There's no option to use a global shared resource file. `ViewLocalizer` implements the localizer using `IHtmlLocalizer`, so Razor doesn't HTML-encode the localized string. You can parameterize resource strings, and `IViewLocalizer` HTML-encodes the parameters but not the resource string. Consider the following Razor markup:

CSHTML

```
@Localizer["<i>Hello</i> <b>{0}!</b>", UserManager.GetUserName(User)]
```

A French resource file could contain the following values:

Expand table

Key	Value
`<i>Hello</i> <b>{0}!</b>`	`<i>Bonjour</i> <b>{0} !</b>`

The rendered view would contain the HTML markup from the resource file.

Generally, ***only localize text***, not HTML.

To use a shared resource file in a view, inject `IHtmlLocalizer<T>`:

CSHTML

```
@using Microsoft.AspNetCore.Mvc.Localization
@using Localization.Services

@inject IViewLocalizer Localizer
@inject IHtmlLocalizer<SharedResource> SharedLocalizer

@{
    ViewData["Title"] = Localizer["About"];
}
<h2>@ViewData["Title"].</h2>

<h1>@SharedLocalizer["Hello!"]</h1>
```

DataAnnotations localization

DataAnnotations error messages are localized with `IStringLocalizer<T>`. Using the option `ResourcesPath = "Resources"`, the error messages in `RegisterViewModel` can be stored in either of the following paths:

- *Resources/ViewModels.Account.RegisterViewModel.fr.resx*
- *Resources/ViewModels/Account/RegisterViewModel.fr.resx*

C#

```
using System.ComponentModel.DataAnnotations;

namespace Localization.ViewModels.Account;

public class RegisterViewModel
{
    [Required(ErrorMessage = "The Email field is required.")]
    [EmailAddress(ErrorMessage = "The Email field is not a valid email
address.")]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required(ErrorMessage = "The Password field is required.")]
    [StringLength(8, ErrorMessage = "The {0} must be at least {2} characters
long.",

MinimumLength = 6)]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm password")]
    [Compare("Password", ErrorMessage =
                            "The password and confirmation password do not
match.")]
    public string ConfirmPassword { get; set; }
}
```

Non-validation attributes are localized.

How to use one resource string for multiple classes

The following code shows how to use one resource string for validation attributes with multiple classes:

C#

```
    services.AddMvc()
        .AddDataAnnotationsLocalization(options => {
            options.DataAnnotationLocalizerProvider = (type, factory) =>
                factory.Create(typeof(SharedResource));
        });
```

In the preceding code, `SharedResource` is the class corresponding to the *.resx* file where the validation messages are stored. With this approach, DataAnnotations only uses `SharedResource`, rather than the resource for each class.

Configure localization services

Localization services are configured in `Program.cs`:

C#

```
builder.Services.AddLocalization(options => options.ResourcesPath =
"Resources");

builder.Services.AddMvc()
    .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix)
    .AddDataAnnotationsLocalization();
```

- AddLocalization adds the localization services to the services container, including implementations for `IStringLocalizer<T>` and `IStringLocalizerFactory`. The preceding code also sets the resources path to "Resources".
- AddViewLocalization adds support for localized view files. In this sample, view localization is based on the view file suffix. For example "fr" in the `Index.fr.cshtml` file.
- AddDataAnnotationsLocalization adds support for localized `DataAnnotations` validation messages through `IStringLocalizer` abstractions.

> **Note**
>
> You may not be able to enter decimal commas in decimal fields. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See this GitHub comment 4076 for instructions on adding decimal comma.

Next steps

Localizing an app also involves the following tasks:

- Provide localized resources for the languages and cultures the app supports
- Implement a strategy to select the language/culture for each request

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- Globalization and localization in ASP.NET Core
- Provide localized resources for languages and cultures in an ASP.NET Core app
- Strategies for selecting language and culture in a localized ASP.NET Core app
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Localization.StarterWeb project used in the article.
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Provide localized resources for languages and cultures in an ASP.NET Core app

Article • 07/26/2024

By Rick Anderson, Damien Bowden, Bart Calixto, Nadeem Afana, and Hisham Bin Ateya

One task for localizing an app is to provide localized strings in resource files. This article is about working with resource files.

`SupportedCultures` and `SupportedUICultures`

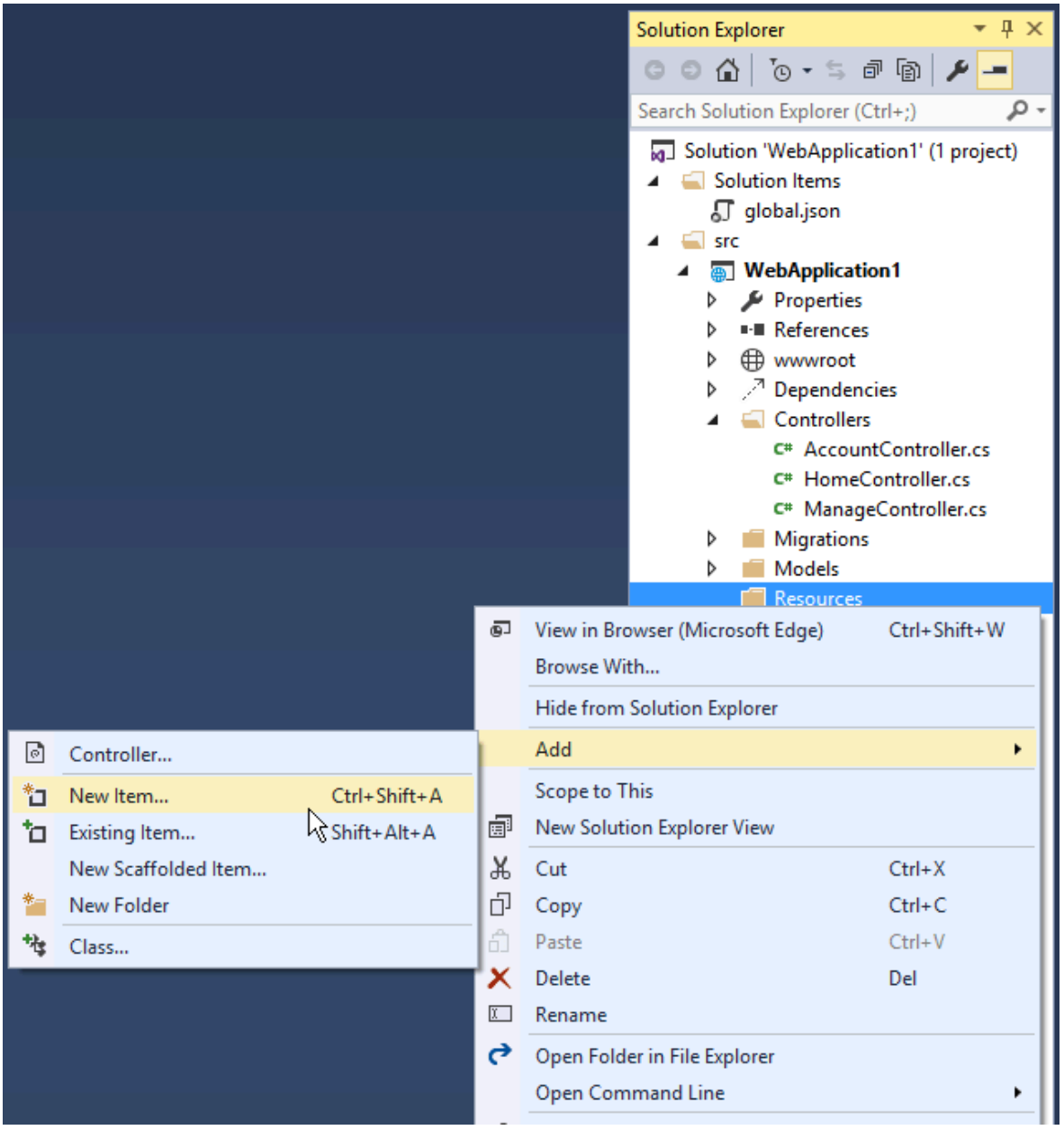
ASP.NET Core has two collections of culture values, `SupportedCultures` and `SupportedUICultures`. The CultureInfo object for `SupportedCultures` determines the results of culture-dependent functions, such as date, time, number, and currency formatting. `SupportedCultures` also determines the sorting order of text, casing conventions, and string comparisons. See StringComparer.CurrentCulture for more info on how the server gets the culture. The `SupportedUICultures` determines which translated strings (from *.resx* files) are looked up by the ResourceManager. The `ResourceManager` simply looks up culture-specific strings that are determined by `CurrentUICulture`. Every thread in .NET has `CurrentCulture` and `CurrentUICulture` objects. ASP.NET Core inspects these values when rendering culture-dependent functions. For example, if the current thread's culture is set to "en-US" (English, United States), `DateTime.Now.ToLongDateString()` displays "Thursday, February 18, 2016", but if `CurrentCulture` is set to "es-ES" (Spanish, Spain) the output will be "jueves, 18 de febrero de 2016".

Resource files

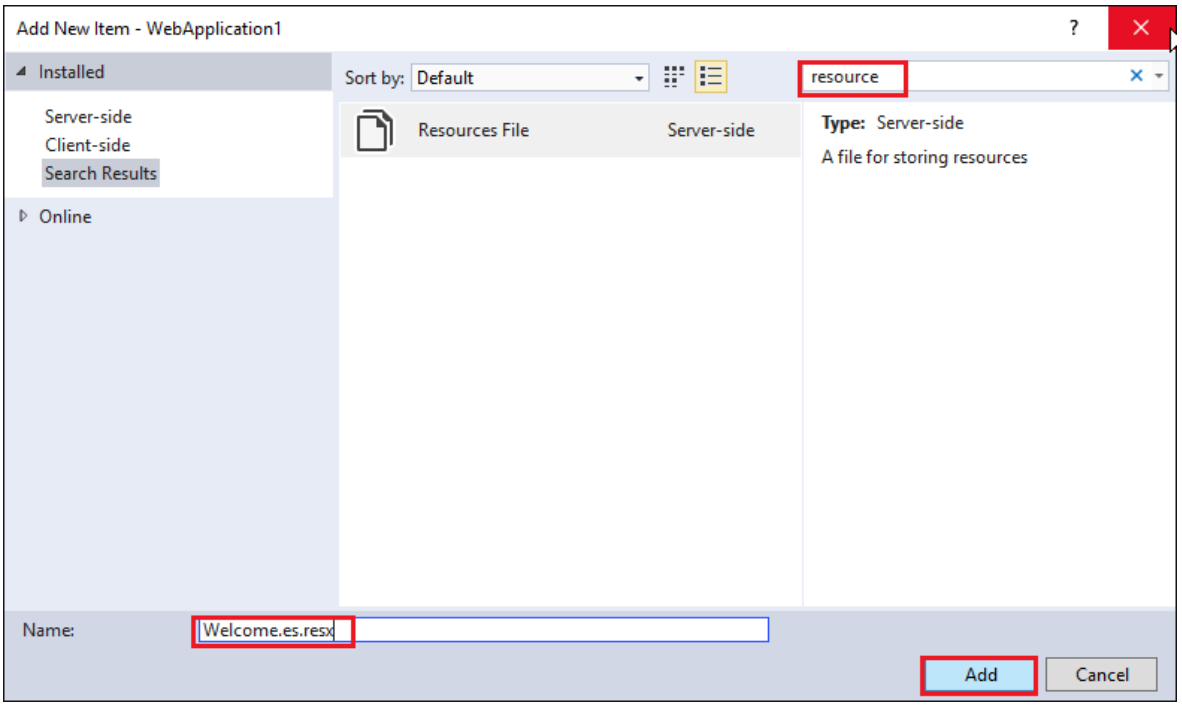
NOTE: ResX Viewer and Editor provides an alternate mechanism to work with resource files using Visual Studio Code.

A resource file is a useful mechanism for separating localizable strings from code. Translated strings for the non-default language are isolated in *.resx* resource files. For example, you might want to create a Spanish resource file named *Welcome.es.resx* containing translated strings. "es" is the language code for Spanish. To create this resource file in Visual Studio:

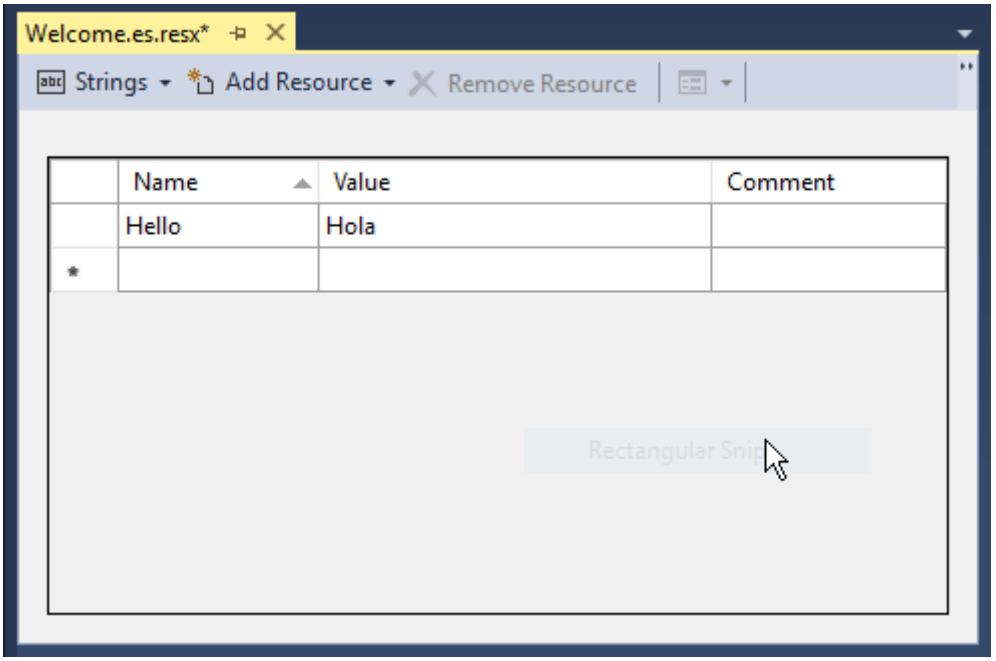
1. In **Solution Explorer**, right click on the folder that will contain the resource file, and then select **Add** > **New Item**.



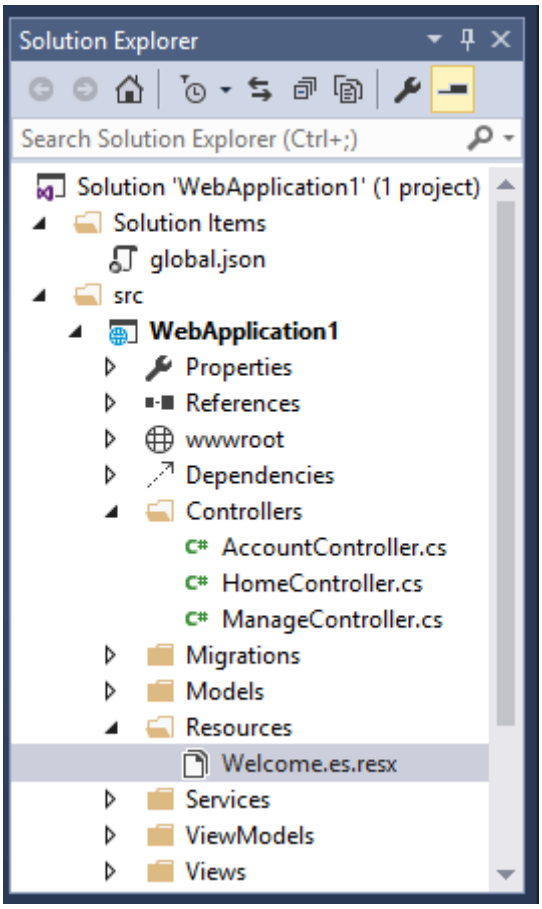
2. In the **Search installed templates** box, enter "resource" and name the file.



3. Enter the key value (native string) in the **Name** column and the translated string in the **Value** column.



Visual Studio shows the *Welcome.es.resx* file.



Resource file naming

Resources are named for the full type name of their class minus the assembly name. For example, a French resource in a project whose main assembly is `LocalizationWebsite.Web.dll` for the class `LocalizationWebsite.Web.Startup` would be named *Startup.fr.resx*. A resource for the class `LocalizationWebsite.Web.Controllers.HomeController` would be named *Controllers.HomeController.fr.resx*. If your targeted class's namespace isn't the same as the assembly name you will need the full type name. For example, in the sample project a resource for the type `ExtraNamespace.Tools` would be named *ExtraNamespace.Tools.fr.resx*.

In the sample project, the `ConfigureServices` method sets the `ResourcesPath` to "Resources", so the project relative path for the home controller's French resource file is *Resources/Controllers.HomeController.fr.resx*. Alternatively, you can use folders to organize resource files. For the home controller, the path would be *Resources/Controllers/HomeController.fr.resx*. If you don't use the `ResourcesPath` option, the *.resx* file would go in the project base directory. The resource file for `HomeController` would be named *Controllers.HomeController.fr.resx*. The choice of using the dot or path naming convention depends on how you want to organize your resource files.

Expand table

Resource name	Dot or path naming
Resources/Controllers.HomeController.fr.resx	Dot
Resources/Controllers/HomeController.fr.resx	Path

Resource files using `@inject IViewLocalizer` in Razor views follow a similar pattern. The resource file for a view can be named using either dot naming or path naming. Razor view resource files mimic the path of their associated view file. Assuming we set the `ResourcesPath` to "Resources", the French resource file associated with the `Views/Home/About.cshtml` view could be either of the following:

- Resources/Views/Home/About.fr.resx
- Resources/Views.Home.About.fr.resx

If you don't use the `ResourcesPath` option, the *.resx* file for a view would be located in the same folder as the view.

RootNamespaceAttribute

The RootNamespaceAttribute attribute provides the root namespace of an assembly when the root namespace of an assembly is different than the assembly name.

> ⚠ **Warning**
>
> This can occur when a project's name is not a valid .NET identifier. For instance `my-project-name.csproj` will use the root namespace `my_project_name` and the assembly name `my-project-name` leading to this error.

If the root namespace of an assembly is different than the assembly name:

- Localization does not work by default.
- Localization fails due to the way resources are searched for within the assembly. `RootNamespace` is a build-time value which is not available to the executing process.

If the `RootNamespace` is different from the `AssemblyName`, include the following in `AssemblyInfo.cs` (with parameter values replaced with the actual values):

```
C#
```

```
using System.Reflection;
using Microsoft.Extensions.Localization;

[assembly: ResourceLocation("Resource Folder Name")]
[assembly: RootNamespace("App Root Namespace")]
```

The preceding code enables the successful resolution of resx files.

Culture fallback behavior

When searching for a resource, localization engages in "culture fallback". Starting from the requested culture, if not found, it reverts to the parent culture of that culture. As an aside, the CultureInfo.Parent property represents the parent culture. This usually (but not always) means removing the national signifier from the language-and-culture code. For example, the dialect of Spanish spoken in Mexico is "es-MX". It has the parent "es"—Spanish non-specific to any country.

Imagine your site receives a request for a "Welcome" resource using culture "fr-CA". The localization system looks for the following resources, in order, and selects the first match:

- *Welcome.fr-CA.resx*
- *Welcome.fr.resx*
- *Welcome.resx* (if the `NeutralResourcesLanguage` is "fr-CA")

As an example, if you remove the ".fr" culture designator and you have the culture set to French, the default resource file is read and strings are localized. The Resource manager designates a default or fallback resource for when nothing meets your requested culture. If you want to just return the key when missing a resource for the requested culture you must not have a default resource file.

Generate resource files with Visual Studio

If you create a resource file in Visual Studio without a culture in the file name (for example, *Welcome.resx*), Visual Studio will create a C# class with a property for each string. That's usually not what you want with ASP.NET Core. You typically don't have a default *.resx* resource file (a *.resx* file without the culture name). We suggest you create the *.resx* file with a culture name (for example *Welcome.fr.resx*). When you create a *.resx* file with a culture name, Visual Studio won't generate the class file.

Add other cultures

Each language and culture combination (other than the default language) requires a unique resource file. You create resource files for different cultures and locales by creating new resource files in which the language codes are part of the file name (for example, **en-us**, **fr-ca**, and **en-gb**). These codes are placed between the file name and the *.resx* file extension, as in *Welcome.es-MX.resx* (Spanish/Mexico).

Next steps

Localizing an app also involves the following tasks:

- Make the app's content localizable.
- Implement a strategy to select the language/culture for each request

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- Globalization and localization in ASP.NET Core
- Make an ASP.NET Core app's content localizable
- Strategies for selecting language and culture in a localized ASP.NET Core app
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Localization.StarterWeb project used in the article.
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Implement a strategy to select the language/culture for each request in a localized ASP.NET Core app

Article • 10/25/2024

> **Note**
>
> This isn't the latest version of this article. For the current release, see the .NET 9 version of this article.

Hisham Bin Ateya, Damien Bowden, Bart Calixto, Nadeem Afana, and Rick Anderson

One task for localizing an app is to implement a strategy for selecting the appropriate culture for each response the app returns.

Configure Localization middleware

The current culture on a request is set in the localization Middleware. The localization middleware is enabled in `Program.cs`. The localization middleware must be configured before any middleware that might check the request culture (for example, `app.UseMvcWithDefaultRoute()`).

C#

```
builder.Services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[] { "en-US", "fr" };
    options.SetDefaultCulture(supportedCultures[0])
        .AddSupportedCultures(supportedCultures)
        .AddSupportedUICultures(supportedCultures);
});
```

UseRequestLocalization initializes a RequestLocalizationOptions object. On every request the list of RequestCultureProvider in the RequestLocalizationOptions is enumerated and the first provider that can successfully determine the request culture is used. The default providers come from the `RequestLocalizationOptions` class:

1. QueryStringRequestCultureProvider
2. CookieRequestCultureProvider

3. AcceptLanguageHeaderRequestCultureProvider

The default list goes from most specific to least specific. Later in the article you'll see how you can change the order and even add a custom culture provider. If none of the providers can determine the request culture, the DefaultRequestCulture is used.

QueryStringRequestCultureProvider

Some apps will use a query string to set the CultureInfo. For apps that use the cookie or Accept-Language header approach, adding a query string to the URL is useful for debugging and testing code. By default, the QueryStringRequestCultureProvider is registered as the first localization provider in the `RequestCultureProvider` list. You pass the query string parameters `culture` and `ui-culture`. The following example sets the specific culture (language and region) to Spanish/Mexico:

```
http://localhost:5000/?culture=es-MX&ui-culture=es-MX
```

If only `culture` or `ui-culture` is passed in, the query string provider sets both values using the one passed in. For example, setting just the culture will set both the `Culture` and the `UICulture`:

```
http://localhost:5000/?culture=es-MX
```

CookieRequestCultureProvider

Production apps will often provide a mechanism to set the culture with the ASP.NET Core culture cookie. Use the MakeCookieValue method to create a cookie.

The CookieRequestCultureProvider DefaultCookieName returns the default cookie name used to track the user's preferred culture information. The default cookie name is `.AspNetCore.Culture`.

The cookie format is `c=%LANGCODE%|uic=%LANGCODE%`, where `c` is `Culture` and `uic` is `UICulture`, for example:

```
```

```
c=en-UK|uic=en-US
```

If only one of culture info or UI culture is provided, the provided culture is used for both culture info and UI culture.

The Accept-Language HTTP header

The Accept-Language header is settable in most browsers and was originally intended to specify the user's language. This setting indicates what the browser has been set to send or has inherited from the underlying operating system. The Accept-Language HTTP header from a browser request isn't an infallible way to detect the user's preferred language (see Setting language preferences in a browser). A production app should include a way for a user to customize their choice of culture.

Set the Accept-Language HTTP header in Edge

1. Search **Settings** for **Preferred languages**.
2. The preferred languages are listed in the **Preferred languages** box.
3. Select **Add languages** to add to the list.
4. Select **More actions ...** next to a language to change the order of preference.

The Content-Language HTTP header

The Content-Language entity header:

- Is used to describe the language(s) intended for the audience.
- Allows a user to differentiate according to the users' own preferred language.

Entity headers are used in both HTTP requests and responses.

The `Content-Language` header can be added by setting the property ApplyCurrentCultureToResponseHeaders.

Adding the Content-Language header:

- Allows the RequestLocalizationMiddleware to set the `Content-Language` header with the `CurrentUICulture`.
- Eliminates the need to set the response header `Content-Language` explicitly.

C#

```
app.UseRequestLocalization(new RequestLocalizationOptions
{
    ApplyCurrentCultureToResponseHeaders = true
});
```

Apply the RouteDataRequest CultureProvider

The RouteDataRequestCultureProvider sets the culture based on the value of the `culture` route value. See Url culture provider using middleware as filters for information on:

- Using the middleware as filters feature of ASP.NET Core.
- How to use `RouteDataRequestCultureProvider` to set the culture of an app from the url.

See Applying the RouteDataRequest CultureProvider globally with middleware as filters for information on how to apply the `RouteDataRequestCultureProvider` globally.

Use a custom provider

Suppose you want to let your customers store their language and culture in your databases. You could write a provider to look up these values for the user. The following code shows how to add a custom provider:

C#

```
private const string enUSCulture = "en-US";

services.Configure<RequestLocalizationOptions>(options =>
{
    var supportedCultures = new[]
    {
        new CultureInfo(enUSCulture),
        new CultureInfo("fr")
    };

    options.DefaultRequestCulture = new RequestCulture(culture: enUSCulture,
uiCulture: enUSCulture);
    options.SupportedCultures = supportedCultures;
    options.SupportedUICultures = supportedCultures;

    options.AddInitialRequestCultureProvider(new
CustomRequestCultureProvider(async context =>
    {
```

```
        // My custom request culture logic
        return await Task.FromResult(new ProviderCultureResult("en"));
    }));
});
```

Use `RequestLocalizationOptions` to add or remove localization providers.

Change request culture providers order

RequestLocalizationOptions has three default request culture providers: QueryStringRequestCultureProvider, CookieRequestCultureProvider, and AcceptLanguageHeaderRequestCultureProvider. Use RequestLocalizationOptions.RequestCultureProviders property to change the order of these providers as shown in the following below:

C#

```
    app.UseRequestLocalization(options =>
    {
        var questStringCultureProvider = options.RequestCultureProviders[0];
        options.RequestCultureProviders.RemoveAt(0);
        options.RequestCultureProviders.Insert(1,
questStringCultureProvider);
    });
```

In the preceding example, the order of `QueryStringRequestCultureProvider` and `CookieRequestCultureProvider` is switched, so the `RequestLocalizationMiddleware` looks for the cultures from the cookies first, then the query string.

As previously mentioned, add a custom provider via AddInitialRequestCultureProvider which sets the order to `0`, so this provider takes the precedence over the others.

User override culture

The RequestLocalizationOptions.CultureInfoUseUserOverride property allows the app to decide whether or not to use non-default Windows settings for the CultureInfo DateTimeFormat and NumberFormat properties. This has ***no*** impact on Linux. This directly corresponds to UseUserOverride.

C#

```
    app.UseRequestLocalization(options =>
    {
```

```
        options.CultureInfoUseUserOverride = false;
    });
```

Set the culture programmatically

This sample **Localization.StarterWeb** project on GitHub contains UI to set the `Culture`. The `Views/Shared/_SelectLanguagePartial.cshtml` file allows you to select the culture from the list of supported cultures:

CSHTML

```
@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Http.Features
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options

@inject IViewLocalizer Localizer
@inject IOptions<RequestLocalizationOptions> LocOptions

@{
    var requestCulture = Context.Features.Get<IRequestCultureFeature>();
    var cultureItems = LocOptions.Value.SupportedUICultures
        .Select(c => new SelectListItem { Value = c.Name, Text =
c.DisplayName })
        .ToList();
    var returnUrl = string.IsNullOrEmpty(Context.Request.Path) ? "~/" : $"~
{Context.Request.Path.Value}";
}

<div title="@Localizer["Request culture provider:"]
@requestCulture?.Provider?.GetType().Name">
    <form id="selectLanguage" asp-controller="Home"
          asp-action="SetLanguage" asp-route-returnUrl="@returnUrl"
          method="post" class="form-horizontal" role="form">
        <label asp-
for="@requestCulture.RequestCulture.UICulture.Name">@Localizer["Language:"]
</label> <select name="culture"
          onchange="this.form.submit();"
          asp-for="@requestCulture.RequestCulture.UICulture.Name" asp-
items="cultureItems">
        </select>
    </form>
</div>
```

The `Views/Shared/_SelectLanguagePartial.cshtml` file is added to the `footer` section of the layout file so it will be available to all views:

CSHTML

```
<div class="container body-content" style="margin-top:60px">
    @RenderBody()
    <hr>
    <footer>
        <div class="row">
            <div class="col-md-6">
                <p>&copy; @System.DateTime.Now.Year - Localization</p>
            </div>
            <div class="col-md-6 text-right">
                @await Html.PartialAsync("_SelectLanguagePartial")
            </div>
        </div>
    </footer>
</div>
```

The `SetLanguage` method sets the culture cookie.

C#

```
[HttpPost]
public IActionResult SetLanguage(string culture, string returnUrl)
{
    Response.Cookies.Append(
        CookieRequestCultureProvider.DefaultCookieName,
        CookieRequestCultureProvider.MakeCookieValue(new
RequestCulture(culture)),
        new CookieOptions { Expires = DateTimeOffset.UtcNow.AddYears(1) }
    );

    return LocalRedirect(returnUrl);
}
```

You can't plug in the `_SelectLanguagePartial.cshtml` to sample code for this project. The **Localization.StarterWeb** project on GitHub has code to flow the `RequestLocalizationOptions` to a Razor partial through the Dependency Injection container.

Model binding route data and query strings

See Globalization behavior of model binding route data and query strings.

Next steps

Localizing an app also involves the following tasks:

- Make the app's content localizable.

- Provide localized resources for the languages and cultures the app supports

Additional resources

- Url culture provider using middleware as filters in ASP.NET Core
- Applying the RouteDataRequest CultureProvider globally with middleware as filters
- Globalization and localization in ASP.NET Core
- Make an ASP.NET Core app's content localizable
- Provide localized resources for languages and cultures in an ASP.NET Core app
- Troubleshoot ASP.NET Core localization
- Globalizing and localizing .NET applications
- Localization.StarterWeb project used in the article.
- Resources in .resx Files
- Microsoft Multilingual App Toolkit
- Localization & Generics

Configure portable object localization in ASP.NET Core

Article • 02/15/2023

By Hisham Bin Ateya and Sébastien Ros.

This article walks through the steps for using Portable Object (PO) files in an ASP.NET Core application with the Orchard Core framework.

Note: Orchard Core isn't a Microsoft product. Microsoft provides no support for this feature.

View or download sample code (how to download)

What is a PO file?

PO files are distributed as text files containing the translated strings for a given language. Some advantages of using PO files instead of *.resx* files include:

- PO files support pluralization; *.resx* files don't support pluralization.
- PO files aren't compiled like *.resx* files. As such, specialized tooling and build steps aren't required.
- PO files work well with collaborative online editing tools.

Example

The following sample PO file contains the translation for two strings in French, including one with its plural form:

fr.po

text

```
#: Pages/Index.cshtml:13
msgid "Hello world!"
msgstr "Bonjour le monde!"

msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

This example uses the following syntax:

- `#:`: A comment indicating the context of the string to be translated. The same string might be translated differently depending on where it's being used.
- `msgid`: The untranslated string.
- `msgstr`: The translated string.

For pluralization support, more entries can be defined.

- `msgid_plural`: The untranslated plural string.
- `msgstr[0]`: The translated string for the case 0.
- `msgstr[N]`: The translated string for the case N.

The PO file specification can be found here.

Configuring PO file support in ASP.NET Core

This example is based on an ASP.NET Core Web application generated from a Visual Studio 2022 project template.

Referencing the package

Add a reference to the `OrchardCore.Localization.Core` NuGet package.

The `.csproj` file now contains a line similar to the following (version number may vary):

XML

```xml
<PackageReference Include="OrchardCore.Localization.Core" Version="1.5.0" />
```

Registering the service

Add the required services to `Program.cs`:

C#

```
builder.Services.AddPortableObjectLocalization();

builder.Services
    .Configure<RequestLocalizationOptions>(options => options
        .AddSupportedCultures("fr", "cs")
        .AddSupportedUICultures("fr", "cs"));

builder.Services
```

```
    .AddRazorPages()
    .AddViewLocalization();
```

Add the following code to your Razor page of choice. `Index.cshtml` is used in this example.

CSHTML

```
@page
@using Microsoft.AspNetCore.Mvc.Localization
@inject IViewLocalizer Localizer
@{
    ViewData["Title"] = "Home";
}

<div class="text-center">
    <h1 class="display-4">Welcome</h1>
    <p>Learn about <a href="https://docs.microsoft.com/aspnet/core">building
Web apps with ASP.NET Core</a>.</p>
</div>

<p>@Localizer["Hello world!"]</p>
```

An `IViewLocalizer` instance is injected and used to translate the text "Hello world!".

Creating a PO file

Create a file named *<culture code>.po* in your application root folder. In this example, the file name is *fr.po* because the French language is used:

text

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

This file stores both the string to translate and the French-translated string. Translations revert to their parent culture, if necessary. In this example, the *fr.po* file is used if the requested culture is `fr-FR` or `fr-CA`.

Testing the application

Run your application, the text **Hello world!** is displayed.

Navigate to the URL `/Index?culture=fr-FR`. The text **Bonjour le monde!** is displayed.

Pluralization

PO files support pluralization forms, which is useful when the same string needs to be translated differently based on a cardinality. This task is made complicated by the fact that each language defines custom rules to select which string to use based on the cardinality.

The Orchard Localization package provides an API to invoke these different plural forms automatically.

Creating pluralization PO files

Add the following content to the previously mentioned *fr.po* file:

```text
msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Il y a un élément."
msgstr[1] "Il y a {0} éléments."
```

See What is a PO file? for an explanation of what each entry in this example represents.

Adding a language using different pluralization forms

English and French strings were used in the previous example. English and French have only two pluralization forms and share the same form rules, which is that a cardinality of one is mapped to the first plural form. Any other cardinality is mapped to the second plural form.

Not all languages share the same rules. This is illustrated with the Czech language, which has three plural forms.

Create the `cs.po` file as follows, and note how the pluralization needs three different translations:

```text
msgid "Hello world!"
msgstr "Ahoj světe!!"

msgid "There is one item."
msgid_plural "There are {0} items."
msgstr[0] "Existuje jedna položka."
```

```
msgstr[1] "Existují {0} položky."
msgstr[2] "Existuje {0} položek."
```

To accept Czech localizations, add `"cs"` to the list of supported cultures in the `Configure` method:

C#

```
builder.Services
    .Configure<RequestLocalizationOptions>(options => options
        .AddSupportedCultures("fr", "cs")
        .AddSupportedUICultures("fr", "cs"));
```

Edit the `Pages/Index.cshtml` file to render localized, plural strings for several cardinalities:

CSHTML

```
<p>@Localizer.Plural(1, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(2, "There is one item.", "There are {0} items.")</p>
<p>@Localizer.Plural(5, "There is one item.", "There are {0} items.")</p>
```

Note: In a real world scenario, a variable would be used to represent the count. Here, we repeat the same code with three different values to expose a specific case.

Upon switching cultures, you see the following:

For `/Index`:

HTML

```
There is one item.
There are 2 items.
There are 5 items.
```

For `/Index?culture=fr`:

HTML

```
Il y a un élément.
Il y a 2 éléments.
Il y a 5 éléments.
```

For `/Index?culture=cs`:

```
```

HTML

```
Existuje jedna položka.
Existují 2 položky.
Existuje 5 položek.
```

For the Czech culture, the three translations are different. The French and English cultures share the same construction for the two last translated strings.

Advanced tasks

Using additional arguments

The argument at index zero `{0}` always represents the count value. When invoking the `Plural` method it is possible to add additional arguments and their index will then start at one (`1`).

CSHTML

```
<p>@Localizer.Plural(count, "There is one item with the color {1}.", "There
are {0} items. The main color is {1}.", color)</p>
```

Contextualizing strings

Applications often contain the strings to be translated in several places. The same string may have a different translation in certain locations within an app (Razor views or class files). A PO file supports the notion of a file context, which can be used to categorize the string being represented. Using a file context, a string can be translated differently, depending on the file context (or lack of a file context).

The PO localization services use the name of the full class or the view that's used when translating a string. This is accomplished by setting the value on the `msgctxt` entry.

Consider a minor addition to the previous *fr.po* example. A Razor page located at `Pages/Index.cshtml` can be defined as the file context by setting the reserved `msgctxt` entry's value:

text

```
msgctxt "Views.Home.About"
msgid "Hello world!"
```

```
msgstr "Bonjour le monde!"
```

With the `msgctxt` set as such, text translation occurs when navigating to `/Index?culture=fr-FR`. The translation doesn't occur when navigating to `/Privacy?culture=fr-FR`.

When no specific entry is matched with a given file context, Orchard Core's fallback mechanism looks for an appropriate PO file without a context. Assuming there's no specific file context defined for `Pages/Privacy.cshtml`, navigating to `/Privacy?culture=fr-FR` loads a PO file such as:

text

```
msgid "Hello world!"
msgstr "Bonjour le monde!"
```

Changing the location of PO files

The default location of PO files can be changed in `Programs.cs`:

C#

```
services.AddPortableObjectLocalization(options => options.ResourcesPath =
"Localization");
```

In this example, the PO files are loaded from the *Localization* folder.

Implementing a custom logic for finding localization files

When more complex logic is needed to locate PO files, the `OrchardCore.Localization.PortableObject.ILocalizationFileLocationProvider` interface can be implemented and registered as a service. This is useful when PO files can be stored in varying locations or when the files have to be found within a hierarchy of folders.

Using a different default pluralized language

The package includes a `Plural` extension method that's specific to two plural forms. For languages requiring more plural forms, create an extension method. With an extension method, you won't need to provide any localization file for the default language — the original strings are already available directly in the code.

You can use the more generic `Plural(int count, string[] pluralForms, params object[] arguments)` overload which accepts a string array of translations.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Localization Extensibility

Article • 07/26/2024

By Hisham Bin Ateya

This article:

- Lists the extensibility points on the localization APIs.
- Provides instructions on how to extend ASP.NET Core app localization.

Extensible Points in Localization APIs

ASP.NET Core localization APIs are built to be extensible. Extensibility allows developers to customize the localization according to their needs. For instance, OrchardCore has a `POStringLocalizer`. `POStringLocalizer` describes in detail using Portable Object localization to use `PO` files to store localization resources.

This article lists the two main extensibility points that localization APIs provide:

- RequestCultureProvider
- IStringLocalizer

Localization Culture Providers

ASP.NET Core localization APIs have four default providers that can determine the current culture of an executing request:

- QueryStringRequestCultureProvider
- CookieRequestCultureProvider
- AcceptLanguageHeaderRequestCultureProvider
- CustomRequestCultureProvider

The preceding providers are described in detail in the Localization Middleware documentation. If the default providers don't meet your needs, build a custom provider using one of the following approaches:

Use CustomRequestCultureProvider

CustomRequestCultureProvider provides a custom RequestCultureProvider that uses a simple delegate to determine the current localization culture:

C#

```
options.AddInitialRequestCultureProvider(new
CustomRequestCultureProvider(async context =>
{
    var currentCulture = "en";
    var segments = context.Request.Path.Value.Split(new char[] { '/' },
        StringSplitOptions.RemoveEmptyEntries);

    if (segments.Length > 1 && segments[0].Length == 2)
    {
        currentCulture = segments[0];
    }

    var requestCulture = new ProviderCultureResult(currentCulture);

    return Task.FromResult(requestCulture);
}));
```

Use a new implementation of RequestCultureProvider

A new implementation of RequestCultureProvider can be created that determines the request culture information from a custom source. For example, the custom source can be a configuration file or database.

The following example shows `AppSettingsRequestCultureProvider`, which extends the RequestCultureProvider to determine the request culture information from `appsettings.json`:

C#

```
public class AppSettingsRequestCultureProvider : RequestCultureProvider
{
    public string CultureKey { get; set; } = "culture";

    public string UICultureKey { get; set; } = "ui-culture";

    public override Task<ProviderCultureResult>
DetermineProviderCultureResult(HttpContext httpContext)
    {
        if (httpContext == null)
        {
            throw new ArgumentNullException();
        }

        var configuration =
httpContext.RequestServices.GetService<IConfigurationRoot>();
        var culture = configuration[CultureKey];
        var uiCulture = configuration[UICultureKey];
```

```
        if (culture == null && uiCulture == null)
        {
            return Task.FromResult((ProviderCultureResult)null);
        }

        if (culture != null && uiCulture == null)
        {
            uiCulture = culture;
        }

        if (culture == null && uiCulture != null)
        {
            culture = uiCulture;
        }

        var providerResultCulture = new ProviderCultureResult(culture,
uiCulture);

        return Task.FromResult(providerResultCulture);
    }
}
```

Localization resources

ASP.NET Core localization provides ResourceManagerStringLocalizer. ResourceManagerStringLocalizer is an implementation of IStringLocalizer that uses `resx` to store localization resources.

You aren't limited to using `resx` files. By implementing `IStringLocalizer`, any data source can be used.

The following example projects implement IStringLocalizer:

- EFStringLocalizer
- JsonStringLocalizer
- SqlLocalizer

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Troubleshoot ASP.NET Core localization

Article • 05/03/2024

By Hisham Bin Ateya

This article provides instructions on how to diagnose ASP.NET Core app localization issues.

Localization configuration issues

Localization middleware order
The app may not localize because the localization middleware isn't ordered as expected.

To resolve this issue, ensure that localization middleware is registered before MVC middleware. Otherwise, the localization middleware isn't applied.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddLocalization(options => options.ResourcesPath =
"Resources");

    services.AddMvc();
}
```

Localization resources path not found

Supported Cultures in RequestCultureProvider don't match with registered once

Resource file naming issues

ASP.NET Core has predefined rules and guidelines for localization resources file naming, which are described in Globalization and localization in ASP.NET Core.

Missing resources

Common causes of resources not being found include:

- Resource names are misspelled in either the .NET XML resource file (`.resx`) or the localizer request.

- The resource is missing from the resource file for some languages, but exists in others.
- If you're still having trouble, check the localization log messages (logged at the `Debug` log level) for more details about the missing resources.

> **Tip**
>
> When using **CookieRequestCultureProvider**, verify single quotes aren't used with the cultures inside the localization cookie value. For example, `c='en-UK'|uic='en-US'` is an invalid cookie value, while `c=en-UK|uic=en-US` is valid.

Resources and class libraries issues

ASP.NET Core by default provides a way to allow the class libraries to find their resource files via ResourceLocationAttribute.

Common issues with class libraries include:

- Missing the ResourceLocationAttribute in a class library prevents ResourceManagerStringLocalizerFactory from discovering the resources.
- Resource file naming. For more information, see the Resource file naming issues section.
- Changing the root namespace of the class library. For more information, see the Root namespace issues section.

`CustomRequestCultureProvider` doesn't work as expected

The RequestLocalizationOptions class has three default providers:

- QueryStringRequestCultureProvider
- CookieRequestCultureProvider
- AcceptLanguageHeaderRequestCultureProvider

The CustomRequestCultureProvider allows you to customize how the localization culture is provided. The CustomRequestCultureProvider is used when the default providers don't meet your requirements.

A common reason for a custom provider not working properly is that it isn't the first provider in the RequestCultureProviders list. To resolve this issue:

- Insert the custom provider at position zero in the RequestCultureProviders list:

```
C#
```

```
options.AddInitialRequestCultureProvider(
    new CustomRequestCultureProvider(async context =>
    {
        // My custom request culture logic
        return new ProviderCultureResult("en");
    }));
```

- Use the AddInitialRequestCultureProvider extension method to set the custom provider as the initial provider.

Root namespace issues

When the root namespace of an assembly is different than the assembly name, localization doesn't work by default. To avoid this issue use the RootNamespace attribute, which is described in Globalization and localization in ASP.NET Core.

> ⚠ **Warning**
>
> A root namespace issue can occur when a project's name isn't a valid .NET identifier. For instance, `my-project-name.csproj` uses the root namespace `my_project_name` and the assembly name `my-project-name`, which results in this error.

Resources and build action

If you use resource files for localization, it's important that they have an appropriate build action. Use **Embedded Resource**; otherwise, the `ResourceStringLocalizer` isn't able to find these resources.

Location override using "Sensors" pane in developer tools

When using the location override using the **Sensors** pane in Google Chrome or Microsoft Edge developer tools, the fallback language is reset after prerendering. Avoid setting the language using the **Sensors** pane when testing. Set the language using the browser's language settings.

For more information, see Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707).

GitHub issues with helpful problem solving tips

- Please add more info about shared files (dotnet/AspNetCore.Docs #28674
- Blazor Localization does not work with InteractiveServer (dotnet/aspnetcore #53707) (Location override using "Sensors" pane)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Model Binding in ASP.NET Core

Article • 06/22/2023

This article explains what model binding is, how it works, and how to customize its behavior.

What is Model binding

Controllers and Razor pages work with data that comes from HTTP requests. For example, route data may provide a record key, and posted form fields may provide values for the properties of the model. Writing code to retrieve each of these values and convert them from strings to .NET types would be tedious and error-prone. Model binding automates this process. The model binding system:

- Retrieves data from various sources such as route data, form fields, and query strings.
- Provides the data to controllers and Razor pages in method parameters and public properties.
- Converts string data to .NET types.
- Updates properties of complex types.

Example

Suppose you have the following action method:

C#

```
[HttpGet("{id}")]
public ActionResult<Pet> GetById(int id, bool dogsOnly)
```

And the app receives a request with this URL:

HTTP

```
https://contoso.com/api/pets/2?DogsOnly=true
```

Model binding goes through the following steps after the routing system selects the action method:

- Finds the first parameter of `GetById`, an integer named `id`.

- Looks through the available sources in the HTTP request and finds `id` = "2" in route data.
- Converts the string "2" into integer 2.
- Finds the next parameter of `GetById`, a boolean named `dogsOnly`.
- Looks through the sources and finds "DogsOnly=true" in the query string. Name matching is not case-sensitive.
- Converts the string "true" into boolean `true`.

The framework then calls the `GetById` method, passing in 2 for the `id` parameter, and `true` for the `dogsOnly` parameter.

In the preceding example, the model binding targets are method parameters that are simple types. Targets may also be the properties of a complex type. After each property is successfully bound, model validation occurs for that property. The record of what data is bound to the model, and any binding or validation errors, is stored in ControllerBase.ModelState or PageModel.ModelState. To find out if this process was successful, the app checks the ModelState.IsValid flag.

Targets

Model binding tries to find values for the following kinds of targets:

- Parameters of the controller action method that a request is routed to.
- Parameters of the Razor Pages handler method that a request is routed to.
- Public properties of a controller or `PageModel` class, if specified by attributes.

[BindProperty] attribute

Can be applied to a public property of a controller or `PageModel` class to cause model binding to target that property:

C#

```
public class EditModel : PageModel
{
    [BindProperty]
    public Instructor? Instructor { get; set; }

    // ...
}
```

[BindProperties] attribute

Can be applied to a controller or `PageModel` class to tell model binding to target all public properties of the class:

```
C#

[BindProperties]
public class CreateModel : PageModel
{
    public Instructor? Instructor { get; set; }

    // ...
}
```

Model binding for HTTP GET requests

By default, properties are not bound for HTTP GET requests. Typically, all you need for a GET request is a record ID parameter. The record ID is used to look up the item in the database. Therefore, there is no need to bind a property that holds an instance of the model. In scenarios where you do want properties bound to data from GET requests, set the `SupportsGet` property to `true`:

```
C#

[BindProperty(Name = "ai_user", SupportsGet = true)]
public string? ApplicationInsightsCookie { get; set; }
```

Model binding simple and complex types

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string using TypeConverter or a `TryParse` method. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter` or `TryParse`. We recommend creating a type converter or using `TryParse` for a `string` to `SomeType` conversion that doesn't require external resources or multiple inputs.

Sources

By default, model binding gets data in the form of key-value pairs from the following sources in an HTTP request:

1. Form fields
2. The request body (For controllers that have the [ApiController] attribute.)

3. Route data
4. Query string parameters
5. Uploaded files

For each target parameter or property, the sources are scanned in the order indicated in the preceding list. There are a few exceptions:

- Route data and query string values are used only for simple types.
- Uploaded files are bound only to target types that implement `IFormFile` or `IEnumerable<IFormFile>`.

If the default source is not correct, use one of the following attributes to specify the source:

- [FromQuery] - Gets values from the query string.
- [FromRoute] - Gets values from route data.
- [FromForm] - Gets values from posted form fields.
- [FromBody] - Gets values from the request body.
- [FromHeader] - Gets values from HTTP headers.

These attributes:

- Are added to model properties individually and not to the model class, as in the following example:

```
C#

public class Instructor
{
    public int Id { get; set; }

    [FromQuery(Name = "Note")]
    public string? NoteFromQueryString { get; set; }

    // ...
}
```

- Optionally accept a model name value in the constructor. This option is provided in case the property name doesn't match the value in the request. For instance, the value in the request might be a header with a hyphen in its name, as in the following example:

```
C#

public void OnGet([FromHeader(Name = "Accept-Language")] string
```

```
language)
```

[FromBody] attribute

Apply the `[FromBody]` attribute to a parameter to populate its properties from the body of an HTTP request. The ASP.NET Core runtime delegates the responsibility of reading the body to an input formatter. Input formatters are explained later in this article.

When `[FromBody]` is applied to a complex type parameter, any binding source attributes applied to its properties are ignored. For example, the following `Create` action specifies that its `pet` parameter is populated from the body:

C#

```
public ActionResult<Pet> Create([FromBody] Pet pet)
```

The `Pet` class specifies that its `Breed` property is populated from a query string parameter:

C#

```
public class Pet
{
    public string Name { get; set; } = null!;

    [FromQuery] // Attribute is ignored.
    public string Breed { get; set; } = null!;
}
```

In the preceding example:

- The `[FromQuery]` attribute is ignored.
- The `Breed` property is not populated from a query string parameter.

Input formatters read only the body and don't understand binding source attributes. If a suitable value is found in the body, that value is used to populate the `Breed` property.

Don't apply `[FromBody]` to more than one parameter per action method. Once the request stream is read by an input formatter, it's no longer available to be read again for binding other `[FromBody]` parameters.

Additional sources

Source data is provided to the model binding system by *value providers*. You can write and register custom value providers that get data for model binding from other sources. For example, you might want data from cookies or session state. To get data from a new source:

- Create a class that implements `IValueProvider`.
- Create a class that implements `IValueProviderFactory`.
- Register the factory class in `Program.cs`.

The sample includes a value provider and factory example that gets values from cookies. Register custom value provider factories in `Program.cs`:

```
C#

builder.Services.AddControllers(options =>
{
    options.ValueProviderFactories.Add(new CookieValueProviderFactory());
});
```

The preceding code puts the custom value provider after all built-in value providers. To make it the first in the list, call `Insert(0, new CookieValueProviderFactory())` instead of `Add`.

No source for a model property

By default, a model state error isn't created if no value is found for a model property. The property is set to null or a default value:

- Nullable simple types are set to `null`.
- Non-nullable value types are set to `default(T)`. For example, a parameter `int id` is set to 0.
- For complex Types, model binding creates an instance by using the default constructor, without setting properties.
- Arrays are set to `Array.Empty<T>()`, except that `byte[]` arrays are set to `null`.

If model state should be invalidated when nothing is found in form fields for a model property, use the [BindRequired] attribute.

Note that this `[BindRequired]` behavior applies to model binding from posted form data, not to JSON or XML data in a request body. Request body data is handled by input formatters.

Type conversion errors

If a source is found but can't be converted into the target type, model state is flagged as invalid. The target parameter or property is set to null or a default value, as noted in the previous section.

In an API controller that has the `[ApiController]` attribute, invalid model state results in an automatic HTTP 400 response.

In a Razor page, redisplay the page with an error message:

```
C#

public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // ...

    return RedirectToPage("./Index");
}
```

When the page is redisplayed by the preceding code, the invalid input isn't shown in the form field. This is because the model property has been set to null or a default value. The invalid input does appear in an error message. If you want to redisplay the bad data in the form field, consider making the model property a string and doing the data conversion manually.

The same strategy is recommended if you don't want type conversion errors to result in model state errors. In that case, make the model property a string.

Simple types

See Model binding simple and complex types for explanation of simple and complex types.

The simple types that the model binder can convert source strings into include the following:

- Boolean
- Byte, SByte
- Char

- DateOnly
- DateTime
- DateTimeOffset
- Decimal
- Double
- Enum
- Guid
- Int16, Int32, Int64
- Single
- TimeOnly
- TimeSpan
- UInt16, UInt32, UInt64
- Uri
- Version

Bind with `IParsable<T>.TryParse`

The IParsable<TSelf>.TryParse API supports binding controller action parameter values:

```
C#

public static bool TryParse (string? s, IFormatProvider? provider, out TSelf
result);
```

The following `DateRange` class implements IParsable<TSelf> to support binding a date range:

```
C#

public class DateRange : IParsable<DateRange>
{
    public DateOnly? From { get; init; }
    public DateOnly? To { get; init; }

    public static DateRange Parse(string value, IFormatProvider? provider)
    {
        if (!TryParse(value, provider, out var result))
        {
           throw new ArgumentException("Could not parse supplied value.",
nameof(value));
        }

        return result;
    }

    public static bool TryParse(string? value,
```

```
                                IFormatProvider? provider, out DateRange
dateRange)
    {
        var segments = value?.Split(',',
StringSplitOptions.RemoveEmptyEntries
                                    | StringSplitOptions.TrimEntries);

        if (segments?.Length == 2
            && DateOnly.TryParse(segments[0], provider, out var fromDate)
            && DateOnly.TryParse(segments[1], provider, out var toDate))
        {
            dateRange = new DateRange { From = fromDate, To = toDate };
            return true;
        }

        dateRange = new DateRange { From = default, To = default };
        return false;
    }
}
```

The preceding code:

- Converts a string representing two dates to a `DateRange` object
- The model binder uses the IParsable<TSelf>.TryParse method to bind the `DateRange`.

The following controller action uses the `DateRange` class to bind a date range:

C#

```
// GET /WeatherForecast/ByRange?range=7/24/2022,07/26/2022
public IActionResult ByRange([FromQuery] DateRange range)
{
    if (!ModelState.IsValid)
        return View("Error", ModelState.Values.SelectMany(v => v.Errors));

    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Where(wf => DateOnly.FromDateTime(wf.Date) >= range.From
                    && DateOnly.FromDateTime(wf.Date) <= range.To)
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d"),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int)(wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });
```

```
    return View("Index", weatherForecasts);
}
```

The following `Locale` class implements IParsable<TSelf> to support binding to `CultureInfo`:

C#

```
public class Locale : CultureInfo, IParsable<Locale>
{
    public Locale(string culture) : base(culture)
    {
    }

    public static Locale Parse(string value, IFormatProvider? provider)
    {
        if (!TryParse(value, provider, out var result))
        {
           throw new ArgumentException("Could not parse supplied value.",
nameof(value));
        }

        return result;
    }

    public static bool TryParse([NotNullWhen(true)] string? value,
                                IFormatProvider? provider, out Locale
locale)
    {
        if (value is null)
        {
            locale = new Locale(CurrentCulture.Name);
            return false;
        }

        try
        {
            locale = new Locale(value);
            return true;
        }
        catch (CultureNotFoundException)
        {
            locale = new Locale(CurrentCulture.Name);
            return false;
        }
    }
}
```

The following controller action uses the `Locale` class to bind a `CultureInfo` string:

C#

```
// GET /en-GB/WeatherForecast
public IActionResult Index([FromRoute] Locale locale)
{
    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d", locale),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int)(wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });

    return View(weatherForecasts);
}
```

The following controller action uses the `DateRange` and `Locale` classes to bind a date range with `CultureInfo`:

C#

```
// GET /af-ZA/WeatherForecast/RangeByLocale?range=2022-07-24,2022-07-29
public IActionResult RangeByLocale([FromRoute] Locale locale, [FromQuery]
string range)
{
    if (!ModelState.IsValid)
        return View("Error", ModelState.Values.SelectMany(v => v.Errors));

    if (!DateRange.TryParse(range, locale, out DateRange rangeResult))
    {
        ModelState.TryAddModelError(nameof(range),
            $"Invalid date range: {range} for locale {locale.DisplayName}");

        return View("Error", ModelState.Values.SelectMany(v => v.Errors));
    }

    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Where(wf => DateOnly.FromDateTime(wf.Date) >= rangeResult.From
                     && DateOnly.FromDateTime(wf.Date) <= rangeResult.To)
```

```
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d", locale),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int) (wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });

    return View("Index", weatherForecasts);
}
```

The API sample app on GitHub shows the preceding sample for an API controller.

Bind with `TryParse`

The `TryParse` API supports binding controller action parameter values:

C#

```
public static bool TryParse(string value, T out result);
public static bool TryParse(string value, IFormatProvider provider, T out
result);
```

IParsable<T>.TryParse is the recommended approach for parameter binding because unlike `TryParse`, it doesn't depend on reflection.

The following `DateRangeTP` class implements `TryParse`:

C#

```
public class DateRangeTP
{
    public DateOnly? From { get; }
    public DateOnly? To { get; }

    public DateRangeTP(string from, string to)
    {
        if (string.IsNullOrEmpty(from))
            throw new ArgumentNullException(nameof(from));
        if (string.IsNullOrEmpty(to))
            throw new ArgumentNullException(nameof(to));

        From = DateOnly.Parse(from);
        To = DateOnly.Parse(to);
    }

    public static bool TryParse(string? value, out DateRangeTP? result)
    {
        var range = value?.Split(',', StringSplitOptions.RemoveEmptyEntries
```

```
| StringSplitOptions.TrimEntries);
        if (range?.Length != 2)
        {
            result = default;
            return false;
        }

        result = new DateRangeTP(range[0], range[1]);
        return true;
    }
}
```

The following controller action uses the `DateRangeTP` class to bind a date range:

C#

```
// GET /WeatherForecast/ByRangeTP?range=7/24/2022,07/26/2022
public IActionResult ByRangeTP([FromQuery] DateRangeTP range)
{
    if (!ModelState.IsValid)
        return View("Error", ModelState.Values.SelectMany(v => v.Errors));

    var weatherForecasts = Enumerable
        .Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .Where(wf => DateOnly.FromDateTime(wf.Date) >= range.From
                    && DateOnly.FromDateTime(wf.Date) <= range.To)
        .Select(wf => new WeatherForecastViewModel
        {
            Date = wf.Date.ToString("d"),
            TemperatureC = wf.TemperatureC,
            TemperatureF = 32 + (int)(wf.TemperatureC / 0.5556),
            Summary = wf.Summary
        });

    return View("Index", weatherForecasts);
}
```

Complex types

A complex type must have a public default constructor and public writable properties to bind. When model binding occurs, the class is instantiated using the public default constructor.

For each property of the complex type, model binding looks through the sources for the name pattern *prefix.property_name*. If nothing is found, it looks for just *property_name* without the prefix. The decision to use the prefix isn't made per property. For example, with a query containing `?Instructor.Id=100&Name=foo`, bound to method `OnGet(Instructor instructor)`, the resulting object of type `Instructor` contains:

- `Id` set to `100`.
- `Name` set to `null`. Model binding expects `Instructor.Name` because `Instructor.Id` was used in the preceding query parameter.

For binding to a parameter, the prefix is the parameter name. For binding to a `PageModel` public property, the prefix is the public property name. Some attributes have a `Prefix` property that lets you override the default usage of parameter or property name.

For example, suppose the complex type is the following `Instructor` class:

C#

```
public class Instructor
{
    public int ID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

Prefix = parameter name

If the model to be bound is a parameter named `instructorToUpdate`:

C#

```
public IActionResult OnPost(int? id, Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `instructorToUpdate.ID`. If that isn't found, it looks for `ID` without a prefix.

Prefix = property name

If the model to be bound is a property named `Instructor` of the controller or `PageModel` class:

C#

```
[BindProperty]
public Instructor Instructor { get; set; }
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Custom prefix

If the model to be bound is a parameter named `instructorToUpdate` and a `Bind` attribute specifies `Instructor` as the prefix:

C#

```
public IActionResult OnPost(
    int? id, [Bind(Prefix = "Instructor")] Instructor instructorToUpdate)
```

Model binding starts by looking through the sources for the key `Instructor.ID`. If that isn't found, it looks for `ID` without a prefix.

Attributes for complex type targets

Several built-in attributes are available for controlling model binding of complex types:

- [Bind]
- [BindRequired]
- [BindNever]

> ⚠ **Warning**
>
> These attributes affect model binding when posted form data is the source of values. They do ***not*** affect input formatters, which process posted JSON and XML request bodies. Input formatters are explained later in this article.

[Bind] attribute

Can be applied to a class or a method parameter. Specifies which properties of a model should be included in model binding. `[Bind]` does ***not*** affect input formatters.

In the following example, only the specified properties of the `Instructor` model are bound when any handler or action method is called:

C#

```
[Bind("LastName,FirstMidName,HireDate")]
public class Instructor
```

In the following example, only the specified properties of the `Instructor` model are bound when the `OnPost` method is called:

C#

```
[HttpPost]
public IActionResult OnPost(
    [Bind("LastName,FirstMidName,HireDate")] Instructor instructor)
```

The `[Bind]` attribute can be used to protect against overposting in *create* scenarios. It doesn't work well in edit scenarios because excluded properties are set to null or a default value instead of being left unchanged. For protection against overposting, view models are recommended rather than the `[Bind]` attribute. For more information, see Security note about overposting.

[ModelBinder] attribute

ModelBinderAttribute can be applied to types, properties, or parameters. It allows specifying the type of model binder used to bind the specific instance or type. For example:

C#

```
[HttpPost]
public IActionResult OnPost(
    [ModelBinder<MyInstructorModelBinder>] Instructor instructor)
```

The `[ModelBinder]` attribute can also be used to change the name of a property or parameter when it's being model bound:

C#

```
public class Instructor
{
    [ModelBinder(Name = "instructor_id")]
    public string Id { get; set; }

    // ...
}
```

[BindRequired] attribute

Causes model binding to add a model state error if binding cannot occur for a model's property. Here's an example:

```
C#

public class InstructorBindRequired
{
    // ...

    [BindRequired]
    public DateTime HireDate { get; set; }
}
```

See also the discussion of the `[Required]` attribute in Model validation.

[BindNever] attribute

Can be applied to a property or a type. Prevents model binding from setting a model's property. When applied to a type, the model binding system excludes all properties the type defines. Here's an example:

```
C#

public class InstructorBindNever
{
    [BindNever]
    public int Id { get; set; }

    // ...
}
```

Collections

For targets that are collections of simple types, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the parameter to be bound is an array named `selectedCourses`:

  ```
  C#

  public IActionResult OnPost(int? id, int[] selectedCourses)
  ```

- Form or query string data can be in one of the following formats:

```
selectedCourses=1050&selectedCourses=2000
```

```
selectedCourses[0]=1050&selectedCourses[1]=2000
```

```
[0]=1050&[1]=2000
```

```
selectedCourses[a]=1050&selectedCourses[b]=2000&selectedCourses.index=a
&selectedCourses.index=b
```

```
[a]=1050&[b]=2000&index=a&index=b
```

  Avoid binding a parameter or a property named `index` or `Index` if it is adjacent to a collection value. Model binding attempts to use `index` as the index for the collection which might result in incorrect binding. For example, consider the following action:

```
C#
public IActionResult Post(string index, List<Product> products)
```

  In the preceding code, the `index` query string parameter binds to the `index` method parameter and also is used to bind the product collection. Renaming the `index` parameter or using a model binding attribute to configure binding avoids this issue:

```
C#
public IActionResult Post(string productIndex, List<Product> products)
```

- The following format is supported only in form data:

```
selectedCourses[]=1050&selectedCourses[]=2000
```

- For all of the preceding example formats, model binding passes an array of two items to the `selectedCourses` parameter:
  - selectedCourses[0]=1050
  - selectedCourses[1]=2000

  Data formats that use subscript numbers (... [0] ... [1] ...) must ensure that they are numbered sequentially starting at zero. If there are any gaps in subscript numbering, all items after the gap are ignored. For example, if the subscripts are 0 and 2 instead of 0 and 1, the second item is ignored.

Dictionaries

For `Dictionary` targets, model binding looks for matches to *parameter_name* or *property_name*. If no match is found, it looks for one of the supported formats without the prefix. For example:

- Suppose the target parameter is a `Dictionary<int, string>` named `selectedCourses`:

```
C#

public IActionResult OnPost(int? id, Dictionary<int, string>
selectedCourses)
```

- The posted form or query string data can look like one of the following examples:

```
selectedCourses[1050]=Chemistry&selectedCourses[2000]=Economics
```

```
[1050]=Chemistry&selectedCourses[2000]=Economics
```

```
selectedCourses[0].Key=1050&selectedCourses[0].Value=Chemistry&
selectedCourses[1].Key=2000&selectedCourses[1].Value=Economics
```

```
[0].Key=1050&[0].Value=Chemistry&[1].Key=2000&[1].Value=Economics
```

- For all of the preceding example formats, model binding passes a dictionary of two items to the `selectedCourses` parameter:
  - selectedCourses["1050"]="Chemistry"
  - selectedCourses["2000"]="Economics"

Constructor binding and record types

Model binding requires that complex types have a parameterless constructor. Both `System.Text.Json` and `Newtonsoft.Json` based input formatters support deserialization of classes that don't have a parameterless constructor.

Record types are a great way to succinctly represent data over the network. ASP.NET Core supports model binding and validating record types with a single constructor:

C#

```
public record Person(
    [Required] string Name, [Range(0, 150)] int Age, [BindNever] int Id);

public class PersonController
{
    public IActionResult Index() => View();

    [HttpPost]
    public IActionResult Index(Person person)
    {
        // ...
    }
}
```

`Person/Index.cshtml`:

CSHTML

```
@model Person

<label>Name: <input asp-for="Name" /></label>
<br />
<label>Age: <input asp-for="Age" /></label>
```

When validating record types, the runtime searches for binding and validation metadata specifically on parameters rather than on properties.

The framework allows binding to and validating record types:

```
C#

public record Person([Required] string Name, [Range(0, 100)] int Age);
```

For the preceding to work, the type must:

- Be a record type.
- Have exactly one public constructor.
- Contain parameters that have a property with the same name and type. The names must not differ by case.

POCOs without parameterless constructors

POCOs that do not have parameterless constructors can't be bound.

The following code results in an exception saying that the type must have a parameterless constructor:

```
C#

public class Person(string Name)

public record Person([Required] string Name, [Range(0, 100)] int Age)
{
    public Person(string Name) : this (Name, 0);
}
```

Record types with manually authored constructors

Record types with manually authored constructors that look like primary constructors work

```
C#

public record Person
{
    public Person([Required] string Name, [Range(0, 100)] int Age)
        => (this.Name, this.Age) = (Name, Age);

    public string Name { get; set; }
```

```
    public int Age { get; set; }
}
```

Record types, validation and binding metadata

For record types, validation and binding metadata on parameters is used. Any metadata on properties is ignored

C#

```
public record Person (string Name, int Age)
{
   [BindProperty(Name = "SomeName")] // This does not get used
   [Required] // This does not get used
   public string Name { get; init; }
}
```

Validation and metadata

Validation uses metadata on the parameter but uses the property to read the value. In the ordinary case with primary constructors, the two would be identical. However, there are ways to defeat it:

C#

```
public record Person([Required] string Name)
{
    private readonly string _name;

    // The following property is never null.
    // However this object could have been constructed as "new
Person(null)".
    public string Name { get; init => _name = value ?? string.Empty; }
}
```

TryUpdateModel does not update parameters on a record type

C#

```
public record Person(string Name)
{
    public int Age { get; set; }
}
```

```
var person = new Person("initial-name");
TryUpdateModel(person, ...);
```

In this case, MVC will not attempt to bind `Name` again. However, `Age` is allowed to be updated

Globalization behavior of model binding route data and query strings

The ASP.NET Core route value provider and query string value provider:

- Treat values as invariant culture.
- Expect that URLs are culture-invariant.

In contrast, values coming from form data undergo a culture-sensitive conversion. This is by design so that URLs are shareable across locales.

To make the ASP.NET Core route value provider and query string value provider undergo a culture-sensitive conversion:

- Inherit from IValueProviderFactory
- Copy the code from QueryStringValueProviderFactory or RouteValueValueProviderFactory
- Replace the culture value passed to the value provider constructor with CultureInfo.CurrentCulture
- Replace the default value provider factory in MVC options with your new one:

C#

```
public class CultureQueryStringValueProviderFactory : IValueProviderFactory
{
    public Task CreateValueProviderAsync(ValueProviderFactoryContext
context)
    {
        _ = context ?? throw new ArgumentNullException(nameof(context));

        var query = context.ActionContext.HttpContext.Request.Query;
        if (query?.Count > 0)
        {
            context.ValueProviders.Add(
                new QueryStringValueProvider(
                    BindingSource.Query,
                    query,
                    CultureInfo.CurrentCulture));
        }
```

```
        return Task.CompletedTask;
    }
}
```

C#

```
builder.Services.AddControllers(options =>
{
    var index = options.ValueProviderFactories.IndexOf(

options.ValueProviderFactories.OfType<QueryStringValueProviderFactory>()
            .Single());

    options.ValueProviderFactories[index] =
        new CultureQueryStringValueProviderFactory();
});
```

Special data types

There are some special data types that model binding can handle.

IFormFile and IFormFileCollection

An uploaded file included in the HTTP request. Also supported is `IEnumerable<IFormFile>` for multiple files.

CancellationToken

Actions can optionally bind a `CancellationToken` as a parameter. This binds RequestAborted that signals when the connection underlying the HTTP request is aborted. Actions can use this parameter to cancel long running async operations that are executed as part of the controller actions.

FormCollection

Used to retrieve all the values from posted form data.

Input formatters

Data in the request body can be in JSON, XML, or some other format. To parse this data, model binding uses an *input formatter* that is configured to handle a particular content

type. By default, ASP.NET Core includes JSON based input formatters for handling JSON data. You can add other formatters for other content types.

ASP.NET Core selects input formatters based on the Consumes attribute. If no attribute is present, it uses the Content-Type header.

To use the built-in XML input formatters:

- In `Program.cs`, call AddXmlSerializerFormatters or AddXmlDataContractSerializerFormatters.

  C#

  ```
  builder.Services.AddControllers()
      .AddXmlSerializerFormatters();
  ```

- Apply the `Consumes` attribute to controller classes or action methods that should expect XML in the request body.

  C#

  ```
  [HttpPost]
  [Consumes("application/xml")]
  public ActionResult<Pet> Create(Pet pet)
  ```

  For more information, see Introducing XML Serialization.

Customize model binding with input formatters

An input formatter takes full responsibility for reading data from the request body. To customize this process, configure the APIs used by the input formatter. This section describes how to customize the `System.Text.Json`-based input formatter to understand a custom type named `ObjectId`.

Consider the following model, which contains a custom `ObjectId` property:

C#

```
public class InstructorObjectId
{
    [Required]
    public ObjectId ObjectId { get; set; } = null!;
}
```

To customize the model binding process when using `System.Text.Json`, create a class derived from JsonConverter<T>:

```
C#

internal class ObjectIdConverter : JsonConverter<ObjectId>
{
    public override ObjectId Read(
        ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions
options)
        => new(JsonSerializer.Deserialize<int>(ref reader, options));

    public override void Write(
        Utf8JsonWriter writer, ObjectId value, JsonSerializerOptions
options)
        => writer.WriteNumberValue(value.Id);
}
```

To use a custom converter, apply the JsonConverterAttribute attribute to the type. In the following example, the `ObjectId` type is configured with `ObjectIdConverter` as its custom converter:

```
C#

[JsonConverter(typeof(ObjectIdConverter))]
public record ObjectId(int Id);
```

For more information, see How to write custom converters.

Exclude specified types from model binding

The model binding and validation systems' behavior is driven by ModelMetadata. You can customize `ModelMetadata` by adding a details provider to MvcOptions.ModelMetadataDetailsProviders. Built-in details providers are available for disabling model binding or validation for specified types.

To disable model binding on all models of a specified type, add an ExcludeBindingMetadataProvider in `Program.cs`. For example, to disable model binding on all models of type `System.Version`:

```
C#

builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
```

```
        options.ModelMetadataDetailsProviders.Add(
            new ExcludeBindingMetadataProvider(typeof(Version)));
        options.ModelMetadataDetailsProviders.Add(
            new SuppressChildValidationMetadataProvider(typeof(Guid)));
    });
```

To disable validation on properties of a specified type, add a SuppressChildValidationMetadataProvider in `Program.cs`. For example, to disable validation on properties of type `System.Guid`:

C#

```
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.ModelMetadataDetailsProviders.Add(
            new ExcludeBindingMetadataProvider(typeof(Version)));
        options.ModelMetadataDetailsProviders.Add(
            new SuppressChildValidationMetadataProvider(typeof(Guid)));
    });
```

Custom model binders

You can extend model binding by writing a custom model binder and using the `[ModelBinder]` attribute to select it for a given target. Learn more about custom model binding.

Manual model binding

Model binding can be invoked manually by using the TryUpdateModelAsync method. The method is defined on both `ControllerBase` and `PageModel` classes. Method overloads let you specify the prefix and value provider to use. The method returns `false` if model binding fails. Here's an example:

C#

```
if (await TryUpdateModelAsync(
    newInstructor,
    "Instructor",
    x => x.Name, x => x.HireDate!))
{
    _instructorStore.Add(newInstructor);
    return RedirectToPage("./Index");
}
```

```
return Page();
```

TryUpdateModelAsync uses value providers to get data from the form body, query string, and route data. `TryUpdateModelAsync` is typically:

- Used with Razor Pages and MVC apps using controllers and views to prevent over-posting.
- Not used with a web API unless consumed from form data, query strings, and route data. Web API endpoints that consume JSON use Input formatters to deserialize the request body into an object.

For more information, see TryUpdateModelAsync.

[FromServices] attribute

This attribute's name follows the pattern of model binding attributes that specify a data source. But it's not about binding data from a value provider. It gets an instance of a type from the dependency injection container. Its purpose is to provide an alternative to constructor injection for when you need a service only if a particular method is called.

If an instance of the type isn't registered in the dependency injection container, the app throws an exception when attempting to bind the parameter. To make the parameter optional, use one of the following approaches:

- Make the parameter nullable.
- Set a default value for the parameter.

For nullable parameters, ensure that the parameter isn't `null` before accessing it.

Additional resources

- View or download sample code (how to download)
- Model validation in ASP.NET Core MVC
- Custom Model Binding in ASP.NET Core

Custom Model Binding in ASP.NET Core

Article • 04/03/2024

By Kirk Larkin

Model binding allows controller actions to work directly with model types (passed in as method arguments), rather than HTTP requests. Mapping between incoming request data and application models is handled by model binders. Developers can extend the built-in model binding functionality by implementing custom model binders (though typically, you don't need to write your own provider).

View or download sample code (how to download)

Default model binder limitations

The default model binders support most of the common .NET Core data types and should meet most developers' needs. They expect to bind text-based input from the request directly to model types. You might need to transform the input prior to binding it. For example, when you have a key that can be used to look up model data. You can use a custom model binder to fetch data based on the key.

Model binding simple and complex types

Model binding uses specific definitions for the types it operates on. A *simple type* is converted from a single string using TypeConverter or a `TryParse` method. A *complex type* is converted from multiple input values. The framework determines the difference based on the existence of a `TypeConverter` or `TryParse`. We recommend creating a type converter or using `TryParse` for a `string` to `SomeType` conversion that doesn't require external resources or multiple inputs.

See Simple types for a list of types that the model binder can convert from strings.

Before creating your own custom model binder, it's worth reviewing how existing model binders are implemented. Consider the ByteArrayModelBinder which can be used to convert base64-encoded strings into byte arrays. The byte arrays are often stored as files or database BLOB fields.

Working with the ByteArrayModelBinder

Base64-encoded strings can be used to represent binary data. For example, an image can be encoded as a string. The sample includes an image as a base64-encoded string in Base64String.txt.

ASP.NET Core MVC can take a base64-encoded string and use a `ByteArrayModelBinder` to convert it into a byte array. The ByteArrayModelBinderProvider maps `byte[]` arguments to `ByteArrayModelBinder`:

C#

```
public IModelBinder GetBinder(ModelBinderProviderContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException(nameof(context));
    }

    if (context.Metadata.ModelType == typeof(byte[]))
    {
        var loggerFactory =
context.Services.GetRequiredService<ILoggerFactory>();
        return new ByteArrayModelBinder(loggerFactory);
    }

    return null;
}
```

When creating your own custom model binder, you can implement your own `IModelBinderProvider` type, or use the ModelBinderAttribute.

The following example shows how to use `ByteArrayModelBinder` to convert a base64-encoded string to a `byte[]` and save the result to a file:

C#

```
[HttpPost]
public void Post([FromForm] byte[] file, string filename)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
```

```
    }

    System.IO.File.WriteAllBytes(filePath, file);
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

You can POST a base64-encoded string to the previous api method using a tool like curl.

As long as the binder can bind request data to appropriately named properties or arguments, model binding will succeed. The following example shows how to use `ByteArrayModelBinder` with a view model:

C#

```
[HttpPost("Profile")]
public void SaveProfile([FromForm] ProfileViewModel model)
{
    // Don't trust the file name sent by the client. Use
    // Path.GetRandomFileName to generate a safe random
    // file name. _targetFilePath receives a value
    // from configuration (the appsettings.json file in
    // the sample app).
    var trustedFileName = Path.GetRandomFileName();
    var filePath = Path.Combine(_targetFilePath, trustedFileName);

    if (System.IO.File.Exists(filePath))
    {
        return;
    }

    System.IO.File.WriteAllBytes(filePath, model.File);
}

public class ProfileViewModel
{
    public byte[] File { get; set; }
    public string FileName { get; set; }
}
```

Custom model binder sample

In this section we'll implement a custom model binder that:

- Converts incoming request data into strongly typed key arguments.
- Uses Entity Framework Core to fetch the associated entity.

- Passes the associated entity as an argument to the action method.

The following sample uses the `ModelBinder` attribute on the `Author` model:

C#

```
using CustomModelBindingSample.Binders;
using Microsoft.AspNetCore.Mvc;

namespace CustomModelBindingSample.Data
{
    [ModelBinder(BinderType = typeof(AuthorEntityBinder))]
    public class Author
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string GitHub { get; set; }
        public string Twitter { get; set; }
        public string BlogUrl { get; set; }
    }
}
```

In the preceding code, the `ModelBinder` attribute specifies the type of `IModelBinder` that should be used to bind `Author` action parameters.

The following `AuthorEntityBinder` class binds an `Author` parameter by fetching the entity from a data source using Entity Framework Core and an `authorId`:

C#

```
public class AuthorEntityBinder : IModelBinder
{
    private readonly AuthorContext _context;

    public AuthorEntityBinder(AuthorContext context)
    {
        _context = context;
    }

    public Task BindModelAsync(ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException(nameof(bindingContext));
        }

        var modelName = bindingContext.ModelName;

        // Try to fetch the value of the argument by name
        var valueProviderResult =
bindingContext.ValueProvider.GetValue(modelName);
```

```
        if (valueProviderResult == ValueProviderResult.None)
        {
            return Task.CompletedTask;
        }

        bindingContext.ModelState.SetModelValue(modelName,
valueProviderResult);

        var value = valueProviderResult.FirstValue;

        // Check if the argument value is null or empty
        if (string.IsNullOrEmpty(value))
        {
            return Task.CompletedTask;
        }

        if (!int.TryParse(value, out var id))
        {
            // Non-integer arguments result in model state errors
            bindingContext.ModelState.TryAddModelError(
                modelName, "Author Id must be an integer.");

            return Task.CompletedTask;
        }

        // Model will be null if not found, including for
        // out of range id values (0, -3, etc.)
        var model = _context.Authors.Find(id);
        bindingContext.Result = ModelBindingResult.Success(model);
        return Task.CompletedTask;
    }
}
```

> **Note**
>
> The preceding `AuthorEntityBinder` class is intended to illustrate a custom model binder. The class isn't intended to illustrate best practices for a lookup scenario. For lookup, bind the `authorId` and query the database in an action method. This approach separates model binding failures from `NotFound` cases.

The following code shows how to use the `AuthorEntityBinder` in an action method:

C#

```
[HttpGet("get/{author}")]
public IActionResult Get(Author author)
{
    if (author == null)
    {
```

```
        return NotFound();
    }

    return Ok(author);
}
```

The `ModelBinder` attribute can be used to apply the `AuthorEntityBinder` to parameters that don't use default conventions:

C#

```
[HttpGet("{id}")]
public IActionResult GetById([ModelBinder(Name = "id")] Author author)
{
    if (author == null)
    {
        return NotFound();
    }

    return Ok(author);
}
```

In this example, since the name of the argument isn't the default `authorId`, it's specified on the parameter using the `ModelBinder` attribute. Both the controller and action method are simplified compared to looking up the entity in the action method. The logic to fetch the author using Entity Framework Core is moved to the model binder. This can be a considerable simplification when you have several methods that bind to the `Author` model.

You can apply the `ModelBinder` attribute to individual model properties (such as on a viewmodel) or to action method parameters to specify a certain model binder or model name for just that type or action.

Implementing a ModelBinderProvider

Instead of applying an attribute, you can implement `IModelBinderProvider`. This is how the built-in framework binders are implemented. When you specify the type your binder operates on, you specify the type of argument it produces, **not** the input your binder accepts. The following binder provider works with the `AuthorEntityBinder`. When it's added to MVC's collection of providers, you don't need to use the `ModelBinder` attribute on `Author` or `Author`-typed parameters.

C#

```
using CustomModelBindingSample.Data;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using System;

namespace CustomModelBindingSample.Binders
{
    public class AuthorEntityBinderProvider : IModelBinderProvider
    {
        public IModelBinder GetBinder(ModelBinderProviderContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException(nameof(context));
            }

            if (context.Metadata.ModelType == typeof(Author))
            {
                return new
BinderTypeModelBinder(typeof(AuthorEntityBinder));
            }

            return null;
        }
    }
}
```

> Note: The preceding code returns a `BinderTypeModelBinder`. `BinderTypeModelBinder` acts as a factory for model binders and provides dependency injection (DI). The `AuthorEntityBinder` requires DI to access EF Core. Use `BinderTypeModelBinder` if your model binder requires services from DI.

To use a custom model binder provider, add it in `ConfigureServices`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<AuthorContext>(options =>
options.UseInMemoryDatabase("Authors"));

    services.AddControllers(options =>
    {
        options.ModelBinderProviders.Insert(0, new
AuthorEntityBinderProvider());
    });
}
```

When evaluating model binders, the collection of providers is examined in order. The first provider that returns a binder that matches the input model is used. Adding your provider to the end of the collection may thus result in a built-in model binder being called before your custom binder has a chance. In this example, the custom provider is added to the beginning of the collection to ensure it's always used for `Author` action arguments.

Polymorphic model binding

Binding to different models of derived types is known as polymorphic model binding. Polymorphic custom model binding is required when the request value must be bound to the specific derived model type. Polymorphic model binding:

- Isn't typical for a REST API that's designed to interoperate with all languages.
- Makes it difficult to reason about the bound models.

However, if an app requires polymorphic model binding, an implementation might look like the following code:

C#

```
public abstract class Device
{
    public string Kind { get; set; }
}

public class Laptop : Device
{
    public string CPUIndex { get; set; }
}

public class SmartPhone : Device
{
    public string ScreenSize { get; set; }
}

public class DeviceModelBinderProvider : IModelBinderProvider
{
    public IModelBinder GetBinder(ModelBinderProviderContext context)
    {
        if (context.Metadata.ModelType != typeof(Device))
        {
            return null;
        }

        var subclasses = new[] { typeof(Laptop), typeof(SmartPhone), };

        var binders = new Dictionary<Type, (ModelMetadata, IModelBinder)>();
        foreach (var type in subclasses)
```

```
        {
            var modelMetadata =
context.MetadataProvider.GetMetadataForType(type);
            binders[type] = (modelMetadata,
context.CreateBinder(modelMetadata));
        }

        return new DeviceModelBinder(binders);
    }
}

public class DeviceModelBinder : IModelBinder
{
    private Dictionary<Type, (ModelMetadata, IModelBinder)> binders;

    public DeviceModelBinder(Dictionary<Type, (ModelMetadata, IModelBinder)>
binders)
    {
        this.binders = binders;
    }

    public async Task BindModelAsync(ModelBindingContext bindingContext)
    {
        var modelKindName =
ModelNames.CreatePropertyModelName(bindingContext.ModelName,
nameof(Device.Kind));
        var modelTypeValue =
bindingContext.ValueProvider.GetValue(modelKindName).FirstValue;

        IModelBinder modelBinder;
        ModelMetadata modelMetadata;
        if (modelTypeValue == "Laptop")
        {
            (modelMetadata, modelBinder) = binders[typeof(Laptop)];
        }
        else if (modelTypeValue == "SmartPhone")
        {
            (modelMetadata, modelBinder) = binders[typeof(SmartPhone)];
        }
        else
        {
            bindingContext.Result = ModelBindingResult.Failed();
            return;
        }

        var newBindingContext =
DefaultModelBindingContext.CreateBindingContext(
            bindingContext.ActionContext,
            bindingContext.ValueProvider,
            modelMetadata,
            bindingInfo: null,
            bindingContext.ModelName);

        await modelBinder.BindModelAsync(newBindingContext);
        bindingContext.Result = newBindingContext.Result;
```

```
        if (newBindingContext.Result.IsModelSet)
        {
            // Setting the ValidationState ensures properties on derived
types are correctly
            bindingContext.ValidationState[newBindingContext.Result.Model] =
new ValidationStateEntry
            {
                Metadata = modelMetadata,
            };
        }
    }
}
```

Recommendations and best practices

Custom model binders:

- Shouldn't attempt to set status codes or return results (for example, 404 Not Found). If model binding fails, an action filter or logic within the action method itself should handle the failure.
- Are most useful for eliminating repetitive code and cross-cutting concerns from action methods.
- Typically shouldn't be used to convert a string into a custom type, a TypeConverter is usually a better option.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Model validation in ASP.NET Core MVC and Razor Pages

Article • 09/08/2023

This article explains how to validate user input in an ASP.NET Core MVC or Razor Pages app.

View or download sample code (how to download).

Model state

Model state represents errors that come from two subsystems: model binding and model validation. Errors that originate from model binding are generally data conversion errors. For example, an "x" is entered in an integer field. Model validation occurs after model binding and reports errors where data doesn't conform to business rules. For example, a 0 is entered in a field that expects a rating between 1 and 5.

Both model binding and model validation occur before the execution of a controller action or a Razor Pages handler method. For web apps, it's the app's responsibility to inspect `ModelState.IsValid` and react appropriately. Web apps typically redisplay the page with an error message, as shown in the following Razor Pages example:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _context.Movies.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

For ASP.NET Core MVC with controllers and views, the following example shows how to check `ModelState.IsValid` inside of a controller action:

C#

```
public async Task<IActionResult> Create(Movie movie)
{
    if (!ModelState.IsValid)
    {
        return View(movie);
    }

    _context.Movies.Add(movie);
    await _context.SaveChangesAsync();

    return RedirectToAction(nameof(Index));
}
```

Web API controllers don't have to check `ModelState.IsValid` if they have the [ApiController] attribute. In that case, an automatic HTTP 400 response containing error details is returned when model state is invalid. For more information, see Automatic HTTP 400 responses.

Rerun validation

Validation is automatic, but you might want to repeat it manually. For example, you might compute a value for a property and want to rerun validation after setting the property to the computed value. To rerun validation, call ModelStateDictionary.ClearValidationState to clear validation specific to the model being validated followed by `TryValidateModel`:

C#

```
public async Task<IActionResult> OnPostTryValidateAsync()
{
    var modifiedReleaseDate = DateTime.Now.Date;
    Movie.ReleaseDate = modifiedReleaseDate;

    ModelState.ClearValidationState(nameof(Movie));
    if (!TryValidateModel(Movie, nameof(Movie)))
    {
        return Page();
    }

    _context.Movies.Add(Movie);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

Validation attributes

Validation attributes let you specify validation rules for model properties. The following example from the sample app shows a model class that is annotated with validation attributes. The `[ClassicMovie]` attribute is a custom validation attribute and the others are built in. Not shown is `[ClassicMovieWithClientValidator]`, which shows an alternative way to implement a custom attribute.

C#

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; } = null!;

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; } = null!;

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

Built-in attributes

Here are some of the built-in validation attributes:

- [ValidateNever]: Indicates that a property or parameter should be excluded from validation.
- [CreditCard]: Validates that the property has a credit card format. Requires jQuery Validation Additional Methods.
- [Compare]: Validates that two properties in a model match.
- [EmailAddress]: Validates that the property has an email format.
- [Phone]: Validates that the property has a telephone number format.

- [Range]: Validates that the property value falls within a specified range.
- [RegularExpression]: Validates that the property value matches a specified regular expression.
- [Required]: Validates that the field isn't null. See [Required] attribute for details about this attribute's behavior.
- [StringLength]: Validates that a string property value doesn't exceed a specified length limit.
- [Url]: Validates that the property has a URL format.
- [Remote]: Validates input on the client by calling an action method on the server. See [Remote] attribute for details about this attribute's behavior.

A complete list of validation attributes can be found in the System.ComponentModel.DataAnnotations namespace.

Error messages

Validation attributes let you specify the error message to be displayed for invalid input. For example:

```
C#

[StringLength(8, ErrorMessage = "Name length can't be more than 8.")]
```

Internally, the attributes call String.Format with a placeholder for the field name and sometimes additional placeholders. For example:

```
C#

[StringLength(8, ErrorMessage = "{0} length must be between {2} and {1}.",
MinimumLength = 6)]
```

When applied to a `Name` property, the error message created by the preceding code would be "Name length must be between 6 and 8.".

To find out which parameters are passed to `String.Format` for a particular attribute's error message, see the DataAnnotations source code.

Use JSON property names in validation errors

By default, when a validation error occurs, model validation produces a ModelStateDictionary with the property name as the error key. Some apps, such as single page apps, benefit from using JSON property names for validation errors

generated from Web APIs. The following code configures validation to use the SystemTextJsonValidationMetadataProvider to use JSON property names:

C#

```
using Microsoft.AspNetCore.Mvc.ModelBinding.Metadata;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
SystemTextJsonValidationMetadataProvider());
});

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

The following code configures validation to use the NewtonsoftJsonValidationMetadataProvider to use JSON property name when using Json.NET:

C#

```
using Microsoft.AspNetCore.Mvc.NewtonsoftJson;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>
{
    options.ModelMetadataDetailsProviders.Add(new
NewtonsoftJsonValidationMetadataProvider());
}).AddNewtonsoftJson();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

For an example of the policy to use camel-casing, see Program.cs on GitHub.

Non-nullable reference types and [Required] attribute

The validation system treats non-nullable parameters or bound properties as if they had a `[Required(AllowEmptyStrings = true)]` attribute. By enabling Nullable contexts, MVC implicitly starts validating non-nullable properties or parameters as if they had been attributed with the `[Required(AllowEmptyStrings = true)]` attribute. Consider the following code:

```
C#

public class Person
{
    public string Name { get; set; }
}
```

If the app was built with `<Nullable>enable</Nullable>`, a missing value for `Name` in a JSON or form post results in a validation error. This may seem contradictory since the `[Required(AllowEmptyStrings = true)]` attribute is implied, but this is expected behavior because empty strings are converted to null by default. Use a nullable reference type to allow null or missing values to be specified for the `Name` property:

```
C#

public class Person
{
    public string? Name { get; set; }
}
```

This behavior can be disabled by configuring SuppressImplicitRequiredAttributeForNonNullableReferenceTypes in `Program.cs`:

```
C#

builder.Services.AddControllers(
    options =>
options.SuppressImplicitRequiredAttributeForNonNullableReferenceTypes =
true);
```

[Required] validation on the server

On the server, a required value is considered missing if the property is null. A non-nullable field is always valid, and the `[Required]` attribute's error message is never displayed.

However, model binding for a non-nullable property may fail, resulting in an error message such as `The value '' is invalid`. To specify a custom error message for server-side validation of non-nullable types, you have the following options:

- Make the field nullable (for example, `decimal?` instead of `decimal`). Nullable<T> value types are treated like standard nullable types.
- Specify the default error message to be used by model binding, as shown in the following example:

  C#

  ```
  builder.Services.AddRazorPages()
      .AddMvcOptions(options =>
      {
          options.MaxModelValidationErrors = 50;

  options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
              _ => "The field is required.");
      });

  builder.Services.AddSingleton
      <IValidationAttributeAdapterProvider,
  CustomValidationAttributeAdapterProvider>();
  ```

  For more information about model binding errors that you can set default messages for, see DefaultModelBindingMessageProvider.

[Required] validation on the client

Non-nullable types and strings are handled differently on the client compared to the server. On the client:

- A value is considered present only if input is entered for it. Therefore, client-side validation handles non-nullable types the same as nullable types.
- Whitespace in a string field is considered valid input by the jQuery Validation required method. Server-side validation considers a required string field invalid if only whitespace is entered.

As noted earlier, non-nullable types are treated as though they had a `[Required(AllowEmptyStrings = true)]` attribute. That means you get client-side

validation even if you don't apply the `[Required(AllowEmptyStrings = true)]` attribute. But if you don't use the attribute, you get a default error message. To specify a custom error message, use the attribute.

[Remote] attribute

The [Remote] attribute implements client-side validation that requires calling a method on the server to determine whether field input is valid. For example, the app may need to verify whether a user name is already in use.

To implement remote validation:

1. Create an action method for JavaScript to call. The jQuery Validation remote method expects a JSON response:

   - `true` means the input data is valid.
   - `false`, `undefined`, or `null` means the input is invalid. Display the default error message.
   - Any other string means the input is invalid. Display the string as a custom error message.

   Here's an example of an action method that returns a custom error message:

   C#

   ```
   [AcceptVerbs("GET", "POST")]
   public IActionResult VerifyEmail(string email)
   {
       if (!_userService.VerifyEmail(email))
       {
           return Json($"Email {email} is already in use.");
       }

       return Json(true);
   }
   ```

2. In the model class, annotate the property with a `[Remote]` attribute that points to the validation action method, as shown in the following example:

   C#

   ```
   [Remote(action: "VerifyEmail", controller: "Users")]
   public string Email { get; set; } = null!;
   ```

Server side validation also needs to be implemented for clients that have disabled JavaScript.

Additional fields

The AdditionalFields property of the `[Remote]` attribute lets you validate combinations of fields against data on the server. For example, if the `User` model had `FirstName` and `LastName` properties, you might want to verify that no existing users already have that pair of names. The following example shows how to use `AdditionalFields`:

C#

```
[Remote(action: "VerifyName", controller: "Users", AdditionalFields =
nameof(LastName))]
[Display(Name = "First Name")]
public string FirstName { get; set; } = null!;

[Remote(action: "VerifyName", controller: "Users", AdditionalFields =
nameof(FirstName))]
[Display(Name = "Last Name")]
public string LastName { get; set; } = null!;
```

`AdditionalFields` could be set explicitly to the strings "FirstName" and "LastName", but using the nameof operator simplifies later refactoring. The action method for this validation must accept both `firstName` and `lastName` arguments:

C#

```
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyName(string firstName, string lastName)
{
    if (!_userService.VerifyName(firstName, lastName))
    {
        return Json($"A user named {firstName} {lastName} already exists.");
    }

    return Json(true);
}
```

When the user enters a first or last name, JavaScript makes a remote call to see if that pair of names has been taken.

To validate two or more additional fields, provide them as a comma-delimited list. For example, to add a `MiddleName` property to the model, set the `[Remote]` attribute as shown in the following example:

C#

```
[Remote(action: "VerifyName", controller: "Users",
    AdditionalFields = nameof(FirstName) + "," + nameof(LastName))]
public string MiddleName { get; set; }
```

`AdditionalFields`, like all attribute arguments, must be a constant expression. Therefore, don't use an interpolated string or call Join to initialize `AdditionalFields`.

Alternatives to built-in attributes

If you need validation not provided by built-in attributes, you can:

- Create custom attributes.
- Implement IValidatableObject.

Custom attributes

For scenarios that the built-in validation attributes don't handle, you can create custom validation attributes. Create a class that inherits from ValidationAttribute, and override the IsValid method.

The `IsValid` method accepts an object named *value*, which is the input to be validated. An overload also accepts a ValidationContext object, which provides additional information, such as the model instance created by model binding.

The following example validates that the release date for a movie in the *Classic* genre isn't later than a specified year. The `[ClassicMovie]` attribute:

- Is only run on the server.
- For Classic movies, validates the release date:

C#

```
public class ClassicMovieAttribute : ValidationAttribute
{
    public ClassicMovieAttribute(int year)
        => Year = year;

    public int Year { get; }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than {Year}.";

    protected override ValidationResult? IsValid(
```

```
        object? value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value!).Year;

        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }
}
```

The `movie` variable in the preceding example represents a `Movie` object that contains the data from the form submission. When validation fails, a ValidationResult with an error message is returned.

IValidatableObject

The preceding example works only with `Movie` types. Another option for class-level validation is to implement IValidatableObject in the model class, as shown in the following example:

C#

```
public class ValidatableMovie : IValidatableObject
{
    private const int _classicYear = 1960;

    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; } = null!;

    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; } = null!;

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
```

```
    public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
    {
        if (Genre == Genre.Classic && ReleaseDate.Year > _classicYear)
        {
            yield return new ValidationResult(
                $"Classic movies must have a release year no later than
{_classicYear}.",
                new[] { nameof(ReleaseDate) });
        }
    }
}
```

Custom validation

The following code shows how to add a model error after examining the model:

C#

```
if (Contact.Name == Contact.ShortName)
{
    ModelState.AddModelError("Contact.ShortName",
                             "Short name can't be the same as Name.");
}
```

The following code implements the validation test in a controller:

C#

```
if (contact.Name == contact.ShortName)
{
    ModelState.AddModelError(nameof(contact.ShortName),
                             "Short name can't be the same as Name.");
}
```

The following code verifies the phone number and email are unique:

C#

```
public async Task<IActionResult> OnPostAsync()
{
    // Attach Validation Error Message to the Model on validation failure.

    if (Contact.Name == Contact.ShortName)
    {
        ModelState.AddModelError("Contact.ShortName",
                                 "Short name can't be the same as Name.");
```

```
    }

    if (_context.Contact.Any(i => i.PhoneNumber == Contact.PhoneNumber))
    {
        ModelState.AddModelError("Contact.PhoneNumber",
                                 "The Phone number is already in use.");
    }
    if (_context.Contact.Any(i => i.Email == Contact.Email))
    {
        ModelState.AddModelError("Contact.Email", "The Email is already in
use.");
    }

    if (!ModelState.IsValid || _context.Contact == null || Contact == null)
    {
        // if model is invalid, return the page with the model state errors.
        return Page();
    }
    _context.Contact.Add(Contact);
    await _context.SaveChangesAsync();

    return RedirectToPage("./Index");
}
```

The following code implements the validation test in a controller:

C#

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("Id,Name,ShortName,Email,PhoneNumber")] Contact contact)
{
    // Attach Validation Error Message to the Model on validation failure.
    if (contact.Name == contact.ShortName)
    {
        ModelState.AddModelError(nameof(contact.ShortName),
                                 "Short name can't be the same as Name.");
    }

    if (_context.Contact.Any(i => i.PhoneNumber == contact.PhoneNumber))
    {
        ModelState.AddModelError(nameof(contact.PhoneNumber),
                                 "The Phone number is already in use.");
    }
    if (_context.Contact.Any(i => i.Email == contact.Email))
    {
        ModelState.AddModelError(nameof(contact.Email), "The Email is
already in use.");
    }

    if (ModelState.IsValid)
    {
```

```
            _context.Add(contact);
            await _context.SaveChangesAsync();
            return RedirectToAction(nameof(Index));
        }
        return View(contact);
    }
```

Checking for a unique phone number or email is typically also done with remote validation.

ValidationResult

Consider the following custom `ValidateNameAttribute`:

C#

```
public class ValidateNameAttribute : ValidationAttribute
{
    public ValidateNameAttribute()
    {
        const string defaultErrorMessage = "Error with Name";
        ErrorMessage ??= defaultErrorMessage;
    }

    protected override ValidationResult? IsValid(object? value,
                                        ValidationContext
validationContext)
    {
        if (value == null || string.IsNullOrWhiteSpace(value.ToString()))
        {
            return new ValidationResult("Name is required.");
        }

        if (value.ToString()!.ToLower().Contains("zz"))
        {

            return new ValidationResult(
                        FormatErrorMessage(validationContext.DisplayName));
        }

        return ValidationResult.Success;
    }
}
```

In the following code, the custom `[ValidateName]` attribute is applied:

C#

```
public class Contact
{
    public Guid Id { get; set; }

    [ValidateName(ErrorMessage = "Name must not contain `zz`")]
    public string? Name { get; set; }
    public string? Email { get; set; }
    public string? PhoneNumber { get; set; }
}
```

When the model contains `zz`, a new ValidationResult is returned.

Top-level node validation

Top-level nodes include:

- Action parameters
- Controller properties
- Page handler parameters
- Page model properties

Model-bound top-level nodes are validated in addition to validating model properties. In the following example from the sample app, the `VerifyPhone` method uses the RegularExpressionAttribute to validate the `phone` action parameter:

C#

```
[AcceptVerbs("GET", "POST")]
public IActionResult VerifyPhone(
    [RegularExpression(@"^\d{3}-\d{3}-\d{4}$")] string phone)
{
    if (!ModelState.IsValid)
    {
        return Json($"Phone {phone} has an invalid format. Format: ###-###-
####");
    }

    return Json(true);
}
```

Top-level nodes can use BindRequiredAttribute with validation attributes. In the following example from the sample app, the `CheckAge` method specifies that the `age` parameter must be bound from the query string when the form is submitted:

C#

```
[HttpPost]
public IActionResult CheckAge([BindRequired, FromQuery] int age)
{
```

In the Check Age page (`CheckAge.cshtml`), there are two forms. The first form submits an `Age` value of `99` as a query string parameter: `https://localhost:5001/Users/CheckAge?Age=99`.

When a properly formatted `age` parameter from the query string is submitted, the form validates.

The second form on the Check Age page submits the `Age` value in the body of the request, and validation fails. Binding fails because the `age` parameter must come from a query string.

Maximum errors

Validation stops when the maximum number of errors is reached (200 by default). You can configure this number with the following code in `Program.cs`:

C#

```
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
        options.MaxModelValidationErrors = 50;
        options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

builder.Services.AddSingleton
    <IValidationAttributeAdapterProvider,
CustomValidationAttributeAdapterProvider>();
```

Maximum recursion

ValidationVisitor traverses the object graph of the model being validated. For models that are deep or are infinitely recursive, validation may result in stack overflow. MvcOptions.MaxValidationDepth provides a way to stop validation early if the visitor recursion exceeds a configured depth. The default value of `MvcOptions.MaxValidationDepth` is 32.

Automatic short-circuit

Validation is automatically short-circuited (skipped) if the model graph doesn't require validation. Objects that the runtime skips validation for include collections of primitives (such as `byte[]`, `string[]`, `Dictionary<string, string>`) and complex object graphs that don't have any validators.

Client-side validation

Client-side validation prevents submission until the form is valid. The Submit button runs JavaScript that either submits the form or displays error messages.

Client-side validation avoids an unnecessary round trip to the server when there are input errors on a form. The following script references in `_Layout.cshtml` and `_ValidationScriptsPartial.cshtml` support client-side validation:

CSHTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.js">
</script>
```

CSHTML

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-
validate/1.19.3/jquery.validate.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery-validation-
unobtrusive/3.2.12/jquery.validate.unobtrusive.js"></script>
```

The jQuery Unobtrusive Validation script is a custom Microsoft front-end library that builds on the popular jQuery Validation plugin. Without jQuery Unobtrusive Validation, you would have to code the same validation logic in two places: once in the server-side validation attributes on model properties, and then again in client-side scripts. Instead, Tag Helpers and HTML helpers use the validation attributes and type metadata from model properties to render HTML 5 `data-` attributes for the form elements that need validation. jQuery Unobtrusive Validation parses the `data-` attributes and passes the logic to jQuery Validation, effectively "copying" the server-side validation logic to the client. You can display validation errors on the client using tag helpers as shown here:

CSHTML

```
<div class="form-group">
    <label asp-for="Movie.ReleaseDate" class="control-label"></label>
    <input asp-for="Movie.ReleaseDate" class="form-control" />
    <span asp-validation-for="Movie.ReleaseDate" class="text-danger"></span>
</div>
```

The preceding tag helpers render the following HTML:

HTML

```
<div class="form-group">
    <label class="control-label" for="Movie_ReleaseDate">Release
Date</label>
    <input class="form-control" type="date" data-val="true"
        data-val-required="The Release Date field is required."
        id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
    <span class="text-danger field-validation-valid"
        data-valmsg-for="Movie.ReleaseDate" data-valmsg-replace="true">
</span>
</div>
```

Notice that the `data-` attributes in the HTML output correspond to the validation attributes for the `Movie.ReleaseDate` property. The `data-val-required` attribute contains an error message to display if the user doesn't fill in the release date field. jQuery Unobtrusive Validation passes this value to the jQuery Validation required() method, which then displays that message in the accompanying **** element.

Data type validation is based on the .NET type of a property, unless that is overridden by a [DataType] attribute. Browsers have their own default error messages, but the jQuery Validation Unobtrusive Validation package can override those messages. `[DataType]` attributes and subclasses such as [EmailAddress] let you specify the error message.

Unobtrusive validation

For information on unobtrusive validation, see this GitHub issue.

Add Validation to Dynamic Forms

jQuery Unobtrusive Validation passes validation logic and parameters to jQuery Validation when the page first loads. Therefore, validation doesn't work automatically on dynamically generated forms. To enable validation, tell jQuery Unobtrusive Validation to parse the dynamic form immediately after you create it. For example, the following code sets up client-side validation on a form added via AJAX.

JavaScript

```
$.get({
    url: "https://url/that/returns/a/form",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add form. " + errorThrown);
    },
    success: function(newFormHTML) {
        var container = document.getElementById("form-container");
        container.insertAdjacentHTML("beforeend", newFormHTML);
        var forms = container.getElementsByTagName("form");
        var newForm = forms[forms.length - 1];
        $.validator.unobtrusive.parse(newForm);
    }
})
```

The `$.validator.unobtrusive.parse()` method accepts a jQuery selector for its one argument. This method tells jQuery Unobtrusive Validation to parse the `data-` attributes of forms within that selector. The values of those attributes are then passed to the jQuery Validation plugin.

Add Validation to Dynamic Controls

The `$.validator.unobtrusive.parse()` method works on an entire form, not on individual dynamically generated controls, such as `<input>` and `<select/>`. To reparse the form, remove the validation data that was added when the form was parsed earlier, as shown in the following example:

JavaScript

```
$.get({
    url: "https://url/that/returns/a/control",
    dataType: "html",
    error: function(jqXHR, textStatus, errorThrown) {
        alert(textStatus + ": Couldn't add control. " + errorThrown);
    },
    success: function(newInputHTML) {
        var form = document.getElementById("my-form");
        form.insertAdjacentHTML("beforeend", newInputHTML);
        $(form).removeData("validator")    // Added by jQuery Validation
               .removeData("unobtrusiveValidation");   // Added by jQuery
Unobtrusive Validation
        $.validator.unobtrusive.parse(form);
    }
})
```

Custom client-side validation

Custom client-side validation is done by generating `data-` HTML attributes that work with a custom jQuery Validation adapter. The following sample adapter code was written for the `[ClassicMovie]` and `[ClassicMovieWithClientValidator]` attributes that were introduced earlier in this article:

JavaScript

```
$.validator.addMethod('classicmovie', function (value, element, params) {
    var genre = $(params[0]).val(), year = params[1], date = new
Date(value);

    // The Classic genre has a value of '0'.
    if (genre && genre.length > 0 && genre[0] === '0') {
        // The release date for a Classic is valid if it's no greater than
the given year.
        return date.getUTCFullYear() <= year;
    }

    return true;
});

$.validator.unobtrusive.adapters.add('classicmovie', ['year'], function
(options) {
    var element = $(options.form).find('select#Movie_Genre')[0];

    options.rules['classicmovie'] = [element,
parseInt(options.params['year'])];
    options.messages['classicmovie'] = options.message;
});
```

For information about how to write adapters, see the jQuery Validation documentation.

The use of an adapter for a given field is triggered by `data-` attributes that:

- Flag the field as being subject to validation (`data-val="true"`).
- Identify a validation rule name and error message text (for example, `data-val-rulename="Error message."`).
- Provide any additional parameters the validator needs (for example, `data-val-rulename-param1="value"`).

The following example shows the `data-` attributes for the sample app's `ClassicMovie` attribute:

HTML

```
<input class="form-control" type="date"
    data-val="true"
    data-val-classicmovie="Classic movies must have a release year no later
than 1960."
    data-val-classicmovie-year="1960"
    data-val-required="The Release Date field is required."
    id="Movie_ReleaseDate" name="Movie.ReleaseDate" value="">
```

As noted earlier, Tag Helpers and HTML helpers use information from validation attributes to render `data-` attributes. There are two options for writing code that results in the creation of custom `data-` HTML attributes:

- Create a class that derives from AttributeAdapterBase<TAttribute> and a class that implements IValidationAttributeAdapterProvider, and register your attribute and its adapter in DI. This method follows the single responsibility principle in that server-related and client-related validation code is in separate classes. The adapter also has the advantage that since it's registered in DI, other services in DI are available to it if needed.
- Implement IClientModelValidator in your ValidationAttribute class. This method might be appropriate if the attribute doesn't do any server-side validation and doesn't need any services from DI.

AttributeAdapter for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovie` attribute in the sample app. To add client validation by using this method:

1. Create an attribute adapter class for the custom validation attribute. Derive the class from AttributeAdapterBase<TAttribute>. Create an `AddValidation` method that adds `data-` attributes to the rendered output, as shown in this example:

   C#

   ```
   public class ClassicMovieAttributeAdapter :
   AttributeAdapterBase<ClassicMovieAttribute>
   {
       public ClassicMovieAttributeAdapter(
           ClassicMovieAttribute attribute, IStringLocalizer?
   stringLocalizer)
           : base(attribute, stringLocalizer)
       {

       }

       public override void AddValidation(ClientModelValidationContext
   ```

```
context)
    {
        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie",
GetErrorMessage(context));

        var year =
Attribute.Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-
year", year);
    }

    public override string GetErrorMessage(ModelValidationContextBase
validationContext)
        => Attribute.GetErrorMessage();
}
```

2. Create an adapter provider class that implements IValidationAttributeAdapterProvider. In the GetAttributeAdapter method pass in the custom attribute to the adapter's constructor, as shown in this example:

C#

```
public class CustomValidationAttributeAdapterProvider :
IValidationAttributeAdapterProvider
{
    private readonly IValidationAttributeAdapterProvider baseProvider =
        new ValidationAttributeAdapterProvider();

    public IAttributeAdapter? GetAttributeAdapter(
        ValidationAttribute attribute, IStringLocalizer?
stringLocalizer)
    {
        if (attribute is ClassicMovieAttribute classicMovieAttribute)
        {
            return new
ClassicMovieAttributeAdapter(classicMovieAttribute, stringLocalizer);
        }

        return baseProvider.GetAttributeAdapter(attribute,
stringLocalizer);
    }
}
```

3. Register the adapter provider for DI in `Program.cs`:

C#

```
builder.Services.AddRazorPages()
    .AddMvcOptions(options =>
    {
```

```
        options.MaxModelValidationErrors = 50;

options.ModelBindingMessageProvider.SetValueMustNotBeNullAccessor(
            _ => "The field is required.");
    });

builder.Services.AddSingleton
    <IValidationAttributeAdapterProvider,
CustomValidationAttributeAdapterProvider>();
```

IClientModelValidator for client-side validation

This method of rendering `data-` attributes in HTML is used by the `ClassicMovieWithClientValidator` attribute in the sample app. To add client validation by using this method:

- In the custom validation attribute, implement the IClientModelValidator interface and create an AddValidation method. In the `AddValidation` method, add `data-` attributes for validation, as shown in the following example:

C#

```
public class ClassicMovieWithClientValidatorAttribute :
    ValidationAttribute, IClientModelValidator
{
    public ClassicMovieWithClientValidatorAttribute(int year)
        => Year = year;

    public int Year { get; }

    public void AddValidation(ClientModelValidationContext context)
    {
        MergeAttribute(context.Attributes, "data-val", "true");
        MergeAttribute(context.Attributes, "data-val-classicmovie",
GetErrorMessage());

        var year = Year.ToString(CultureInfo.InvariantCulture);
        MergeAttribute(context.Attributes, "data-val-classicmovie-
year", year);
    }

    public string GetErrorMessage() =>
        $"Classic movies must have a release year no later than
{Year}.";

    protected override ValidationResult? IsValid(
        object? value, ValidationContext validationContext)
    {
        var movie = (Movie)validationContext.ObjectInstance;
        var releaseYear = ((DateTime)value!).Year;
```

```
        if (movie.Genre == Genre.Classic && releaseYear > Year)
        {
            return new ValidationResult(GetErrorMessage());
        }

        return ValidationResult.Success;
    }

    private static bool MergeAttribute(IDictionary<string, string>
attributes, string key, string value)
    {
        if (attributes.ContainsKey(key))
        {
            return false;
        }

        attributes.Add(key, value);
        return true;
    }
}
```

Disable client-side validation

The following code disables client validation in Razor Pages:

C#

```
builder.Services.AddRazorPages()
    .AddViewOptions(options =>
    {
        options.HtmlHelperOptions.ClientValidationEnabled = false;
    });
```

Other options to disable client-side validation:

- Comment out the reference to `_ValidationScriptsPartial` in all the `.cshtml` files.
- Remove the contents of the *Pages\Shared_ValidationScriptsPartial.cshtml* file.

The preceding approach won't prevent client-side validation of ASP.NET Core Identity Razor class library. For more information, see Scaffold Identity in ASP.NET Core projects.

Problem details

Problem Details are not the only response format to describe an HTTP API error, however, they are commonly used to report errors for HTTP APIs.

The problem details service implements the IProblemDetailsService interface, which supports creating problem details in ASP.NET Core. The AddProblemDetails(IServiceCollection) extension method on IServiceCollection registers the default `IProblemDetailsService` implementation.

In ASP.NET Core apps, the following middleware generates problem details HTTP responses when `AddProblemDetails` is called, except when the Accept request HTTP header doesn't include one of the content types supported by the registered IProblemDetailsWriter (default: `application/json`):

- ExceptionHandlerMiddleware: Generates a problem details response when a custom handler is not defined.
- StatusCodePagesMiddleware: Generates a problem details response by default.
- DeveloperExceptionPageMiddleware: Generates a problem details response in development when the `Accept` request HTTP header doesn't include `text/html`.

Additional resources

- System.ComponentModel.DataAnnotations
- Model Binding

Compatibility version for ASP.NET Core MVC

Article • 06/03/2022

By Rick Anderson

The SetCompatibilityVersion method is a no-op for ASP.NET Core 3.0 apps. That is, calling `SetCompatibilityVersion` with any value of CompatibilityVersion has no impact on the application.

- The next minor version of ASP.NET Core may provide a new `CompatibilityVersion` value.
- `CompatibilityVersion` values `Version_2_0` through `Version_2_2` are marked `[Obsolete(...)]`.
- See Breaking API changes in Antiforgery, CORS, Diagnostics, Mvc, and Routing. This list includes breaking changes for compatibility switches.

To see how `SetCompatibilityVersion` works with ASP.NET Core 2.x apps, select the ASP.NET Core 2.2 version of this article.

Write custom ASP.NET Core middleware

Article • 07/26/2024

By Fiyaz Hasan, Rick Anderson, and Steve Smith

Middleware is software that's assembled into an app pipeline to handle requests and responses. ASP.NET Core provides a rich set of built-in middleware components, but in some scenarios you might want to write a custom middleware.

This topic describes how to write *convention-based* middleware. For an approach that uses strong typing and per-request activation, see Factory-based middleware activation in ASP.NET Core.

Middleware class

Middleware is generally encapsulated in a class and exposed with an extension method. Consider the following inline middleware, which sets the culture for the current request from a query string:

C#

```
using System.Globalization;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseHttpsRedirection();

app.Use(async (context, next) =>
{
    var cultureQuery = context.Request.Query["culture"];
    if (!string.IsNullOrWhiteSpace(cultureQuery))
    {
        var culture = new CultureInfo(cultureQuery);

        CultureInfo.CurrentCulture = culture;
        CultureInfo.CurrentUICulture = culture;
    }

    // Call the next delegate/middleware in the pipeline.
    await next(context);
});

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
```

```
{CultureInfo.CurrentCulture.DisplayName}");
});

app.Run();
```

The preceding highlighted inline middleware is used to demonstrate creating a middleware component by calling Microsoft.AspNetCore.Builder.UseExtensions.Use. The preceding `Use` extension method adds a middleware delegate defined in-line to the application's request pipeline.

There are two overloads available for the `Use` extension:

- One takes a HttpContext and a `Func<Task>`. Invoke the `Func<Task>` without any parameters.
- The other takes a `HttpContext` and a RequestDelegate. Invoke the `RequestDelegate` by passing the `HttpContext`.

Prefer using the later overload as it saves two internal per-request allocations that are required when using the other overload.

Test the middleware by passing in the culture. For example, request `https://localhost:5001/?culture=es-es`.

For ASP.NET Core's built-in localization support, see Globalization and localization in ASP.NET Core.

The following code moves the middleware delegate to a class:

C#

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
    private readonly RequestDelegate _next;

    public RequestCultureMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
```

```
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
        await _next(context);
    }
}
```

The middleware class must include:

- A public constructor with a parameter of type RequestDelegate.
- A public method named `Invoke` or `InvokeAsync`. This method must:
  - Return a `Task`.
  - Accept a first parameter of type HttpContext.

Additional parameters for the constructor and `Invoke`/`InvokeAsync` are populated by dependency injection (DI).

Typically, an extension method is created to expose the middleware through IApplicationBuilder:

C#

```
using System.Globalization;

namespace Middleware.Example;

public class RequestCultureMiddleware
{
    private readonly RequestDelegate _next;

    public RequestCultureMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        var cultureQuery = context.Request.Query["culture"];
        if (!string.IsNullOrWhiteSpace(cultureQuery))
        {
            var culture = new CultureInfo(cultureQuery);

            CultureInfo.CurrentCulture = culture;
            CultureInfo.CurrentUICulture = culture;
        }

        // Call the next delegate/middleware in the pipeline.
```

```
        await _next(context);
    }
}

public static class RequestCultureMiddlewareExtensions
{
    public static IApplicationBuilder UseRequestCulture(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<RequestCultureMiddleware>();
    }
}
```

The following code calls the middleware from `Program.cs`:

C#

```
using Middleware.Example;
using System.Globalization;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseHttpsRedirection();

app.UseRequestCulture();

app.Run(async (context) =>
{
    await context.Response.WriteAsync(
        $"CurrentCulture.DisplayName:
{CultureInfo.CurrentCulture.DisplayName}");
});

app.Run();
```

Middleware dependencies

Middleware should follow the Explicit Dependencies Principle by exposing its dependencies in its constructor. Middleware is constructed once per *application lifetime*.

Middleware components can resolve their dependencies from dependency injection (DI) through constructor parameters. UseMiddleware can also accept additional parameters directly.

Per-request middleware dependencies

Middleware is constructed at app startup and therefore has application life time. Scoped lifetime services used by middleware constructors aren't shared with other dependency-injected types during each request. To share a *scoped* service between middleware and other types, add these services to the `InvokeAsync` method's signature. The `InvokeAsync` method can accept additional parameters that are populated by DI:

C#

```
namespace Middleware.Example;

public class MyCustomMiddleware
{
    private readonly RequestDelegate _next;

    public MyCustomMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    // IMessageWriter is injected into InvokeAsync
    public async Task InvokeAsync(HttpContext httpContext, IMessageWriter
svc)
    {
        svc.Write(DateTime.Now.Ticks.ToString());
        await _next(httpContext);
    }
}

public static class MyCustomMiddlewareExtensions
{
    public static IApplicationBuilder UseMyCustomMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<MyCustomMiddleware>();
    }
}
```

Lifetime and registration options contains a complete sample of middleware with *scoped* lifetime services.

The following code is used to test the preceding middleware:

C#

```
using Middleware.Example;
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IMessageWriter, LoggingMessageWriter>();

var app = builder.Build();
```

```
app.UseHttpsRedirection();

app.UseMyCustomMiddleware();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The `IMessageWriter` interface and implementation:

C#

```
namespace Middleware.Example;

public interface IMessageWriter
{
    void Write(string message);
}

public class LoggingMessageWriter : IMessageWriter
{

    private readonly ILogger<LoggingMessageWriter> _logger;

    public LoggingMessageWriter(ILogger<LoggingMessageWriter> logger) =>
        _logger = logger;

    public void Write(string message) =>
        _logger.LogInformation(message);
}
```

Additional resources

- Sample code used in this article
- UseExtensions source on GitHub
- Lifetime and registration options contains a complete sample of middleware with *scoped*, *transient*, and *singleton* lifetime services.
- DEEP DIVE: HOW IS THE ASP.NET CORE MIDDLEWARE PIPELINE BUILT
- ASP.NET Core Middleware
- Test ASP.NET Core middleware
- Migrate HTTP handlers and modules to ASP.NET Core middleware
- App startup in ASP.NET Core
- Request Features in ASP.NET Core
- Factory-based middleware activation in ASP.NET Core
- Middleware activation with a third-party container in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Request and response operations in ASP.NET Core

Article • 07/26/2024

By Justin Kotalik

This article explains how to read from the request body and write to the response body. Code for these operations might be required when writing middleware. Outside of writing middleware, custom code isn't generally required because the operations are handled by MVC and Razor Pages.

There are two abstractions for the request and response bodies: Stream and Pipe. For request reading, HttpRequest.Body is a Stream, and `HttpRequest.BodyReader` is a PipeReader. For response writing, HttpResponse.Body is a Stream, and `HttpResponse.BodyWriter` is a PipeWriter.

Pipelines are recommended over streams. Streams can be easier to use for some simple operations, but pipelines have a performance advantage and are easier to use in most scenarios. ASP.NET Core is starting to use pipelines instead of streams internally. Examples include:

- `FormReader`
- `TextReader`
- `TextWriter`
- `HttpResponse.WriteAsync`

Streams aren't being removed from the framework. Streams continue to be used throughout .NET, and many stream types don't have pipe equivalents, such as `FileStreams` and `ResponseCompression`.

Stream examples

Suppose the goal is to create a middleware that reads the entire request body as a list of strings, splitting on new lines. A simple stream implementation might look like the following example:

> ⚠ **Warning**
>
> The following code:

- Is used to demonstrate the problems with not using a pipe to read the request body.
- Is not intended to be used in production apps.

C#

```
private async Task<List<string>> GetListOfStringsFromStream(Stream
requestBody)
{
    // Build up the request body in a string builder.
    StringBuilder builder = new StringBuilder();

    // Rent a shared buffer to write the request body into.
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);
        if (bytesRemaining == 0)
        {
            break;
        }

        // Append the encoded string into the string builder.
        var encodedString = Encoding.UTF8.GetString(buffer, 0,
bytesRemaining);
        builder.Append(encodedString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    var entireRequestBody = builder.ToString();

    // Split on \n in the string.
    return new List<string>(entireRequestBody.Split("\n"));
}
```

If you would like to see code comments translated to languages other than English, let us know in this GitHub discussion issue.

This code works, but there are some issues:

- Before appending to the `StringBuilder`, the example creates another string (`encodedString`) that is thrown away immediately. This process occurs for all bytes in the stream, so the result is extra memory allocation the size of the entire request body.

- The example reads the entire string before splitting on new lines. It's more efficient to check for new lines in the byte array.

Here's an example that fixes some of the preceding issues:

> **⚠ Warning**
>
> The following code:
>
> - Is used to demonstrate the solutions to some problems in the preceding code while not solving all the problems.
> - Is not intended to be used in production apps.

C#

```
private async Task<List<string>>
GetListOfStringsFromStreamMoreEfficient(Stream requestBody)
{
    StringBuilder builder = new StringBuilder();
    byte[] buffer = ArrayPool<byte>.Shared.Rent(4096);
    List<string> results = new List<string>();

    while (true)
    {
        var bytesRemaining = await requestBody.ReadAsync(buffer, offset: 0,
buffer.Length);

        if (bytesRemaining == 0)
        {
            results.Add(builder.ToString());
            break;
        }

        // Instead of adding the entire buffer into the StringBuilder
        // only add the remainder after the last \n in the array.
        var prevIndex = 0;
        int index;
        while (true)
        {
            index = Array.IndexOf(buffer, (byte)'\n', prevIndex);
            if (index == -1)
            {
                break;
            }

            var encodedString = Encoding.UTF8.GetString(buffer, prevIndex,
index - prevIndex);

            if (builder.Length > 0)
            {
```

```
                // If there was a remainder in the string buffer, include it
in the next string.
                results.Add(builder.Append(encodedString).ToString());
                builder.Clear();
            }
            else
            {
                results.Add(encodedString);
            }

            // Skip past last \n
            prevIndex = index + 1;
        }

        var remainingString = Encoding.UTF8.GetString(buffer, prevIndex,
bytesRemaining - prevIndex);
        builder.Append(remainingString);
    }

    ArrayPool<byte>.Shared.Return(buffer);

    return results;
}
```

This preceding example:

- Doesn't buffer the entire request body in a `StringBuilder` unless there aren't any newline characters.
- Doesn't call `Split` on the string.

However, there are still a few issues:

- If newline characters are sparse, much of the request body is buffered in the string.
- The code continues to create strings (`remainingString`) and adds them to the string buffer, which results in an extra allocation.

These issues are fixable, but the code is becoming progressively more complicated with little improvement. Pipelines provide a way to solve these problems with minimal code complexity.

Pipelines

The following example shows how the same scenario can be handled using a PipeReader:

```
C#
```

```
private async Task<List<string>> GetListOfStringFromPipe(PipeReader reader)
{
    List<string> results = new List<string>();

    while (true)
    {
        ReadResult readResult = await reader.ReadAsync();
        var buffer = readResult.Buffer;

        SequencePosition? position = null;

        do
        {
            // Look for a EOL in the buffer
            position = buffer.PositionOf((byte)'\n');

            if (position != null)
            {
                var readOnlySequence = buffer.Slice(0, position.Value);
                AddStringToList(results, in readOnlySequence);

                // Skip the line + the \n character (basically position)
                buffer = buffer.Slice(buffer.GetPosition(1,
position.Value));
            }
        }
        while (position != null);


        if (readResult.IsCompleted && buffer.Length > 0)
        {
            AddStringToList(results, in buffer);
        }

        reader.AdvanceTo(buffer.Start, buffer.End);

        // At this point, buffer will be updated to point one byte after the
last
        // \n character.
        if (readResult.IsCompleted)
        {
            break;
        }
    }

    return results;
}

private static void AddStringToList(List<string> results, in
ReadOnlySequence<byte> readOnlySequence)
{
    // Separate method because Span/ReadOnlySpan cannot be used in async
methods
    ReadOnlySpan<byte> span = readOnlySequence.IsSingleSegment ?
```

```
        readOnlySequence.First.Span : readOnlySequence.ToArray().AsSpan();
        results.Add(Encoding.UTF8.GetString(span));
    }
```

This example fixes many issues that the streams implementations had:

- There's no need for a string buffer because the `PipeReader` handles bytes that haven't been used.
- Encoded strings are directly added to the list of returned strings.
- Other than the `ToArray` call, and the memory used by the string, string creation is allocation free.

Adapters

The `Body`, `BodyReader`, and `BodyWriter` properties are available for `HttpRequest` and `HttpResponse`. When you set `Body` to a different stream, a new set of adapters automatically adapt each type to the other. If you set `HttpRequest.Body` to a new stream, `HttpRequest.BodyReader` is automatically set to a new `PipeReader` that wraps `HttpRequest.Body`.

StartAsync

`HttpResponse.StartAsync` is used to indicate that headers are unmodifiable and to run `OnStarting` callbacks. When using Kestrel as a server, calling `StartAsync` before using the `PipeReader` guarantees that memory returned by `GetMemory` belongs to Kestrel's internal Pipe rather than an external buffer.

Additional resources

- System.IO.Pipelines in .NET
- Write custom ASP.NET Core middleware

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Request decompression in ASP.NET Core

Article • 07/26/2024

By David Acker

Request decompression middleware:

- Enables API endpoints to accept requests with compressed content.
- Uses the Content-Encoding HTTP header to automatically identify and decompress requests which contain compressed content.
- Eliminates the need to write code to handle compressed requests.

When the `Content-Encoding` header value on a request matches one of the available decompression providers, the middleware:

- Uses the matching provider to wrap the HttpRequest.Body in an appropriate decompression stream.
- Removes the `Content-Encoding` header, indicating that the request body is no longer compressed.

Requests that don't include a `Content-Encoding` header are ignored by the request decompression middleware.

Decompression:

- Occurs when the body of the request is read. That is, decompression occurs at the endpoint on model binding. The request body isn't decompressed eagerly.
- When attempting to read the decompressed request body with invalid compressed data for the specified `Content-Encoding`, an exception is thrown. Brotli can throw System.InvalidOperationException: Decoder ran into invalid data. Deflate and GZip can throw System.IO.InvalidDataException: The archive entry was compressed using an unsupported compression method.

If the middleware encounters a request with compressed content but is unable to decompress it, the request is passed to the next delegate in the pipeline. For example, a request with an unsupported `Content-Encoding` header value or multiple `Content-Encoding` header values is passed to the next delegate in the pipeline.

Configuration

The following code uses AddRequestDecompression(IServiceCollection) and UseRequestDecompression to enable request decompression for the default `Content-`

`Encoding` types:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRequestDecompression();

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Default decompression providers

The `Content-Encoding` header values that the request decompression middleware supports by default are listed in the following table:

Expand table

Content-Encoding header values	Description
`br`	Brotli compressed data format
`deflate`	DEFLATE compressed data format
`gzip`	Gzip file format

Custom decompression providers

Support for custom encodings can be added by creating custom decompression provider classes that implement IDecompressionProvider:

```
C#

public class CustomDecompressionProvider : IDecompressionProvider
{
    public Stream GetDecompressionStream(Stream stream)
    {
        // Perform custom decompression logic here
        return stream;
    }
}
```

Custom decompression providers are registered with RequestDecompressionOptions along with their corresponding `Content-Encoding` header values:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRequestDecompression(options =>
{
    options.DecompressionProviders.Add("custom", new
CustomDecompressionProvider());
});

var app = builder.Build();

app.UseRequestDecompression();

app.MapPost("/", (HttpRequest request) => Results.Stream(request.Body));

app.Run();
```

Request size limits

In order to protect against zip bombs or decompression bombs :

- The maximum size of the decompressed request body is limited to the request body size limit enforced by the endpoint or server.
- If the number of bytes read from the decompressed request body stream exceeds the limit, an InvalidOperationException is thrown to prevent additional bytes from being read from the stream.

In order of precedence, the maximum request size for an endpoint is set by:

1. IRequestSizeLimitMetadata.MaxRequestBodySize, such as RequestSizeLimitAttribute or DisableRequestSizeLimitAttribute for MVC endpoints.
2. The global server size limit IHttpMaxRequestBodySizeFeature.MaxRequestBodySize. `MaxRequestBodySize` can be overridden per request with IHttpMaxRequestBodySizeFeature.MaxRequestBodySize, but defaults to the limit configured for the web server implementation.

Expand table

Web server implementation	`MaxRequestBodySize` configuration
HTTP.sys	HttpSysOptions.MaxRequestBodySize
IIS	IISServerOptions.MaxRequestBodySize
Kestrel	KestrelServerLimits.MaxRequestBodySize

> ⚠ **Warning**
>
> Disabling the request body size limit poses a security risk in regards to uncontrolled resource consumption, particularly if the request body is being buffered. Ensure that safeguards are in place to mitigate the risk of denial-of-service (DoS) attacks.

Additional Resources

- ASP.NET Core Middleware
- Mozilla Developer Network: Content-Encoding
- Brotli Compressed Data Format
- DEFLATE Compressed Data Format Specification version 1.3
- GZIP file format specification version 4.3

URL Rewriting Middleware in ASP.NET Core

Article • 09/25/2023

By Kirk Larkin and Rick Anderson

This article introduces URL rewriting with instructions on how to use URL Rewriting Middleware in ASP.NET Core apps.

URL rewriting is the act of modifying request URLs based on one or more predefined rules. URL rewriting creates an abstraction between resource locations and their addresses so that the locations and addresses aren't tightly linked. URL rewriting is valuable in several scenarios to:

- Move or replace server resources temporarily or permanently and maintain stable locators for those resources.
- Split request processing across different apps or across areas of one app.
- Remove, add, or reorganize URL segments on incoming requests.
- Optimize public URLs for Search Engine Optimization (SEO).
- Permit the use of friendly public URLs to help visitors predict the content returned by requesting a resource.
- Redirect insecure requests to secure endpoints.
- Prevent hotlinking, where an external site uses a hosted static asset on another site by linking the asset into its own content.

URL rewriting can reduce the performance of an app. Limit the number and complexity of rules.

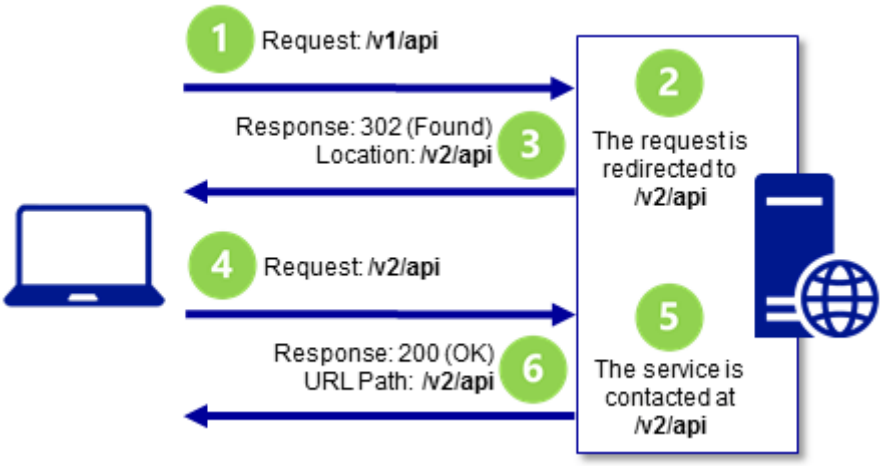
URL redirect and URL rewrite

The difference in wording between *URL redirect* and *URL rewrite* is subtle but has important implications for providing resources to clients. ASP.NET Core's URL Rewriting Middleware is capable of meeting the need for both.

A *URL redirect* involves a client-side operation, where the client is instructed to access a resource at a different address than the client originally requested. This requires a round trip to the server. The redirect URL returned to the client appears in the browser's address bar when the client makes a new request for the resource.

If `/resource` is *redirected* to `/different-resource`, the server responds that the client should obtain the resource at `/different-resource` with a status code indicating that

the redirect is either temporary or permanent.



When redirecting requests to a different URL, indicate whether the redirect is permanent or temporary by specifying the status code with the response:

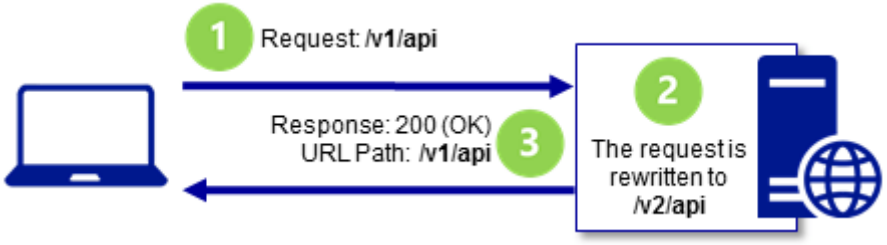
- The 301 - Moved Permanently status code is used where the resource has a new, permanent URL and that all future requests for the resource should use the new URL. *The client may cache and reuse the response when a 301 status code is received.*
- The 302 - Found status code is used where the redirection is temporary or generally subject to change. The 302 status code indicates to the client not to store the URL and use it in the future.

For more information on status codes, see RFC 9110: Status Code Definitions.

A *URL rewrite* is a server-side operation that provides a resource from a different resource address than the client requested. Rewriting a URL doesn't require a round trip to the server. The rewritten URL isn't returned to the client and doesn't appear in the browser's address bar.

If `/resource` is *rewritten* to `/different-resource`, the server *internally* fetches and returns the resource at `/different-resource`.

Although the client might be able to retrieve the resource at the rewritten URL, the client isn't informed that the resource exists at the rewritten URL when it makes its request and receives the response.



URL rewriting sample app

Explore the features of the URL Rewriting Middleware with the sample app. The app applies redirect and rewrite rules and shows the redirected or rewritten URL for several scenarios.

When to use URL rewriting middleware

Use URL Rewriting Middleware when the following approaches aren't satisfactory:

- URL Rewrite module with IIS on Windows Server
- Apache mod_rewrite module on Apache Server
- URL rewriting on Nginx

Use the URL rewriting middleware when the app is hosted on HTTP.sys server.

The main reasons to use the server-based URL rewriting technologies in IIS, Apache, and Nginx are:

- The middleware doesn't support the full features of these modules.

  Some of the features of the server modules don't work with ASP.NET Core projects, such as the `IsFile` and `IsDirectory` constraints of the IIS Rewrite module. In these scenarios, use the middleware instead.

- The performance of the middleware probably doesn't match that of the modules.

  Benchmarking is the only way to know with certainty which approach degrades performance the most or if degraded performance is negligible.

Extension and options

Establish URL rewrite and redirect rules by creating an instance of the RewriteOptions class with extension methods for each of the rewrite rules. Chain multiple rules ***in the order that they should be processed***. The `RewriteOptions` are passed into the URL Rewriting Middleware as it's added to the request pipeline with UseRewriter:

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
```

```
using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

In the preceding code, MethodRules is a user defined class. See RewriteRules.cs in this article for more information.

Redirect non-www to www

Three options permit the app to redirect non-`www` requests to `www`:

- AddRedirectToWwwPermanent: Permanently redirect the request to the `www` subdomain if the request is non-`www`. Redirects with a Status308PermanentRedirect status code.
- AddRedirectToWww: Redirect the request to the `www` subdomain if the incoming request is non-`www`. Redirects with a Status307TemporaryRedirect status code. An overload permits providing the status code for the response. Use a field of the StatusCodes class for a status code assignment.

URL redirect

Use AddRedirect to redirect requests. The first parameter contains the .NET regular expression (Regex) for matching on the path of the incoming URL. The second parameter is the replacement string. The third parameter, if present, specifies the status code. If the status code isn't specified, the status code defaults to 302 - Found, which indicates that the resource is temporarily moved or replaced.

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

In a browser with developer tools enabled, make a request to the sample app with the path `/redirect-rule/1234/5678`. The regular expression matches the request path on `redirect-rule/(.*)`, and the path is replaced with `/redirected/1234/5678`. The redirect URL is sent back to the client with a *302 - Found* status code. The browser makes a new request at the redirect URL, which appears in the browser's address bar. Since no rules in the sample app match on the redirect URL:

- The second request receives a *200 - OK* response from the app.
- The body of the response shows the redirect URL.

A round trip is made to the server when a URL is *redirected*.

> ⚠ **Warning**
>
> Be cautious when establishing redirect rules. Redirect rules are evaluated on every request to the app, including after a redirect. It's easy to accidentally create a loop of ***infinite*** redirects.

The part of the expression contained within parentheses is called a capture group. The dot (`.`) of the expression means *match any character*. The asterisk (`*`) indicates *match the preceding character zero or more times*. Therefore, the last two path segments of the URL, `1234/5678`, are captured by capture group `(.*)`. Any value provided in the request URL after `redirect-rule/` is captured by this single capture group.

In the replacement string, captured groups are injected into the string with the dollar sign (`$`) followed by the sequence number of the capture. The first capture group value is obtained with `$1`, the second with `$2`, and they continue in sequence for the capture groups in the regular expression. There's only one captured group in the redirect rule regular expression in `redirect-rule/(.*)`, so there's only one injected group in the replacement string, which is `$1`. When the rule is applied, the URL becomes `/redirected/1234/5678`.

Try `/redirect-rule/1234/5678` with the browser tools on the network tab.

URL redirect to a secure endpoint

Use AddRedirectToHttps to redirect HTTP requests to the same host and path using the HTTPS protocol. If the status code isn't supplied, the middleware defaults to *302 - Found*. If the port isn't supplied:

- The middleware defaults to `null`.
- The scheme changes to `https` (HTTPS protocol), and the client accesses the resource on port 443.

The following example shows how to set the status code to `301 - Moved Permanently` and change the port to the HTTPS port used by Kestrel on localhost. In production, the HTTPS port is set to null:

```
```

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

int? localhostHTTPSport = null;
if (app.Environment.IsDevelopment())
{
    localhostHTTPSport = Int32.Parse(Environment.GetEnvironmentVariable(
                    "ASPNETCORE_URLS")!.Split(new Char[] { ':', ';' })[2]);
}

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        // localhostHTTPport not needed for production, used only with
localhost.
        .AddRedirectToHttps(301, localhostHTTPSport)
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

Use AddRedirectToHttpsPermanent to redirect insecure requests to the same host and path with secure HTTPS protocol on port 443. The middleware sets the status code to `301 - Moved Permanently`.

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

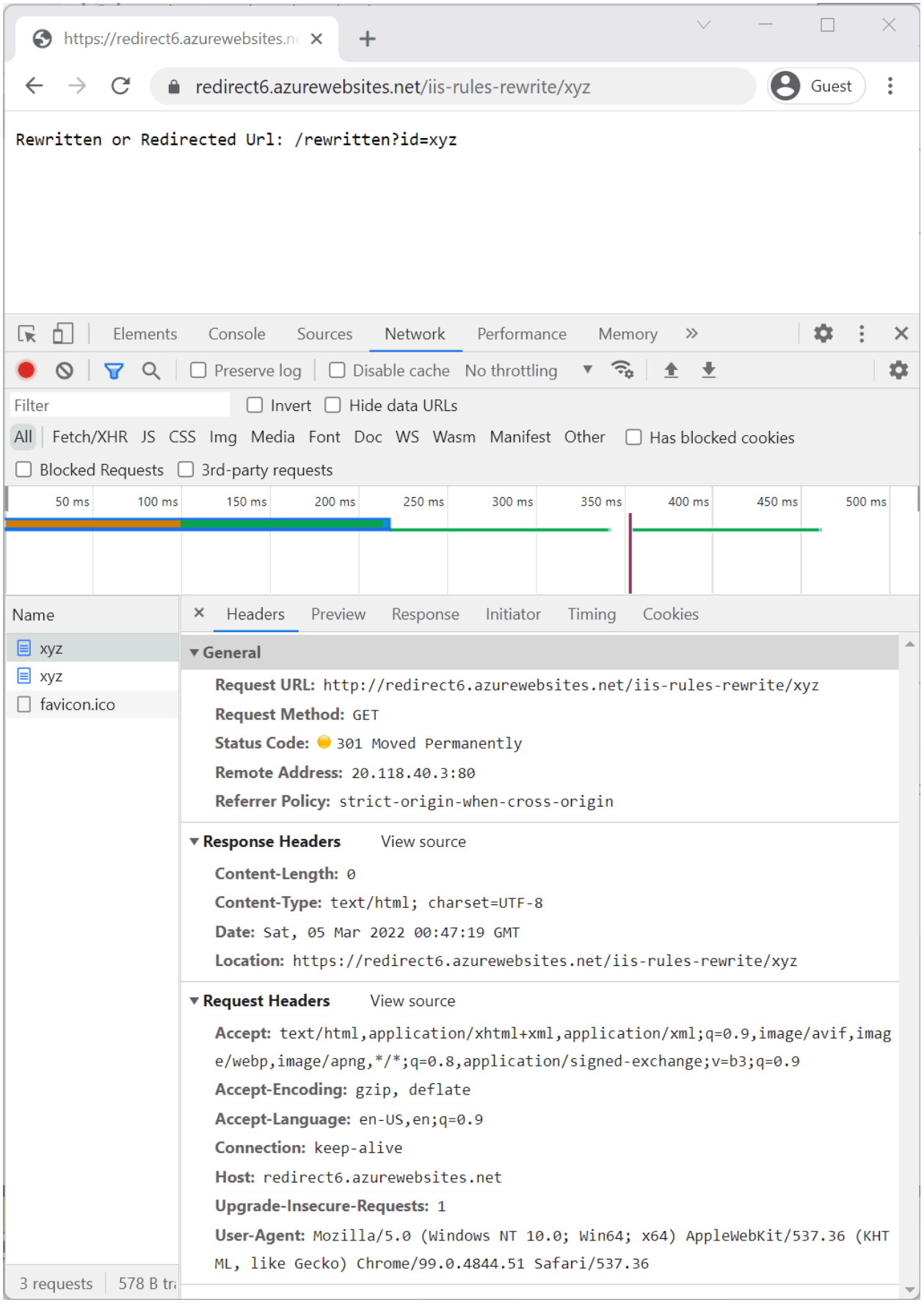
> ⓘ Note
>
> When redirecting to a secure endpoint without the requirement for additional redirect rules, we recommend using HTTPS Redirection Middleware. For more information, see Enforce HTTPS.

The sample app demonstrates how to use `AddRedirectToHttps` or `AddRedirectToHttpsPermanent`. Make an insecure HTTP request to the app at `http://redirect6.azurewebsites.net/iis-rules-rewrite/xyz`. When testing HTTP to HTTPS redirection with localhost:

- Use the HTTP URL, which has a different port than the HTTPS URL. The HTTP URL is in the `Properties/launchSettings.json` file.

- Removing the `s` from `https://localhost/{port}` fails because localhost doesn't respond on HTTP to the HTTPS port.

The following image shows the F12 browser tools image of a request to `http://redirect6.azurewebsites.net/iis-rules-rewrite/xyz` using the preceding code:



URL rewrite

Use AddRewrite to create a rule for rewriting URLs. The first parameter contains the regular expression for matching on the incoming URL path. The second parameter is the replacement string. The third parameter, `skipRemainingRules: {true|false}`, indicates to the middleware whether or not to skip additional rewrite rules if the current rule is applied.

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

Try the request to `https://redirect6.azurewebsites.net/rewrite-rule/1234/5678`

The caret (`^`) at the beginning of the expression means that matching starts at the beginning of the URL path.

In the earlier example with the redirect rule, `redirect-rule/(.*)`, there's no caret (`^`) at the start of the regular expression. Therefore, any characters may precede `redirect-rule/` in the path for a successful match.

Expand table

Path	Match
`/redirect-rule/1234/5678`	Yes
`/my-cool-redirect-rule/1234/5678`	Yes
`/anotherredirect-rule/1234/5678`	Yes

The rewrite rule, `^rewrite-rule/(\d+)/(\d+)`, only matches paths if they start with `rewrite-rule/`. In the following table, note the difference in matching.

Expand table

Path	Match
`/rewrite-rule/1234/5678`	Yes
`/my-cool-rewrite-rule/1234/5678`	No
`/anotherrewrite-rule/1234/5678`	No

Following the `^rewrite-rule/` portion of the expression, there are two capture groups, `(\d+)/(\d+)`. The `\d` signifies *match a digit (number)*. The plus sign (`+`) means *match one or more of the preceding character*. Therefore, the URL must contain a number followed by a forward-slash followed by another number. These capture groups are injected into the rewritten URL as `$1` and `$2`. The rewrite rule replacement string places the captured groups into the query string. The requested path `/rewrite-rule/1234/5678` is rewritten to return the resource at `/rewritten?var1=1234&var2=5678`. If a query string is present on the original request, it's preserved when the URL is rewritten.

There's no round trip to the server to return the resource. If the resource exists, it's fetched and returned to the client with a *200 - OK* status code. Because the client isn't redirected, the URL in the browser's address bar doesn't change. Clients can't detect that a URL rewrite operation occurred on the server.

Performance tips for URL rewrite and redirect

For the fastest response:

- Order rewrite rules from the most frequently matched rule to the least frequently matched rule.
- Use `skipRemainingRules: true` whenever possible because matching rules is computationally expensive and increases app response time. Skip the processing of the remaining rules when a match occurs and no additional rule processing is required.

> ⚠ **Warning**
>
> A malicious user can provide expensive to process input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout. For example, the RedirectRule and RewriteRule classes both pass in a one second timeout.

Apache mod_rewrite

Apply Apache mod_rewrite rules with AddApacheModRewrite. Make sure that the rules file is deployed with the app. For more information and examples of mod_rewrite rules, see Apache mod_rewrite.

A StreamReader is used to read the rules from the *ApacheModRewrite.txt* rules file:

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
```

```
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The sample app redirects requests from `/apache-mod-rules-redirect/(.\*)` to `/redirected?id=$1`. The response status code is *302 - Found*.

```

# Rewrite path with additional sub directory
RewriteRule ^/apache-mod-rules-redirect/(.*) /redirected?id=$1 [L,R=302]
```

Try the request to `https://redirect6.azurewebsites.net/apache-mod-rules-redirect/1234`

The Apache middleware supports the following Apache mod_rewrite server variables:

- CONN_REMOTE_ADDR
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_FORWARDED
- HTTP_HOST
- HTTP_REFERER
- HTTP_USER_AGENT
- HTTPS
- IPV6
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_METHOD
- REQUEST_SCHEME
- REQUEST_URI
- SCRIPT_FILENAME

- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL
- TIME
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_MON
- TIME_SEC
- TIME_WDAY
- TIME_YEAR

IIS URL Rewrite Module rules

To use the same rule set that applies to the IIS URL Rewrite Module, use AddIISUrlRewrite. Make sure that the rules file is deployed with the app. Don't direct the middleware to use the app's *web.config* file when running on Windows Server IIS. With IIS, these rules should be stored outside of the app's *web.config* file in order to avoid conflicts with the IIS Rewrite module. For more information and examples of IIS URL Rewrite Module rules, see Using Url Rewrite Module 2.0 and URL Rewrite Module Configuration Reference.

A StreamReader is used to read the rules from the `IISUrlRewrite.xml` rules file:

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
```

```
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The sample app rewrites requests from `/iis-rules-rewrite/(.*)` to `/rewritten?id=$1`. The response is sent to the client with a *200 - OK* status code.

XML

```
<rewrite>
  <rules>
    <rule name="Rewrite segment to id querystring" stopProcessing="true">
      <match url="^iis-rules-rewrite/(.*)$" />
      <action type="Rewrite" url="rewritten?id={R:1}"
appendQueryString="false"/>
    </rule>
  </rules>
</rewrite>
```

Try the request to `https://redirect6.azurewebsites.net/iis-rules-rewrite/xyz`

Apps that have an active IIS Rewrite Module with server-level rules configured that impacts the app in undesirable ways:

- Consider disabling the IIS Rewrite Module for the app.
- For more information, see Disabling IIS modules.

Unsupported features

The middleware doesn't support the following IIS URL Rewrite Module features:

- Outbound Rules
- Custom Server Variables
- Wildcards
- LogRewrittenUrl

Supported server variables

The middleware supports the following IIS URL Rewrite Module server variables:

- CONTENT_LENGTH
- CONTENT_TYPE
- HTTP_ACCEPT
- HTTP_CONNECTION
- HTTP_COOKIE
- HTTP_HOST
- HTTP_REFERER
- HTTP_URL
- HTTP_USER_AGENT
- HTTPS
- LOCAL_ADDR
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_PORT
- REQUEST_FILENAME
- REQUEST_URI

IFileProvider can be obtained via a PhysicalFileProvider. This approach may provide greater flexibility for the location of rewrite rules files. Make sure that the rewrite rules files are deployed to the server at the path provided.

```
C#

var fileProvider = new
PhysicalFileProvider(Directory.GetCurrentDirectory());
```

Method-based rule

Use Add to implement custom rule logic in a method. `Add` exposes the RewriteContext, which makes available the HttpContext for use in redirect methods. The RewriteContext.Result property determines how additional pipeline processing is handled. Set the value to one of the RuleResult fields described in the following table.

Expand table

Rewrite context result	Action
`RuleResult.ContinueRules`	Continue applying rules.

Rewrite context result	Action
(default)	
`RuleResult.EndResponse`	Stop applying rules and send the response.
`RuleResult.SkipRemainingRules`	Stop applying rules and send the context to the next middleware.

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The sample app demonstrates a method that redirects requests for paths that end with `.xml`. When a request is made for `/file.xml`:

- The request is redirected to `/xmlfiles/file.xml`

- The status code is set to `301 - Moved Permanently`. When the browser makes a new request for `/xmlfiles/file.xml`, Static File Middleware serves the file to the client from the *wwwroot/xmlfiles* folder. For a redirect, explicitly set the status code of the response. Otherwise, a *200 - OK* status code is returned, and the redirect doesn't occur on the client.

`RewriteRules.cs`:

C#

```
public static void RedirectXmlFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    // Because the client is redirecting back to the same app, stop
    // processing if the request has already been redirected.
    if (request.Path.StartsWithSegments(new PathString("/xmlfiles")) ||
        request.Path.Value==null)
    {
        return;
    }

    if (request.Path.Value.EndsWith(".xml",
StringComparison.OrdinalIgnoreCase))
    {
        var response = context.HttpContext.Response;
        response.StatusCode = (int) HttpStatusCode.MovedPermanently;
        context.Result = RuleResult.EndResponse;
        response.Headers[HeaderNames.Location] =
            "/xmlfiles" + request.Path + request.QueryString;
    }
}
```

This approach can also rewrite requests. The sample app demonstrates rewriting the path for any text file request to serve the *file.txt* text file from the *wwwroot* folder. Static File Middleware serves the file based on the updated request path:

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
```

```
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

`RewriteRules.cs`:

C#

```
public static void RewriteTextFileRequests(RewriteContext context)
{
    var request = context.HttpContext.Request;

    if (request.Path.Value != null &&
        request.Path.Value.EndsWith(".txt",
StringComparison.OrdinalIgnoreCase))
    {
        context.Result = RuleResult.SkipRemainingRules;
        request.Path = "/file.txt";
    }
}
```

IRule-based rule

Use Add to use rule logic in a class that implements the IRule interface. `IRule` provides greater flexibility over using the method-based rule approach. The implementation class may include a constructor that allows passing in parameters for the ApplyRule method.

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

The values of the parameters in the sample app for the `extension` and the `newPath` are checked to meet several conditions. The `extension` must contain a value, and the value must be `.png`, `.jpg`, or `.gif`. If the `newPath` isn't valid, an ArgumentException is thrown. If a request is made for `image.png`, the request is redirected to `/png-images/image.png`. If a request is made for `image.jpg`, the request is redirected to `/jpg-images/image.jpg`. The status code is set to `301 - Moved Permanently`, and the `context.Result` is set to stop processing rules and send the response.

C#

```
public class RedirectImageRequests : IRule
{
    private readonly string _extension;
    private readonly PathString _newPath;
```

```
    public RedirectImageRequests(string extension, string newPath)
    {
        if (string.IsNullOrEmpty(extension))
        {
            throw new ArgumentException(nameof(extension));
        }

        if (!Regex.IsMatch(extension, @"^\.(png|jpg|gif)$"))
        {
            throw new ArgumentException("Invalid extension",
nameof(extension));
        }

        if (!Regex.IsMatch(newPath, @"(/[A-Za-z0-9]+)+?"))
        {
            throw new ArgumentException("Invalid path", nameof(newPath));
        }

        _extension = extension;
        _newPath = new PathString(newPath);
    }

    public void ApplyRule(RewriteContext context)
    {
        var request = context.HttpContext.Request;

        // Because we're redirecting back to the same app, stop
        // processing if the request has already been redirected
        if (request.Path.StartsWithSegments(new PathString(_newPath)) ||
            request.Path.Value == null)
        {
            return;
        }

        if (request.Path.Value.EndsWith(_extension,
StringComparison.OrdinalIgnoreCase))
        {
            var response = context.HttpContext.Response;
            response.StatusCode = (int) HttpStatusCode.MovedPermanently;
            context.Result = RuleResult.EndResponse;
            response.Headers[HeaderNames.Location] =
                _newPath + request.Path + request.QueryString;
        }
    }
}
```

Try:

- PNG request: `https://redirect6.azurewebsites.net/image.png`
- JPG request: `https://redirect6.azurewebsites.net/image.jpg`

Regex examples

Expand table

Goal	Regex String & Match Example	Replacement String & Output Example
Rewrite path into querystring	`^path/(.*)/(.*)` `/path/abc/123`	`path?var1=$1&var2=$2` `/path?var1=abc&var2=123`
Strip trailing slash	`^path2/(.*)/$` `/path2/xyz/`	`$1` `/path2/xyz`
Enforce trailing slash	`^path3/(.*[^/])$` `/path3/xyz`	`$1/` `/path3/xyz/`

Avoid rewriting specific requests	`^(.*)(?<!\.axd)$` or
	`^(?!.*\.axd$)(.*)$`
	Yes: `/path4/resource.htm`
	No: `/path4/resource.axd`
		`rewritten/$1`
		`/rewritten/resource.htm`
		`/resource.axd`

Rearrange URL segments	`path5/(.*)/(.*)/(.*)` `path5/1/2/3`	`path5/$3/$2/$1` `path5/3/2/1`
Replace a URL segment	`^path6/(.*)/segment2/(.*)` `^path6/segment1/segment2/segment3`	`path6/$1/replaced/$2` `/path6/segment1/replaced/segment3`

The links in the preceding table use the following code deployed to Azure:

C#

```
using Microsoft.AspNetCore.Rewrite;
using RewriteRules;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

using (StreamReader apacheModRewriteStreamReader =
    File.OpenText("ApacheModRewrite.txt"))
using (StreamReader iisUrlRewriteStreamReader =
    File.OpenText("IISUrlRewrite.xml"))
{
    var options = new RewriteOptions()
        .AddRedirectToHttpsPermanent()
        .AddRedirect("redirect-rule/(.*)", "redirected/$1")
        .AddRewrite(@"^rewrite-rule/(\d+)/(\d+)", "rewritten?
var1=$1&var2=$2",
            skipRemainingRules: true)
```

```
        // Rewrite path to QS.
        .AddRewrite(@"^path/(.*)/(.*)", "path?var1=$1&var2=$2",
            skipRemainingRules: true)
        // Skip trailing slash.
        .AddRewrite(@"^path2/(.*)/$", "path2/$1",
            skipRemainingRules: true)
         // Enforce trailing slash.
         .AddRewrite(@"^path3/(.*[^/])$", "path3/$1/",
            skipRemainingRules: true)
         // Avoid rewriting specific requests.
         .AddRewrite(@"^path4/(.*)(?<!\.axd)$", "rewritten/$1",
            skipRemainingRules: true)
         // Rearrange URL segments
         .AddRewrite(@"^path5/(.*)/(.*)/(.*)", "path5/$3/$2/$1",
            skipRemainingRules: true)
          // Replace a URL segment
          .AddRewrite(@"^path6/(.*)/segment2/(.*)", "path6/$1/replaced/$2",
            skipRemainingRules: true)
```

```
        .AddApacheModRewrite(apacheModRewriteStreamReader)
        .AddIISUrlRewrite(iisUrlRewriteStreamReader)
        .Add(MethodRules.RedirectXmlFileRequests)
        .Add(MethodRules.RewriteTextFileRequests)
        .Add(new RedirectImageRequests(".png", "/png-images"))
        .Add(new RedirectImageRequests(".jpg", "/jpg-images"));

    app.UseRewriter(options);
}

app.UseStaticFiles();

app.Run(context => context.Response.WriteAsync(
    $"Rewritten or Redirected Url: " +
    $"{context.Request.Path + context.Request.QueryString}"));

app.Run();
```

In most of the preceding regular expression samples, the literal `path` is used to make unique testable rewrite rules for the deployed sample. Typically the regular expression wouldn't include `path`. For example, see these regular expression examples table.

Additional resources

- View or download sample code (how to download)
- RewriteMiddleware source on GitHub
- App startup in ASP.NET Core
- ASP.NET Core Middleware
- Regular expressions in .NET
- Regular expression language - quick reference

- Apache mod_rewrite
- Using Url Rewrite Module 2.0 (for IIS)
- URL Rewrite Module Configuration Reference
- Keep a simple URL structure
- 10 URL Rewriting Tips and Tricks
- To slash or not to slash

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

File Providers in ASP.NET Core

Article • 07/26/2024

By Steve Smith

ASP.NET Core abstracts file system access through the use of File Providers. File Providers are used throughout the ASP.NET Core framework. For example:

- IWebHostEnvironment exposes the app's content root and web root as `IFileProvider` types.
- Static File Middleware uses File Providers to locate static files.
- Razor uses File Providers to locate pages and views.
- .NET Core tooling uses File Providers and glob patterns to specify which files should be published.

View or download sample code (how to download)

File Provider interfaces

The primary interface is IFileProvider. `IFileProvider` exposes methods to:

- Obtain file information (IFileInfo).
- Obtain directory information (IDirectoryContents).
- Set up change notifications (using an IChangeToken).

`IFileInfo` provides methods and properties for working with files:

- Exists
- IsDirectory
- Name
- Length (in bytes)
- LastModified date

You can read from the file using the IFileInfo.CreateReadStream method.

The `FileProviderSample` sample app demonstrates how to configure a File Provider in `Startup.ConfigureServices` for use throughout the app via dependency injection.

File Provider implementations

The following table lists implementations of `IFileProvider`.

Expand table

Implementation	Description
Composite File Provider	Used to provide combined access to files and directories from one or more other providers.
Manifest Embedded File Provider	Used to access files embedded in assemblies.
Physical File Provider	Used to access the system's physical files.

Physical File Provider

The PhysicalFileProvider provides access to the physical file system. `PhysicalFileProvider` uses the System.IO.File type (for the physical provider) and scopes all paths to a directory and its children. This scoping prevents access to the file system outside of the specified directory and its children. The most common scenario for creating and using a `PhysicalFileProvider` is to request an `IFileProvider` in a constructor through dependency injection.

When instantiating this provider directly, an absolute directory path is required and serves as the base path for all requests made using the provider. Glob patterns aren't supported in the directory path.

The following code shows how to use `PhysicalFileProvider` to obtain directory contents and file information:

C#

```
var provider = new PhysicalFileProvider(applicationRoot);
var contents = provider.GetDirectoryContents(string.Empty);
var filePath = Path.Combine("wwwroot", "js", "site.js");
var fileInfo = provider.GetFileInfo(filePath);
```

Types in the preceding example:

- `provider` is an `IFileProvider`.
- `contents` is an `IDirectoryContents`.
- `fileInfo` is an `IFileInfo`.

The File Provider can be used to iterate through the directory specified by `applicationRoot` or call `GetFileInfo` to obtain a file's information. Glob patterns can't

be passed to the `GetFileInfo` method. The File Provider has no access outside of the `applicationRoot` directory.

The `FileProviderSample` sample app creates the provider in the `Startup.ConfigureServices` method using IHostEnvironment.ContentRootFileProvider:

C#

```
var physicalProvider = _env.ContentRootFileProvider;
```

Manifest Embedded File Provider

The ManifestEmbeddedFileProvider is used to access files embedded within assemblies. The `ManifestEmbeddedFileProvider` uses a manifest compiled into the assembly to reconstruct the original paths of the embedded files.

To generate a manifest of the embedded files:

1. Add the Microsoft.Extensions.FileProviders.Embedded NuGet package to your project.
2. Set the `<GenerateEmbeddedFilesManifest>` property to `true`. Specify the files to embed with <EmbeddedResource>:

   XML

   ```
   <Project Sdk="Microsoft.NET.Sdk.Web">

     <PropertyGroup>
       <TargetFramework>netcoreapp3.1</TargetFramework>
       <GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>
     </PropertyGroup>

     <ItemGroup>
       <PackageReference
   Include="Microsoft.Extensions.FileProviders.Embedded" Version="3.1.0"
   />
     </ItemGroup>

     <ItemGroup>
       <EmbeddedResource Include="Resource.txt" />
     </ItemGroup>

   </Project>
   ```

Use glob patterns to specify one or more files to embed into the assembly.

The `FileProviderSample` sample app creates an `ManifestEmbeddedFileProvider` and passes the currently executing assembly to its constructor.

`Startup.cs`:

```
C#

var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
```

Additional overloads allow you to:

- Specify a relative file path.
- Scope files to a last modified date.
- Name the embedded resource containing the embedded file manifest.

Expand table

Overload	Description
`ManifestEmbeddedFileProvider(Assembly, String)`	Accepts an optional `root` relative path parameter. Specify the `root` to scope calls to GetDirectoryContents to those resources under the provided path.
`ManifestEmbeddedFileProvider(Assembly, String, DateTimeOffset)`	Accepts an optional `root` relative path parameter and a `lastModified` date (DateTimeOffset) parameter. The `lastModified` date scopes the last modification date for the IFileInfo instances returned by the IFileProvider.
`ManifestEmbeddedFileProvider(Assembly, String, String, DateTimeOffset)`	Accepts an optional `root` relative path, `lastModified` date, and `manifestName` parameters. The `manifestName` represents the name of the embedded resource containing the manifest.

Composite File Provider

The CompositeFileProvider combines `IFileProvider` instances, exposing a single interface for working with files from multiple providers. When creating the `CompositeFileProvider`, pass one or more `IFileProvider` instances to its constructor.

In the `FileProviderSample` sample app, a `PhysicalFileProvider` and a `ManifestEmbeddedFileProvider` provide files to a `CompositeFileProvider` registered in the

app's service container. The following code is found in the project's `Startup.ConfigureServices` method:

```
C#

var physicalProvider = _env.ContentRootFileProvider;
var manifestEmbeddedProvider =
    new ManifestEmbeddedFileProvider(typeof(Program).Assembly);
var compositeProvider =
    new CompositeFileProvider(physicalProvider, manifestEmbeddedProvider);

services.AddSingleton<IFileProvider>(compositeProvider);
```

Watch for changes

The IFileProvider.Watch method provides a scenario to watch one or more files or directories for changes. The `Watch` method:

- Accepts a file path string, which can use glob patterns to specify multiple files.
- Returns an IChangeToken.

The resulting change token exposes:

- HasChanged: A property that can be inspected to determine if a change has occurred.
- RegisterChangeCallback: Called when changes are detected to the specified path string. Each change token only calls its associated callback in response to a single change. To enable constant monitoring, use a TaskCompletionSource<TResult> (shown below) or recreate `IChangeToken` instances in response to changes.

The `WatchConsole` sample app writes a message whenever a `.txt` file in the `TextFiles` directory is modified:

```
C#

private static readonly string _fileFilter = Path.Combine("TextFiles",
"*.txt");

public static void Main(string[] args)
{
    Console.WriteLine($"Monitoring for changes with filter '{_fileFilter}'
(Ctrl + C to quit)...");

    while (true)
    {
        MainAsync().GetAwaiter().GetResult();
```

```
    }
}

private static async Task MainAsync()
{
    var fileProvider = new
PhysicalFileProvider(Directory.GetCurrentDirectory());
    IChangeToken token = fileProvider.Watch(_fileFilter);
    var tcs = new TaskCompletionSource<object>();

    token.RegisterChangeCallback(state =>
        ((TaskCompletionSource<object>)state).TrySetResult(null), tcs);

    await tcs.Task.ConfigureAwait(false);

    Console.WriteLine("file changed");
}
```

Some file systems, such as Docker containers and network shares, may not reliably send change notifications. Set the `DOTNET_USE_POLLING_FILE_WATCHER` environment variable to `1` or `true` to poll the file system for changes every four seconds (not configurable).

Glob patterns

File system paths use wildcard patterns called *glob (or globbing) patterns*. Specify groups of files with these patterns. The two wildcard characters are `*` and `**`:

`*`

Matches anything at the current folder level, any filename, or any file extension. Matches are terminated by `/` and `.` characters in the file path.

`**`

Matches anything across multiple directory levels. Can be used to recursively match many files within a directory hierarchy.

The following table provides common examples of glob patterns.

Expand table

Pattern	Description
`directory/file.txt`	Matches a specific file in a specific directory.
`directory/*.txt`	Matches all files with `.txt` extension in a specific directory.
`directory/*/appsettings.json`	Matches all `appsettings.json` files in directories exactly one level below the `directory` folder.

Pattern	Description
`directory/**/*.txt`	Matches all files with a `.txt` extension found anywhere under the `directory` folder.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Request Features in ASP.NET Core

Article • 01/09/2023

By Steve Smith

The `HttpContext` API that applications and middleware use to process requests has an abstraction layer underneath it called *feature interfaces*. Each feature interface provides a granular subset of the functionality exposed by `HttpContext`. These interfaces can be added, modified, wrapped, replaced, or even removed by the server or middleware as the request is processed without having to re-implement the entire `HttpContext`. They can also be used to mock functionality when testing.

Feature collections

The Features property of `HttpContext` provides access to the collection of feature interfaces for the current request. Since the feature collection is mutable even within the context of a request, middleware can be used to modify the collection and add support for additional features. Some advanced features are only available by accessing the associated interface through the feature collection.

Feature interfaces

ASP.NET Core defines a number of common HTTP feature interfaces in Microsoft.AspNetCore.Http.Features, which are shared by various servers and middleware to identify the features that they support. Servers and middleware may also provide their own interfaces with additional functionality.

Most feature interfaces provide optional, light-up functionality, and their associated `HttpContext` APIs provide defaults if the feature isn't present. A few interfaces are indicated in the following content as required because they provide core request and response functionality and must be implemented in order to process the request.

The following feature interfaces are from Microsoft.AspNetCore.Http.Features:

IHttpRequestFeature: Defines the structure of an HTTP request, including the protocol, path, query string, headers, and body. This feature is required in order to process requests.

IHttpResponseFeature: Defines the structure of an HTTP response, including the status code, headers, and body of the response. This feature is required in order to process

requests.

IHttpResponseBodyFeature: Defines different ways of writing out the response body, using either a `Stream`, a `PipeWriter`, or a file. This feature is required in order to process requests. This replaces `IHttpResponseFeature.Body` and `IHttpSendFileFeature`.

IHttpAuthenticationFeature: Holds the ClaimsPrincipal currently associated with the request.

IFormFeature: Used to parse and cache incoming HTTP and multipart form submissions.

IHttpBodyControlFeature: Used to control if synchronous IO operations are allowed for the request or response bodies.

`IHttpActivityFeature`: Used to add `Activity` information for diagnostic listeners.

IHttpConnectionFeature: Defines properties for the connection id and local and remote addresses and ports.

IHttpMaxRequestBodySizeFeature: Controls the maximum allowed request body size for the current request.

`IHttpRequestBodyDetectionFeature`: Indicates if the request can have a body.

IHttpRequestIdentifierFeature: Adds a property that can be implemented to uniquely identify requests.

IHttpRequestLifetimeFeature: Defines support for aborting connections or detecting if a request has been terminated prematurely, such as by a client disconnect.

IHttpRequestTrailersFeature: Provides access to the request trailer headers, if any.

IHttpResetFeature: Used to send reset messages for protocols that support them such as HTTP/2 or HTTP/3.

IHttpResponseTrailersFeature: Enables the application to provide response trailer headers if supported.

IHttpUpgradeFeature: Defines support for HTTP Upgrades, which allow the client to specify which additional protocols it would like to use if the server wishes to switch protocols.

IHttpWebSocketFeature: Defines an API for supporting WebSockets.

IHttpsCompressionFeature: Controls if response compression should be used over HTTPS connections.

IItemsFeature: Stores the Items collection for per request application state.

IQueryFeature: Parses and caches the query string.

IRequestBodyPipeFeature: Represents the request body as a PipeReader.

IRequestCookiesFeature: Parses and caches the request `Cookie` header values.

IResponseCookiesFeature: Controls how response cookies are applied to the `Set-Cookie` header.

IServerVariablesFeature: This feature provides access to request server variables such as those provided by IIS.

IServiceProvidersFeature: Provides access to an IServiceProvider with scoped request services.

ISessionFeature: Defines `ISessionFactory` and ISession abstractions for supporting user sessions. `ISessionFeature` is implemented by the SessionMiddleware (see Session in ASP.NET Core).

ITlsConnectionFeature: Defines an API for retrieving client certificates.

ITlsTokenBindingFeature: Defines methods for working with TLS token binding parameters.

ITrackingConsentFeature: Used to query, grant, and withdraw user consent regarding the storage of user information related to site activity and functionality.

Additional resources

- Web server implementations in ASP.NET Core
- ASP.NET Core Middleware

Access `HttpContext` in ASP.NET Core

Article • 07/26/2024

HttpContext encapsulates all information about an individual HTTP request and response. An `HttpContext` instance is initialized when an HTTP request is received. The `HttpContext` instance is accessible by middleware and app frameworks such as Web API controllers, Razor Pages, SignalR, gRPC, and more.

For information about using `HttpContext` with a HTTP request and response, see Use HttpContext in ASP.NET Core.

Access `HttpContext` from Razor Pages

The Razor Pages PageModel exposes the PageModel.HttpContext property:

C#

```
public class IndexModel : PageModel
{
    public void OnGet()
    {
        var message = HttpContext.Request.PathBase;

        // ...
    }
}
```

The same property can be used in the corresponding Razor Page View:

CSHTML

```
@page
@model IndexModel

@{
    var message = HttpContext.Request.PathBase;

    // ...
}
```

Access `HttpContext` from a Razor view in MVC

Razor views in the MVC pattern expose the `HttpContext` via the RazorPage.Context property on the view. The following example retrieves the current username in an intranet app using Windows Authentication:

CSHTML

```
@{
    var username = Context.User.Identity.Name;

    // ...
}
```

Access `HttpContext` from a controller

Controllers expose the ControllerBase.HttpContext property:

C#

```
public class HomeController : Controller
{
    public IActionResult About()
    {
        var pathBase = HttpContext.Request.PathBase;

        // ...

        return View();
    }
}
```

Access `HttpContext` from minimal APIs

To use `HttpContext` from minimal APIs, add a `HttpContext` parameter:

C#

```
app.MapGet("/", (HttpContext context) => context.Response.WriteAsync("Hello
World"));
```

Access `HttpContext` from middleware

To use `HttpContext` from custom middleware components, use the `HttpContext` parameter passed into the `Invoke` or `InvokeAsync` method:

C#

```
public class MyCustomMiddleware
{
    // ...

    public async Task InvokeAsync(HttpContext context)
    {
        // ...
    }
}
```

Access `HttpContext` from SignalR

To use `HttpContext` from SignalR, call the GetHttpContext method on Hub.Context:

C#

```
public class MyHub : Hub
{
    public async Task SendMessage()
    {
        var httpContext = Context.GetHttpContext();

        // ...
    }
}
```

Access `HttpContext` from gRPC methods

To use `HttpContext` from gRPC methods, see Resolve HttpContext in gRPC methods.

Access `HttpContext` from custom components

For other framework and custom components that require access to `HttpContext`, the recommended approach is to register a dependency using the built-in Dependency Injection (DI) container. The DI container supplies the `IHttpContextAccessor` to any classes that declare it as a dependency in their constructors:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
```

```
builder.Services.AddHttpContextAccessor();
builder.Services.AddTransient<IUserRepository, UserRepository>();
```

In the following example:

- `UserRepository` declares its dependency on `IHttpContextAccessor`.
- The dependency is supplied when DI resolves the dependency chain and creates an instance of `UserRepository`.

C#

```
public class UserRepository : IUserRepository
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public UserRepository(IHttpContextAccessor httpContextAccessor) =>
        _httpContextAccessor = httpContextAccessor;

    public void LogCurrentUser()
    {
        var username = _httpContextAccessor.HttpContext.User.Identity.Name;

        // ...
    }
}
```

`HttpContext` access from a background thread

`HttpContext` isn't thread-safe. Reading or writing properties of the `HttpContext` outside of processing a request can result in a NullReferenceException.

> ⓘ Note
>
> If your app generates sporadic `NullReferenceException` errors, review parts of the code that start background processing or that continue processing after a request completes. Look for mistakes, such as defining a controller method as `async void`.

To safely do background work with `HttpContext` data:

- Copy the required data during request processing.
- Pass the copied data to a background task.
- Do ***not*** reference `HttpContext` data in parallel tasks. Extract the data needed from the context before starting the parallel tasks.

To avoid unsafe code, never pass `HttpContext` into a method that does background work. Pass the required data instead. In the following example, `SendEmail` calls `SendEmailCoreAsync` to start sending an email. The value of the `X-Correlation-Id` header is passed to `SendEmailCoreAsync` instead of the `HttpContext`. Code execution doesn't wait for `SendEmailCoreAsync` to complete:

C#

```
public class EmailController : Controller
{
    public IActionResult SendEmail(string email)
    {
        var correlationId = HttpContext.Request.Headers["X-Correlation-
Id"].ToString();

        _ = SendEmailCoreAsync(correlationId);

        return View();
    }

    private async Task SendEmailCoreAsync(string correlationId)
    {
        // ...
    }
}
```

`IHttpContextAccessor` / `HttpContext` in Razor components (Blazor)

IHttpContextAccessor must be avoided with interactive rendering because there isn't a valid `HttpContext` available.

IHttpContextAccessor can be used for components that are statically rendered on the server. **However, we recommend avoiding it if possible.**

HttpContext can be used as a cascading parameter only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

C#

```
[CascadingParameter]
public HttpContext? HttpContext { get; set; }
```

For scenarios where the HttpContext is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see Server-side ASP.NET Core Blazor additional security scenarios.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Detect changes with change tokens in ASP.NET Core

Article • 08/18/2023

A *change token* is a general-purpose, low-level building block used to track state changes.

View or download sample code (how to download)

IChangeToken interface

IChangeToken propagates notifications that a change has occurred. `IChangeToken` resides in the Microsoft.Extensions.Primitives namespace. The Microsoft.Extensions.Primitives NuGet package is implicitly provided to the ASP.NET Core apps.

`IChangeToken` has two properties:

- ActiveChangeCallbacks indicate if the token proactively raises callbacks. If `ActiveChangedCallbacks` is set to `false`, a callback is never called, and the app must poll `HasChanged` for changes. It's also possible for a token to never be cancelled if no changes occur or the underlying change listener is disposed or disabled.
- HasChanged receives a value that indicates if a change has occurred.

The `IChangeToken` interface includes the RegisterChangeCallback(Action<Object>, Object) method, which registers a callback that's invoked when the token has changed. `HasChanged` must be set before the callback is invoked.

ChangeToken class

ChangeToken is a static class used to propagate notifications that a change has occurred. `ChangeToken` resides in the Microsoft.Extensions.Primitives namespace. The Microsoft.Extensions.Primitives NuGet package is implicitly provided to the ASP.NET Core apps.

The ChangeToken.OnChange(Func<IChangeToken>, Action) method registers an `Action` to call whenever the token changes:

- `Func<IChangeToken>` produces the token.

- `Action` is called when the token changes.

The ChangeToken.OnChange<TState>(Func<IChangeToken>, Action<TState>, TState) overload takes an additional `TState` parameter that's passed into the token consumer `Action`.

`OnChange` returns an IDisposable. Calling Dispose stops the token from listening for further changes and releases the token's resources.

Example uses of change tokens in ASP.NET Core

Change tokens are used in prominent areas of ASP.NET Core to monitor for changes to objects:

- For monitoring changes to files, IFileProvider's Watch method creates an `IChangeToken` for the specified files or folder to watch.
- `IChangeToken` tokens can be added to cache entries to trigger cache evictions on change.
- For `TOptions` changes, the default OptionsMonitor<TOptions> implementation of IOptionsMonitor<TOptions> has an overload that accepts one or more IOptionsChangeTokenSource<TOptions> instances. Each instance returns an `IChangeToken` to register a change notification callback for tracking options changes.

Monitor for configuration changes

By default, ASP.NET Core templates use JSON configuration files (`appsettings.json`, `appsettings.Development.json`, and `appsettings.Production.json`) to load app configuration settings.

These files are configured using the AddJsonFile(IConfigurationBuilder, String, Boolean, Boolean) extension method on ConfigurationBuilder that accepts a `reloadOnChange` parameter. `reloadOnChange` indicates if configuration should be reloaded on file changes. This setting appears in the Host convenience method CreateDefaultBuilder:

C#

```
config.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
      .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true,
          reloadOnChange: true);
```

File-based configuration is represented by FileConfigurationSource. `FileConfigurationSource` uses IFileProvider to monitor files.

By default, the `IFileMonitor` is provided by a PhysicalFileProvider, which uses FileSystemWatcher to monitor for configuration file changes.

The sample app demonstrates two implementations for monitoring configuration changes. If any of the `appsettings` files change, both of the file monitoring implementations execute custom code—the sample app writes a message to the console.

A configuration file's `FileSystemWatcher` can trigger multiple token callbacks for a single configuration file change. To ensure that the custom code is only run once when multiple token callbacks are triggered, the sample's implementation checks file hashes. The sample uses SHA1 file hashing. A retry is implemented with an exponential back-off.

`Utilities/Utilities.cs`:

C#

```
public static byte[] ComputeHash(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fs = File.OpenRead(filePath))
                {
                    return System.Security.Cryptography.SHA1
                        .Create().ComputeHash(fs);
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3)
            {
                throw;
            }

            Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
            runCount++;
```

```
        }
    }

    return new byte[20];
}
```

Simple startup change token

Register a token consumer `Action` callback for change notifications to the configuration reload token.

In `Startup.Configure`:

C#

```
ChangeToken.OnChange(
    () => config.GetReloadToken(),
    (state) => InvokeChanged(state),
    env);
```

`config.GetReloadToken()` provides the token. The callback is the `InvokeChanged` method:

C#

```
private void InvokeChanged(IWebHostEnvironment env)
{
    byte[] appsettingsHash = ComputeHash("appSettings.json");
    byte[] appsettingsEnvHash =
        ComputeHash($"appSettings.{env.EnvironmentName}.json");

    if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
        !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
    {
        _appsettingsHash = appsettingsHash;
        _appsettingsEnvHash = appsettingsEnvHash;

        WriteConsole("Configuration changed (Simple Startup Change Token)");
    }
}
```

The `state` of the callback is used to pass in the `IWebHostEnvironment`, which is useful for specifying the correct `appsettings` configuration file to monitor (for example, `appsettings.Development.json` when in the Development environment). File hashes are used to prevent the `WriteConsole` statement from running multiple times due to multiple token callbacks when the configuration file has only changed once.

This system runs as long as the app is running and can't be disabled by the user.

Monitor configuration changes as a service

The sample implements:

- Basic startup token monitoring.
- Monitoring as a service.
- A mechanism to enable and disable monitoring.

The sample establishes an `IConfigurationMonitor` interface.

`Extensions/ConfigurationMonitor.cs`:

```
C#

public interface IConfigurationMonitor
{
    bool MonitoringEnabled { get; set; }
    string CurrentState { get; set; }
}
```

The constructor of the implemented class, `ConfigurationMonitor`, registers a callback for change notifications:

```
C#

public ConfigurationMonitor(IConfiguration config, IWebHostEnvironment env)
{
    _env = env;

    ChangeToken.OnChange<IConfigurationMonitor>(
        () => config.GetReloadToken(),
        InvokeChanged,
        this);
}

public bool MonitoringEnabled { get; set; } = false;
public string CurrentState { get; set; } = "Not monitoring";
```

`config.GetReloadToken()` supplies the token. `InvokeChanged` is the callback method. The `state` in this instance is a reference to the `IConfigurationMonitor` instance that's used to access the monitoring state. Two properties are used:

- `MonitoringEnabled`: Indicates if the callback should run its custom code.
- `CurrentState`: Describes the current monitoring state for use in the UI.

The `InvokeChanged` method is similar to the earlier approach, except that it:

- Doesn't run its code unless `MonitoringEnabled` is `true`.
- Outputs the current `state` in its `WriteConsole` output.

C#

```
private void InvokeChanged(IConfigurationMonitor state)
{
    if (MonitoringEnabled)
    {
        byte[] appsettingsHash = ComputeHash("appSettings.json");
        byte[] appsettingsEnvHash =
            ComputeHash($"appSettings.{_env.EnvironmentName}.json");

        if (!_appsettingsHash.SequenceEqual(appsettingsHash) ||
            !_appsettingsEnvHash.SequenceEqual(appsettingsEnvHash))
        {
            string message = $"State updated at {DateTime.Now}";


            _appsettingsHash = appsettingsHash;
            _appsettingsEnvHash = appsettingsEnvHash;

            WriteConsole("Configuration changed (ConfigurationMonitor Class)
" +
                $"{message}, state:{state.CurrentState}");
        }
    }
}
```

An instance `ConfigurationMonitor` is registered as a service in `Startup.ConfigureServices`:

C#

```
services.AddSingleton<IConfigurationMonitor, ConfigurationMonitor>();
```

The Index page offers the user control over configuration monitoring. The instance of `IConfigurationMonitor` is injected into the `IndexModel`.

`Pages/Index.cshtml.cs`:

C#

```
public IndexModel(
    IConfiguration config,
    IConfigurationMonitor monitor,
    FileService fileService)
```

```
{
    _config = config;
    _monitor = monitor;
    _fileService = fileService;
}
```

The configuration monitor (`_monitor`) is used to enable or disable monitoring and set the current state for UI feedback:

C#

```
public IActionResult OnPostStartMonitoring()
{
    _monitor.MonitoringEnabled = true;
    _monitor.CurrentState = "Monitoring!";

    return RedirectToPage();
}

public IActionResult OnPostStopMonitoring()
{
    _monitor.MonitoringEnabled = false;
    _monitor.CurrentState = "Not monitoring";

    return RedirectToPage();
}
```

When `OnPostStartMonitoring` is triggered, monitoring is enabled, and the current state is cleared. When `OnPostStopMonitoring` is triggered, monitoring is disabled, and the state is set to reflect that monitoring isn't occurring.

Buttons in the UI enable and disable monitoring.

`Pages/Index.cshtml`:

CSHTML

```
<button class="btn btn-success" asp-page-handler="StartMonitoring">
    Start Monitoring
</button>

<button class="btn btn-danger" asp-page-handler="StopMonitoring">
    Stop Monitoring
</button>
```

Monitor cached file changes

File content can be cached in-memory using IMemoryCache. In-memory caching is described in the Cache in-memory topic. Without taking additional steps, such as the implementation described below, *stale* (outdated) data is returned from a cache if the source data changes.

For example, not taking into account the status of a cached source file when renewing a sliding expiration period leads to stale cached file data. Each request for the data renews the sliding expiration period, but the file is never reloaded into the cache. Any app features that use the file's cached content are subject to possibly receiving stale content.

Using change tokens in a file caching scenario prevents the presence of stale file content in the cache. The sample app demonstrates an implementation of the approach.

The sample uses `GetFileContent` to:

- Return file content.
- Implement a retry algorithm with exponential back-off to cover cases where a file access problem temporarily delays reading the file's content.

`Utilities/Utilities.cs`:

C#

```
public async static Task<string> GetFileContent(string filePath)
{
    var runCount = 1;

    while(runCount < 4)
    {
        try
        {
            if (File.Exists(filePath))
            {
                using (var fileStreamReader = File.OpenText(filePath))
                {
                    return await fileStreamReader.ReadToEndAsync();
                }
            }
            else
            {
                throw new FileNotFoundException();
            }
        }
        catch (IOException ex)
        {
            if (runCount == 3)
            {
                throw;
            }
```

```
            Thread.Sleep(TimeSpan.FromSeconds(Math.Pow(2, runCount)));
            runCount++;
        }
    }

    return null;
}
```

A `FileService` is created to handle cached file lookups. The `GetFileContent` method call of the service attempts to obtain file content from the in-memory cache and return it to the caller (`Services/FileService.cs`).

If cached content isn't found using the cache key, the following actions are taken:

1. The file content is obtained using `GetFileContent`.
2. A change token is obtained from the file provider with IFileProviders.Watch. The token's callback is triggered when the file is modified.
3. The file content is cached with a sliding expiration period. The change token is attached with MemoryCacheEntryExtensions.AddExpirationToken to evict the cache entry if the file changes while it's cached.

In the following example, files are stored in the app's content root. `IWebHostEnvironment.ContentRootFileProvider` is used to obtain an IFileProvider pointing at the app's `IWebHostEnvironment.ContentRootPath`. The `filePath` is obtained with IFileInfo.PhysicalPath.

C#

```
public class FileService
{
    private readonly IMemoryCache _cache;
    private readonly IFileProvider _fileProvider;
    private List<string> _tokens = new List<string>();

    public FileService(IMemoryCache cache, IWebHostEnvironment env)
    {
        _cache = cache;
        _fileProvider = env.ContentRootFileProvider;
    }

    public async Task<string> GetFileContents(string fileName)
    {
        var filePath = _fileProvider.GetFileInfo(fileName).PhysicalPath;
        string fileContent;

        // Try to obtain the file contents from the cache.
        if (_cache.TryGetValue(filePath, out fileContent))
        {
            return fileContent;
```

```
        }

        // The cache doesn't have the entry, so obtain the file
        // contents from the file itself.
        fileContent = await GetFileContent(filePath);

        if (fileContent != null)
        {
            // Obtain a change token from the file provider whose
            // callback is triggered when the file is modified.
            var changeToken = _fileProvider.Watch(fileName);

            // Configure the cache entry options for a five minute
            // sliding expiration and use the change token to
            // expire the file in the cache if the file is
            // modified.
            var cacheEntryOptions = new MemoryCacheEntryOptions()
                .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                .AddExpirationToken(changeToken);

            // Put the file contents into the cache.
            _cache.Set(filePath, fileContent, cacheEntryOptions);

            return fileContent;
        }

        return string.Empty;
    }
}
```

The `FileService` is registered in the service container along with the memory caching service.

In `Startup.ConfigureServices`:

C#

```
services.AddMemoryCache();
services.AddSingleton<FileService>();
```

The page model loads the file's content using the service.

In the Index page's `OnGet` method (`Pages/Index.cshtml.cs`):

C#

```
var fileContent = await _fileService.GetFileContents("poem.txt");
```

CompositeChangeToken class

For representing one or more `IChangeToken` instances in a single object, use the CompositeChangeToken class.

C#

```
var firstCancellationTokenSource = new CancellationTokenSource();
var secondCancellationTokenSource = new CancellationTokenSource();

var firstCancellationToken = firstCancellationTokenSource.Token;
var secondCancellationToken = secondCancellationTokenSource.Token;

var firstCancellationChangeToken = new
CancellationChangeToken(firstCancellationToken);
var secondCancellationChangeToken = new
CancellationChangeToken(secondCancellationToken);

var compositeChangeToken =
    new CompositeChangeToken(
        new List<IChangeToken>
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken
        });
```

`HasChanged` on the composite token reports `true` if any represented token `HasChanged` is `true`. `ActiveChangeCallbacks` on the composite token reports `true` if any represented token `ActiveChangeCallbacks` is `true`. If multiple concurrent change events occur, the composite change callback is invoked one time.

Additional resources

- Cache in-memory in ASP.NET Core
- Distributed caching in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

Open Web Interface for .NET (OWIN) with ASP.NET Core

Article • 12/02/2024

By Steve Smith and Rick Anderson

ASP.NET Core:

- Supports the Open Web Interface for .NET (OWIN).
- Has .NET Core compatible replacements for the `Microsoft.Owin.*` (Katana) libraries.

OWIN allows web apps to be decoupled from web servers. It defines a standard way for middleware to be used in a pipeline to handle requests and associated responses. ASP.NET Core applications and middleware can interoperate with OWIN-based applications, servers, and middleware.

OWIN provides a decoupling layer that allows two frameworks with disparate object models to be used together. The `Microsoft.AspNetCore.Owin` package provides two adapter implementations:

- ASP.NET Core to OWIN
- OWIN to ASP.NET Core

This allows ASP.NET Core to be hosted on top of an OWIN compatible server/host or for other OWIN compatible components to be run on top of ASP.NET Core.

> **Note**
>
> Using these adapters comes with a performance cost. Apps using only ASP.NET Core components shouldn't use the `Microsoft.AspNetCore.Owin` package or adapters.

View or download sample code (how to download)

Running OWIN middleware in the ASP.NET Core pipeline

ASP.NET Core's OWIN support is deployed as part of the `Microsoft.AspNetCore.Owin` package. You can import OWIN support into your project by installing this package.

OWIN middleware conforms to the OWIN specification, which requires a `Func<IDictionary<string, object>, Task>` interface, and specific keys be set (such as `owin.ResponseBody`). The following simple OWIN middleware displays "Hello World":

C#

```
public Task OwinHello(IDictionary<string, object> environment)
{
    string responseText = "Hello World via OWIN";
    byte[] responseBytes = Encoding.UTF8.GetBytes(responseText);

    // OWIN Environment Keys: https://owin.org/spec/spec/owin-1.0.0.html
    var responseStream = (Stream)environment["owin.ResponseBody"];
    var responseHeaders = (IDictionary<string,
string[]>)environment["owin.ResponseHeaders"];

    responseHeaders["Content-Length"] = new string[] {
responseBytes.Length.ToString(CultureInfo.InvariantCulture) };
    responseHeaders["Content-Type"] = new string[] { "text/plain" };

    return responseStream.WriteAsync(responseBytes, 0,
responseBytes.Length);
}
```

The sample signature returns a `Task` and accepts an `IDictionary<string, object>` as required by OWIN.

The following code shows how to add the `OwinHello` middleware (shown above) to the ASP.NET Core pipeline with the `UseOwin` extension method.

C#

```
public void Configure(IApplicationBuilder app)
{
    app.UseOwin(pipeline =>
    {
        pipeline(next => OwinHello);
    });
}
```

You can configure other actions to take place within the OWIN pipeline.

> **Note**
>
> Response headers should only be modified prior to the first write to the response stream.

> **Note**
>
> Multiple calls to `UseOwin` is discouraged for performance reasons. OWIN components will operate best if grouped together.

C#

```
app.UseOwin(pipeline =>
{
    pipeline(next =>
    {
        return async environment =>
        {
            // Do something before.
            await next(environment);
            // Do something after.
        };
    });
});
```

OWIN environment

You can construct an OWIN environment using the `HttpContext`.

C#

```
var environment = new OwinEnvironment(HttpContext);
var features = new OwinFeatureCollection(environment);
```

OWIN keys

OWIN depends on an `IDictionary<string,object>` object to communicate information throughout an HTTP Request/Response exchange. ASP.NET Core implements the keys listed below. See the primary specification, extensions, and OWIN Key Guidelines and Common Keys.

Request data (OWIN v1.0.0)

Expand table

Key	Value (type)	Description
owin.RequestScheme	`String`	
owin.RequestMethod	`String`	
owin.RequestPathBase	`String`	
owin.RequestPath	`String`	
owin.RequestQueryString	`String`	
owin.RequestProtocol	`String`	
owin.RequestHeaders	`IDictionary<string,string[]>`	
owin.RequestBody	`Stream`	

Request data (OWIN v1.1.0)

Expand table

Key	Value (type)	Description
owin.RequestId	`String`	Optional

Response data (OWIN v1.0.0)

Expand table

Key	Value (type)	Description
owin.ResponseStatusCode	`int`	Optional
owin.ResponseReasonPhrase	`String`	Optional
owin.ResponseHeaders	`IDictionary<string,string[]>`	
owin.ResponseBody	`Stream`	

Other data (OWIN v1.0.0)

Expand table

Key	Value (type)	Description
owin.CallCancelled	`CancellationToken`	
owin.Version	`String`	

Common keys

Expand table

Key	Value (type)	Description
ssl.ClientCertificate	`X509Certificate`	
ssl.LoadClientCertAsync	`Func<Task>`	
server.RemoteIpAddress	`String`	
server.RemotePort	`String`	
server.LocalIpAddress	`String`	
server.LocalPort	`String`	
server.OnSendingHeaders	`Action<Action<object>,object>`	

SendFiles v0.3.0

Expand table

Key	Value (type)	Description
sendfile.SendAsync	See delegate signature	Per Request

Opaque v0.3.0

Expand table

Key	Value (type)	Description
opaque.Version	`String`	
opaque.Upgrade	`OpaqueUpgrade`	See delegate signature
opaque.Stream	`Stream`	

Key	Value (type)	Description
opaque.CallCancelled	`CancellationToken`	

WebSocket v0.3.0

Expand table

Key	Value (type)	Description
websocket.Version	`String`	
websocket.Accept	`WebSocketAccept`	See delegate signature
websocket.AcceptAlt		Non-spec
websocket.SubProtocol	`String`	See RFC6455 Section 4.2.2 Step 5.5
websocket.SendAsync	`WebSocketSendAsync`	See delegate signature
websocket.ReceiveAsync	`WebSocketReceiveAsync`	See delegate signature
websocket.CloseAsync	`WebSocketCloseAsync`	See delegate signature
websocket.CallCancelled	`CancellationToken`	
websocket.ClientCloseStatus	`int`	Optional
websocket.ClientCloseDescription	`String`	Optional

Additional resources

- See the source on GitHub for OWIN keys supported in the translation layer.
- Middleware
- Servers

Background tasks with hosted services in ASP.NET Core

Article • 05/10/2024

By Jeow Li Huan

In ASP.NET Core, background tasks can be implemented as *hosted services*. A hosted service is a class with background task logic that implements the IHostedService interface. This article provides three hosted service examples:

- Background task that runs on a timer.
- Hosted service that activates a scoped service. The scoped service can use dependency injection (DI).
- Queued background tasks that run sequentially.

Worker Service template

The ASP.NET Core Worker Service template provides a starting point for writing long running service apps. An app created from the Worker Service template specifies the Worker SDK in its project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

To use the template as a basis for a hosted services app:

Visual Studio

1. Create a new project.
2. Select **Worker Service**. Select **Next**.
3. Provide a project name in the **Project name** field or accept the default project name. Select **Next**.
4. In the **Additional information** dialog, Choose a **Framework**. Select **Create**.

Package

An app based on the Worker Service template uses the `Microsoft.NET.Sdk.Worker` SDK and has an explicit package reference to the Microsoft.Extensions.Hosting package. For example, see the sample app's project file (`BackgroundTasksSample.csproj`).

For web apps that use the `Microsoft.NET.Sdk.Web` SDK, the Microsoft.Extensions.Hosting package is referenced implicitly from the shared framework. An explicit package reference in the app's project file isn't required.

IHostedService interface

The IHostedService interface defines two methods for objects that are managed by the host:

- StartAsync(CancellationToken)
- StopAsync(CancellationToken)

`StartAsync`

StartAsync(CancellationToken) contains the logic to start the background task. `StartAsync` is called *before*:

- The app's request processing pipeline is configured.
- The server is started and IApplicationLifetime.ApplicationStarted is triggered.

`StartAsync` should be limited to short running tasks because hosted services are run sequentially, and no further services are started until `StartAsync` runs to completion.

`StopAsync`

- StopAsync(CancellationToken) is triggered when the host is performing a graceful shutdown. `StopAsync` contains the logic to end the background task. Implement IDisposable and finalizers (destructors) to dispose of any unmanaged resources.

The cancellation token has a default 30 second timeout to indicate that the shutdown process should no longer be graceful. When cancellation is requested on the token:

- Any remaining background operations that the app is performing should be aborted.
- Any methods called in `StopAsync` should return promptly.

However, tasks aren't abandoned after cancellation is requested—the caller awaits all tasks to complete.

If the app shuts down unexpectedly (for example, the app's process fails), `StopAsync` might not be called. Therefore, any methods called or operations conducted in `StopAsync` might not occur.

To extend the default 30 second shutdown timeout, set:

- ShutdownTimeout when using Generic Host. For more information, see .NET Generic Host in ASP.NET Core.
- Shutdown timeout host configuration setting when using Web Host. For more information, see ASP.NET Core Web Host.

The hosted service is activated once at app startup and gracefully shut down at app shutdown. If an error is thrown during background task execution, `Dispose` should be called even if `StopAsync` isn't called.

BackgroundService base class

BackgroundService is a base class for implementing a long running IHostedService.

ExecuteAsync(CancellationToken) is called to run the background service. The implementation returns a Task that represents the entire lifetime of the background service. No further services are started until ExecuteAsync becomes asynchronous, such as by calling `await`. Avoid performing long, blocking initialization work in `ExecuteAsync`. The host blocks in StopAsync(CancellationToken) waiting for `ExecuteAsync` to complete.

The cancellation token is triggered when IHostedService.StopAsync is called. Your implementation of `ExecuteAsync` should finish promptly when the cancellation token is fired in order to gracefully shut down the service. Otherwise, the service ungracefully shuts down at the shutdown timeout. For more information, see the IHostedService interface section.

For more information, see the BackgroundService source code.

Timed background tasks

A timed background task makes use of the System.Threading.Timer class. The timer triggers the task's `DoWork` method. The timer is disabled on `StopAsync` and disposed when the service container is disposed on `Dispose`:

```
C#
```

```
public class TimedHostedService : IHostedService, IDisposable
{
    private int executionCount = 0;
    private readonly ILogger<TimedHostedService> _logger;
    private Timer? _timer = null;

    public TimedHostedService(ILogger<TimedHostedService> logger)
    {
        _logger = logger;
    }

    public Task StartAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service running.");

        _timer = new Timer(DoWork, null, TimeSpan.Zero,
            TimeSpan.FromSeconds(5));

        return Task.CompletedTask;
    }

    private void DoWork(object? state)
    {
        var count = Interlocked.Increment(ref executionCount);

        _logger.LogInformation(
            "Timed Hosted Service is working. Count: {Count}", count);
    }

    public Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service is stopping.");

        _timer?.Change(Timeout.Infinite, 0);

        return Task.CompletedTask;
    }

    public void Dispose()
    {
        _timer?.Dispose();
    }
}
```

The Timer doesn't wait for previous executions of `DoWork` to finish, so the approach shown might not be suitable for every scenario. Interlocked.Increment is used to increment the execution counter as an atomic operation, which ensures that multiple threads don't update `executionCount` concurrently.

The service is registered in `IHostBuilder.ConfigureServices` (`Program.cs`) with the `AddHostedService` extension method:

C#

```
services.AddHostedService<TimedHostedService>();
```

Consuming a scoped service in a background task

To use scoped services within a BackgroundService, create a scope. No scope is created for a hosted service by default.

The scoped background task service contains the background task's logic. In the following example:

- The service is asynchronous. The `DoWork` method returns a `Task`. For demonstration purposes, a delay of ten seconds is awaited in the `DoWork` method.
- An ILogger is injected into the service.

C#

```
internal interface IScopedProcessingService
{
    Task DoWork(CancellationToken stoppingToken);
}

internal class ScopedProcessingService : IScopedProcessingService
{
    private int executionCount = 0;
    private readonly ILogger _logger;

    public ScopedProcessingService(ILogger<ScopedProcessingService> logger)
    {
        _logger = logger;
    }

    public async Task DoWork(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            executionCount++;

            _logger.LogInformation(
                "Scoped Processing Service is working. Count: {Count}",
executionCount);

            await Task.Delay(10000, stoppingToken);
        }
```

```
    }
}
```

The hosted service creates a scope to resolve the scoped background task service to call its `DoWork` method. `DoWork` returns a `Task`, which is awaited in `ExecuteAsync`:

C#

```
public class ConsumeScopedServiceHostedService : BackgroundService
{
    private readonly ILogger<ConsumeScopedServiceHostedService> _logger;

    public ConsumeScopedServiceHostedService(IServiceProvider services,
        ILogger<ConsumeScopedServiceHostedService> logger)
    {
        Services = services;
        _logger = logger;
    }

    public IServiceProvider Services { get; }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service running.");

        await DoWork(stoppingToken);
    }

    private async Task DoWork(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is working.");

        using (var scope = Services.CreateScope())
        {
            var scopedProcessingService =
                scope.ServiceProvider
                    .GetRequiredService<IScopedProcessingService>();

            await scopedProcessingService.DoWork(stoppingToken);
        }
    }

    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation(
            "Consume Scoped Service Hosted Service is stopping.");

        await base.StopAsync(stoppingToken);
```

```
    }
}
```

The services are registered in `IHostBuilder.ConfigureServices` (`Program.cs`). The hosted service is registered with the `AddHostedService` extension method:

C#

```
services.AddHostedService<ConsumeScopedServiceHostedService>();
services.AddScoped<IScopedProcessingService, ScopedProcessingService>();
```

Queued background tasks

A background task queue is based on the .NET 4.x QueueBackgroundWorkItem:

C#

```
public interface IBackgroundTaskQueue
{
    ValueTask QueueBackgroundWorkItemAsync(Func<CancellationToken,
ValueTask> workItem);

    ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken);
}

public class BackgroundTaskQueue : IBackgroundTaskQueue
{
    private readonly Channel<Func<CancellationToken, ValueTask>> _queue;

    public BackgroundTaskQueue(int capacity)
    {
        // Capacity should be set based on the expected application load and
        // number of concurrent threads accessing the queue.
        // BoundedChannelFullMode.Wait will cause calls to WriteAsync() to
return a task,
        // which completes only when space became available. This leads to
backpressure,
        // in case too many publishers/calls start accumulating.
        var options = new BoundedChannelOptions(capacity)
        {
            FullMode = BoundedChannelFullMode.Wait
        };
        _queue = Channel.CreateBounded<Func<CancellationToken, ValueTask>>
(options);
    }

    public async ValueTask QueueBackgroundWorkItemAsync(
        Func<CancellationToken, ValueTask> workItem)
```

```
    {
        if (workItem == null)
        {
            throw new ArgumentNullException(nameof(workItem));
        }

        await _queue.Writer.WriteAsync(workItem);
    }

    public async ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        var workItem = await _queue.Reader.ReadAsync(cancellationToken);

        return workItem;
    }
}
```

In the following `QueueHostedService` example:

- The `BackgroundProcessing` method returns a `Task`, which is awaited in `ExecuteAsync`.
- Background tasks in the queue are dequeued and executed in `BackgroundProcessing`.
- Work items are awaited before the service stops in `StopAsync`.

C#

```
public class QueuedHostedService : BackgroundService
{
    private readonly ILogger<QueuedHostedService> _logger;

    public QueuedHostedService(IBackgroundTaskQueue taskQueue,
        ILogger<QueuedHostedService> logger)
    {
        TaskQueue = taskQueue;
        _logger = logger;
    }

    public IBackgroundTaskQueue TaskQueue { get; }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        _logger.LogInformation(
            $"Queued Hosted Service is running.{Environment.NewLine}" +
            $"{Environment.NewLine}Tap W to add a work item to the " +
            $"background queue.{Environment.NewLine}");

        await BackgroundProcessing(stoppingToken);
    }
```

```
    private async Task BackgroundProcessing(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            var workItem =
                await TaskQueue.DequeueAsync(stoppingToken);

            try
            {
                await workItem(stoppingToken);
            }
            catch (Exception ex)
            {
                _logger.LogError(ex,
                    "Error occurred executing {WorkItem}.",
nameof(workItem));
            }
        }
    }

    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Queued Hosted Service is stopping.");

        await base.StopAsync(stoppingToken);
    }
}
```

A `MonitorLoop` service handles enqueuing tasks for the hosted service whenever the `w` key is selected on an input device:

- The `IBackgroundTaskQueue` is injected into the `MonitorLoop` service.
- `IBackgroundTaskQueue.QueueBackgroundWorkItem` is called to enqueue a work item.
- The work item simulates a long-running background task:
  - Three 5-second delays are executed (`Task.Delay`).
  - A `try-catch` statement traps OperationCanceledException if the task is cancelled.

C#

```
public class MonitorLoop
{
    private readonly IBackgroundTaskQueue _taskQueue;
    private readonly ILogger _logger;
    private readonly CancellationToken _cancellationToken;

    public MonitorLoop(IBackgroundTaskQueue taskQueue,
        ILogger<MonitorLoop> logger,
        IHostApplicationLifetime applicationLifetime)
```

```
    {
        _taskQueue = taskQueue;
        _logger = logger;
        _cancellationToken = applicationLifetime.ApplicationStopping;
    }

    public void StartMonitorLoop()
    {
        _logger.LogInformation("MonitorAsync Loop is starting.");

        // Run a console user input loop in a background thread
        Task.Run(async () => await MonitorAsync());
    }

    private async ValueTask MonitorAsync()
    {
        while (!_cancellationToken.IsCancellationRequested)
        {
            var keyStroke = Console.ReadKey();

            if (keyStroke.Key == ConsoleKey.W)
            {
                // Enqueue a background work item
                await
_taskQueue.QueueBackgroundWorkItemAsync(BuildWorkItem);
            }
        }
    }

    private async ValueTask BuildWorkItem(CancellationToken token)
    {
        // Simulate three 5-second tasks to complete
        // for each enqueued work item

        int delayLoop = 0;
        var guid = Guid.NewGuid().ToString();

        _logger.LogInformation("Queued Background Task {Guid} is starting.",
guid);

        while (!token.IsCancellationRequested && delayLoop < 3)
        {
            try
            {
                await Task.Delay(TimeSpan.FromSeconds(5), token);
            }
            catch (OperationCanceledException)
            {
                // Prevent throwing if the Delay is cancelled
            }

            delayLoop++;

            _logger.LogInformation("Queued Background Task {Guid} is
running. "
```

```
                                    + "{DelayLoop}/3", guid, delayLoop);
        }

        if (delayLoop == 3)
        {
            _logger.LogInformation("Queued Background Task {Guid} is
complete.", guid);
        }
        else
        {
            _logger.LogInformation("Queued Background Task {Guid} was
cancelled.", guid);
        }
    }
}
```

The services are registered in `IHostBuilder.ConfigureServices` (`Program.cs`). The hosted service is registered with the `AddHostedService` extension method:

C#

```
services.AddSingleton<MonitorLoop>();
services.AddHostedService<QueuedHostedService>();
services.AddSingleton<IBackgroundTaskQueue>(ctx =>
{
    if (!int.TryParse(hostContext.Configuration["QueueCapacity"], out var
queueCapacity))
        queueCapacity = 100;
    return new BackgroundTaskQueue(queueCapacity);
});
```

`MonitorLoop` is started in `Program.cs`:

C#

```
var monitorLoop = host.Services.GetRequiredService<MonitorLoop>();
monitorLoop.StartMonitorLoop();
```

Asynchronous timed background task

The following code creates an asynchronous timed background task:

C#

```
namespace TimedBackgroundTasks;

public class TimedHostedService : BackgroundService
{
```

```
    private readonly ILogger<TimedHostedService> _logger;
    private int _executionCount;

    public TimedHostedService(ILogger<TimedHostedService> logger)
    {
        _logger = logger;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        _logger.LogInformation("Timed Hosted Service running.");

        // When the timer should have no due-time, then do the work once
now.
        DoWork();

        using PeriodicTimer timer = new(TimeSpan.FromSeconds(1));

        try
        {
            while (await timer.WaitForNextTickAsync(stoppingToken))
            {
                DoWork();
            }
        }
        catch (OperationCanceledException)
        {
            _logger.LogInformation("Timed Hosted Service is stopping.");
        }
    }

    // Could also be a async method, that can be awaited in ExecuteAsync
above
    private void DoWork()
    {
        int count = Interlocked.Increment(ref _executionCount);

        _logger.LogInformation("Timed Hosted Service is working. Count:
{Count}", count);
    }
}
```

Native AOT

The Worker Service templates support .NET native ahead-of-time (AOT) with the `--aot` flag:

Visual Studio

1. Create a new project.
2. Select **Worker Service**. Select **Next**.
3. Provide a project name in the **Project name** field or accept the default project name. Select **Next**.
4. In the **Additional information** dialog:
5. Choose a **Framework**.
6. Check the **Enable Native AOT publish** checkbox.
7. Select **Create**.

The AOT option adds `<PublishAot>true</PublishAot>` to the project file:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <InvariantGlobalization>true</InvariantGlobalization>
+   <PublishAot>true</PublishAot>
    <UserSecretsId>dotnet-WorkerWithAot-e94b2</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="8.0.0-
preview.4.23259.5" />
  </ItemGroup>
</Project>
```

Additional resources

- Background services unit tests on GitHub.
- View or download sample code (how to download)
- Implement background tasks in microservices with IHostedService and the BackgroundService class
- Run background tasks with WebJobs in Azure App Service
- Timer

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Use hosting startup assemblies in ASP.NET Core

Article • 04/11/2023

By Pavel Krymets

An IHostingStartup (hosting startup) implementation adds enhancements to an app at startup from an external assembly. For example, an external library can use a hosting startup implementation to provide additional configuration providers or services to an app.

View or download sample code (how to download)

HostingStartup attribute

A HostingStartup attribute indicates the presence of a hosting startup assembly to activate at runtime.

The entry assembly or the assembly containing the `Startup` class is automatically scanned for the `HostingStartup` attribute. The list of assemblies to search for `HostingStartup` attributes is loaded at runtime from configuration in the WebHostDefaults.HostingStartupAssembliesKey. The list of assemblies to exclude from discovery is loaded from the WebHostDefaults.HostingStartupExcludeAssembliesKey.

In the following example, the namespace of the hosting startup assembly is `StartupEnhancement`. The class containing the hosting startup code is `StartupEnhancementHostingStartup`:

C#

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

The `HostingStartup` attribute is typically located in the hosting startup assembly's `IHostingStartup` implementation class file.

Discover loaded hosting startup assemblies

To discover loaded hosting startup assemblies, enable logging and check the app's logs. Errors that occur when loading assemblies are logged. Loaded hosting startup assemblies are logged at the Debug level, and all errors are logged.

Disable automatic loading of hosting startup assemblies

To disable automatic loading of hosting startup assemblies, use one of the following approaches:

- To prevent all hosting startup assemblies from loading, set one of the following to `true` or `1`:
  - Prevent Hosting Startup host configuration setting:

    C#

    ```
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseSetting(
                        WebHostDefaults.PreventHostingStartupKey, "true")
                    .UseStartup<Startup>();
            });
    ```

  - `ASPNETCORE_PREVENTHOSTINGSTARTUP` environment variable.
- To prevent specific hosting startup assemblies from loading, set one of the following to a semicolon-delimited string of hosting startup assemblies to exclude at startup:
  - Hosting Startup Exclude Assemblies host configuration setting:

    C#

    ```
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseSetting(
                        WebHostDefaults.HostingStartupExcludeAssembliesKey,
                        "{ASSEMBLY1;ASSEMBLY2; ...}")
                    .UseStartup<Startup>();
            });
    ```

    The `{ASSEMBLY1;ASSEMBLY2; ...}` placeholder represents the semicolon-separated list of assemblies.

  - `ASPNETCORE_HOSTINGSTARTUPEXCLUDEASSEMBLIES` environment variable.

If both the host configuration setting and the environment variable are set, the host setting controls the behavior.

Disabling hosting startup assemblies using the host setting or environment variable disables the assembly globally and may disable several characteristics of an app.

Project

Create a hosting startup with either of the following project types:

- Class library
- Console app without an entry point

Class library

A hosting startup enhancement can be provided in a class library. The library contains a `HostingStartup` attribute.

The sample code includes a Razor Pages app, *HostingStartupApp*, and a class library, *HostingStartupLibrary*. The class library:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration using the in-memory configuration provider (AddInMemoryCollection).
- Includes a `HostingStartup` attribute that identifies the hosting startup's namespace and class.

The `ServiceKeyInjection` class's Configure method uses an IWebHostBuilder to add enhancements to an app.

`HostingStartupLibrary/ServiceKeyInjection.cs`:

C#

```
[assembly: HostingStartup(typeof(HostingStartupLibrary.ServiceKeyInjection))]

namespace HostingStartupLibrary
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromLibrary", "DEV_1111111-1111"},
                    {"ProdAccount_FromLibrary", "PROD_2222222-2222"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

The app's Index page reads and renders the configuration values for the two keys set by the class library's hosting startup assembly:

`HostingStartupApp/Pages/Index.cshtml.cs`:

C#

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
```

```
            ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
            ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
            ServiceKey_Development_Package = config["DevAccount_FromPackage"];
            ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
        }

        public string ServiceKey_Development_Library { get; private set; }
        public string ServiceKey_Production_Library { get; private set; }
        public string ServiceKey_Development_Package { get; private set; }
        public string ServiceKey_Production_Package { get; private set; }

        public void OnGet()
        {
        }
    }
```

The sample code also includes a NuGet package project that provides a separate hosting startup, *HostingStartupPackage*. The package has the same characteristics of the class library described earlier. The package:

- Contains a hosting startup class, `ServiceKeyInjection`, which implements `IHostingStartup`. `ServiceKeyInjection` adds a pair of service strings to the app's configuration.
- Includes a `HostingStartup` attribute.

`HostingStartupPackage/ServiceKeyInjection.cs`:

C#

```
[assembly: HostingStartup(typeof(HostingStartupPackage.ServiceKeyInjection))]

namespace HostingStartupPackage
{
    public class ServiceKeyInjection : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureAppConfiguration(config =>
            {
                var dict = new Dictionary<string, string>
                {
                    {"DevAccount_FromPackage", "DEV_3333333-3333"},
                    {"ProdAccount_FromPackage", "PROD_4444444-4444"}
                };

                config.AddInMemoryCollection(dict);
            });
        }
    }
}
```

The app's Index page reads and renders the configuration values for the two keys set by the package's hosting startup assembly:

`HostingStartupApp/Pages/Index.cshtml.cs`:

C#

```
public class IndexModel : PageModel
{
    public IndexModel(IConfiguration config)
    {
        ServiceKey_Development_Library = config["DevAccount_FromLibrary"];
        ServiceKey_Production_Library = config["ProdAccount_FromLibrary"];
        ServiceKey_Development_Package = config["DevAccount_FromPackage"];
        ServiceKey_Production_Package = config["ProdAccount_FromPackage"];
    }

    public string ServiceKey_Development_Library { get; private set; }
    public string ServiceKey_Production_Library { get; private set; }
    public string ServiceKey_Development_Package { get; private set; }
    public string ServiceKey_Production_Package { get; private set; }

    public void OnGet()
    {
    }
}
```

Console app without an entry point

This approach is only available for .NET Core apps, not .NET Framework.

A dynamic hosting startup enhancement that doesn't require a compile-time reference for activation can be provided in a console app without an entry point that contains a `HostingStartup` attribute. Publishing the console app produces a hosting startup assembly that can be consumed from the runtime store.

A console app without an entry point is used in this process because:

- A dependencies file is required to consume the hosting startup in the hosting startup assembly. A dependencies file is a runnable app asset that's produced by publishing an app, not a library.
- A library can't be added directly to the runtime package store, which requires a runnable project that targets the shared runtime.

In the creation of a dynamic hosting startup:

- A hosting startup assembly is created from the console app without an entry point that:
  - Includes a class that contains the `IHostingStartup` implementation.
  - Includes a HostingStartup attribute to identify the `IHostingStartup` implementation class.
- The console app is published to obtain the hosting startup's dependencies. A consequence of publishing the console app is that unused dependencies are trimmed from the dependencies file.
- The dependencies file is modified to set the runtime location of the hosting startup assembly.
- The hosting startup assembly and its dependencies file is placed into the runtime package store. To discover the hosting startup assembly and its dependencies file, they're listed in a pair of environment variables.

The console app references the Microsoft.AspNetCore.Hosting.Abstractions package:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Hosting.Abstractions"
                      Version="3.0.0" />
  </ItemGroup>

</Project>
```

A HostingStartup attribute identifies a class as an implementation of `IHostingStartup` for loading and execution when building the IWebHost. In the following example, the namespace is `StartupEnhancement`, and the class is `StartupEnhancementHostingStartup`:

C#

```
[assembly: HostingStartup(typeof(StartupEnhancement.StartupEnhancementHostingStartup))]
```

A class implements `IHostingStartup`. The class's Configure method uses an IWebHostBuilder to add enhancements to an app. `IHostingStartup.Configure` in the hosting startup assembly is called by the runtime before `Startup.Configure` in user code, which allows user code to overwrite any configuration provided by the hosting startup assembly.

C#

```
namespace StartupEnhancement
{
    public class StartupEnhancementHostingStartup : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            // Use the IWebHostBuilder to add app enhancements.
        }
    }
}
```

When building an `IHostingStartup` project, the dependencies file (`.deps.json`) sets the `runtime` location of the assembly to the *bin* folder:

JSON

```
"targets": {
  ".NETCoreApp,Version=v3.0": {
    "StartupEnhancement/1.0.0": {
      "dependencies": {
        "Microsoft.AspNetCore.Hosting.Abstractions": "3.0.0"
```

```
      },
      "runtime": {
        "StartupEnhancement.dll": {}
      }
    }
  }
}
```

Only part of the file is shown. The assembly name in the example is `StartupEnhancement`.

Configuration provided by the hosting startup

There are two approaches to handling configuration depending on whether you want the hosting startup's configuration to take precedence or the app's configuration to take precedence:

1. Provide configuration to the app using ConfigureAppConfiguration to load the configuration after the app's ConfigureAppConfiguration delegates execute. Hosting startup configuration takes priority over the app's configuration using this approach.
2. Provide configuration to the app using UseConfiguration to load the configuration before the app's ConfigureAppConfiguration delegates execute. The app's configuration values take priority over those provided by the hosting startup using this approach.

C#

```
public class ConfigurationInjection : IHostingStartup
{
    public void Configure(IWebHostBuilder builder)
    {
        Dictionary<string, string> dict;

        builder.ConfigureAppConfiguration(config =>
        {
            dict = new Dictionary<string, string>
            {
                {"ConfigurationKey1",
                    "From IHostingStartup: Higher priority " +
                    "than the app's configuration."},
            };

            config.AddInMemoryCollection(dict);
        });

        dict = new Dictionary<string, string>
        {
            {"ConfigurationKey2",
                "From IHostingStartup: Lower priority " +
                "than the app's configuration."},
        };

        var builtConfig = new ConfigurationBuilder()
            .AddInMemoryCollection(dict)
            .Build();

        builder.UseConfiguration(builtConfig);
```

```
    }
}
```

Specify the hosting startup assembly

For either a class library- or console app-supplied hosting startup, specify the hosting startup assembly's name in the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable. The environment variable is a semicolon-delimited list of assemblies.

Only hosting startup assemblies are scanned for the `HostingStartup` attribute. For the sample app, *HostingStartupApp*, to discover the hosting startups described earlier, the environment variable is set to the following value:

```
HostingStartupLibrary;HostingStartupPackage;StartupDiagnostics
```

A hosting startup assembly can also be set using the Hosting Startup Assemblies host configuration setting:

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseSetting(
                    WebHostDefaults.HostingStartupAssembliesKey,
                    "{ASSEMBLY1;ASSEMBLY2; ...}")
                .UseStartup<Startup>();
        });
```

The `{ASSEMBLY1;ASSEMBLY2; ...}` placeholder represents the semicolon-separated list of assemblies.

When multiple hosting startup assembles are present, their Configure methods are executed in the order that the assemblies are listed.

Activation

Options for hosting startup activation are:

- Runtime store: Activation doesn't require a compile-time reference for activation. The sample app places the hosting startup assembly and dependencies files into a folder, *deployment*, to facilitate deployment of the hosting startup in a multimachine environment. The *deployment* folder also includes a PowerShell script that creates or modifies environment variables on the deployment system to enable the hosting startup.
- Compile-time reference required for activation
  - NuGet package

- Project bin folder

Runtime store

The hosting startup implementation is placed in the runtime store. A compile-time reference to the assembly isn't required by the enhanced app.

After the hosting startup is built, a runtime store is generated using the manifest project file and the dotnet store command.

.NET CLI

```
dotnet store --manifest {MANIFEST FILE} --runtime {RUNTIME IDENTIFIER} --output {OUTPUT
LOCATION} --skip-optimization
```

In the sample app (*RuntimeStore* project) the following command is used:

.NET CLI

```
dotnet store --manifest store.manifest.csproj --runtime win7-x64 --output
./deployment/store --skip-optimization
```

For the runtime to discover the runtime store, the runtime store's location is added to the `DOTNET_SHARED_STORE` environment variable.

Modify and place the hosting startup's dependencies file

To activate the enhancement without a package reference to the enhancement, specify additional dependencies to the runtime with `additionalDeps`. `additionalDeps` allows you to:

- Extend the app's library graph by providing a set of additional `.deps.json` files to merge with the app's own `.deps.json` file on startup.
- Make the hosting startup assembly discoverable and loadable.

The recommended approach for generating the additional dependencies file is to:

1. Execute `dotnet publish` on the runtime store manifest file referenced in the previous section.
2. Remove the manifest reference from libraries and the `runtime` section of the resulting `.deps.json` file.

In the example project, the `store.manifest/1.0.0` property is removed from the `targets` and `libraries` section:

JSON

```
{
  "runtimeTarget": {
    "name": ".NETCoreApp,Version=v3.0",
    "signature": ""
  },
```

```
  "compilationOptions": {},
  "targets": {
    ".NETCoreApp,Version=v3.0": {
      "store.manifest/1.0.0": {
        "dependencies": {
          "StartupDiagnostics": "1.0.0"
        },
        "runtime": {
          "store.manifest.dll": {}
        }
      },
      "StartupDiagnostics/1.0.0": {
        "runtime": {
          "lib/netcoreapp3.0/StartupDiagnostics.dll": {
            "assemblyVersion": "1.0.0.0",
            "fileVersion": "1.0.0.0"
          }
        }
      }
    }
  },
  "libraries": {
    "store.manifest/1.0.0": {
      "type": "project",
      "serviceable": false,
      "sha512": ""
    },
    "StartupDiagnostics/1.0.0": {
      "type": "package",
      "serviceable": true,
      "sha512": "sha512-
xrhzuNSyM5/f4ZswhooJ9dmIYLP64wMnqUJSyTKVDKDVj5T+qtzypl8JmM/aFJLLpYrf0FYpVWvGujd7/FfMEw=
=",
      "path": "startupdiagnostics/1.0.0",
      "hashPath": "startupdiagnostics.1.0.0.nupkg.sha512"
    }
  }
}
```

Place the `.deps.json` file into the following location:

```
{ADDITIONAL DEPENDENCIES PATH}/shared/{SHARED FRAMEWORK NAME}/{SHARED FRAMEWORK
VERSION}/{ENHANCEMENT ASSEMBLY NAME}.deps.json
```

- `{ADDITIONAL DEPENDENCIES PATH}`: Location added to the `DOTNET_ADDITIONAL_DEPS` environment variable.
- `{SHARED FRAMEWORK NAME}`: Shared framework required for this additional dependencies file.
- `{SHARED FRAMEWORK VERSION}`: Minimum shared framework version.
- `{ENHANCEMENT ASSEMBLY NAME}`: The enhancement's assembly name.

In the sample app (*RuntimeStore* project), the additional dependencies file is placed into the following location:

```
deployment/additionalDeps/shared/Microsoft.AspNetCore.App/3.0.0/StartupDiagnostics.deps
.json
```

For runtime to discover the runtime store location, the additional dependencies file location is added to the `DOTNET_ADDITIONAL_DEPS` environment variable.

In the sample app (*RuntimeStore* project), building the runtime store and generating the additional dependencies file is accomplished using a PowerShell script.

For examples of how to set environment variables for various operating systems, see Use multiple environments.

Deployment

To facilitate the deployment of a hosting startup in a multimachine environment, the sample app creates a *deployment* folder in published output that contains:

- The hosting startup runtime store.
- The hosting startup dependencies file.
- A PowerShell script that creates or modifies the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES`, `DOTNET_SHARED_STORE`, and `DOTNET_ADDITIONAL_DEPS` to support the activation of the hosting startup. Run the script from an administrative PowerShell command prompt on the deployment system.

NuGet package

A hosting startup enhancement can be provided in a NuGet package. The package has a `HostingStartup` attribute. The hosting startup types provided by the package are made available to the app using either of the following approaches:

- The enhanced app's project file makes a package reference for the hosting startup in the app's project file (a compile-time reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (`.deps.json`). This approach applies to a hosting startup assembly package published to nuget.org.
- The hosting startup's dependencies file is made available to the enhanced app as described in the Runtime store section (without a compile-time reference).

For more information on NuGet packages and the runtime store, see the following topics:

- How to Create a NuGet Package with Cross Platform Tools
- Publishing packages
- Runtime package store

Project bin folder

A hosting startup enhancement can be provided by a *bin*-deployed assembly in the enhanced app. The hosting startup types provided by the assembly are made available to the app using one of the following approaches:

- The enhanced app's project file makes an assembly reference to the hosting startup (a compile-time reference). With the compile-time reference in place, the hosting startup assembly and all of its dependencies are incorporated into the app's dependency file (`.deps.json`). This approach applies when the deployment scenario calls for making a compile-time reference to the hosting startup's assembly (*.dll* file) and moving the assembly to either:
  - The consuming project.
  - A location accessible by the consuming project.
- The hosting startup's dependencies file is made available to the enhanced app as described in the Runtime store section (without a compile-time reference).
- When targeting the .NET Framework, the assembly is loadable in the default load context, which on .NET Framework means that the assembly is located at either of the following locations:
  - Application base path: The *bin* folder where the app's executable (*.exe*) is located.
  - Global Assembly Cache (GAC): The GAC stores assemblies that several .NET Framework apps share. For more information, see How to: Install an assembly into the global assembly cache in the .NET Framework documentation.

Sample code

The sample code (how to download) demonstrates hosting startup implementation scenarios:

- Two hosting startup assemblies (class libraries) set a pair of in-memory configuration key-value pairs each:
  - NuGet package (*HostingStartupPackage*)
  - Class library (*HostingStartupLibrary*)
- A hosting startup is activated from a runtime store-deployed assembly (*StartupDiagnostics*). The assembly adds two middlewares to the app at startup that provide diagnostic information on:
  - Registered services
  - Address (scheme, host, path base, path, query string)
  - Connection (remote IP, remote port, local IP, local port, client certificate)
  - Request headers
  - Environment variables

To run the sample:

Activation from a NuGet package

1. Compile the *HostingStartupPackage* package with the dotnet pack command.
2. Add the package's assembly name of the *HostingStartupPackage* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.

3. Compile and run the app. A package reference is present in the enhanced app (a compile-time reference). A `<PropertyGroup>` in the app's project file specifies the package project's output (*../HostingStartupPackage/bin/Debug*) as a package source. This allows the app to use the package without uploading the package to nuget.org. For more information, see the notes in the HostingStartupApp's project file.

   XML

   ```
   <PropertyGroup>

   <RestoreSources>$(RestoreSources);https://api.nuget.org/v3/index.json;../HostingSt
   artupPackage/bin/Debug</RestoreSources>
   </PropertyGroup>
   ```

4. Observe that the service configuration key values rendered by the Index page match the values set by the package's `ServiceKeyInjection.Configure` method.

If you make changes to the *HostingStartupPackage* project and recompile it, clear the local NuGet package caches to ensure that the *HostingStartupApp* receives the updated package and not a stale package from the local cache. To clear the local NuGet caches, execute the following dotnet nuget locals command:

.NET CLI

```
dotnet nuget locals all --clear
```

Activation from a class library

1. Compile the *HostingStartupLibrary* class library with the dotnet build command.
2. Add the class library's assembly name of *HostingStartupLibrary* to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
3. *bin*-deploy the class library's assembly to the app by copying the *HostingStartupLibrary.dll* file from the class library's compiled output to the app's *bin/Debug* folder.
4. Compile and run the app. An `<ItemGroup>` in the app's project file references the class library's assembly (*.\bin\Debug\netcoreapp3.0\HostingStartupLibrary.dll*) (a compile-time reference). For more information, see the notes in the HostingStartupApp's project file.

   XML

   ```
   <ItemGroup>
     <Reference Include=".\\bin\\Debug\\netcoreapp3.0\\HostingStartupLibrary.dll">
       <HintPath>.\bin\Debug\netcoreapp3.0\HostingStartupLibrary.dll</HintPath>
       <SpecificVersion>False</SpecificVersion>
     </Reference>
   </ItemGroup>
   ```

5. Observe that the service configuration key values rendered by the Index page match the values set by the class library's `ServiceKeyInjection.Configure` method.

Activation from a runtime store-deployed assembly

1. The *StartupDiagnostics* project uses PowerShell to modify its `StartupDiagnostics.deps.json` file. PowerShell is installed by default on Windows starting with Windows 7 SP1 and Windows Server 2008 R2 SP1. To obtain PowerShell on other platforms, see Installing various versions of PowerShell.
2. Execute the *build.ps1* script in the *RuntimeStore* folder. The script:
   - Generates the `StartupDiagnostics` package in the *obj\packages* folder.
   - Generates the runtime store for `StartupDiagnostics` in the *store* folder. The `dotnet store` command in the script uses the `win7-x64` runtime identifier (RID) for a hosting startup deployed to Windows. When providing the hosting startup for a different runtime, substitute the correct RID on line 37 of the script. The runtime store for `StartupDiagnostics` would later be moved to the user's or system's runtime store on the machine where the assembly will be consumed. The user runtime store install location for the `StartupDiagnostics` assembly is *.dotnet/store/x64/netcoreapp3.0/startupdiagnostics/1.0.0/lib/netcoreapp3.0/StartupDiagnostics.dll*.
   - Generates the `additionalDeps` for `StartupDiagnostics` in the *additionalDeps* folder. The additional dependencies would later be moved to the user's or system's additional dependencies. The user `StartupDiagnostics` additional dependencies install location is `.dotnet/x64/additionalDeps/StartupDiagnostics/shared/Microsoft.NETCore.App/3.0.0/StartupDiagnostics.deps.json`.
   - Places the *deploy.ps1* file in the *deployment* folder.
3. Run the *deploy.ps1* script in the *deployment* folder. The script appends:
   - `StartupDiagnostics` to the `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` environment variable.
   - The hosting startup dependencies path (in the RuntimeStore project's *deployment* folder) to the `DOTNET_ADDITIONAL_DEPS` environment variable.
   - The runtime store path (in the RuntimeStore project's *deployment* folder) to the `DOTNET_SHARED_STORE` environment variable.
4. Run the sample app.
5. Request the `/services` endpoint to see the app's registered services. Request the `/diag` endpoint to see the diagnostic information.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

information, see our contributor guide.

Provide product feedback

Use ASP.NET Core APIs in a class library

Article • 09/19/2023

By Scott Addie

This document provides guidance for using ASP.NET Core APIs in a class library. For all other library guidance, see Open-source library guidance.

Determine which ASP.NET Core versions to support

ASP.NET Core adheres to the .NET Core support policy. Consult the support policy when determining which ASP.NET Core versions to support in a library. A library should:

- Make an effort to support all ASP.NET Core versions classified as *Long-Term Support* (LTS).
- Not feel obligated to support ASP.NET Core versions classified as *End of Life* (EOL).

As preview releases of ASP.NET Core are made available, breaking changes are posted in the aspnet/Announcements GitHub repository. Compatibility testing of libraries can be conducted as framework features are being developed.

Use the ASP.NET Core shared framework

With the release of .NET Core 3.0, many ASP.NET Core assemblies are no longer published to NuGet as packages. Instead, the assemblies are included in the `Microsoft.AspNetCore.App` shared framework, which is installed with the .NET Core SDK and runtime installers. For a list of packages no longer being published, see Remove obsolete package references.

As of .NET Core 3.0, projects using the `Microsoft.NET.Sdk.Web` MSBuild SDK implicitly reference the shared framework. Projects using the `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Razor` SDK must reference ASP.NET Core to use ASP.NET Core APIs in the shared framework.

To reference ASP.NET Core, add the following `<FrameworkReference>` element to your project file:

```
XML
```

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

Include Blazor extensibility

Blazor supports creating Razor components class libraries for server-side and client-side apps. To support Razor components in a class library, the class library must use the Microsoft.NET.Sdk.Razor SDK.

Support server-side and client-side apps

To support Razor component consumption by server-side and client-side apps from a single library, use the following instructions for your editor.

Visual Studio

Use the **Razor Class Library** project template.

> ⓘ Note
>
> Do ***not*** select the **Support pages and views** checkbox. Selecting the checkbox results in a class library that only supports server-side apps.

The library generated from the project template:

- Targets the current .NET framework based on the installed SDK.
- Enables browser compatibility checks for platform dependencies by including `browser` as a supported platform with the `SupportedPlatform` MSBuild item.
- Adds a NuGet package reference for Microsoft.AspNetCore.Components.Web.

RazorClassLibrary-CSharp.csproj (reference source)

> **Note**
>
> Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see How to select a version tag of ASP.NET Core source code (dotnet/AspNetCore.Docs #26205).

Support multiple framework versions

If the library must support features added to Blazor in the current release while also supporting one or more earlier releases, multi-target the library. Provide a semicolon-separated list of Target Framework Monikers (TFMs) in the `TargetFrameworks` MSBuild property:

XML

```xml
<TargetFrameworks>{TARGET FRAMEWORKS}</TargetFrameworks>
```

In the preceding example, the `{TARGET FRAMEWORKS}` placeholder represents the semicolon-separated TFMs list. For example, `netcoreapp3.1;net5.0`.

Only support server-side consumption

Class libraries are rarely built to only support server-side apps. If the class library only requires server-side-specific features, such as access to CircuitHandler or Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage, or uses ASP.NET Core-specific features, such as middleware, MVC controllers, or Razor Pages, use **one** of the following approaches:

- Specify that the library supports pages and views when the library is created with the **Support pages and views** checkbox (Visual Studio) or the `-s|--support-pages-and-views` option with the `dotnet new` command:

  .NET CLI

  ```
  dotnet new razorclasslib -s
  ```

- Only provide a framework reference to ASP.NET Core in the library's project file in addition to any other required MSBuild properties:

XML

```xml
<ItemGroup>
  <FrameworkReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

For more information on libraries containing Razor components, see Consume ASP.NET Core Razor components from a Razor class library (RCL).

Include MVC extensibility

This section outlines recommendations for libraries that include:

- Razor views or Razor Pages
- Tag Helpers
- View components

This section doesn't discuss multi-targeting to support multiple versions of MVC. For guidance on supporting multiple ASP.NET Core versions, see Support multiple ASP.NET Core versions.

Razor views or Razor Pages

A project that includes Razor views or Razor Pages must use the Microsoft.NET.Sdk.Razor SDK.

If the project targets .NET Core 3.x, it requires:

- An `AddRazorSupportForMvc` MSBuild property set to `true`.
- A `<FrameworkReference>` element for the shared framework.

The **Razor Class Library** project template satisfies the preceding requirements for projects targeting .NET Core. Use the following instructions for your editor.

Visual Studio

Use the **Razor Class Library** project template. The template's **Support pages and views** checkbox should be selected.

For example:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

If the project targets .NET Standard instead, a Microsoft.AspNetCore.Mvc package reference is required. The `Microsoft.AspNetCore.Mvc` package moved into the shared framework in ASP.NET Core 3.0 and is therefore no longer published. For example:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  </ItemGroup>

</Project>
```

Tag Helpers

A project that includes Tag Helpers should use the `Microsoft.NET.Sdk` SDK. If targeting .NET Core 3.x, add a `<FrameworkReference>` element for the shared framework. For example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
```

```
</Project>
```

If targeting .NET Standard (to support versions earlier than ASP.NET Core 3.x), add a package reference to Microsoft.AspNetCore.Mvc.Razor. The `Microsoft.AspNetCore.Mvc.Razor` package moved into the shared framework and is therefore no longer published. For example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  </ItemGroup>

</Project>
```

View components

A project that includes View components should use the `Microsoft.NET.Sdk` SDK. If targeting .NET Core 3.x, add a `<FrameworkReference>` element for the shared framework. For example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

If targeting .NET Standard (to support versions earlier than ASP.NET Core 3.x), add a package reference to Microsoft.AspNetCore.Mvc.ViewFeatures. The `Microsoft.AspNetCore.Mvc.ViewFeatures` package moved into the shared framework and is therefore no longer published. For example:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Mvc.ViewFeatures"
Version="2.2.0" />
  </ItemGroup>

</Project>
```

Support multiple ASP.NET Core versions

Multi-targeting is required to author a library that supports multiple variants of ASP.NET Core. Consider a scenario in which a Tag Helpers library must support the following ASP.NET Core variants:

- ASP.NET Core 2.1 targeting .NET Framework 4.6.1
- ASP.NET Core 2.x targeting .NET Core 2.x
- ASP.NET Core 3.x targeting .NET Core 3.x

The following project file supports these variants via the `TargetFrameworks` property:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netcoreapp2.1;netcoreapp3.1;net461</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.1'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor"
Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.1'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```

With the preceding project file:

- The `Markdig` package is added for all consumers.
- A reference to Microsoft.AspNetCore.Mvc.Razor is added for consumers targeting .NET Framework 4.6.1 or later or .NET Core 2.x. Version 2.1.0 of the package works with ASP.NET Core 2.2 because of backwards compatibility.
- The shared framework is referenced for consumers targeting .NET Core 3.x. The `Microsoft.AspNetCore.Mvc.Razor` package is included in the shared framework.

Alternatively, .NET Standard 2.0 could be targeted instead of targeting both .NET Core 2.1 and .NET Framework 4.6.1:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netstandard2.0;netcoreapp3.1</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.1'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor"
Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.1'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```

With the preceding project file, the following caveats exist:

- Since the library only contains Tag Helpers, it's more straightforward to target the specific platforms on which ASP.NET Core runs: .NET Core and .NET Framework. Tag Helpers can't be used by other .NET Standard 2.0-compliant target frameworks such as Unity, UWP, and Xamarin.
- Using .NET Standard 2.0 from .NET Framework has some issues that were addressed in .NET Framework 4.7.2. You can improve the experience for consumers using .NET Framework 4.6.1 through 4.7.1 by targeting .NET Framework 4.6.1.

If your library needs to call platform-specific APIs, target specific .NET implementations instead of .NET Standard. For more information, see Multi-targeting.

Use an API that hasn't changed

Imagine a scenario in which you're upgrading a middleware library from .NET Core 2.2 to 3.1. The ASP.NET Core middleware APIs being used in the library haven't changed between ASP.NET Core 2.2 and 3.1. To continue supporting the middleware library in .NET Core 3.1, take the following steps:

- Follow the standard library guidance.
- Add a package reference for each API's NuGet package if the corresponding assembly doesn't exist in the shared framework.

Use an API that changed

Imagine a scenario in which you're upgrading a library from .NET Core 2.2 to .NET Core 3.1. An ASP.NET Core API being used in the library has a breaking change in ASP.NET Core 3.1. Consider whether the library can be rewritten to not use the broken API in all versions.

If you can rewrite the library, do so and continue to target an earlier target framework (for example, .NET Standard 2.0 or .NET Framework 4.6.1) with package references.

If you can't rewrite the library, take the following steps:

- Add a target for .NET Core 3.1.
- Add a `<FrameworkReference>` element for the shared framework.
- Use the #if preprocessor directive with the appropriate target framework symbol to conditionally compile code.

For example, synchronous reads and writes on HTTP request and response streams are disabled by default as of ASP.NET Core 3.1. ASP.NET Core 2.2 supports the synchronous behavior by default. Consider a middleware library in which synchronous reads and writes should be enabled where I/O is occurring. The library should enclose the code to enable synchronous features in the appropriate preprocessor directive. For example:

C#

```
public async Task Invoke(HttpContext httpContext)
{
    if (httpContext.Request.Path.StartsWithSegments(_path,
StringComparison.Ordinal))
    {
        httpContext.Response.StatusCode = (int) HttpStatusCode.OK;
        httpContext.Response.ContentType = "application/json";
        httpContext.Response.ContentLength = _bufferSize;
```

```
#if !NETCOREAPP3_1 && !NETCOREAPP5_0
        var syncIOFeature =
httpContext.Features.Get<IHttpBodyControlFeature>();
        if (syncIOFeature != null)
        {
            syncIOFeature.AllowSynchronousIO = true;
        }

        using (var sw = new StreamWriter(
            httpContext.Response.Body, _encoding, bufferSize: _bufferSize))
        {
            _json.Serialize(sw, new JsonMessage { message = "Hello, World!"
});
        }
#else
        await JsonSerializer.SerializeAsync<JsonMessage>(
            httpContext.Response.Body, new JsonMessage { message = "Hello,
World!" });
#endif
        return;
    }

    await _next(httpContext);
}
```

Use an API introduced in 3.1

Imagine that you want to use an ASP.NET Core API that was introduced in ASP.NET Core 3.1. Consider the following questions:

1. Does the library functionally require the new API?
2. Can the library implement this feature in a different way?

If the library functionally requires the API and there's no way to implement it down-level:

- Target .NET Core 3.x only.
- Add a `<FrameworkReference>` element for the shared framework.

If the library can implement the feature in a different way:

- Add .NET Core 3.x as a target framework.
- Add a `<FrameworkReference>` element for the shared framework.
- Use the #if preprocessor directive with the appropriate target framework symbol to conditionally compile code.

For example, the following Tag Helper uses the IWebHostEnvironment interface introduced in ASP.NET Core 3.1. Consumers targeting .NET Core 3.1 execute the code

path defined by the `NETCOREAPP3_1` target framework symbol. The Tag Helper's constructor parameter type changes to IHostingEnvironment for .NET Core 2.1 and .NET Framework 4.6.1 consumers. This change was necessary because ASP.NET Core 3.1 marked `IHostingEnvironment` as obsolete and recommended `IWebHostEnvironment` as the replacement.

C#

```
[HtmlTargetElement("script", Attributes = "asp-inline")]
public class ScriptInliningTagHelper : TagHelper
{
    private readonly IFileProvider _wwwroot;

#if NETCOREAPP3_1
    public ScriptInliningTagHelper(IWebHostEnvironment env)
#else
    public ScriptInliningTagHelper(IHostingEnvironment env)
#endif
    {
        _wwwroot = env.WebRootFileProvider;
    }

    // code omitted for brevity
}
```

The following multi-targeted project file supports this Tag Helper scenario:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netcoreapp2.1;netcoreapp3.1;net461</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Markdig" Version="0.16.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' != 'netcoreapp3.1'">
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor"
Version="2.1.0" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.1'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>
</Project>
```

Use an API removed from the shared framework

To use an ASP.NET Core assembly that was removed from the shared framework, add the appropriate package reference. For a list of packages removed from the shared framework in ASP.NET Core 3.1, see Remove obsolete package references.

For example, to add the web API client:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNet.WebApi.Client"
Version="5.2.7" />
  </ItemGroup>

</Project>
```

Additional resources

- Reusable Razor UI in class libraries with ASP.NET Core
- Consume ASP.NET Core Razor components from a Razor class library (RCL)
- .NET implementation support
- .NET support policies

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

more information, see our contributor guide.

Microsoft.AspNetCore.App for ASP.NET Core

Article • 07/26/2024

The ASP.NET Core shared framework (`Microsoft.AspNetCore.App`) contains assemblies that are developed and supported by Microsoft. `Microsoft.AspNetCore.App` is installed when the .NET Core 3.0 or later SDK is installed. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and includes a runtime component and a targeting pack. For more information, see The shared framework.

- Projects that target the `Microsoft.NET.Sdk.Web` SDK implicitly reference the `Microsoft.AspNetCore.App` framework.

No additional references are required for these projects:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>
    ...
</Project>
```

The ASP.NET Core shared framework:

- Doesn't include third-party dependencies.
- Includes all supported packages by the ASP.NET Core team.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0

Article • 07/26/2024

> **Note**
>
> The `Microsoft.AspNetCore.All` metapackage isn't used in ASP.NET Core 3.0 and later. For more information, see this GitHub issue.

> **Note**
>
> We recommend applications targeting ASP.NET Core 2.1 and later use the Microsoft.AspNetCore.App metapackage rather than this package. See Migrating from Microsoft.AspNetCore.All to Microsoft.AspNetCore.App in this article.

This feature requires ASP.NET Core 2.x targeting .NET Core 2.x.

Microsoft.AspNetCore.All is a metapackage that refers to a shared framework. A *shared framework* is a set of assemblies (*.dll* files) that are not in the app's folders. The shared framework must be installed on the machine to run the app. For more information, see The shared framework.

The shared framework that `Microsoft.AspNetCore.All` refers to includes:

- All supported packages by the ASP.NET Core team.
- All supported packages by the Entity Framework Core.
- Internal and 3rd-party dependencies used by ASP.NET Core and Entity Framework Core.

All the features of ASP.NET Core 2.x and Entity Framework Core 2.x are included in the `Microsoft.AspNetCore.All` package. The default project templates targeting ASP.NET Core 2.0 use this package.

The version number of the `Microsoft.AspNetCore.All` metapackage represents the minimum ASP.NET Core version and Entity Framework Core version.

The following `.csproj` file references the `Microsoft.AspNetCore.All` metapackage for ASP.NET Core:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
  </ItemGroup>

</Project>
```

Implicit versioning

In ASP.NET Core 2.1 or later, you can specify the `Microsoft.AspNetCore.All` package reference without a version. When the version isn't specified, an implicit version is specified by the SDK (`Microsoft.NET.Sdk.Web`). We recommend relying on the implicit version specified by the SDK and not explicitly setting the version number on the package reference. If you have questions about this approach, leave a GitHub comment at the Discussion for the Microsoft.AspNetCore.App implicit version.

The implicit version is set to `major.minor.0` for portable apps. The shared framework roll-forward mechanism runs the app on the latest compatible version among the installed shared frameworks. To guarantee the same version is used in development, test, and production, ensure the same version of the shared framework is installed in all environments. For self-contained apps, the implicit version number is set to the `major.minor.patch` of the shared framework bundled in the installed SDK.

Specifying a version number on the `Microsoft.AspNetCore.All` package reference does **not** guarantee that version of the shared framework is chosen. For example, suppose version "2.1.1" is specified, but "2.1.3" is installed. In that case, the app will use "2.1.3". Although not recommended, you can disable roll forward (patch and/or minor). For more information regarding dotnet host roll-forward and how to configure its behavior, see dotnet host roll forward.

The project's SDK must be set to `Microsoft.NET.Sdk.Web` in the project file to use the implicit version of `Microsoft.AspNetCore.All`. When the `Microsoft.NET.Sdk` SDK is specified (`<Project Sdk="Microsoft.NET.Sdk">` at the top of the project file), the following warning is generated:

Warning NU1604: Project dependency Microsoft.AspNetCore.All does not contain an inclusive lower bound. Include a lower bound in the dependency version to ensure

consistent restore results.

This is a known issue with the .NET Core 2.1 SDK and will be fixed in the .NET Core 2.2 SDK.

Migrating from Microsoft.AspNetCore.All to Microsoft.AspNetCore.App

The following packages are included in `Microsoft.AspNetCore.All` but not the `Microsoft.AspNetCore.App` package.

- `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServices.HostingStartup`
- `Microsoft.AspNetCore.AzureAppServicesIntegration`
- `Microsoft.AspNetCore.DataProtection.AzureKeyVault`
- `Microsoft.AspNetCore.DataProtection.AzureStorage`
- `Microsoft.AspNetCore.Server.Kestrel.Transport.Libuv`
- `Microsoft.AspNetCore.SignalR.Redis`
- `Microsoft.Data.Sqlite`
- `Microsoft.Data.Sqlite.Core`
- `Microsoft.EntityFrameworkCore.Sqlite`
- `Microsoft.EntityFrameworkCore.Sqlite.Core`
- `Microsoft.Extensions.Caching.Redis`
- `Microsoft.Extensions.Configuration.AzureKeyVault`
- `Microsoft.Extensions.Logging.AzureAppServices`
- `Microsoft.VisualStudio.Web.BrowserLink`

To move from `Microsoft.AspNetCore.All` to `Microsoft.AspNetCore.App`, if your app uses any APIs from the above packages, or packages brought in by those packages, add references to those packages in your project.

Any dependencies of the preceding packages that otherwise aren't dependencies of `Microsoft.AspNetCore.App` are not included implicitly. For example:

- `StackExchange.Redis` as a dependency of `Microsoft.Extensions.Caching.Redis`
- `Microsoft.ApplicationInsights` as a dependency of `Microsoft.AspNetCore.ApplicationInsights.HostingStartup`

Update ASP.NET Core 2.1

We recommend migrating to the `Microsoft.AspNetCore.App` metapackage for 2.1 and later. To keep using the `Microsoft.AspNetCore.All` metapackage and ensure the latest patch version is deployed:

- On development machines and build servers: Install the latest .NET Core SDK.
- On deployment servers: Install the latest .NET Core runtime. Your app will roll forward to the latest installed version on an application restart.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

High-performance logging in .NET

Article • 04/11/2024

The LoggerMessage class exposes functionality to create cacheable delegates that require fewer object allocations and reduced computational overhead compared to logger extension methods, such as LogInformation and LogDebug. For high-performance logging scenarios, use the LoggerMessage pattern.

LoggerMessage provides the following performance advantages over logger extension methods:

- Logger extension methods require "boxing" (converting) value types, such as `int`, into `object`. The LoggerMessage pattern avoids boxing by using static Action fields and extension methods with strongly typed parameters.
- Logger extension methods must parse the message template (named format string) every time a log message is written. LoggerMessage only requires parsing a template once when the message is defined.

> **Important**
>
> Instead of using the **LoggerMessage class** to create high-performance logs, you can use the **LoggerMessage attribute** in .NET 6 and later versions. The `LoggerMessageAttribute` provides source-generation logging support designed to deliver a highly usable and highly performant logging solution for modern .NET applications. For more information, see **Compile-time logging source generation (.NET Fundamentals)**.

The sample app demonstrates LoggerMessage features with a priority queue processing worker service. The app processes work items in priority order. As these operations occur, log messages are generated using the LoggerMessage pattern.

> **Tip**
>
> All of the logging example source code is available in the **Samples Browser** for download. For more information, see **Browse code samples: Logging in .NET**.

Define a logger message

Use Define(LogLevel, EventId, String) to create an Action delegate for logging a message. Define overloads permit passing up to six type parameters to a named format string (template).

The string provided to the Define method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend Pascal casing for placeholder names. For example, `{Item}`, `{DateTime}`.

Each log message is an Action held in a static field created by LoggerMessage.Define. For example, the sample app creates a field to describe a log message for the processing of work items:

C#

```
private static readonly Action<ILogger, Exception>
s_failedToProcessWorkItem;
```

For the Action, specify:

- The log level.
- A unique event identifier (EventId) with the name of the static extension method.
- The message template (named format string).

As work items are dequeued for processing, the worker service app sets the:

- Log level to LogLevel.Critical.
- Event ID to `13` with the name of the `FailedToProcessWorkItem` method.
- Message template (named format string) to a string.

C#

```
s_failedToProcessWorkItem = LoggerMessage.Define(
    LogLevel.Critical,
    new EventId(13, nameof(FailedToProcessWorkItem)),
    "Epic failure processing item!");
```

The LoggerMessage.Define method is used to configure and define an Action delegate, which represents a log message.

Structured logging stores may use the event name when it's supplied with the event ID to enrich logging. For example, Serilog uses the event name.

The Action is invoked through a strongly typed extension method. The `PriorityItemProcessed` method logs a message every time a work item is processed. `FailedToProcessWorkItem` is called if and when an exception occurs:

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                WorkItem? nextItem =
priorityQueue.ProcessNextHighestPriority();

                if (nextItem is not null)
                {
                    logger.PriorityItemProcessed(nextItem);
                }
            }
            catch (Exception ex)
            {
                logger.FailedToProcessWorkItem(ex);
            }

            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

Inspect the app's console output:

Console

```
crit: WorkerServiceOptions.Example.Worker[13]
      Epic failure processing item!
      System.Exception: Failed to verify communications.
         at
WorkerServiceOptions.Example.Worker.ExecuteAsync(CancellationToken
stoppingToken) in
         ..\Worker.cs:line 27
```

To pass parameters to a log message, define up to six types when creating the static field. The sample app logs the work item details when processing items by defining a `WorkItem` type for the Action field:

C#

```
private static readonly Action<ILogger, WorkItem, Exception>
s_processingPriorityItem;
```

The delegate's log message template receives its placeholder values from the types provided. The sample app defines a delegate for adding a work item where the item parameter is a `WorkItem`:

C#

```
s_processingPriorityItem = LoggerMessage.Define<WorkItem>(
    LogLevel.Information,
    new EventId(1, nameof(PriorityItemProcessed)),
    "Processing priority item: {Item}");
```

The static extension method for logging that a work item is being processed, `PriorityItemProcessed`, receives the work item argument value and passes it to the Action delegate:

C#

```
public static void PriorityItemProcessed(
    this ILogger logger, WorkItem workItem) =>
    s_processingPriorityItem(logger, workItem, default!);
```

In the worker service's `ExecuteAsync` method, `PriorityItemProcessed` is called to log the message:

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                WorkItem? nextItem =
priorityQueue.ProcessNextHighestPriority();

                if (nextItem is not null)
                {
                    logger.PriorityItemProcessed(nextItem);
                }
            }
```

```
            catch (Exception ex)
            {
                logger.FailedToProcessWorkItem(ex);
            }

            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

Inspect the app's console output:

Console

```
info: WorkerServiceOptions.Example.Worker[1]
      Processing priority item: Priority-Extreme (50db062a-9732-4418-936d-
110549ad79e4): 'Verify communications'
```

Define logger message scope

The DefineScope(string) method creates a Func<TResult> delegate for defining a log scope. DefineScope overloads permit passing up to six type parameters to a named format string (template).

As is the case with the Define method, the string provided to the DefineScope method is a template and not an interpolated string. Placeholders are filled in the order that the types are specified. Placeholder names in the template should be descriptive and consistent across templates. They serve as property names within structured log data. We recommend Pascal casing for placeholder names. For example, `{Item}`, `{DateTime}`.

Define a log scope to apply to a series of log messages using the DefineScope method. Enable `IncludeScopes` in the console logger section of *appsettings.json*:

JSON

```
{
    "Logging": {
        "Console": {
            "IncludeScopes": true
        },
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
```

```
    }
}
```

To create a log scope, add a field to hold a Func<TResult> delegate for the scope. The sample app creates a field named `s_processingWorkScope` (*Internal/LoggerExtensions.cs*):

C#

```
private static readonly Func<ILogger, DateTime, IDisposable?>
s_processingWorkScope;
```

Use DefineScope to create the delegate. Up to six types can be specified for use as template arguments when the delegate is invoked. The sample app uses a message template that includes the date time in which processing started:

C#

```
s_processingWorkScope =
    LoggerMessage.DefineScope<DateTime>(
        "Processing scope, started at: {DateTime}");
```

Provide a static extension method for the log message. Include any type parameters for named properties that appear in the message template. The sample app takes in a `DateTime` for a custom time stamp to log and returns `_processingWorkScope`:

C#

```
public static IDisposable? ProcessingWorkScope(
    this ILogger logger, DateTime time) =>
    s_processingWorkScope(logger, time);
```

The scope wraps the logging extension calls in a using block:

C#

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using (IDisposable? scope = logger.ProcessingWorkScope(DateTime.Now))
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                WorkItem? nextItem =
priorityQueue.ProcessNextHighestPriority();
```

```
                if (nextItem is not null)
                {
                    logger.PriorityItemProcessed(nextItem);
                }
            }
            catch (Exception ex)
            {
                logger.FailedToProcessWorkItem(ex);
            }

            await Task.Delay(1_000, stoppingToken);
        }
    }
}
```

Inspect the log messages in the app's console output. The following result shows priority ordering of log messages with the log scope message included:

Console

```
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
      Processing priority item: Priority-Extreme (7d153ef9-8894-4282-836a-
8e5e38319fb3): 'Verify communications'
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\source\repos\dotnet-
docs\docs\core\extensions\snippets\logging\worker-service-options
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
      Processing priority item: Priority-High (dbad6558-60cd-4eb1-8531-
231e90081f62): 'Validate collection'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
      Processing priority item: Priority-Medium (1eabe213-dc64-4e3a-9920-
f67fe1dfb0f6): 'Propagate selections'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
      Processing priority item: Priority-Medium (1142688d-d4dc-4f78-95c5-
04ec01cbfac7): 'Enter pooling [contention]'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
      Processing priority item: Priority-Low (e85e0c4d-0840-476e-b8b0-
22505c08e913): 'Health check network'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
      Processing priority item: Priority-Deferred (07571363-d559-4e72-bc33-
cd8398348786): 'Ping weather service'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing scope, started at: 04/11/2024 11:27:52
```

```
      Processing priority item: Priority-Deferred (2bf74f2f-0198-4831-8138-
03368e60bd6b): 'Set process state'
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
```

Log level guarded optimizations

Another performance optimization can be made by checking the LogLevel, with ILogger.IsEnabled(LogLevel) before an invocation to the corresponding `Log*` method. When logging isn't configured for the given `LogLevel`, the following statements are true:

- ILogger.Log isn't called.
- An allocation of `object[]` representing the parameters is avoided.
- Value type boxing is avoided.

For more information:

- Micro benchmarks in the .NET runtime
- Background and motivation for log level checks

See also

- Logging in .NET

Develop ASP.NET Core apps using a file watcher

Article • 04/11/2023

By Rick Anderson and Victor Hurdugaci

`dotnet watch` is a tool that runs a .NET CLI command when source files change. For example, a file change can trigger compilation, test execution, or deployment.

This tutorial uses an existing web API with two endpoints: one that returns a sum and one that returns a product. The product method has a bug, which is fixed in this tutorial.

Download the sample app. It consists of two projects: *WebApp* (an ASP.NET Core web API) and *WebAppTests* (unit tests for the web API).

In a command shell, navigate to the *WebApp* folder. Run the following command:

.NET CLI

```
dotnet run
```

> **Note**
>
> You can use `dotnet run --project <PROJECT>` to specify a project to run. For example, running `dotnet run --project WebApp` from the root of the sample app will also run the *WebApp* project.

The console output shows messages similar to the following (indicating that the app is running and awaiting requests):

Console

```
$ dotnet run
Hosting environment: Development
Content root path: C:/Docs/aspnetcore/tutorials/dotnet-watch/sample/WebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

In a web browser, navigate to `http://localhost:<port number>/api/math/sum?a=4&b=5`. You should see the result of `9`.

Navigate to the product API (`http://localhost:<port number>/api/math/product?a=4&b=5`). It returns `9`, not `20` as you'd expect. That problem is fixed later in the tutorial.

Run .NET CLI commands using `dotnet watch`

Any .NET CLI command can be run with `dotnet watch`. For example:

Expand table

Command	Command with watch
dotnet run	dotnet watch run
dotnet run -f netcoreapp3.1	dotnet watch run -f netcoreapp3.1
dotnet run -f netcoreapp3.1 -- --arg1	dotnet watch run -f netcoreapp3.1 -- --arg1
dotnet test	dotnet watch test

Run `dotnet watch run` in the *WebApp* folder. The console output indicates `watch` has started.

Running `dotnet watch run` on a web app launches a browser that navigates to the app's URL once ready. `dotnet watch` does this by reading the app's console output and waiting for the ready message displayed by WebHost.

`dotnet watch` refreshes the browser when it detects changes to watched files. To do this, the watch command injects a middleware to the app that modifies HTML responses created by the app. The middleware adds a JavaScript script block to the page that allows `dotnet watch` to instruct the browser to refresh. Currently, changes to all watched files, including static content such as `.html` and `.css` files cause the app to be rebuilt.

`dotnet watch`:

- Only watches files that impact builds by default.
- Any additionally watched files (via configuration) still results in a build taking place.

For more information on configuration, see dotnet-watch configuration in this document.

> **Note**
>
> You can use `dotnet watch --project <PROJECT>` to specify a project to watch. For example, running `dotnet watch --project WebApp run` from the root of the sample

app will also run and watch the *WebApp* project.

Make changes with `dotnet watch`

Make sure `dotnet watch` is running.

Fix the bug in the `Product` method of `MathController.cs` so it returns the product and not the sum:

```C#
public static int Product(int a, int b)
{
    return a * b;
}
```

Save the file. The console output indicates that `dotnet watch` detected a file change and restarted the app.

Verify `http://localhost:<port number>/api/math/product?a=4&b=5` returns the correct result.

Run tests using `dotnet watch`

1. Change the `Product` method of `MathController.cs` back to returning the sum. Save the file.
2. In a command shell, navigate to the *WebAppTests* folder.
3. Run dotnet restore.
4. Run `dotnet watch test`. Its output indicates that a test failed and that the watcher is awaiting file changes:

   ```Console
   Total tests: 2. Passed: 1. Failed: 1. Skipped: 0.
   Test Run Failed.
   ```

5. Fix the `Product` method code so it returns the product. Save the file.

`dotnet watch` detects the file change and reruns the tests. The console output indicates the tests passed.

Customize files list to watch

By default, `dotnet-watch` tracks all files matching the following glob patterns:

- `**/*.cs`
- `*.csproj`
- `**/*.resx`
- Content files: `wwwroot/**`, `**/*.config`, `**/*.json`

More items can be added to the watch list by editing the `.csproj` file. Items can be specified individually or by using glob patterns.

XML

```
<ItemGroup>
    <!-- extends watching group to include *.js files -->
    <Watch Include="**\*.js"
Exclude="node_modules\**\*;**\*.js.map;obj\**\*;bin\**\*" />
</ItemGroup>
```

Opt-out of files to be watched

`dotnet-watch` can be configured to ignore its default settings. To ignore specific files, add the `Watch="false"` attribute to an item's definition in the `.csproj` file:

XML

```
<ItemGroup>
    <!-- exclude Generated.cs from dotnet-watch -->
    <Compile Include="Generated.cs" Watch="false" />

    <!-- exclude Strings.resx from dotnet-watch -->
    <EmbeddedResource Include="Strings.resx" Watch="false" />

    <!-- exclude changes in this referenced project -->
    <ProjectReference Include="..\ClassLibrary1\ClassLibrary1.csproj"
Watch="false" />
</ItemGroup>
```

XML

```
<ItemGroup>
     <!-- Exclude all Content items from being watched. -->
    <Content Update="@(Content)" Watch="false" />
</ItemGroup>
```

Custom watch projects

`dotnet-watch` isn't restricted to C# projects. Custom watch projects can be created to handle different scenarios. Consider the following project layout:

- **test/**
  - `UnitTests/UnitTests.csproj`
  - `IntegrationTests/IntegrationTests.csproj`

If the goal is to watch both projects, create a custom project file configured to watch both projects:

XML

```xml
<Project>
    <ItemGroup>
        <TestProjects Include="**\*.csproj" />
        <Watch Include="**\*.cs" />
    </ItemGroup>

    <Target Name="Test">
        <MSBuild Targets="VSTest" Projects="@(TestProjects)" />
    </Target>

    <Import Project="$(MSBuildExtensionsPath)\Microsoft.Common.targets" />
</Project>
```

To start file watching on both projects, change to the *test* folder. Execute the following command:

.NET CLI

```dotnetcli
dotnet watch msbuild /t:Test
```

VSTest executes when any file changes in either test project.

dotnet-watch configuration

Some configuration options can be passed to `dotnet watch` through environment variables. The available variables are:

Expand table

Setting	Description
`DOTNET_USE_POLLING_FILE_WATCHER`	If set to "1" or "true", `dotnet watch` uses a polling file watcher instead of CoreFx's `FileSystemWatcher`. Used when watching files on network shares or Docker mounted volumes.
`DOTNET_WATCH_SUPPRESS_MSBUILD_INCREMENTALISM`	By default, `dotnet watch` optimizes the build by avoiding certain operations such as running restore or re-evaluating the set of watched files on every file change. If set to "1" or "true", these optimizations are disabled.
`DOTNET_WATCH_SUPPRESS_LAUNCH_BROWSER`	`dotnet watch run` attempts to launch browsers for web apps with `launchBrowser` configured in `launchSettings.json`. If set to "1" or "true", this behavior is suppressed.
`DOTNET_WATCH_SUPPRESS_BROWSER_REFRESH`	`dotnet watch run` attempts to refresh browsers when it detects file changes. If set to "1" or "true", this behavior is suppressed. This behavior is also suppressed if `DOTNET_WATCH_SUPPRESS_LAUNCH_BROWSER` is set.

Browser refresh

`dotnet watch` injects a script into the app that allows it to refresh the browser when the content changes. In some scenarios, such as when the app enables response compression, `dotnet watch` might ***not*** be able to inject the script. For such cases in development, manually inject the script into the app. For example, to configure the web app to manually inject the script, update the layout file to include `_framework/aspnet-browser-refresh.js`:

razor

```razor
@* _Layout.cshtml *@
<environment names="Development">
    <script src="/_framework/aspnetcore-browser-refresh.js"></script>
</environment>
```

Non-ASCII characters

Visual Studio 17.2 and later includes the .NET SDK 6.0.300 and later. With the .NET SDK and 6.0.300 later, `dotnet-watch` emits non-ASCII characters to the console during a hot

reload session. On certain console hosts, such as the Windows conhost, these characters may appear garbled. To avoid garbled characters, consider one of the following approaches:

- Configure the `DOTNET_WATCH_SUPPRESS_EMOJIS=1` environment variable to suppress emitting these values.
- Switch to a different terminal, such as https://github.com/microsoft/terminal, that supports rendering non-ASCII characters.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Factory-based middleware activation in ASP.NET Core

Article • 07/26/2024

IMiddlewareFactory/IMiddleware is an extensibility point for middleware activation that offers the following benefits:

- Activation per client request (injection of scoped services)
- Strong typing of middleware

UseMiddleware extension methods check if a middleware's registered type implements IMiddleware. If it does, the IMiddlewareFactory instance registered in the container is used to resolve the IMiddleware implementation instead of using the convention-based middleware activation logic. The middleware is registered as a scoped or transient service in the app's service container.

IMiddleware is activated per client request (connection), so scoped services can be injected into the middleware's constructor.

IMiddleware

IMiddleware defines middleware for the app's request pipeline. The InvokeAsync(HttpContext, RequestDelegate) method handles requests and returns a Task that represents the execution of the middleware.

Middleware activated by convention:

C#

```
public class ConventionalMiddleware
{
    private readonly RequestDelegate _next;

    public ConventionalMiddleware(RequestDelegate next)
        => _next = next;

    public async Task InvokeAsync(HttpContext context, SampleDbContext
dbContext)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            dbContext.Requests.Add(new Request("Conventional", keyValue));
```

```
            await dbContext.SaveChangesAsync();
        }

        await _next(context);
    }
}
```

Middleware activated by MiddlewareFactory:

C#

```
public class FactoryActivatedMiddleware : IMiddleware
{
    private readonly SampleDbContext _dbContext;

    public FactoryActivatedMiddleware(SampleDbContext dbContext)
        => _dbContext = dbContext;

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _dbContext.Requests.Add(new Request("Factory", keyValue));

            await _dbContext.SaveChangesAsync();
        }

        await next(context);
    }
}
```

Extensions are created for the middleware:

C#

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseConventionalMiddleware(
        this IApplicationBuilder app)
        => app.UseMiddleware<ConventionalMiddleware>();

    public static IApplicationBuilder UseFactoryActivatedMiddleware(
        this IApplicationBuilder app)
        => app.UseMiddleware<FactoryActivatedMiddleware>();
}
```

It isn't possible to pass objects to the factory-activated middleware with UseMiddleware:

```
C#

public static IApplicationBuilder UseFactoryActivatedMiddleware(
    this IApplicationBuilder app, bool option)
{
    // Passing 'option' as an argument throws a NotSupportedException at
runtime.
    return app.UseMiddleware<FactoryActivatedMiddleware>(option);
}
```

The factory-activated middleware is added to the built-in container in `Program.cs`:

```
C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<SampleDbContext>
    (options => options.UseInMemoryDatabase("SampleDb"));

builder.Services.AddTransient<FactoryActivatedMiddleware>();
```

Both middleware are registered in the request processing pipeline, also in `Program.cs`:

```
C#

var app = builder.Build();

app.UseConventionalMiddleware();
app.UseFactoryActivatedMiddleware();
```

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware. The middleware factory implementation is registered in the container as a scoped service.

The default IMiddlewareFactory implementation, MiddlewareFactory, is found in the Microsoft.AspNetCore.Http package.

Additional resources

- View or download sample code (how to download)
- ASP.NET Core Middleware
- Middleware activation with a third-party container in ASP.NET Core

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Middleware activation with a third-party container in ASP.NET Core

Article • 07/26/2024

This article demonstrates how to use IMiddlewareFactory and IMiddleware as an extensibility point for middleware activation with a third-party container. For introductory information on `IMiddlewareFactory` and `IMiddleware`, see Factory-based middleware activation in ASP.NET Core.

View or download sample code (how to download)

The sample app demonstrates middleware activation by an `IMiddlewareFactory` implementation, `SimpleInjectorMiddlewareFactory`. The sample uses the Simple Injector dependency injection (DI) container.

The sample's middleware implementation records the value provided by a query string parameter (`key`). The middleware uses an injected database context (a scoped service) to record the query string value in an in-memory database.

> **Note**
>
> The sample app uses Simple Injector purely for demonstration purposes. Use of Simple Injector isn't an endorsement. Middleware activation approaches described in the Simple Injector documentation and GitHub issues are recommended by the maintainers of Simple Injector. For more information, see the Simple Injector documentation and Simple Injector GitHub repository.

IMiddlewareFactory

IMiddlewareFactory provides methods to create middleware.

In the sample app, a middleware factory is implemented to create a `SimpleInjectorActivatedMiddleware` instance. The middleware factory uses the Simple Injector container to resolve the middleware:

```
C#

public class SimpleInjectorMiddlewareFactory : IMiddlewareFactory
{
    private readonly Container _container;
```

```
    public SimpleInjectorMiddlewareFactory(Container container)
    {
        _container = container;
    }

    public IMiddleware Create(Type middlewareType)
    {
        return _container.GetInstance(middlewareType) as IMiddleware;
    }

    public void Release(IMiddleware middleware)
    {
        // The container is responsible for releasing resources.
    }
}
```

IMiddleware

IMiddleware defines middleware for the app's request pipeline.

Middleware activated by an `IMiddlewareFactory` implementation (`Middleware/SimpleInjectorActivatedMiddleware.cs`):

C#

```
public class SimpleInjectorActivatedMiddleware : IMiddleware
{
    private readonly AppDbContext _db;

    public SimpleInjectorActivatedMiddleware(AppDbContext db)
    {
        _db = db;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var keyValue = context.Request.Query["key"];

        if (!string.IsNullOrWhiteSpace(keyValue))
        {
            _db.Add(new Request()
                {
                    DT = DateTime.UtcNow,
                    MiddlewareActivation =
"SimpleInjectorActivatedMiddleware",
                    Value = keyValue
                });

            await _db.SaveChangesAsync();
        }
```

```
            await next(context);
        }
    }
```

An extension is created for the middleware (`Middleware/MiddlewareExtensions.cs`):

C#

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSimpleInjectorActivatedMiddleware(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<SimpleInjectorActivatedMiddleware>();
    }
}
```

`Startup.ConfigureServices` must perform several tasks:

- Set up the Simple Injector container.
- Register the factory and middleware.
- Make the app's database context available from the Simple Injector container.

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    // Replace the default middleware factory with the
    // SimpleInjectorMiddlewareFactory.
    services.AddTransient<IMiddlewareFactory>(_ =>
    {
        return new SimpleInjectorMiddlewareFactory(_container);
    });

    // Wrap ASP.NET Core requests in a Simple Injector execution
    // context.
    services.UseSimpleInjectorAspNetRequestScoping(_container);

    // Provide the database context from the Simple
    // Injector container whenever it's requested from
    // the default service container.
    services.AddScoped<AppDbContext>(provider =>
        _container.GetInstance<AppDbContext>());

    _container.Options.DefaultScopedLifestyle = new AsyncScopedLifestyle();

    _container.Register<AppDbContext>(() =>
    {
        var optionsBuilder = new DbContextOptionsBuilder<DbContext>();
```

```
            optionsBuilder.UseInMemoryDatabase("InMemoryDb");
            return new AppDbContext(optionsBuilder.Options);
        }, Lifestyle.Scoped);

        _container.Register<SimpleInjectorActivatedMiddleware>();

        _container.Verify();
    }
```

The middleware is registered in the request processing pipeline in `Startup.Configure`:

C#

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseSimpleInjectorActivatedMiddleware();

    app.UseStaticFiles();
    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Additional resources

- Middleware
- Factory-based middleware activation
- Simple Injector GitHub repository
- Simple Injector documentation

Collaborate with us on GitHub

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

feedback:

Open a documentation issue

Provide product feedback

Migrate from ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9

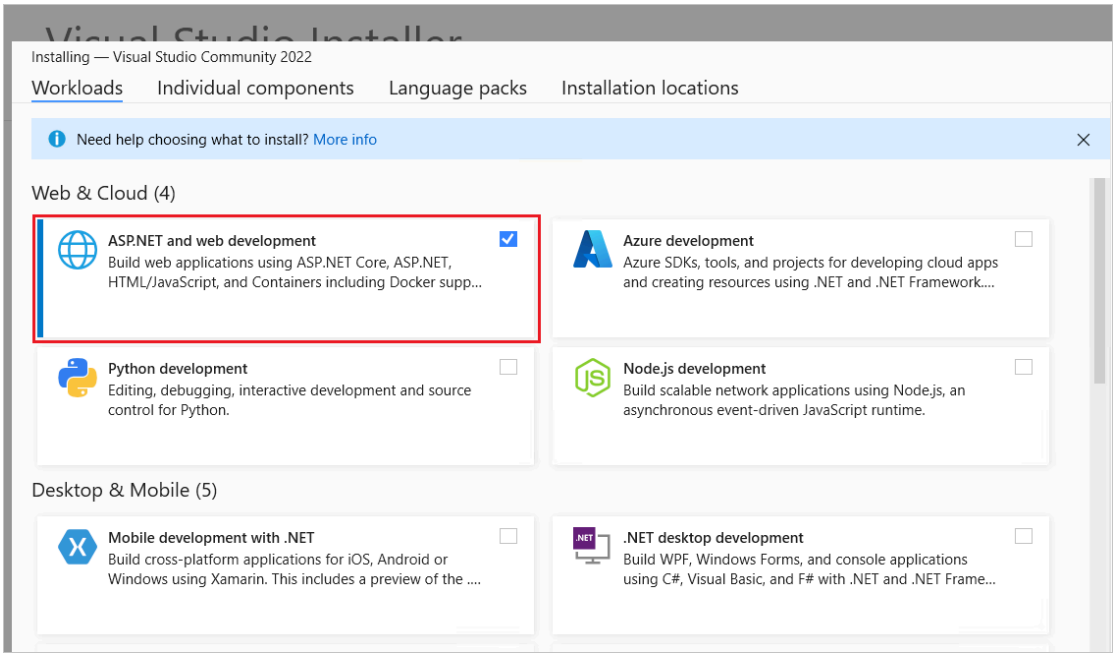
Article • 11/06/2024

This article explains how to update an ASP.NET Core in .NET 8 to ASP.NET Core in .NET 9.

Prerequisites

Visual Studio

- Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



Update the .NET SDK version in `global.json`

If you rely on a global.json file to target a specific .NET Core SDK version, update the `version` property to the .NET 9.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "8.0.100"
```

```
+    "version": "9.0.100"
  }
}
```

Update the target framework

Update the project file's Target Framework Moniker (TFM) to `net9.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>net8.0</TargetFramework>
+    <TargetFramework>net9.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Update package references

In the project file, update each Microsoft.AspNetCore.* , Microsoft.EntityFrameworkCore.* , Microsoft.Extensions.* , and System.Net.Http.Json package reference's `Version` attribute to 9.0.0 or later. For example:

diff

```
<ItemGroup>
-   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="8.0.2" />
-   <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="8.0.2" />
-   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="8.0.0" />
-   <PackageReference Include="System.Net.Http.Json" Version="8.0.0" />
+   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="9.0.0" />
+   <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="9.0.0" />
+   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="9.0.0" />
+   <PackageReference Include="System.Net.Http.Json" Version="9.0.0" />
</ItemGroup>
```

Replace `UseStaticFiles` with `MapStaticAssets`

Optimize the handling of static files in your web apps by replacing UseStaticFiles with MapStaticAssets in the app's `Program` file:

diff

```
- app.UseStaticFiles();
+ app.MapStaticAssets();
```

In MVC & Razor Pages apps you additionally need to chain a call to `.WithStaticAssets` after `MapRazorPages` or `MapControllerRoute` in `Program.cs`. For an example, see the Static files in ASP.NET Core.

ASP.NET Core automatically fingerprints and precompresses your static files at build and publish time, and then MapStaticAssets surfaces the optimized files as endpoints using endpoint routing with appropriate caching headers.

To resolve the fingerprinted file names from your app:

- In Blazor apps, use the ComponentBase.Assets property. Update explicit references to static assets in Razor component files (`.razor`) to use `@Assets["{ASSET PATH}"]`, where the `{ASSET PATH}` placeholder is the path to the asset. Note that this should ***NOT*** be done for the Blazor framework scripts (`blazor.*.js`). In the following example, Bootstrap, the Blazor project template app stylesheet (`app.css`), and the CSS isolation stylesheet (based on an app's namespace of `BlazorSample`) are linked in a root component, typically the `App` component (`Components/App.razor`):

  razor

  ```
  <link rel="stylesheet" href="@Assets["bootstrap/bootstrap.min.css"]" />
  <link rel="stylesheet" href="@Assets["app.css"]" />
  <link rel="stylesheet" href="@Assets["BlazorSample.styles.css"]" />
  ```

- In MVC & Razor Pages apps, the script and link tag helpers will automatically resolve the fingerprinted file names.

To resolve the fingerprinted file names when importing JavaScript modules, add a generated import map:

- In Blazor apps, add the (ImportMap) component to the `<head>` content of the app's root component, typically in the `App` component (`App.razor`):

  razor

```
<ImportMap />
```

- In MVC & Razor pages apps, add `<script type="importmap"></script>` to the head of the main layout file, which is updated by the Import Map Tag Helper.

For more information, see the following resources:

- What's new in ASP.NET Core 9.0
- ASP.NET Core Blazor static files

Blazor

Adopt simplified authentication state serialization for Blazor Web Apps

Blazor Web Apps can optionally adopt simplified authentication state serialization.

In the server project:

- Remove the Persisting Authentication State Provider (`PersistingAuthenticationStateProvider.cs`).

- Remove the service registration from the `Program` file. Instead, chain a call to AddAuthenticationStateSerialization on AddRazorComponents:

diff

```
- builder.Services.AddScoped<AuthenticationStateProvider,
PersistingAuthenticationStateProvider>();

  builder.Services.AddRazorComponents()
      .AddInteractiveServerComponents()
      .AddInteractiveWebAssemblyComponents()
+     .AddAuthenticationStateSerialization();
```

The API only serializes the server-side name and role claims for access in the browser. To include all claims, set SerializeAllClaims to `true`:

C#

```
.AddAuthenticationStateSerialization(options => options.SerializeAllClaims =
true);
```

In the client project (`.Client`):

- Remove the Persistent Authentication State Provider (`PersistentAuthenticationStateProvider.cs`).
- Remove the service registration from the `Program` file. Instead, call AddAuthenticationStateDeserialization on the service collection:

  diff

  ```
  - builder.Services.AddSingleton<AuthenticationStateProvider,
  PersistentAuthenticationStateProvider>();
  + builder.Services.AddAuthenticationStateDeserialization();
  ```

For more information, see What's new in ASP.NET Core 9.0.

Additional resources

Migrate from ASP.NET Core in .NET 7 to .NET 8

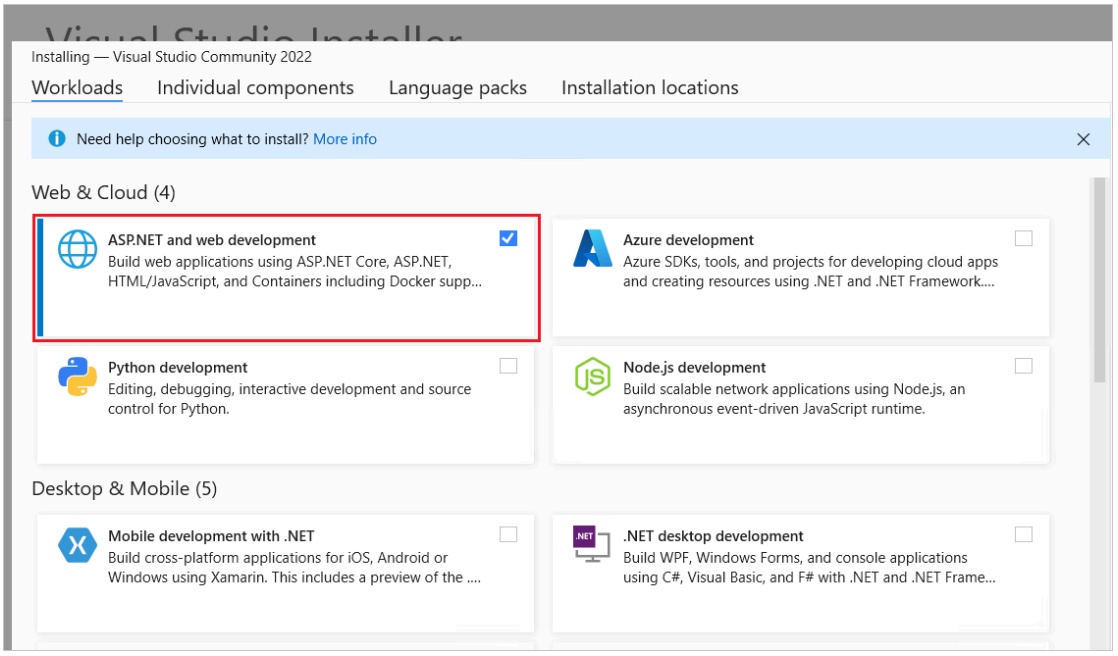
Article • 10/25/2024

This article explains how to update an existing ASP.NET Core 7.0 project to ASP.NET Core 8.0.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Update the .NET SDK version in `global.json`

If you rely on a global.json file to target a specific .NET Core SDK version, update the `version` property to the .NET 8.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "7.0.100"
+    "version": "8.0.100"
```

```
  }
}
```

Update the target framework

Update the project file's Target Framework Moniker (TFM) to `net8.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>net7.0</TargetFramework>
+    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Update package references

In the project file, update each Microsoft.AspNetCore.* , Microsoft.EntityFrameworkCore.* , Microsoft.Extensions.* , and System.Net.Http.Json package reference's `Version` attribute to 8.00 or later. For example:

diff

```
<ItemGroup>
-   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="7.0.12" />
-   <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="7.0.12" />
-   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="7.0.0" />
-   <PackageReference Include="System.Net.Http.Json" Version="7.0.1" />
+   <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="8.0.0" />
+   <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="8.0.0" />
+   <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="8.0.0" />
+   <PackageReference Include="System.Net.Http.Json" Version="8.0.0" />
</ItemGroup>
```

Blazor

The following migration scenarios are covered:

- Update a Blazor Server app
- Adopt all Blazor Web App conventions
- Convert a Blazor Server app into a Blazor Web App
- Update a Blazor WebAssembly app
- Convert a hosted Blazor WebAssembly app into a Blazor Web App
- Update service and endpoint option configuration
- Drop Blazor Server with Yarp routing workaround
- Migrate CascadingValue components in layout components
- Migrate the BlazorEnableCompression MSBuild property
- Migrate the <CascadingAuthenticationState> component to cascading authentication state services
- *New article*: HTTP caching issues during migration
- *New article*: New article on class libraries with static server-side rendering (static SSR)
- Discover components from additional assemblies
- Drop [Parameter] attribute when the parameter is supplied from a query string
- Blazor Server script fallback policy authorization

For guidance on adding Blazor support to an ASP.NET Core app, see Integrate ASP.NET Core Razor components with MVC or Razor Pages.

Update a Blazor Server app

We recommend using Blazor Web Apps in .NET 8, but Blazor Server is supported. To continue using Blazor Server with .NET 8, follow the guidance in the first three sections of this article:

- Update the .NET SDK version in global.json
- Update the target framework
- Update package references

New Blazor features introduced for Blazor Web Apps aren't available to a Blazor Server app updated to run under .NET 8. If you wish to adopt the new .NET 8 Blazor features, follow the guidance in either of the following sections:

- Adopt all Blazor Web App conventions
- Convert a Blazor Server app into a Blazor Web App

Adopt all Blazor Web App conventions

To optionally adopt all of the new Blazor Web App conventions, we recommend the following process:

- Create a new app from the Blazor Web App project template. For more information, see Tooling for ASP.NET Core Blazor.
- Move the your app's components and code to the new Blazor Web App, making modifications to adopt new features.
- Update the layout and styles of the Blazor Web App.

New .NET 8 features are covered in What's new in ASP.NET Core 8.0. When updating an app from .NET 6 or earlier, see the migration and release notes (*What's new* articles) for intervening releases.

Convert a Blazor Server app into a Blazor Web App

Blazor Server apps are supported in .NET 8 without any code changes. Use the following guidance to convert a Blazor Server app into an equivalent .NET 8 Blazor Web App, which makes all of the new .NET 8 features available.

> **ⓘ Important**
>
> This section focuses on the minimal changes required to convert a .NET 7 Blazor Server app into a .NET 8 Blazor Web App. To adopt all of the new Blazor Web App conventions, follow the guidance in the **Adopt all Blazor Web App conventions** section.

1. Follow the guidance in the first three sections of this article:
   - Update the .NET SDK version in global.json
   - Update the target framework
   - Update package references
2. Move the contents of the `App` component (`App.razor`) to a new `Routes` component file (`Routes.razor`) added to the project's root folder. Leave the empty `App.razor` file in the app in the project's root folder.
3. Add an entry to the `_Imports.razor` file to make shorthand render modes available to the app:

   razor

   ```
   @using static Microsoft.AspNetCore.Components.Web.RenderMode
   ```

4. Move the content in the `_Host` page (`Pages/_Host.cshtml`) to the empty `App.razor` file. Proceed to make the following changes to the `App` component.

> **Note**
>
> In the following example, the project's namespace is `BlazorServerApp`. Adjust the namespace to match your project.

Remove the following lines from the top of the file:

```
diff
- @page "/"
- @using Microsoft.AspNetCore.Components.Web
- @namespace BlazorServerApp.Pages
- @addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Replace the preceding lines with a line that injects an IHostEnvironment instance:

```
razor
@inject IHostEnvironment Env
```

Remove the tilde (`~`) from the `href` of the `<base>` tag and replace with the base path for your app:

```
diff
- <base href="~/" />
+ <base href="/" />
```

Remove the Component Tag Helper for the HeadOutlet component and replace it with the HeadOutlet component.

Remove the following line:

```
diff
- <component type="typeof(HeadOutlet)" render-mode="ServerPrerendered"
/>
```

Replace the preceding line with the following:

```
```

razor

```razor
<HeadOutlet @rendermode="InteractiveServer" />
```

Remove the Component Tag Helper for the `App` component and replace it with the `Routes` component.

Remove the following line:

diff

```diff
- <component type="typeof(App)" render-mode="ServerPrerendered" />
```

Replace the preceding line with the following:

razor

```razor
<Routes @rendermode="InteractiveServer" />
```

> **Note**
>
> The preceding configuration assumes that the app's components adopt interactive server rendering. For more information, including how to adopt static server-side rendering (SSR), see ASP.NET Core Blazor render modes.

Remove the Environment Tag Helpers for error UI and replace them with the following Razor markup.

Remove the following lines:

diff

```diff
- <environment include="Staging,Production">
-     An error has occurred. This application may no longer respond
until reloaded.
- </environment>
- <environment include="Development">
-     An unhandled exception has occurred. See browser dev tools for
details.
- </environment>
```

Replace the preceding lines with the following:

razor

```
@if (Env.IsDevelopment())
{
    <text>
        An unhandled exception has occurred. See browser dev tools for
details.
    </text>
}
else
{
    <text>
        An error has occurred. This app may no longer respond until
reloaded.
    </text>
}
```

Change the Blazor script from `blazor.server.js` to `blazor.web.js`:

diff

```
- <script src="_framework/blazor.server.js"></script>
+ <script src="_framework/blazor.web.js"></script>
```

5. Delete the `Pages/_Host.cshtml` file.

6. Update `Program.cs`:

> **Note**
>
> In the following example, the project's namespace is `BlazorServerApp`. Adjust the namespace to match your project.

Add a `using` statement to the top of the file for the project's namespace:

C#

```
using BlazorServerApp;
```

Replace AddServerSideBlazor with AddRazorComponents and a chained call to AddInteractiveServerComponents.

Remove the following line:

diff

```
- builder.Services.AddServerSideBlazor();
```

Replace the preceding line with Razor component and interactive server component services. Calling AddRazorComponents adds antiforgery services (AddAntiforgery) by default.

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
```

Remove the following line:

diff

```
- app.MapBlazorHub();
```

Replace the preceding line with a call to MapRazorComponents, supplying the `App` component as the root component type, and add a chained call to AddInteractiveServerRenderMode:

C#

```
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

Remove the following line:

diff

```
- app.MapFallbackToPage("/_Host");
```

Remove Routing Middleware:

diff

```
- app.UseRouting();
```

Add Antiforgery Middleware to the request processing pipeline after the line that adds HTTPS Redirection Middleware (`app.UseHttpsRedirection`):

C#

```
app.UseAntiforgery();
```

The preceding call to `app.UseAntiforgery` must be placed after calls, if present, to `app.UseAuthentication` and `app.UseAuthorization`. There's no need to explicitly add antiforgery services (`builder.Services.AddAntiforgery`), as they're added automatically by AddRazorComponents, which was covered earlier.

7. If the Blazor Server app was configured to disable prerendering, you can continue to disable prerendering for the updated app. In the `App` component, change the value assigned to the `@rendermode` Razor directive attributes for the HeadOutlet and `Routes` components.

   Change the value of the `@rendermode` directive attribute for both the HeadOutlet and `Routes` components to disable prerendering:

   ```
   diff
   ```

   ```
   - @rendermode="InteractiveServer"
   + @rendermode="new InteractiveServerRenderMode(prerender: false)"
   ```

   For more information, see ASP.NET Core Blazor render modes.

Update a Blazor WebAssembly app

Follow the guidance in the first three sections of this article:

- Update the .NET SDK version in global.json
- Update the target framework
- Update package references

For apps that adopt lazy assembly loading, change the file extension from `.dll` to `.wasm` in the app's implementation to reflect Blazor WebAssembly's adoption of Webcil assembly packaging.

Prior to the release of .NET 8, guidance in Deployment layout for ASP.NET Core hosted Blazor WebAssembly apps addresses environments that block clients from downloading and executing DLLs with a multipart bundling approach. In .NET 8 or later, Blazor uses the Webcil file format to address this problem. Multipart bundling using the experimental NuGet package described by the *WebAssembly deployment layout* article isn't supported for Blazor apps in .NET 8 or later. If you desire to continue using the multipart bundle package in .NET 8 or later apps, you can use the guidance in the article

to create your own multipart bundling NuGet package, but it won't be supported by Microsoft.

Convert a hosted Blazor WebAssembly app into a Blazor Web App

Blazor WebAssembly apps are supported in .NET 8 without any code changes. Use the following guidance to convert an ASP.NET Core hosted Blazor WebAssembly app into an equivalent .NET 8 Blazor Web App, which makes all of the new .NET 8 features available.

> **ⓘ Important**
>
> This section focuses on the minimal changes required to convert a .NET 7 ASP.NET Core hosted Blazor WebAssembly app into a .NET 8 Blazor Web App. To adopt all of the new Blazor Web App conventions, follow the guidance in the **Adopt all Blazor Web App conventions** section.

1. Follow the guidance in the first three sections of this article:

   - Update the .NET SDK version in global.json
   - Update the target framework
   - Update package references

   > **ⓘ Important**
   >
   > Using the preceding guidance, update the `.Client`, `.Server`, and `.Shared` projects of the solution.

2. In the `.Client` project file (`.csproj`), add the following MSBuild properties:

   XML

   ```xml
   <NoDefaultLaunchSettingsFile>true</NoDefaultLaunchSettingsFile>
   <StaticWebAssetProjectMode>Default</StaticWebAssetProjectMode>
   ```

   Also in the `.Client` project file, remove the Microsoft.AspNetCore.Components.WebAssembly.DevServer package reference:

   diff

   ```
   - <PackageReference
   ```

```
Include="Microsoft.AspNetCore.Components.WebAssembly.DevServer"... />
```

3. Move the file content from the `.Client/wwwroot/index.html` file to a new `App` component file (`App.razor`) created at the root of the `.Server` project. After you move the file's contents, delete the `index.html` file.

   Rename `App.razor` in the `.Client` project to `Routes.razor`.

   In `Routes.razor`, update the value of the `AppAssembly` attribute to `typeof(Program).Assembly`.

4. In the `.Client` project, add an entry to the `_Imports.razor` file to make shorthand render modes available to the app:

   ```razor
   @using static Microsoft.AspNetCore.Components.Web.RenderMode
   ```

   Make a copy of the `.Client` project's `_Imports.razor` file and add it to the `.Server` project.

5. Make the following changes to the `App.razor` file:

   Replace the website's default website title (`<title>...</title>`) with a HeadOutlet component. Note the website title for use later and remove the title tags and title:

   ```diff
   - <title>...</title>
   ```

   Where you removed the title, place a HeadOutlet component assigning the Interactive WebAssembly render mode (prerendering disabled):

   ```razor
   <HeadOutlet @rendermode="new
   InteractiveWebAssemblyRenderMode(prerender: false)" />
   ```

   Change the CSS style bundle:

   ```diff
   - <link href="{CLIENT PROJECT ASSEMBLY NAME}.styles.css"
   rel="stylesheet">
   ```

```
+ <link href="{SERVER PROJECT ASSEMBLY NAME}.styles.css"
rel="stylesheet">
```

Placeholders in the preceding code:

- `{CLIENT PROJECT ASSEMBLY NAME}`: Client project assembly name. Example: `BlazorSample.Client`
- `{SERVER PROJECT ASSEMBLY NAME}`: Server project assembly name. Example: `BlazorSample.Server`

Locate following `<div>...</div>` HTML markup:

diff

```
- <div id="app">
-     ...
- </div>
```

Replace the preceding `<div>...</div>` HTML markup with the `Routes` component using the Interactive WebAssembly render mode (prerendering disabled):

razor

```
<Routes @rendermode="new InteractiveWebAssemblyRenderMode(prerender:
false)" />
```

Update the `blazor.webassembly.js` script to `blazor.web.js`:

diff

```
- <script src="_framework/blazor.webassembly.js"></script>
+ <script src="_framework/blazor.web.js"></script>
```

6. Open the `.Client` project's layout file (`.Client/Shared/MainLayout.razor`) and add a PageTitle component with the website's default title (`{TITLE}` placeholder):

razor

```
<PageTitle>{TITLE}</PageTitle>
```

> ⓘ **Note**

> Other layout files should also receive a PageTitle component with the default website title.
>
> For more information, see Control head content in ASP.NET Core Blazor apps.

7. Remove the following lines from `.Client/Program.cs`:

```diff
- builder.RootComponents.Add<App>("#app");
- builder.RootComponents.Add<HeadOutlet>("head::after");
```

8. Update `.Server/Program.cs`:

   Add Razor component and interactive WebAssembly component services to the project. Call AddRazorComponents with a chained call to AddInteractiveWebAssemblyComponents. Calling AddRazorComponents adds antiforgery services (AddAntiforgery) by default.

```C#
builder.Services.AddRazorComponents()
    .AddInteractiveWebAssemblyComponents();
```

   Add Antiforgery Middleware to the request processing pipeline.

   Place the following line after the call to `app.UseHttpsRedirection`. The call to `app.UseAntiforgery` must be placed after calls, if present, to `app.UseAuthentication` and `app.UseAuthorization`. There's no need to explicitly add antiforgery services (`builder.Services.AddAntiforgery`), as they're added automatically by AddRazorComponents, which was covered earlier.

```C#
app.UseAntiforgery();
```

   Remove the following line:

```diff
- app.UseBlazorFrameworkFiles();
```

Remove the following line:

```diff
- app.MapFallbackToFile("index.html");
```

Replace the preceding line with a call to MapRazorComponents, supplying the `App` component as the root component type, and add chained calls to AddInteractiveWebAssemblyRenderMode and AddAdditionalAssemblies:

```C#
app.MapRazorComponents<App>()
    .AddInteractiveWebAssemblyRenderMode()
    .AddAdditionalAssemblies(typeof({CLIENT APP
NAMESPACE}._Imports).Assembly);
```

In the preceding example, the `{CLIENT APP NAMESPACE}` placeholder is the namespace of the `.Client` project (for example, `HostedBlazorApp.Client`).

9. Run the solution from the `.Server` project:

   For Visual Studio, confirm that the `.Server` project is selected in **Solution Explorer** when running the app.

   If using the .NET CLI, run the project from the `.Server` project's folder.

Update service and endpoint option configuration

With the release of Blazor Web Apps in .NET 8, Blazor service and endpoint option configuration is updated with the introduction of new API for interactive component services and component endpoint configuration.

Updated configuration guidance appears in the following locations:

- Setting and reading the app's environment: Contains updated guidance, especially in the section titled *Read the environment client-side in a Blazor Web App*.
- Server-side circuit handler options: Covers new Blazor-SignalR circuit and hub options configuration.
- Render Razor components from JavaScript: Covers dynamic component registration with RegisterForJavaScript.
- Blazor custom elements: Blazor Web App registration: Covers root component custom element registration with `RegisterCustomElement`.

- Prefix for Blazor WebAssembly assets: Covers control of the path string that indicates the prefix for Blazor WebAssembly assets.
- Temporary redirection URL validity duration: Covers control of the lifetime of data protection validity for temporary redirection URLs emitted by Blazor server-side rendering.
- Detailed errors: Covers enabling detailed errors for Razor component server-side rendering.
- Prerendering configuration: Prerendering is enabled by default for Blazor Web Apps. Follow this link for guidance on how to disable prerendering if you have special circumstances that require an app to disable prerendering.
- Form binding options: Covers form binding options configuration.

Drop Blazor Server with Yarp routing workaround

If you previously followed the guidance in Enable ASP.NET Core Blazor Server support with Yarp in incremental migration for migrating a Blazor Server app with Yarp to .NET 6 or .NET 7, you can reverse the workaround steps that you took when following the article's guidance. Routing and deep linking for Blazor Server with Yarp work correctly in .NET 8.

Migrate `CascadingValue` components in layout components

Cascading parameters don't pass data across render mode boundaries, and layouts are statically rendered in otherwise interactive apps. Therefore, apps that seek to use cascading parameters in interactively rendered components won't be able to cascade the values from a layout.

The two approaches for migration are:

- (*Recommended*) Pass the state as a root-level cascading value. For more information, see Root-level cascading values.
- Wrap the router in the `Routes` component with the CascadingValue component and make the `Routes` component interactively rendered. For an example, see CascadingValue component.

For more information, see Cascading values/parameters and render mode boundaries.

Migrate the `BlazorEnableCompression` MSBuild property

For Blazor WebAssembly apps that disable compression and target .NET 7 or earlier but are built with the .NET 8 SDK, the `BlazorEnableCompression` MSBuild property has changed to `CompressionEnabled`:

diff

```
<PropertyGroup>
-   <BlazorEnableCompression>false</BlazorEnableCompression>
+   <CompressionEnabled>false</CompressionEnabled>
</PropertyGroup>
```

When using the .NET CLI publish command, use the new property:

.NET CLI

```
dotnet publish -p:CompressionEnabled=false
```

For more information, see the following resources:

- Static Web Assets Compression Flag Breaking Change (dotnet/announcements #283)
- Host and deploy ASP.NET Core Blazor WebAssembly

Migrate the `<CascadingAuthenticationState>` component to cascading authentication state services

In .NET 7 or earlier, the CascadingAuthenticationState component is wrapped around some part of the UI tree, for example around the Blazor router, to provide cascading authentication state:

razor

```
<CascadingAuthenticationState>
    <Router ...>
        ...
    </Router>
</CascadingAuthenticationState>
```

In .NET 8, don't use the CascadingAuthenticationState component:

diff

```
- <CascadingAuthenticationState>
      <Router ...>
          ...
```

```
        </Router>
- </CascadingAuthenticationState>
```

Instead, add cascading authentication state services to the service collection by calling AddCascadingAuthenticationState in the `Program` file:

```
C#

builder.Services.AddCascadingAuthenticationState();
```

For more information, see the following resources:

- *ASP.NET Core Blazor authentication and authorization* article
  - AuthenticationStateProvider service
  - Expose the authentication state as a cascading parameter
  - Customize unauthorized content with the Router component
- ASP.NET Core Blazor authentication and authorization

New article on HTTP caching issues

We've added a new article that discusses some of the common HTTP caching issues that can occur when upgrading Blazor apps across major versions and how to address HTTP caching issues.

For more information, see Avoid HTTP caching issues when upgrading ASP.NET Core Blazor apps.

New article on class libraries with static server-side rendering (static SSR)

We've added a new article that discusses component library authorship in Razor class libraries (RCLs) with static server-side rendering (static SSR).

For more information, see ASP.NET Core Razor class libraries (RCLs) with static server-side rendering (static SSR).

Discover components from additional assemblies

When migrating from a Blazor Server app to a Blazor Web App, access the guidance in ASP.NET Core Blazor routing and navigation if the app uses routable components from additional assemblies, such as component class libraries.

Drop `[Parameter]` attribute when the parameter is supplied from a query string

The `[Parameter]` attribute is no longer required when supplying a parameter from the query string:

```diff
- [Parameter]
  [SupplyParameterFromQuery]
```

Blazor Server script fallback policy authorization

In .NET 7, the Blazor Server script (`blazor.server.js`) is served by Static Files Middleware. Placing the call for Static Files Middleware (UseStaticFiles) in the request processing pipeline before the call to Authorization Middleware (UseAuthorization) is sufficient in .NET 7 apps to serve the Blazor script to anonymous users.

In .NET 8, the Blazor Server script is served by its own endpoint, using endpoint routing. This change is introduced by Fixed bug - Passing options to UseStaticFiles breaks Blazor Server (dotnet/aspnetcore #45897).

Consider a multi-tenant scenario where:

- Both the default and fallback policies are set identically.
- The tenant is resolved using the first segment in the request path (for example, `tld.com/tenant-name/...`).
- The requests to tenant endpoints are authenticated by an additional authentication scheme, which adds an additional identity to the request principal.
- The fallback authorization policy has requirements that check claims via the additional identity.

Requests for the Blazor script file (`blazor.server.js`) are served at `/_framework/blazor.server.js`, which is hardcoded in the framework. Requests for the file aren't authenticated by the additional authentication scheme for tenants ***but are still challenged by the fallback policy***, which results in returning an unauthorized result.

This problem is under evaluation for a new framework feature in MapRazorComponents broken with FallbackPolicy RequireAuthenticatedUser (dotnet/aspnetcore 51836), which is currently scheduled for .NET 9's release in November, 2024. Until then, you can work around this problem using any of the following three approaches:

- Don't use a fallback policy. Apply the `[Authorize]` attribute in the `_Imports.razor` file to apply it to all of the components of the app. For non-blazor endpoints, explicitly use `[Authorize]` or `RequireAuthorization`.

- Add `[AllowAnonymous]` to the `/_framework/blazor.server.js` endpoint in the `Program` file:

  C#

  ```
  app.MapBlazorHub().Add(endpointBuilder =>
  {
      if (endpointBuilder is
          RouteEndpointBuilder
          {
              RoutePattern: { RawText: "/_framework/blazor.server.js" }
          })
      {
          endpointBuilder.Metadata.Add(new AllowAnonymousAttribute());
      }
  });
  ```

- Register a custom `AuthorizationHandler` that checks the HttpContext to allow the `/_framework/blazor.server.js` file through.

Docker

Update Docker images

For apps using Docker, update the *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 8.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 7.0 and 8.0:

diff

```
- docker pull mcr.microsoft.com/dotnet/aspnet:7.0
+ docker pull mcr.microsoft.com/dotnet/aspnet:8.0
```

Update Docker port

The default ASP.NET Core port configured in .NET container images has been updated from port 80 to 8080.

The new `ASPNETCORE_HTTP_PORTS` environment variable was added as a simpler alternative to `ASPNETCORE_URLS`.

For more information, see:

- Default ASP.NET Core port changed from 80 to 8080.
- Specify ports only with ASPNETCORE_HTTP_PORTS

Review breaking changes

For breaking changes from .NET Core .NET 7.0 to 8.0, see Breaking changes in .NET 8, which includes ASP.NET Core and Entity Framework Core sections.

Migrate from ASP.NET Core 6.0 to 7.0

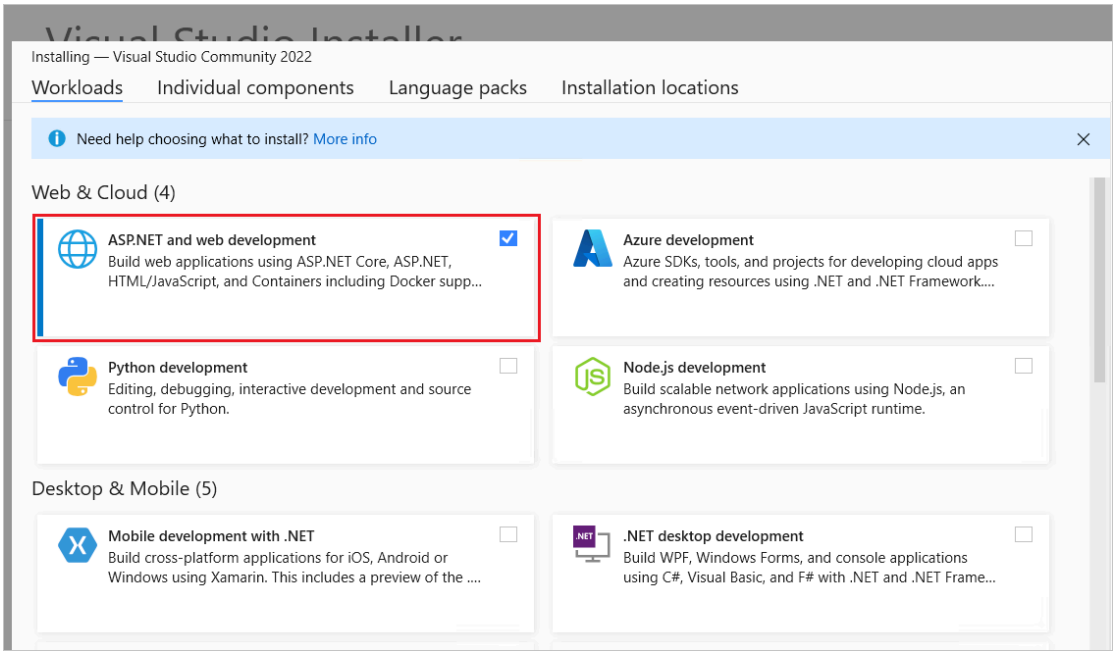
Article • 11/16/2024

This article explains how to update an existing ASP.NET Core 6.0 project to ASP.NET Core 7.0.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.



Update .NET Core SDK version in global.json

If you rely on a global.json file to target a specific .NET Core SDK version, update the `version` property to the .NET 7.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "6.0.200"
+    "version": "7.0.100"
```

```
  }
}
```

Update the target framework

Update the project file's Target Framework Moniker (TFM) to `net7.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>net6.0</TargetFramework>
+    <TargetFramework>net7.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Update package references

In the project file, update each Microsoft.AspNetCore.* , Microsoft.EntityFrameworkCore.* , Microsoft.Extensions.* , and System.Net.Http.Json package reference's `Version` attribute to 7.0.0 or later. For example:

diff

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="6.0.9" />
-    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="6.0.9" />
-    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="6.0.9" />
-    <PackageReference Include="System.Net.Http.Json" Version="6.0.0" />
+    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="7.0.0" />
+    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="7.0.0" />
+    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="7.0.0" />
+    <PackageReference Include="System.Net.Http.Json" Version="7.0.0" />
</ItemGroup>
```

Blazor

Adopt .NET 7 features

After following the guidance earlier in this article to update an app to 7.0, adopt specific features by following the links in What's new in ASP.NET Core 7.0.

To adopt all of the new 7.0 features for Blazor apps, we recommend the following process:

- Create a new 7.0 Blazor project from one of the Blazor project templates. For more information, see Tooling for ASP.NET Core Blazor.
- Move the app's components and code to the 7.0 app making modifications to adopt the new 7.0 features.

Simplify component parameter binding

In prior Blazor releases, binding across multiple components required binding to properties with `get`/`set` accessors.

In .NET 6 and earlier:

razor

```
<NestedGrandchild @bind-GrandchildMessage="BoundValue" />

@code {
    ...

    private string BoundValue
    {
        get => ChildMessage ?? string.Empty;
        set => ChildMessageChanged.InvokeAsync(value);
    }
}
```

In .NET 7, you can use the new `@bind:get` and `@bind:set` modifiers to support two-way data binding and simplify the binding syntax:

razor

```
<NestedGrandchild @bind-GrandchildMessage:get="ChildMessage"
    @bind-GrandchildMessage:set="ChildMessageChanged" />
```

For more information, see the following content in the *Data binding* article:

- Introduction
- Bind across more than two components

Migrate unmarshalled JavaScript interop

Unmarshalled interop using the IJSUnmarshalledRuntime interface is obsolete and should be replaced with JavaScript `[JSImport]`/`[JSExport]` interop.

For more information, see JavaScript JSImport/JSExport interop with ASP.NET Core Blazor.

Blazor WebAssembly authentication uses history state for redirects

The support for authentication in Blazor WebAssembly apps changed to rely on navigation history state instead of query strings in the URL. As a result, passing the return URL through the query string fails to redirect back to the original page after a successful login in .NET 7.

The following example demonstrates **the prior redirection approach for apps that target .NET 6 or earlier** in `RedirectToLogin.razor`, which is based on a redirect URL (`?returnUrl=`) with NavigateTo:

razor

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code {
    protected override void OnInitialized()
    {
        Navigation.NavigateTo(
            $"authentication/login?returnUrl=
{Uri.EscapeDataString(Navigation.Uri)}");
    }
}
```

The following example demonstrates **the new redirection approach for apps that target .NET 7 or later** in `RedirectToLogin.razor`, which is based on navigation history state with NavigateToLogin:

razor

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Options

@inject
IOptionsSnapshot<RemoteAuthenticationOptions<ApiAuthorizationProviderOptions
>> OptionsSnapshot
@code {
    protected override void OnInitialized()
    {

Navigation.NavigateToLogin(OptionsSnapshot.Get(Options.DefaultName).Authenti
cationPaths.LogInPath);
    }
}
```

As part of this change, SignOutSessionStateManager is obsolete in .NET 7 or later and replaced with NavigateToLogout.

The following example demonstrates **the prior approach** in `Shared/LoginDisplay.razor` of an app generated from the Blazor WebAssembly project template:

razor

```
@inject SignOutSessionStateManager SignOutManager

...

@code{
    private async Task BeginLogout(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }
}
```

The following example demonstrates **the new approach** in `Shared/LoginDisplay.razor` that calls NavigateToLogout. The injection (`@inject`) of the SignOutSessionStateManager is removed from the component's directives at the top of the file, and the `BeginLogOut` method is updated to the following code:

razor

```
@code{
    public void BeginLogOut()
    {
        Navigation.NavigateToLogout("authentication/logout");
```

```
    }
}
```

The SignOutSessionStateManager service registration is removed in `Program.cs`:

```diff
- builder.Services.AddScoped<SignOutSessionStateManager>();
```

For more information, see the following resources:

- [Breaking change]: Updates to Authentication in webassembly applications
- ASP.NET Core Blazor routing and navigation
- Secure ASP.NET Core Blazor WebAssembly
- ASP.NET Core Blazor authentication and authorization

.NET WebAssembly build tools for .NET 6 projects

You can now use the .NET WebAssembly build tools with a .NET 6 project when working with the .NET 7 SDK. The new `wasm-tools-net6` workload includes the .NET WebAssembly build tools for .NET 6 projects so that they can be used with the .NET 7 SDK. The existing `wasm-tools` workload installs the .NET WebAssembly build tools for .NET 7 projects. However, the .NET 7 version of the .NET WebAssembly build tools are incompatible with existing projects built with .NET 6. Projects using the .NET WebAssembly build tools that need to support both .NET 6 and .NET 7 must use multi-targeting.

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 7.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 6.0 and 7.0:

```diff
- docker pull mcr.microsoft.com/dotnet/aspnet:6.0
+ docker pull mcr.microsoft.com/dotnet/aspnet:7.0
```

Review breaking changes

For breaking changes from .NET Core 6.0 to .NET 7.0, see Breaking changes in .NET 7. ASP.NET Core and Entity Framework Core are included in the list.

Migrate from ASP.NET Core 5.0 to 6.0

Article • 09/17/2023

This article explains how to update an existing ASP.NET Core 5.0 project to ASP.NET Core 6.0. For instructions on how to migrate from ASP.NET Core 3.1 to ASP.NET Core 6.0, see Migrate from ASP.NET Core 3.1 to 6.0.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

Update .NET SDK version in global.json

If you rely upon a global.json file to target a specific .NET SDK version, update the `version` property to the .NET 6.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "5.0.100"
+    "version": "6.0.100"
  }
}
```

Update the target framework

Update the project file's Target Framework Moniker (TFM) to `net6.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>net5.0</TargetFramework>
+    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
```

```
</Project>
```

Update package references

In the project file, update each Microsoft.AspNetCore.* and Microsoft.Extensions.* package reference's `Version` attribute to 6.0.0 or later. For example:

diff

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="5.0.3" />
-    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="5.0.0" />
+    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="6.0.0" />
+    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="6.0.0" />
</ItemGroup>
```

New hosting model

The new .NET 6 minimal hosting model for ASP.NET Core apps requires only one file and a few lines of code. **Apps migrating to 6.0 don't need to use the new minimal hosting model.** For more information, see Apps migrating to 6.0 don't need to use the new minimal hosting model in the following section.

The following code from the ASP.NET Core empty template creates an app using the new minimal hosting model:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

The minimal hosting model:

- Significantly reduces the number of files and lines of code required to create an app. Only one file is needed with four lines of code.

- Unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file.
- Uses top-level statements to minimize the code required for an app.
- Uses global using directives to eliminate or minimize the number of using statement lines required.

The following code displays the `Startup.cs` and `Program.cs` files from an ASP.NET Core 5 Web App template (Razor Pages) with unused `using` statements removed:

C#

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
// Unused usings removed.

namespace WebAppRPv5
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add
services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddRazorPages();
        }

        // This method gets called by the runtime. Use this method to
configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                // The default HSTS value is 30 days. You may want to change
this for production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
```

```
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapRazorPages();
            });
        }
    }
}
```

C#

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
// Unused usings removed.

namespace WebAppRPv5
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
    }
}
```

In ASP.NET Core 6, the preceding code is replaced by the following:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
```

```
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

The preceding ASP.NET Core 6 sample shows how:

- ConfigureServices is replaced with WebApplication.Services.
- `builder.Build()` returns a configured WebApplication to the variable `app`. Configure is replaced with configuration calls to same services using `app`.

Detailed examples of migrating ASP.NET Core 5 `Startup` code to ASP.NET Core 6 using the minimal hosting model are provided later in this document.

There are a few changes to the other files generated for the Web App template:

- `Index.cshtml` and `Privacy.cshtml` have the unused `using` statements removed.
- `RequestId` in `Error.cshtml` is declared as a nullable reference type (NRT):

diff

```
- public string RequestId { get; set; }
+ public string? RequestId { get; set; }
```

- Log level defaults have changed in `appsettings.json` and `appsettings.Development.json`:

diff

```
- "Microsoft": "Warning",
- "Microsoft.Hosting.Lifetime": "Information"
+ "Microsoft.AspNetCore": "Warning"
```

In the preceding ASP.NET Core template code, `"Microsoft": "Warning"` has been changed to `"Microsoft.AspNetCore": "Warning"`. This change results in logging all informational messages from the `Microsoft` namespace ***except*** `Microsoft.AspNetCore`. For example, `Microsoft.EntityFrameworkCore` is now logged at the informational level.

For more details on the new hosting model, see the Frequently asked questions section. For more information on the adoption of NRTs and .NET compiler null-state analysis, see the Nullable reference types (NRTs) and .NET compiler null-state static analysis section.

Apps migrating to or using 6.0 and later don't need to use the new minimal hosting model

Using Startup and the Generic Host used by the ASP.NET Core 3.1 and 5.0 templates is fully supported.

Use Startup with the new minimal hosting model

ASP.NET Core 3.1 and 5.0 apps can use their `Startup` code with the new minimal hosting model. Using `Startup` with the minimal hosting model has the following advantages:

- No hidden reflection is used to call the `Startup` class.
- Asynchronous code can be written because the developer controls the call to `Startup`.
- Code can be written that interleaves `ConfigureServices` and `Configure`.

One minor limitation in using `Startup` code with the new minimal hosting model is that to inject a dependency into `Configure`, the service in `Program.cs` must be manually resolved.

Consider the following code generated by the ASP.NET Core 3.1 or 5.0 Razor Pages template:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
```

```
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

C#

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

The preceding code migrated to the new minimal hosting model:

C#

```
using Microsoft.AspNetCore.Builder;

var builder = WebApplication.CreateBuilder(args);
```

```
var startup = new Startup(builder.Configuration);

startup.ConfigureServices(builder.Services);

var app = builder.Build();

startup.Configure(app, app.Environment);

app.Run();
```

C#

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (!env.IsDevelopment())
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

In the preceding code, the `if (env.IsDevelopment())` block is removed because in development mode, the developer exception page middleware is enabled by default. For more information, see Differences between the ASP.NET Core 5 and 6 hosting models in the next section.

When using a custom dependency injection (DI) container, add the following highlighted code:

C#

```
using Autofac;
using Autofac.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Hosting;

var builder = WebApplication.CreateBuilder(args);

var startup = new Startup(builder.Configuration);

startup.ConfigureServices(builder.Services);

// Using a custom DI container.
builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());
builder.Host.ConfigureContainer<ContainerBuilder>
(startup.ConfigureContainer);

var app = builder.Build();

startup.Configure(app, app.Environment);

app.Run();
```

C#

```
using Autofac;
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    //  Using a custom DI container
    public void ConfigureContainer(ContainerBuilder builder)
    {
        // Configure custom container.
    }
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (!env.IsDevelopment())
```

```
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

When using the minimal hosting model, the endpoint routing middleware wraps the entire middleware pipeline, therefore there's no need to have explicit calls to `UseRouting` or `UseEndpoints` to register routes. `UseRouting` can still be used to specify where route matching happens, but `UseRouting` doesn't need to be explicitly called if routes should be matched at the beginning of the middleware pipeline.

In the following code, the calls to `UseRouting` and `UseEndpoints` are removed from `Startup`. `MapRazorPages` is called in `Program.cs`:

C#

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (!env.IsDevelopment())
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
```

```
        //app.UseRouting();

        //app.UseEndpoints(endpoints =>
        //{
        //    endpoints.MapRazorPages();
        //});
    }
}
```

C#

```
using Microsoft.AspNetCore.Builder;

var builder = WebApplication.CreateBuilder(args);

var startup = new Startup(builder.Configuration);

startup.ConfigureServices(builder.Services);

var app = builder.Build();

startup.Configure(app, app.Environment);

app.MapRazorPages();

app.Run();
```

When using `Startup` with the new minimal hosting model, keep in mind the following difference:

- `Program.cs` controls the instantiation and lifetime of the `Startup` class.
- Any additional services injected into the `Configure` method need to be manually resolved by the `Program` class.

Differences between the ASP.NET Core 5 and 6 hosting models

- In development mode, the developer exception page middleware is enabled by default.
- The app name defaults to the entry point assembly's name: `Assembly.GetEntryAssembly().GetName().FullName`. When using the WebApplicationBuilder in a library, explicitly change the app name to the library's assembly to allow MVC's application part discovery to work. See Change the content root, app name, and environment in this document for detailed instructions.

- The endpoint routing middleware wraps the entire middleware pipeline, therefore there's no need to have explicit calls to `UseRouting` or `UseEndpoints` to register routes. `UseRouting` can still be used to specify where route matching happens, but `UseRouting` doesn't need to be explicitly called if routes should be matched at the beginning of the middleware pipeline.
- The pipeline is created before any IStartupFilter runs, therefore exceptions caused while building the pipeline aren't visible to the `IStartupFilter` call chain.
- Some tools, such as EF migrations, use Program.CreateHostBuilder to access the app's `IServiceProvider` to execute custom logic in the context of the app. These tools have been updated to use a new technique to execute custom logic in the context of the app. Entity Framework Migrations is an example of a tool that uses `Program.CreateHostBuilder` in this way. We're working to make sure tools are updated to use the new model.
- Unlike the `Startup` class, the minimal host doesn't automatically configure a DI scope when instantiating the service provider. For contexts where a scope is required, it is necessary to invoke IServiceScope with IServiceScopeFactory.CreateScope to instantiate a new scope. For more information, see how to resolve a service at app startup.
- It's ***not*** possible to change any host settings such as app name, environment, or the content root after the creation of the WebApplicationBuilder. For detailed instructions on changing host settings, see Customize IHostBuilder or IWebHostBuilder. The following highlighted APIs throw an exception:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

// WebHost
```

```
try
{
    builder.WebHost.UseContentRoot(Directory.GetCurrentDirectory());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseEnvironment(Environments.Staging);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseSetting(WebHostDefaults.ApplicationKey,
"ApplicationName2");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseSetting(WebHostDefaults.ContentRootKey,
Directory.GetCurrentDirectory());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

try
{
    builder.WebHost.UseSetting(WebHostDefaults.EnvironmentKey,
Environments.Staging);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

// Host
try
{
    builder.Host.UseEnvironment(Environments.Staging);
}
catch (Exception ex)
{
```

```
    Console.WriteLine(ex.Message);
}

try
{
    // TODO: This does not throw
    builder.Host.UseContentRoot(Directory.GetCurrentDirectory());
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapRazorPages();

app.Run();
```

- The `Startup` class can't be used from WebApplicationBuilder.Host or WebApplicationBuilder.WebHost. The following highlighted code throws an exception:

  C#

  ```
  var builder = WebApplication.CreateBuilder(args);

  try
  {
      builder.Host.ConfigureWebHostDefaults(webHostBuilder =>
      {
          webHostBuilder.UseStartup<Startup>();
      });
  }
  catch (Exception ex)
  {
      Console.WriteLine(ex.Message);
      throw;
  }
  ```

```
builder.Services.AddRazorPages();

var app = builder.Build();
```

C#

```
var builder = WebApplication.CreateBuilder(args);

try
{
    builder.WebHost.UseStartup<Startup>();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    throw;
}

builder.Services.AddRazorPages();

var app = builder.Build();
```

- The IHostBuilder implementation on WebApplicationBuilder (`WebApplicationBuilder.Host`), doesn't defer execution of the ConfigureServices, ConfigureAppConfiguration, or ConfigureHostConfiguration methods. Not deferring execution allows code using WebApplicationBuilder to observe changes made to the `IServiceCollection` and `IConfiguration`. The following example only adds `Service1` as an `IService`:

C#

```
using Microsoft.Extensions.DependencyInjection.Extensions;

var builder = WebApplication.CreateBuilder(args);

builder.Host.ConfigureServices(services =>
{
    services.TryAddSingleton<IService, Service1>();
});

builder.Services.TryAddSingleton<IService, Service2>();

var app = builder.Build();

// Displays Service1 only.
Console.WriteLine(app.Services.GetRequiredService<IService>());

app.Run();

class Service1 : IService
```

```
{
}

class Service2 : IService
{
}

interface IService
{
}
```

In the preceding code, the `builder.Host.ConfigureServices` callback gets called inline rather than being deferred until `builder.Build` is called. This means that `Service1` gets added to the `IServiceCollection` before `Service2` and results in `Service1` being resolved for `IService`.

Building libraries for ASP.NET Core 6

The existing .NET ecosystem built extensibility around IServiceCollection, IHostBuilder, and IWebHostBuilder. These properties are available on WebApplicationBuilder as `Services`, `Host`, and `WebHost`.

`WebApplication` implements both Microsoft.AspNetCore.Builder.IApplicationBuilder and Microsoft.AspNetCore.Routing.IEndpointRouteBuilder.

We expect library authors to continue targeting `IHostBuilder`, `IWebHostBuilder`, `IApplicationBuilder`, and `IEndpointRouteBuilder` when building ASP.NET Core specific components. This ensures that your middleware, route handler, or other extensibility points continue to work across different hosting models.

Frequently asked questions (FAQ)

- **Is the new minimal hosting model less capable?**

  No. The new hosting model is functionally equivalent for 98% of scenarios supported by `IHostBuilder` and the `IWebHostBuilder`. There are some advanced scenarios that require specific workarounds on `IHostBuilder`, but we expect those to be extremely rare.

- **Is the generic hosting model deprecated?**

  No. The generic hosting model is an alternative model that is supported indefinitely. The generic host underpins the new hosting model and is still the

primary way to host worker-based applications.

- **Do I have to migrate to the new hosting model?**

  No. The new hosting model is the preferred way to host new apps using .NET 6 and later, but you aren't forced to change the project layout in existing apps. This means apps can upgrade from .NET 5 to .NET 6 by changing the target framework in the project file from `net5.0` to `net6.0`. For more information, see the Update the target framework section in this article. However, we recommend apps migrate to the new hosting model to take advantage of new features only available to the new hosting model.

- **Do I have to use top-level statements?**

  No. The new project templates all use top-level statements, but the new hosting APIs can be used in any .NET 6 app to host a webserver or web app.

- **Where do I put state that was stored as fields in my `Program` or `Startup` class?**

  We strongly recommend using dependency injection (DI) to flow state in ASP.NET Core apps.

  There are two approaches to storing state outside of DI:

  - Store the state in another class. Storing in a class assumes a static state that can be accessed from anywhere in the app.
  - Use the `Program` class generated by top level statements to store state. Using `Program` to store state is the semantic approach:

    C#

    ```
    var builder = WebApplication.CreateBuilder(args);

    ConfigurationValue = builder.Configuration["SomeKey"] ?? "Hello";

    var app = builder.Build();

    app.MapGet("/", () => ConfigurationValue);

    app.Run();

    partial class Program
    {
        public static string? ConfigurationValue { get; private set; }
    }
    ```

- **What if I was using a custom dependency injection container?**

  Custom DI containers are supported. For an example, see Custom dependency injection (DI) container.

- **Do `WebApplicationFactory` and `TestServer` still work?**

  Yes. `WebApplicationFactory<TEntryPoint>` is the way to test the new hosting model. For an example, see Test with WebApplicationFactory or TestServer.

Blazor

After following the guidance earlier in this article to update an app to 6.0, adopt specific features by following the links in What's new in ASP.NET Core 6.0.

To adopt all of the new 6.0 features for Blazor apps, we recommend the following process:

- Create a new 6.0 Blazor project from one of the Blazor project templates. For more information, see Tooling for ASP.NET Core Blazor.
- Move the app's components and code to the 6.0 app making modifications to adopt the new 6.0 features.

Migrating SPA projects

Migrating Angular apps from SPA extensions

See this GitHub issue

Migrating React apps from SPA extensions

See **Migrating React applications from Spa Extensions** in this GitHub issue

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 6.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 5.0 and 6.0:

```
diff
```

```
- docker pull mcr.microsoft.com/dotnet/aspnet:5.0
+ docker pull mcr.microsoft.com/dotnet/aspnet:6.0
```

See GitHub issue Breaking Change: Default console logger format set to JSON.

Changes to the ASP.NET Core Razor SDK

The Razor compiler now leverages the new source generators feature to generate compiled C# files from the Razor views and pages in a project. In previous versions:

- The compilation relied on the `RazorGenerate` and `RazorCompile` targets to produce the generated code. These targets are no longer valid. In .NET 6, both code generation and compilation are supported by a single call to the compiler. `RazorComponentGenerateDependsOn` is still supported to specify dependencies that are required before the build runs.
- A separate Razor assembly, `AppName.Views.dll`, was generated that contained the compiled view types in an application. This behavior has been deprecated and a single assembly `AppName.dll` is produced that contains both the app types and the generated views.
- The app types in `AppName.Views.dll` were public. In .NET 6, the app types are in `AppName.dll` but are `internal sealed`. Apps doing type discover on `AppName.Views.dll` won't be able to do type discover on `AppName.dll`. The following shows the API change:

diff

```
- public class Views_Home_Index :
global::Microsoft.AspNetCore.Mvc.Razor.RazorPage<dynamic>
+ internal sealed class Views_Home_Index :
global::Microsoft.AspNetCore.Mvc.Razor.RazorPage<dynamic>
```

Make the following changes:

- The following properties are no longer applicable with the single-step compilation model.
  - `RazorTargetAssemblyAttribute`
  - `RazorTargetName`
  - `EnableDefaultRazorTargetAssemblyInfoAttributes`
  - `UseRazorBuildServer`
  - `GenerateRazorTargetAssemblyInfo`
  - `GenerateMvcApplicationPartsAssemblyAttributes`

For more information, see Razor compiler no longer produces a Views assembly.

Project templates use Duende Identity Server

Project templates now use Duende Identity Server.

> **Important**
>
> Duende Identity Server is an open source product with a reciprocal license agreement. If you plan to use Duende Identity Server in production, you might be required to obtain a commercial licence from Duende Software and pay a license fee. For more information, see Duende Software: Licenses.
>
> To learn how to use Microsoft Azure Active Directory for ASP.NET Core Identity, see Identity (dotnet/aspnetcore GitHub repository).

Add a `DbSet<Key>` property named `Keys` to every `IdentityDbContext` to satisfy a new requirement from the updated version of `IPersistedGrantDbContext`. The keys are required as part of the contract with Duende Identity Server's stores.

```
C#

public DbSet<Key> Keys { get; set; }
```

> **Note**
>
> Existing migrations must be recreated for Duende Identity Server.

Code samples migrated to ASP.NET Core 6.0

Code samples migrated to the new minimal hosting model in 6.0

Review breaking changes

See the following resources:

- Identity: Default Bootstrap version of UI changed
- Breaking changes for migration from version 5.0 to 6.0: Includes ASP.NET Core and Entity Framework Core.

- Announcements GitHub repository (aspnet/Announcements, 6.0.0 label): Includes breaking and non-breaking information.

Nullable reference types (NRTs) and .NET compiler null-state static analysis

ASP.NET Core project templates use nullable reference types (NRTs), and the .NET compiler performs null-state static analysis. These features were released with C# 8 and are enabled by default for apps generated using ASP.NET Core 6.0 (C# 10) or later.

The .NET compiler's null-state static analysis warnings can either serve as a guide for updating a documentation example or sample app locally or be ignored. Null-state static analysis can be disabled by setting Nullable to disable in the app's project file, which we only recommend for documentation examples and sample apps if the compiler warnings are distracting while learning about .NET. ***We don't recommend disabling null-state checking in production projects.***

For more information on NRTs, the MSBuild `Nullable` property, and updating apps (including `#pragma` guidance), see the following resources in the C# documentation:

- Nullable reference types
- Nullable reference types (C# reference)
- Learn techniques to resolve nullable warnings
- Update a codebase with nullable reference types to improve null diagnostic warnings
- Attributes for null-state static analysis
- ! (null-forgiving) operator (C# reference)

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Application name change

In .NET 6, WebApplicationBuilder normalizes the content root path to end with a DirectorySeparatorChar. Most apps migrating from HostBuilder or WebHostBuilder

won't have the same app name because they aren't normalized. For more information, see SetApplicationName

Additional resources

- Code samples migrated to the new minimal hosting model in 6.0

Code samples migrated to the new minimal hosting model in ASP.NET Core 6.0

Article • 07/14/2023

This article provides samples of code migrated to ASP.NET Core 6.0. ASP.NET Core 6.0 uses a new minimal hosting model. For more information, see New hosting model.

Middleware

The following code adds the Static File Middleware to an ASP.NET Core 5 app:

C#

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseStaticFiles();
    }
}
```

The following code adds the Static File Middleware to an ASP.NET Core 6 app:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.UseStaticFiles();

app.Run();
```

WebApplication.CreateBuilder initializes a new instance of the WebApplicationBuilder class with preconfigured defaults. For more information, see ASP.NET Core Middleware

Routing

The following code adds an endpoint to an ASP.NET Core 5 app:

C#

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", () => "Hello World");
        });
    }
}
```

In .NET 6, routes can be added directly to the WebApplication without an explicit call to UseEndpoints or UseRouting. The following code adds an endpoint to an ASP.NET Core 6 app:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

Note: Routes added directly to the WebApplication execute at the ***end*** of the pipeline.

Change the content root, app name, and environment

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseEnvironment(Environments.Staging)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>()
                      .UseSetting(WebHostDefaults.ApplicationKey,
                                  typeof(Program).Assembly.FullName);
        });
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    ApplicationName = typeof(Program).Assembly.FullName,
    ContentRootPath = Directory.GetCurrentDirectory(),
    EnvironmentName = Environments.Staging,
    WebRootPath = "customwwwroot"
});

Console.WriteLine($"Application Name:
{builder.Environment.ApplicationName}");
Console.WriteLine($"Environment Name:
{builder.Environment.EnvironmentName}");
Console.WriteLine($"ContentRoot Path:
{builder.Environment.ContentRootPath}");
Console.WriteLine($"WebRootPath: {builder.Environment.WebRootPath}");

var app = builder.Build();
```

For more information, see ASP.NET Core fundamentals overview

Change the content root, app name, and environment by environment variables or command line

The following table shows the environment variable and command-line argument used to change the content root, app name, and environment:

feature	Environment variable	Command-line argument
Application name	ASPNETCORE_APPLICATIONNAME	--applicationName
Environment name	ASPNETCORE_ENVIRONMENT	--environment
Content root	ASPNETCORE_CONTENTROOT	--contentRoot

Add configuration providers

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration(config =>
        {
            config.AddIniFile("appsettings.ini");
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Configuration.AddIniFile("appsettings.ini");

var app = builder.Build();
```

For detailed information, see File configuration providers in Configuration in ASP.NET Core.

Add logging providers

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.AddJsonConsole();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(args);

// Configure JSON logging to the console.
builder.Logging.AddJsonConsole();

var app = builder.Build();
```

For more information, see Logging in .NET Core and ASP.NET Core.

Add services

ASP.NET Core 5

C#

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add the memory cache services
        services.AddMemoryCache();

        // Add a custom scoped service
        services.AddScoped<ITodoRepository, TodoRepository>();
    }
}
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add the memory cache services.
builder.Services.AddMemoryCache();

// Add a custom scoped service.
builder.Services.AddScoped<ITodoRepository, TodoRepository>();
var app = builder.Build();
```

For more information, see Dependency injection in ASP.NET Core.

Customize IHostBuilder or IWebHostBuilder

Customize IHostBuilder

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureHostOptions(o => o.ShutdownTimeout =
TimeSpan.FromSeconds(30));
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(args);

// Wait 30 seconds for graceful shutdown.
builder.Host.ConfigureHostOptions(o => o.ShutdownTimeout =
TimeSpan.FromSeconds(30));

var app = builder.Build();
```

Customize IWebHostBuilder

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            // Change the HTTP server implementation to be HTTP.sys based.
            webBuilder.UseHttpSys()
                      .UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(args);

// Change the HTTP server implementation to be HTTP.sys based.
// Windows only.
builder.WebHost.UseHttpSys();

var app = builder.Build();
```

Change the web root

By default, the web root is relative to the content root in the `wwwroot` folder. Web root is where the static files middleware looks for static files. Web root can be changed by setting the WebRootPath property on WebApplicationOptions:

ASP.NET Core 5

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            // Look for static files in webroot.
            webBuilder.UseWebRoot("webroot")
                      .UseStartup<Startup>();
        });
```

ASP.NET Core 6

C#

```
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args,
    // Look for static files in webroot
    WebRootPath = "webroot"
});

var app = builder.Build();
```

Custom dependency injection (DI) container

The following .NET 5 and .NET 6 samples use Autofac

ASP.NET Core 5

Program class

```C#
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .UseServiceProviderFactory(new AutofacServiceProviderFactory())
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

Startup

```C#
public class Startup
{
    public void ConfigureContainer(ContainerBuilder containerBuilder)
    {
    }
}
```

ASP.NET Core 6

```C#
var builder = WebApplication.CreateBuilder(args);

builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());

// Register services directly with Autofac here. Don't
// call builder.Populate(), that happens in AutofacServiceProviderFactory.
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
builder.RegisterModule(new MyApplicationModule()));

var app = builder.Build();
```

Access additional services

`Startup.Configure` can inject any service added via the IServiceCollection.

ASP.NET Core 5

C#

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add
services
    // to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IService, Service>();
    }

    // Anything added to the service collection can be injected into
Configure.
    public void Configure(IApplicationBuilder app,
                          IWebHostEnvironment env,
                          IHostApplicationLifetime lifetime,
                          IService service,
                          ILogger<Startup> logger)
    {
        lifetime.ApplicationStarted.Register(() =>
            logger.LogInformation(
                "The application {Name} started in the injected {Service}",
                env.ApplicationName, service));
    }
}
```

ASP.NET Core 6

In ASP.NET Core 6:

- There are a few common services available as top level properties on WebApplication.
- Additional services need to be manually resolved from the `IServiceProvider` via WebApplication.Services.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<IService, Service>();

var app = builder.Build();

IService service = app.Services.GetRequiredService<IService>();
ILogger logger = app.Logger;
IHostApplicationLifetime lifetime = app.Lifetime;
```

```
IWebHostEnvironment env = app.Environment;

lifetime.ApplicationStarted.Register(() =>
    logger.LogInformation(
        $"The application {env.ApplicationName} started" +
        $" with injected {service}"));
```

Test with WebApplicationFactory or TestServer

ASP.NET Core 5

In the following samples, the test project uses `TestServer` and WebApplicationFactory<TEntryPoint>. These ship as separate packages that require explicit reference:

WebApplicationFactory

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Testing" Version="
{Version}" />
</ItemGroup>
```

TestServer

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.TestHost" Version="
{Version}" />
</ItemGroup>
```

ASP.NET Core 5 code

C#

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IHelloService, HelloService>();
    }
```

```
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
IHelloService helloService)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await
context.Response.WriteAsync(helloService.HelloMessage);
            });
        });
    }
}
```

With TestServer

C#

```
[Fact]
public async Task HelloWorld()
{
    using var host = Host.CreateDefaultBuilder()
        .ConfigureWebHostDefaults(builder =>
        {
            // Use the test server and point to the application's startup
            builder.UseTestServer()
                    .UseStartup<WebApplication1.Startup>();
        })
        .ConfigureServices(services =>
        {
            // Replace the service
            services.AddSingleton<IHelloService, MockHelloService>();
        })
        .Build();

    await host.StartAsync();

    var client = host.GetTestClient();

    var response = await client.GetStringAsync("/");

    Assert.Equal("Test Hello", response);
}
```

```
class MockHelloService : IHelloService
{
    public string HelloMessage => "Test Hello";
}
```

With WebApplicationFactory

C#

```
[Fact]
public async Task HelloWorld()
{
    var application = new WebApplicationFactory<Program>()
        .WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                services.AddSingleton<IHelloService, MockHelloService>();
            });
        });

    var client = application.CreateClient();

    var response = await client.GetStringAsync("/");

    Assert.Equal("Test Hello", response);
}

class MockHelloService : IHelloService
{
    public string HelloMessage => "Test Hello";
}
```

ASP.NET Core 6

C#

```


var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IHelloService, HelloService>();

var app = builder.Build();

var helloService = app.Services.GetRequiredService<IHelloService>();

app.MapGet("/", async context =>
{
    await context.Response.WriteAsync(helloService.HelloMessage);
});
```

```
app.Run();
```

Project file (.csproj)

The project file can contain one of the following:

XML

```
<ItemGroup>
    <InternalsVisibleTo Include="MyTestProject" />
</ItemGroup>
```

Or

```
[assembly: InternalsVisibleTo("MyTestProject")]
```

An alternative solution is to make the `Program` class public. `Program` can be made public with Top-level statements by defining a `public partial Program` class in the project or in `Program.cs`:

C#

```
var builder = WebApplication.CreateBuilder(args);

// ... Configure services, routes, etc.

app.Run();

public partial class Program { }
```

C#

```
[Fact]
public async Task HelloWorld()
{
    var application = new WebApplicationFactory<Program>()
        .WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                services.AddSingleton<IHelloService, MockHelloService>();
            });
        });
```

```
    var client = application.CreateClient();

    var response = await client.GetStringAsync("/");

    Assert.Equal("Test Hello", response);
}

class MockHelloService : IHelloService
{
    public string HelloMessage => "Test Hello";
}
```

The .NET 5 version and .NET 6 version with the `WebApplicationFactory` are identical by design.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Migrate from ASP.NET Core 3.1 to 6.0

Article • 06/04/2022

This article explains how to update an existing ASP.NET Core 3.1 project to ASP.NET Core 6.0. To upgrade from ASP.NET Core 5.0 to 6.0, see Migrate from ASP.NET Core 5.0 to 6.0.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.

Update .NET SDK version in `global.json`

If you rely upon a global.json file to target a specific .NET SDK version, update the `version` property to the .NET 6.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "3.1.200"
+    "version": "6.0.100"
  }
}
```

Update the target framework

Update the project file's Target Framework Moniker (TFM) to `net6.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.1</TargetFramework>
+    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Update package references

In the project file, update each Microsoft.AspNetCore.*, Microsoft.EntityFrameworkCore.*, Microsoft.Extensions.*, and System.Net.Http.Json package reference's `Version` attribute to 6.0.0 or later. For example:

diff

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="3.1.6" />
-    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="3.1.6" />
-    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="3.1.6" />
-    <PackageReference Include="System.Net.Http.Json" Version="3.2.1" />
+    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="6.0.0" />
+    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="6.0.0" />
+    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="6.0.0" />
+    <PackageReference Include="System.Net.Http.Json" Version="6.0.0" />
</ItemGroup>
```

Delete `bin` and `obj` folders

You may need to delete the `bin` and `obj` folders. Run `dotnet nuget locals --clear all` to clear the NuGet package cache.

Minimal hosting model

The ASP.NET Core templates generate code using the new minimal hosting model. The minimal hosting model unifies `Startup.cs` and `Program.cs` into a single `Program.cs` file. `ConfigureServices` and `Configure` are no longer used. Apps migrating from ASP.NET Core 3.1 to 6.0 don't need to use the minimal hosting model, using `Startup` and the Generic Host used by the ASP.NET Core 3.1 templates is fully supported.

To use `Startup` with the new minimal hosting model, see Use Startup with the new minimal hosting model.

To migrate to the new minimal hosting model using the following pattern used by the ASP.NET Core 6.0 templates, see Code samples migrated to the new minimal hosting model in ASP.NET Core 6.0 and Migrate from ASP.NET Core 5.0 to 6.0

Update Razor class libraries (RCLs)

Migrate Razor class libraries (RCLs) to take advantage of new APIs or features that are introduced as part of ASP.NET Core 6.0.

To update a RCL that targets components:

1. Update the following properties in the project file:

   diff

   ```
   <Project Sdk="Microsoft.NET.Sdk.Razor">
     <PropertyGroup>
   -     <TargetFramework>netstandard2.0</TargetFramework>
   -     <RazorLangVersion>3.0</RazorLangVersion>
   +     <TargetFramework>net6.0</TargetFramework>
     </PropertyGroup>
   ```

2. Update other packages to their latest versions. The latest versions can be found at NuGet.org.

To update an RCL targeting MVC, update the following properties in the project file:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.1</TargetFramework>
+    <TargetFramework>net6.0</TargetFramework>
     <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>
```

Blazor

To adopt all of the 5.0 features and 6.0 features for Blazor apps, we recommend the following process:

- Create a new 6.0 Blazor project from one of the Blazor project templates. For more information, see Tooling for ASP.NET Core Blazor.

- Move the app's components and code to the 6.0 app making modifications to adopt the new 5.0 and 6.0 features.

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 6.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 3.1 and 6.0:

```
diff
```

```
- docker pull mcr.microsoft.com/dotnet/core/aspnet:3.1
+ docker pull mcr.microsoft.com/dotnet/aspnet:6.0
```

As part of the move to ".NET" as the product name, the Docker images moved from the `mcr.microsoft.com/dotnet/core` repositories to `mcr.microsoft.com/dotnet`. For more information, see .NET 5.0 - Docker Repo Name Change (dotnet/dotnet-docker #1939).

Model binding changes in ASP.NET Core MVC and Razor Pages

`DateTime` values are model bound as UTC times

In ASP.NET Core 3.1 and earlier, `DateTime` values were model-bound as local time, where the timezone was determined by the server. `DateTime` values bound from input formatting (JSON) and `DateTimeOffset` values were bound as UTC timezones.

In ASP.NET Core 5.0 and later, model binding consistently binds `DateTime` values with the UTC timezone.

To retain the previous behavior, remove the `DateTimeModelBinderProvider` in `Startup.ConfigureServices`:

```
C#
```

```
services.AddControllersWithViews(options =>
    options.ModelBinderProviders.RemoveType<DateTimeModelBinderProvider>());
```

`ComplexObjectModelBinderProvider` \ `ComplexObjectModelBinder` replace `ComplexTypeModelBinderProvider` \ `ComplexTypeModelBinder`

To add support for model binding C# 9 record types, the ComplexTypeModelBinderProvider is:

- Annotated as obsolete.
- No longer registered by default.

Apps that rely on the presence of the `ComplexTypeModelBinderProvider` in the `ModelBinderProviders` collection need to reference the new binder provider:

diff

```
- var complexModelBinderProvider =
options.ModelBinderProviders.OfType<ComplexTypeModelBinderProvider>();
+ var complexModelBinderProvider =
options.ModelBinderProviders.OfType<ComplexObjectModelBinderProvider>();
```

`UseDatabaseErrorPage` obsolete

The ASP.NET Core 3.1 templates that include an option for individual user accounts generate a call to UseDatabaseErrorPage. `UseDatabaseErrorPage` is now obsolete and should be replaced with a combination of `AddDatabaseDeveloperPageExceptionFilter` and `UseMigrationsEndPoint`, as shown in the following code:

diff

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
+   services.AddDatabaseDeveloperPageExceptionFilter();
    services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
```

```
+       app.UseMigrationsEndPoint();
-       app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }
```

For more information, see Obsoleting DatabaseErrorPage middleware (dotnet/aspnetcore #24987).

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Application name change

In .NET 6, WebApplicationBuilder normalizes the content root path to end with a DirectorySeparatorChar. Most apps migrating from HostBuilder or WebHostBuilder won't have the same app name because they aren't normalized. For more information, see SetApplicationName

Review breaking changes

See the following resources:

- Identity: Default Bootstrap version of UI changed
- Breaking changes for migration from version 3.1 to 5.0. ASP.NET Core and Entity Framework Core are also included in the list.
- Breaking changes for migration from version 5.0 to 6.0: Includes ASP.NET Core and Entity Framework Core.
- Announcements GitHub repository (aspnet/Announcements, 6.0.0 label): Includes breaking and non-breaking information.
- Announcements GitHub repository (aspnet/Announcements, 5.0.0 label): Includes breaking and non-breaking information.

Migrate from ASP.NET Core 3.1 to 5.0

Article • 08/30/2023

This article explains how to update an existing ASP.NET Core 3.1 project to ASP.NET Core 5.0. For instructions on how to migrate from ASP.NET Core 3.1 to ASP.NET Core 6.0, see Migrate from ASP.NET Core 3.1 to 6.0.

Prerequisites

Visual Studio

- Visual Studio 2019 16.8 or later with the **ASP.NET and web development** workload
- .NET 5.0 SDK

Update .NET Core SDK version in global.json

If you rely upon a global.json file to target a specific .NET Core SDK version, update the `version` property to the .NET 5.0 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "3.1.200"
+    "version": "5.0.100"
  }
}
```

Update the target framework

If updating a Blazor WebAssembly project, skip to the Update Blazor WebAssembly projects section. For any other ASP.NET Core project type, update the project file's Target Framework Moniker (TFM) to `net5.0`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

```
  <PropertyGroup>
-    <TargetFramework>netcoreapp3.1</TargetFramework>
+    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Delete `bin` and `obj` folders

You may need to delete the `bin` and `obj` folders. Run `dotnet nuget locals --clear all` to clear the NuGet package cache.

Changes to Blazor app routing logic in 5.0.1 and further 5.x releases up to 6.0

The computation of route precedence changed in the ASP.NET Core 5.0.1 patch release. This might affect you if you've defined catch-all routes or routes with optional parameters.

Old behavior

With the prior behavior in ASP.NET Core 5.0.0 or earlier, routes with lower precedence, such as `{*slug}`, are matched before routes with higher precedence, such as `/customer/{id}`.

New behavior

The new behavior in ASP.NET Core 5.0.1 or later more closely matches the routing behavior defined in ASP.NET Core apps, where the framework computes and establishes the route precedence for each segment first and only uses the length of the route to break ties as a secondary criteria.

Reason for change

The original behavior is considered a bug in the implementation because our goal is for the Blazor routing system to behave in the same way as the ASP.NET Core routing system for the subset of features supported by Blazor routing.

Recommended action

Add the `PreferExactMatches` attribute to the `Router` component in the `App.razor` file to opt into the correct behavior:

razor

```
<Router AppAssembly="@typeof(Program).Assembly" PreferExactMatches="@true">
```

When `PreferExactMatches` is set to `@true`, route matching prefers exact matches over wildcards.

> **ⓘ Important**
>
> All apps should explicitly set `PreferExactMatches` to `@true`.
>
> The ability to set `PreferExactMatches` to `@false` or leave it unset *is only provided for backward compatibility*.
>
> When .NET 6 is released, the router will always prefer exact matches, and the `PreferExactMatches` option won't be available.

Update Blazor WebAssembly and Blazor Server projects

The guidance in this section applies to both Blazor hosting models. Sections following this section provide additional guidance specific to hosting models and app types. Apply the guidance from all relevant sections to your app.

1. In `wwwroot/index.html` of a Blazor WebAssembly app or the `Pages/_Host.cshtml` of a Blazor Server app, add a `<link>` element to the `<head>` element for styles. In the following `<link>` element `href` attribute values, the placeholder `{ASSEMBLY NAME}` is the app's assembly name.

   diff

   ```
   +<link href="{ASSEMBLY NAME}.styles.css" rel="stylesheet" />
   ```

   Standalone Blazor WebAssembly or Blazor Server example:

   diff

```
+<link href="BlazorSample.styles.css" rel="stylesheet" />
```

`Client` project of a hosted Blazor WebAssembly solution example:

diff

```
+<link href="BlazorSample.Client.styles.css" rel="stylesheet" />
```

2. Include a new namespace in the app's `_Imports.razor` file for component virtualization, Microsoft.AspNetCore.Components.Web.Virtualization. The following `_Imports.razor` files show the default namespaces in apps generated from the Blazor project templates. The placeholder `{ASSEMBLY NAME}` is the app's assembly name.

   Blazor WebAssembly (`_Imports.razor`):

   razor

   ```
   @using System.Net.Http
   @using System.Net.Http.Json
   @using Microsoft.AspNetCore.Components.Forms
   @using Microsoft.AspNetCore.Components.Routing
   @using Microsoft.AspNetCore.Components.Web
   @using Microsoft.AspNetCore.Components.Web.Virtualization
   @using Microsoft.AspNetCore.Components.WebAssembly.Http
   @using Microsoft.JSInterop
   @using {ASSEMBLY NAME}
   @using {ASSEMBLY NAME}.Shared
   ```

   Blazor Server (`_Imports.razor`):

   razor

   ```
   @using System.Net.Http
   @using Microsoft.AspNetCore.Authorization
   @using Microsoft.AspNetCore.Components.Authorization
   @using Microsoft.AspNetCore.Components.Forms
   @using Microsoft.AspNetCore.Components.Routing
   @using Microsoft.AspNetCore.Components.Web
   @using Microsoft.AspNetCore.Components.Web.Virtualization
   @using Microsoft.JSInterop
   @using {ASSEMBLY NAME}
   @using {ASSEMBLY NAME}.Shared
   ```

3. In the `MainLayout` component (`Shared/MainLayout.razor`), surround the component's HTML markup with a `<div>` element that has a `class` attribute set to `page`:

```razor
<div class="page">

    ...

</div>
```

4. Add the following files to the `Shared` folder:

`MainLayout.razor.css`:

```css
.page {
    position: relative;
    display: flex;
    flex-direction: column;
}

.main {
    flex: 1;
}

.sidebar {
    background-image: linear-gradient(180deg, rgb(5, 39, 103) 0%,
#3a0647 70%);
}

.top-row {
    background-color: #f7f7f7;
    border-bottom: 1px solid #d6d5d5;
    justify-content: flex-end;
    height: 3.5rem;
    display: flex;
    align-items: center;
}

    .top-row ::deep a, .top-row .btn-link {
        white-space: nowrap;
        margin-left: 1.5rem;
    }

    .top-row a:first-child {
        overflow: hidden;
        text-overflow: ellipsis;
    }
```

```
@media (max-width: 767.98px) {
    .top-row:not(.auth) {
        display: none;
    }

    .top-row.auth {
        justify-content: space-between;
    }

    .top-row a, .top-row .btn-link {
        margin-left: 0;
    }
}

@media (min-width: 768px) {
    .page {
        flex-direction: row;
    }

    .sidebar {
        width: 250px;
        height: 100vh;
        position: sticky;
        top: 0;
    }

    .top-row {
        position: sticky;
        top: 0;
        z-index: 1;
    }

    .main > div {
        padding-left: 2rem !important;
        padding-right: 1.5rem !important;
    }
}
```

`NavMenu.razor.css`:

```css
.navbar-toggler {
    background-color: rgba(255, 255, 255, 0.1);
}

.top-row {
    height: 3.5rem;
    background-color: rgba(0,0,0,0.4);
}

.navbar-brand {
```

```
    font-size: 1.1rem;
}

.oi {
    width: 2rem;
    font-size: 1.1rem;
    vertical-align: text-top;
    top: -2px;
}

.nav-item {
    font-size: 0.9rem;
    padding-bottom: 0.5rem;
}

    .nav-item:first-of-type {
        padding-top: 1rem;
    }

    .nav-item:last-of-type {
        padding-bottom: 1rem;
    }

    .nav-item ::deep a {
        color: #d7d7d7;
        border-radius: 4px;
        height: 3rem;
        display: flex;
        align-items: center;
        line-height: 3rem;
    }

.nav-item ::deep a.active {
    background-color: rgba(255,255,255,0.25);
    color: white;
}

.nav-item ::deep a:hover {
    background-color: rgba(255,255,255,0.1);
    color: white;
}

@media (min-width: 768px) {
    .navbar-toggler {
        display: none;
    }

    .collapse {
        /* Never collapse the sidebar for wide screens */
        display: block;
    }
}
```

5. The latest base `wwwroot/css/app.css` file of a Blazor WebAssembly app or `wwwroot/css/site.css` file of a Blazor Server app includes the following styles. Remove extra styles leaving the following styles and any that you've added to the app.

   The following stylesheet only includes base styles and does **not** include custom styles added by the developer:

   ```css
   html, body {
       font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
   }

   a, .btn-link {
       color: #0366d6;
   }

   .btn-primary {
       color: #fff;
       background-color: #1b6ec2;
       border-color: #1861ac;
   }

   .content {
       padding-top: 1.1rem;
   }

   .valid.modified:not([type=checkbox]) {
       outline: 1px solid #26b050;
   }

   .invalid {
       outline: 1px solid red;
   }

   .validation-message {
       color: red;
   }

   #blazor-error-ui {
       background: lightyellow;
       bottom: 0;
       box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
       display: none;
       left: 0;
       padding: 0.6rem 1.25rem 0.7rem 1.25rem;
       position: fixed;
       width: 100%;
       z-index: 1000;
   }
   ```

```
#blazor-error-ui .dismiss {
    cursor: pointer;
    position: absolute;
    right: 0.75rem;
    top: 0.5rem;
}
```

> **Note**
>
> The preceding example doesn't show the `@import` directive for Open Iconic icons (`open-iconic-bootstrap.css`), provided by the Blazor project template. Open Iconic was abandoned by its maintainers.

Update Blazor WebAssembly projects

Follow the guidance in the preceding Update Blazor WebAssembly and Blazor Server projects section.

For a Blazor WebAssembly project, including the *`Client`* project of a hosted Blazor solution, apply the following changes to the project file:

1. Update the SDK from `Microsoft.NET.Sdk.Web` to `Microsoft.NET.Sdk.BlazorWebAssembly`:

   diff

   ```
   - <Project Sdk="Microsoft.NET.Sdk.Web">
   + <Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">
   ```

   > **Note**
   >
   > This update only applies to standalone Blazor WebAssembly projects and the *`Client`* projects of hosted Blazor solutions.

2. Update the following properties:

   diff

   ```
   <Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">

     <PropertyGroup>
   -     <TargetFramework>netstandard2.1</TargetFramework>
   -     <RazorLangVersion>3.0</RazorLangVersion>
   ```

```
+     <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
```

3. Remove the package reference to Microsoft.AspNetCore.Components.WebAssembly.Build:

```
diff

<ItemGroup>
-    <PackageReference
Include="Microsoft.AspNetCore.Components.WebAssembly.Build"
Version="3.2.1" PrivateAssets="all" />
```

4. Update other packages to their latest versions. The latest versions can be found at NuGet.org.

5. In `wwwroot/index.html`, change the element that loads the `App` component to a `<div>` element with an `id` set to `app`:

```
diff

-<app>Loading...</app>
+<div id="app">Loading...</div>
```

6. In `Program.Main` (`Program.cs`), change the reference to the `<app>` element to a CSS selector by adding a hash `#` to it:

```
diff

-builder.RootComponents.Add<App>("app");
+builder.RootComponents.Add<App>("#app");
```

7. In `Program.Main` (`Program.cs`), change a default transient `HttpClient` registration to scoped, if present:

```
diff

-builder.Services.AddTransient(sp => new HttpClient
-    { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
+builder.Services.AddScoped(sp => new HttpClient
+    { BaseAddress = new Uri(builder.HostEnvironment.BaseAddress) });
```

8. In `Program.Main` (`Program.cs`) of the **`Client`** app of hosted Blazor solutions:

- Optionally, substitute `builder.HostEnvironment.BaseAddress` for string client base addresses.
- Change any named transient client factory registrations to scoped.

diff

```
-builder.Services.AddHttpClient("{APP NAMESPACE}.ServerAPI",
-    client => client.BaseAddress = new Uri("https://localhost:5001"))
-    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();
-builder.Services.AddTransient(sp =>
sp.GetRequiredService<IHttpClientFactory>()
-    .CreateClient("{APP NAMESPACE}.ServerAPI"));
+builder.Services.AddHttpClient("{APP NAMESPACE}.ServerAPI",
+    client => client.BaseAddress = new
Uri(builder.HostEnvironment.BaseAddress))
+    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();
+builder.Services.AddScoped(sp =>
sp.GetRequiredService<IHttpClientFactory>()
+    .CreateClient("{APP NAMESPACE}.ServerAPI"));
```

In the preceding code, the `{APP NAMESPACE}` placeholder is the app's namespace.

Standalone Blazor WebAssembly app with Microsoft Accounts

Follow the guidance in the preceding Update Blazor WebAssembly and Blazor Server projects and Update Blazor WebAssembly projects sections.

For a standalone Blazor WebAssembly app registered in the Azure portal to use Microsoft Entra ID (ME-ID) for Microsoft Accounts:

- The app requires the `openid` and `offline_access` scopes:

C#

```
options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
```

- In the Azure portal app registration **Authentication** blade:

  1. Remove the **Web** platform configuration.
  2. Add a **Single-page application** platform configuration with the app's redirect URI.
  3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Accounts.

Standalone Blazor WebAssembly app with Microsoft Entra ID (ME-ID)

Follow the guidance in the preceding Update Blazor WebAssembly and Blazor Server projects and Update Blazor WebAssembly projects sections.

For a standalone Blazor WebAssembly app registered in the Azure portal to use Microsoft Entra ID (ME-ID):

- The app requires the `https://graph.microsoft.com/User.Read` scope:

  ```
  C#

  options.ProviderOptions.DefaultAccessTokenScopes
      .Add("https://graph.microsoft.com/User.Read");
  ```

- In the Azure portal app registration **Authentication** blade:

  1. Remove the **Web** platform configuration.
  2. Add a **Single-page application** platform configuration with the app's redirect URI.
  3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID.

Standalone Blazor WebAssembly app with Azure Active Directory (AAD) B2C

Follow the guidance in the preceding Update Blazor WebAssembly and Blazor Server projects and Update Blazor WebAssembly projects sections.

For a standalone Blazor WebAssembly app registered in the Azure portal to use Azure Active Directory (AAD) B2C:

- The app requires the `openid` and `offline_access` scopes:

  ```
  C#

  options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
  ```

```
options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
```

- In the Azure portal app registration **Authentication** blade:

  1. Remove the **Web** platform configuration.
  2. Add a **Single-page application** platform configuration with the app's redirect URI.
  3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory B2C.

Hosted Blazor WebAssembly app with Microsoft Entra ID (ME-ID) or AAD B2C

Follow the guidance in the preceding Update Blazor WebAssembly and Blazor Server projects and Update Blazor WebAssembly projects sections.

The `Client` app registration of a hosted Blazor solution that uses AAD or AAD B2C for user authentication should use a **Single-page application** Azure Apps platform configuration.

In the Azure portal `Client` app registration **Authentication** blade:

1. Remove the **Web** platform configuration.
2. Add a **Single-page application** platform configuration with the app's redirect URI.
3. Disable **Implicit grant** for **Access tokens** and **ID tokens**.

For more information, see:

- Secure a hosted ASP.NET Core Blazor WebAssembly app with Microsoft Entra ID
- Secure a hosted ASP.NET Core Blazor WebAssembly app with Azure Active Directory B2C

Update the Server project of a hosted Blazor solution

Follow the guidance in the preceding sections:

- Update Blazor WebAssembly and Blazor Server projects
- Update Blazor WebAssembly projects
- The section that applies to the app's provider with Azure Active Directory:
  - Standalone Blazor WebAssembly app with Microsoft Accounts
  - Standalone Blazor WebAssembly app with Microsoft Entra ID (ME-ID)

- Standalone Blazor WebAssembly app with Azure Active Directory (AAD) B2C

Update the `Server` project of a hosted Blazor solution as an ASP.NET Core app following the general guidance in this article.

Additionally, `Server` projects that authenticate users to client Blazor WebAssembly apps with Microsoft Entra ID (ME-ID) or B2C should adopt new Microsoft Identity v2.0 packages:

For AAD:

diff

```
-<PackageReference Include="Microsoft.AspNetCore.Authentication.AzureAD.UI"
Version="..." />
+<PackageReference Include="Microsoft.Identity.Web" Version="{VERSION}" />
+<PackageReference Include="Microsoft.Identity.Web.UI" Version="{VERSION}"
/>
```

For AAD B2C:

diff

```
-<PackageReference
Include="Microsoft.AspNetCore.Authentication.AzureADB2C.UI" Version="..." />
+<PackageReference Include="Microsoft.Identity.Web" Version="{VERSION}" />
+<PackageReference Include="Microsoft.Identity.Web.UI" Version="{VERSION}"
/>
```

For the preceding package references, determine the package versions for the `{VERSION}` placeholders at NuGet.org:

- Microsoft.Identity.Web
- Microsoft.Identity.Web.UI

Note

The SDK of the `Server` project in a hosted Blazor WebAssembly solution remains `Microsoft.NET.Sdk.Web`:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
```

For more information, see:

- Secure a hosted ASP.NET Core Blazor WebAssembly app with Microsoft Entra ID
- Secure a hosted ASP.NET Core Blazor WebAssembly app with Azure Active Directory B2C

Clean and rebuild the solution

After migrating the app or solution to .NET 5, clean and rebuild the app or solution. If package incompatibilities exist between new package references and cached packages:

1. Clear NuGet package caches by executing the following dotnet nuget locals command in a command shell:

   .NET CLI

   ```
   dotnet nuget locals --clear all
   ```

2. Clean and rebuild the app or solution.

Troubleshoot

Follow the *Troubleshoot* guidance at the end of the Blazor WebAssembly security topic that applies to your app:

Standalone Blazor WebAssembly apps:

- General guidance for OIDC providers and the WebAssembly Authentication Library
- Microsoft Accounts
- Microsoft Entra ID (ME-ID)
- Azure Active Directory (AAD) B2C

Hosted Blazor WebAssembly apps:

- Microsoft Entra ID (ME-ID)
- Azure Active Directory (AAD) B2C
- Identity Server

Unauthorized client for Microsoft Entra ID (ME-ID)

After upgrading a Blazor WebAssembly app that uses AAD for authentication, you may receive the following error on the login callback to the app after the user signs in with AAD:

> info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2]
> Authorization failed. These requirements were not met:
> DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

Login callback error from AAD:

- Error: `unauthorized_client`
- Description: `AADB2C90058: The provided application is not configured to allow public clients.`

To resolve the error:

1. In the Azure portal, access the app's manifest.
2. Set the allowPublicClient attribute to `null` or `true`.

Update a Blazor Progressive Web Application (PWA)

Add the following item to the PWA app's project file:

XML

```
<ItemGroup>
  <ServiceWorker Include="wwwroot\service-worker.js"
    PublishedContent="wwwroot\service-worker.published.js" />
</ItemGroup>
```

Remove preview CSS isolation stylesheet link

If the project's `wwwroot/index.html` (Blazor WebAssembly) or `Pages/_Host.cshtml` (Blazor Server) contains a stylesheet `<link>` element for `scoped.styles.css` from an earlier 5.0 preview release, remove the `<link>` tag:

diff

```
-<link href="_framework/scoped.styles.css/" rel="stylesheet" />
```

Update Razor class libraries (RCLs)

Migrate Razor class libraries (RCLs) to take advantage of new APIs or features that are introduced as part of ASP.NET Core 5.0.

To update a RCL that targets components:

1. Update the following properties in the project file:

   diff

   ```
   <Project Sdk="Microsoft.NET.Sdk.Razor">

     <PropertyGroup>
   -     <TargetFramework>netstandard2.0</TargetFramework>
   -     <RazorLangVersion>3.0</RazorLangVersion>
   +     <TargetFramework>net5.0</TargetFramework>
     </PropertyGroup>
   ```

2. Update other packages to their latest versions. The latest versions can be found at NuGet.org.

To update an RCL targeting MVC, update the following properties in the project file:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Razor">

  <PropertyGroup>
-     <TargetFramework>netcoreapp3.1</TargetFramework>
+     <TargetFramework>net5.0</TargetFramework>
    <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
  </PropertyGroup>
```

Update package references

In the project file, update each Microsoft.AspNetCore.*, Microsoft.EntityFrameworkCore.*, Microsoft.Extensions.*, and System.Net.Http.Json package reference's `Version` attribute to 5.0.0 or later. For example:

diff

```
<ItemGroup>
-     <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="3.1.6" />
-     <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="3.1.6">
-     <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
```

```
Version="3.1.6" />
-    <PackageReference Include="System.Net.Http.Json" Version="3.2.1" />
+    <PackageReference Include="Microsoft.AspNetCore.JsonPatch"
Version="5.0.0" />
+    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="5.0.0">
+    <PackageReference Include="Microsoft.Extensions.Caching.Abstractions"
Version="5.0.0" />
+    <PackageReference Include="System.Net.Http.Json" Version="5.0.0" />
</ItemGroup>
```

Update Docker images

For apps using Docker, update your *Dockerfile* `FROM` statements and scripts. Use a base image that includes the ASP.NET Core 5.0 runtime. Consider the following `docker pull` command difference between ASP.NET Core 3.1 and 5.0:

diff

```
- docker pull mcr.microsoft.com/dotnet/core/aspnet:3.1
+ docker pull mcr.microsoft.com/dotnet/aspnet:5.0
```

As part of the move to ".NET" as the product name, the Docker images moved from the `mcr.microsoft.com/dotnet/core` repositories to `mcr.microsoft.com/dotnet`. For more information, see dotnet/dotnet-docker#1939.

Model binding changes in ASP.NET Core MVC and Razor Pages

DateTime values are model bound as UTC times

In ASP.NET Core 3.1 and earlier, `DateTime` values were model-bound as local time, where the timezone was determined by the server. `DateTime` values bound from input formatting (JSON) and `DateTimeOffset` values were bound as UTC timezones.

In ASP.NET Core 5.0 and later, model binding consistently binds `DateTime` values with the UTC timezone.

To retain the previous behavior, remove the `DateTimeModelBinderProvider` in `Startup.ConfigureServices`:

C#

```
services.AddControllersWithViews(options =>
    options.ModelBinderProviders.RemoveType<DateTimeModelBinderProvider>());
```

ComplexObjectModelBinderProvider \ ComplexObjectModelBinder replace ComplexTypeModelBinderProvider \ ComplexTypeModelBinder

To add support for model binding C# 9 record types, the ComplexTypeModelBinderProvider is:

- Annotated as obsolete.
- No longer registered by default.

Apps that rely on the presence of the `ComplexTypeModelBinderProvider` in the `ModelBinderProviders` collection need to reference the new binder provider:

diff

```
- var complexModelBinderProvider =
options.ModelBinderProviders.OfType<ComplexTypeModelBinderProvider>();
+ var complexModelBinderProvider =
options.ModelBinderProviders.OfType<ComplexObjectModelBinderProvider>();
```

UseDatabaseErrorPage obsolete

The ASP.NET Core 3.1 templates that include an option for individual user accounts generate a call to UseDatabaseErrorPage. `UseDatabaseErrorPage` is now obsolete and should be replaced with a combination of `AddDatabaseDeveloperPageExceptionFilter` and `UseMigrationsEndPoint`, as shown in the following code:

diff

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
+   services.AddDatabaseDeveloperPageExceptionFilter();
    services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    services.AddRazorPages();
```

```
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
+       app.UseMigrationsEndPoint();
-       app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }
```

For more information, see this GitHub issue.

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Package reference changes affecting some NuGet packages

With the migration of some `Microsoft.Extensions.*` NuGet packages from the dotnet/extensions repository to dotnet/runtime, as described in Migrating dotnet/extensions content to dotnet/runtime and dotnet/aspnetcore (aspnet/Announcements #411), packaging changes are being applied to some of the migrated packages. These changes often result in namespace changes for .NET API.

To research APIs further for app namespace changes when migrating to 5.0, use the .NET API browser.

Migrate Microsoft.Identity.Web

The following wiki pages explain how to migrate Microsoft.Identity.Web from ASP.NET Core 3.1 to 5.0:

- Microsoft.Identity.Web in web apps

- Microsoft.Identity.Web in web APIs

The following tutorials also explain the migration:

- An ASP.NET Core Web app signing-in users with the Microsoft identity platform in your organization. See **Option 2: Create the sample from the command line**.
- Sign-in a user with the Microsoft identity platform in a WPF Desktop application and call an ASP.NET Core Web API. See **How was the code created**.

Review breaking changes

For breaking changes from .NET Core 3.1 to .NET 5.0, see Breaking changes for migration from version 3.1 to 5.0. ASP.NET Core and Entity Framework Core are also included in the list.

Migrate from ASP.NET Core 3.0 to 3.1

Article • 06/18/2024

By Scott Addie

This article explains how to update an existing ASP.NET Core 3.0 project to ASP.NET Core 3.1.

Prerequisites

Visual Studio

- Visual Studio 2019 16.4 or later with the **ASP.NET and web development** workload
- .NET Core 3.1 SDK

Update .NET Core SDK version in global.json

If you rely upon a global.json file to target a specific .NET Core SDK version, update the `version` property to the 3.1 SDK version that's installed. For example:

diff

```
{
  "sdk": {
-    "version": "3.0.101"
+    "version": "3.1.101"
  }
}
```

Update the target framework

In the project file, update the Target Framework Moniker (TFM) to `netcoreapp3.1`:

diff

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
-    <TargetFramework>netcoreapp3.0</TargetFramework>
```

```
+    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

Update package references

In the project file, update each `Microsoft.AspNetCore.*` package reference's `Version` attribute to 3.1.0 or later. For example:

diff

```
<ItemGroup>
-    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson"
Version="3.0.0" />
-    <PackageReference
Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.0.0"
Condition="'$(Configuration)' == 'Debug'" />
+    <PackageReference Include="Microsoft.AspNetCore.Mvc.NewtonsoftJson"
Version="3.1.1" />
+    <PackageReference
Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="3.1.1"
Condition="'$(Configuration)' == 'Debug'" />
</ItemGroup>
```

Update Docker images

For apps using Docker, use a base image that includes ASP.NET Core 3.1. For example:

Console

```
docker pull mcr.microsoft.com/dotnet/aspnet:3.1
```

React to SameSite cookie changes

The `SameSite` attribute implementations for HTTP cookies changed between ASP.NET Core 3.0 and 3.1. For actions to be taken, see the following resources:

- Work with SameSite cookies in ASP.NET Core
- aspnet/Announcements#390
- Upcoming SameSite cookie changes in ASP.NET and ASP.NET Core

Publish with Visual Studio

In the `.pubxml` file update the `TargetFramework` to 3.1:

XML

```
- <TargetFramework>netcoreapp3.0</TargetFramework>
+ <TargetFramework>netcoreapp3.1</TargetFramework>
```

Review breaking changes

Review 3.0-to-3.1 breaking changes across .NET Core, ASP.NET Core, and Entity Framework Core at Breaking changes for migration from version 3.0 to 3.1.

Optional changes

The following changes are optional.

Use the Component Tag Helper

ASP.NET Core 3.1 introduces a `Component` Tag Helper. The Tag Helper can replace the `RenderComponentAsync<TComponent>` HTML helper method in a Blazor project. For example:

diff

```
- @(await Html.RenderComponentAsync<Counter>(RenderMode.ServerPrerendered,
new { IncrementAmount = 10 }))
+ <component type="typeof(Counter)" render-mode="ServerPrerendered" param-
IncrementAmount="10" />
```

For more information, see Integrate ASP.NET Core Razor components.

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Migrate from ASP.NET Core 2.2 to 3.0

Article • 06/03/2023

By Scott Addie and Rick Anderson

This article explains how to update an existing ASP.NET Core 2.2 project to ASP.NET Core 3.0. It might be helpful to create a new ASP.NET Core 3.0 project to:

- Compare with the ASP.NET Core 2.2 code.
- Copy the relevant changes to your ASP.NET Core 3.0 project.

Prerequisites

Visual Studio

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core 3.0 SDK

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to the 3.0 version installed on your machine:

JSON

```
{
  "sdk": {
    "version": "3.0.100"
  }
}
```

Update the project file

Update the Target Framework

ASP.NET Core 3.0 and later only run on .NET Core. Set the Target Framework Moniker (TFM) to `netcoreapp3.0`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Remove obsolete package references

A large number of NuGet packages aren't produced for ASP.NET Core 3.0. Such package references should be removed from your project file. Consider the following project file for an ASP.NET Core 2.2 web app:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App"/>
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design"
Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

The updated project file for ASP.NET Core 3.0:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

</Project>
```

The updated ASP.NET Core 3.0 project file:

- In the `<PropertyGroup>`:

    - Updates the TFM to `netcoreapp3.0`
    - Removes the `<AspNetCoreHostingModel>` element. For more information, see In-process hosting model in this document.

- In the `<ItemGroup>`:
    - `Microsoft.AspNetCore.App` is removed. For more information, see Framework reference in this document.
    - `Microsoft.AspNetCore.Razor.Design` is removed and in the following list of packages no longer being produced.

To see the full list of packages that are no longer produced, select the following expand list:

▼ Click to expand the list of packages no longer being produced

- Microsoft.AspNetCore
- Microsoft.AspNetCore.All
- Microsoft.AspNetCore.App
- Microsoft.AspNetCore.Antiforgery
- Microsoft.AspNetCore.Authentication
- Microsoft.AspNetCore.Authentication.Abstractions
- Microsoft.AspNetCore.Authentication.Cookies
- Microsoft.AspNetCore.Authentication.Core
- Microsoft.AspNetCore.Authentication.OAuth
- Microsoft.AspNetCore.Authorization.Policy
- Microsoft.AspNetCore.CookiePolicy
- Microsoft.AspNetCore.Cors
- Microsoft.AspNetCore.Diagnostics
- Microsoft.AspNetCore.Diagnostics.HealthChecks
- Microsoft.AspNetCore.HostFiltering
- Microsoft.AspNetCore.Hosting
- Microsoft.AspNetCore.Hosting.Abstractions
- Microsoft.AspNetCore.Hosting.Server.Abstractions
- Microsoft.AspNetCore.Http
- Microsoft.AspNetCore.Http.Abstractions
- Microsoft.AspNetCore.Http.Connections
- Microsoft.AspNetCore.Http.Extensions
- Microsoft.AspNetCore.HttpOverrides
- Microsoft.AspNetCore.HttpsPolicy
- Microsoft.AspNetCore.Identity
- Microsoft.AspNetCore.Localization
- Microsoft.AspNetCore.Localization.Routing

- Microsoft.AspNetCore.Mvc
- Microsoft.AspNetCore.Mvc.Abstractions
- Microsoft.AspNetCore.Mvc.Analyzers
- Microsoft.AspNetCore.Mvc.ApiExplorer
- Microsoft.AspNetCore.Mvc.Api.Analyzers
- Microsoft.AspNetCore.Mvc.Core
- Microsoft.AspNetCore.Mvc.Cors
- Microsoft.AspNetCore.Mvc.DataAnnotations
- Microsoft.AspNetCore.Mvc.Formatters.Json
- Microsoft.AspNetCore.Mvc.Formatters.Xml
- Microsoft.AspNetCore.Mvc.Localization
- Microsoft.AspNetCore.Mvc.Razor
- Microsoft.AspNetCore.Mvc.Razor.ViewCompilation
- Microsoft.AspNetCore.Mvc.RazorPages
- Microsoft.AspNetCore.Mvc.TagHelpers
- Microsoft.AspNetCore.Mvc.ViewFeatures
- Microsoft.AspNetCore.Razor
- Microsoft.AspNetCore.Razor.Runtime
- Microsoft.AspNetCore.Razor.Design
- Microsoft.AspNetCore.ResponseCaching
- Microsoft.AspNetCore.ResponseCaching.Abstractions
- Microsoft.AspNetCore.ResponseCompression
- Microsoft.AspNetCore.Rewrite
- Microsoft.AspNetCore.Routing
- Microsoft.AspNetCore.Routing.Abstractions
- Microsoft.AspNetCore.Server.HttpSys
- Microsoft.AspNetCore.Server.IIS
- Microsoft.AspNetCore.Server.IISIntegration
- Microsoft.AspNetCore.Server.Kestrel
- Microsoft.AspNetCore.Server.Kestrel.Core
- Microsoft.AspNetCore.Server.Kestrel.Https
- Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions
- Microsoft.AspNetCore.Server.Kestrel.Transport.Sockets
- Microsoft.AspNetCore.Session
- Microsoft.AspNetCore.SignalR
- Microsoft.AspNetCore.SignalR.Core
- Microsoft.AspNetCore.StaticFiles
- Microsoft.AspNetCore.WebSockets
- Microsoft.AspNetCore.WebUtilities
- Microsoft.Net.Http.Headers

Review breaking changes

Review breaking changes

Framework reference

Features of ASP.NET Core that were available through one of the packages listed above are available as part of the `Microsoft.AspNetCore.App` shared framework. The *shared framework* is the set of assemblies (*.dll* files) that are installed on the machine and includes a runtime component and a targeting pack. For more information, see The shared framework.

- Projects that target the `Microsoft.NET.Sdk.Web` SDK implicitly reference the `Microsoft.AspNetCore.App` framework.

  No additional references are required for these projects:

  XML

  ```
  <Project Sdk="Microsoft.NET.Sdk.Web">
    <PropertyGroup>
      <TargetFramework>netcoreapp3.0</TargetFramework>
    </PropertyGroup>
      ...
  </Project>
  ```

- Projects that target `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Razor` SDK, should add an explicit `FrameworkReference` to `Microsoft.AspNetCore.App`:

  XML

  ```
  <Project Sdk="Microsoft.NET.Sdk.Razor">
    <PropertyGroup>
      <TargetFramework>netcoreapp3.0</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
      <FrameworkReference Include="Microsoft.AspNetCore.App" />
    </ItemGroup>
      ...
  </Project>
  ```

Framework-dependent builds using Docker

Framework-dependent builds of console apps that use a package that depends on the ASP.NET Core shared framework may give the following runtime error:

Console

```
It was not possible to find any compatible framework version
The specified framework 'Microsoft.AspNetCore.App', version '3.0.0' was not
found.
  - No frameworks were found.
```

`Microsoft.AspNetCore.App` is the shared framework containing the ASP.NET Core runtime and is only present on the `dotnet/core/aspnet` Docker image. The 3.0 SDK reduces the size of framework-dependent builds using ASP.NET Core by not including duplicate copies of libraries that are available in the shared framework. This is a potential savings of up to 18 MB, but it requires that the ASP.NET Core runtime be present / installed to run the app.

To determine if the app has a dependency (either direct or indirect) on the ASP.NET Core shared framework, examine the `runtimeconfig.json` file generated during a build/publish of your app. The following JSON file shows a dependency on the ASP.NET Core shared framework:

JSON

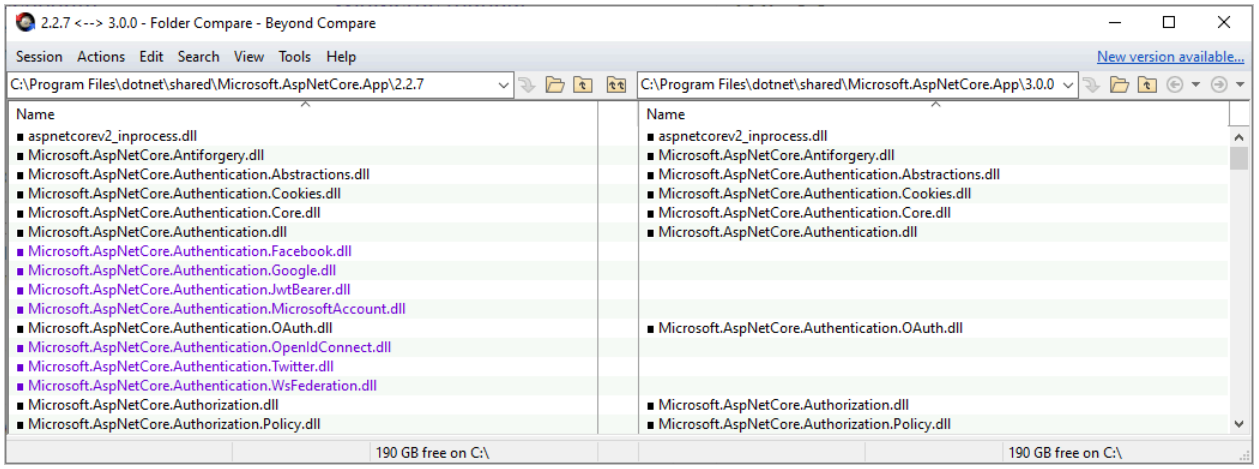
```
{
  "runtimeOptions": {
    "tfm": "netcoreapp3.0",
    "framework": {
      "name": "Microsoft.AspNetCore.App",
      "version": "3.0.0"
    },
    "configProperties": {
      "System.GC.Server": true
    }
  }
}
```

If your app is using Docker, use a base image that includes ASP.NET Core 3.0. For example, `docker pull mcr.microsoft.com/dotnet/core/aspnet:3.0`.

Add package references for removed assemblies

ASP.NET Core 3.0 removes some assemblies that were previously part of the `Microsoft.AspNetCore.App` package reference. To visualize which assemblies were

removed, compare the two shared framework folders. For example, a comparison of versions 2.2.7 and 3.0.0:



To continue using features provided by the removed assemblies, reference the 3.0 versions of the corresponding packages:

- A template-generated web app with **Individual User Accounts** requires adding the following packages:

  XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UserSecretsId>My-secret</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore"
Version="3.0.0" />
    <PackageReference
Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
Version="3.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Identity.UI"
Version="3.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="3.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="3.0.0" />
  </ItemGroup>

</Project>
```

- Microsoft.EntityFrameworkCore

For more information on referencing the database provider-specific package, see Database Providers.

- Identity UI

  Support for Identity UI can be added by referencing the Microsoft.AspNetCore.Identity.UI package.

- SPA Services
  - Microsoft.AspNetCore.SpaServices
  - Microsoft.AspNetCore.SpaServices.Extensions

- Authentication: Support for third-party authentication flows are available as NuGet packages:
  - Facebook OAuth (Microsoft.AspNetCore.Authentication.Facebook)
  - Google OAuth (Microsoft.AspNetCore.Authentication.Google)
  - Microsoft Account authentication (Microsoft.AspNetCore.Authentication.MicrosoftAccount)
  - OpenID Connect authentication (Microsoft.AspNetCore.Authentication.OpenIdConnect)
  - OpenID Connect bearer token (Microsoft.AspNetCore.Authentication.JwtBearer)
  - Twitter OAuth (Microsoft.AspNetCore.Authentication.Twitter)
  - WsFederation authentication (Microsoft.AspNetCore.Authentication.WsFederation)

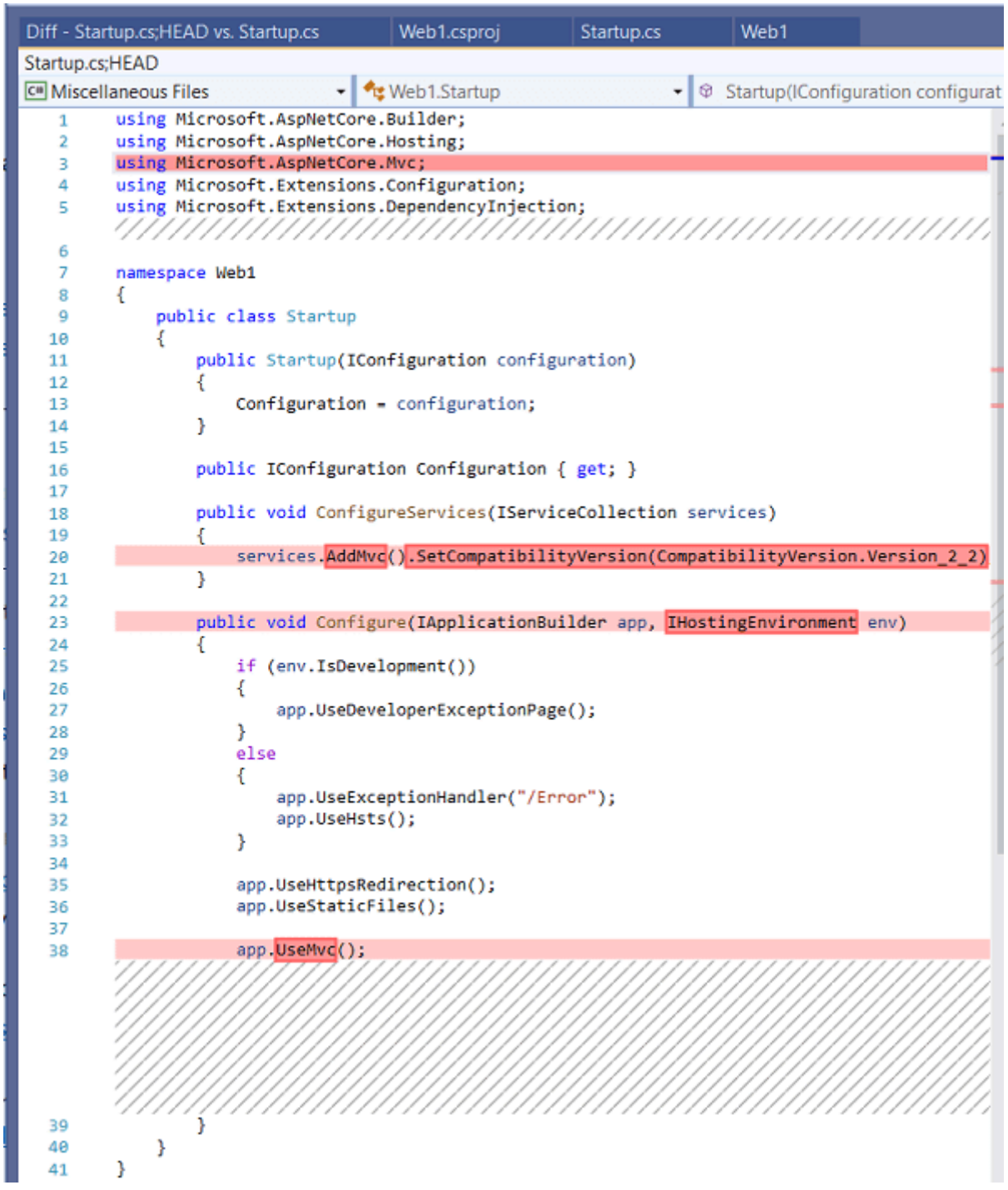
- Formatting and content negotiation support for `System.Net.HttpClient`: The Microsoft.AspNet.WebApi.Client NuGet package provides useful extensibility to `System.Net.HttpClient` with APIs such as `ReadAsAsync` and `PostJsonAsync`. However, this package depends on `Newtonsoft.Json`, not `System.Text.Json`. That means, for example, that serialization property names specified by `JsonPropertyNameAttribute` (`System.Text.Json`) are ignored. There's a newer NuGet package that contains similar extension methods but uses `System.Text.Json`: System.Net.Http.Json.

- Razor runtime compilation: Support for runtime compilation of Razor views and pages is now part of Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation.

- MVC `Newtonsoft.Json` (Json.NET) support: Support for using MVC with `Newtonsoft.Json` is now part of Microsoft.AspNetCore.Mvc.NewtonsoftJson.

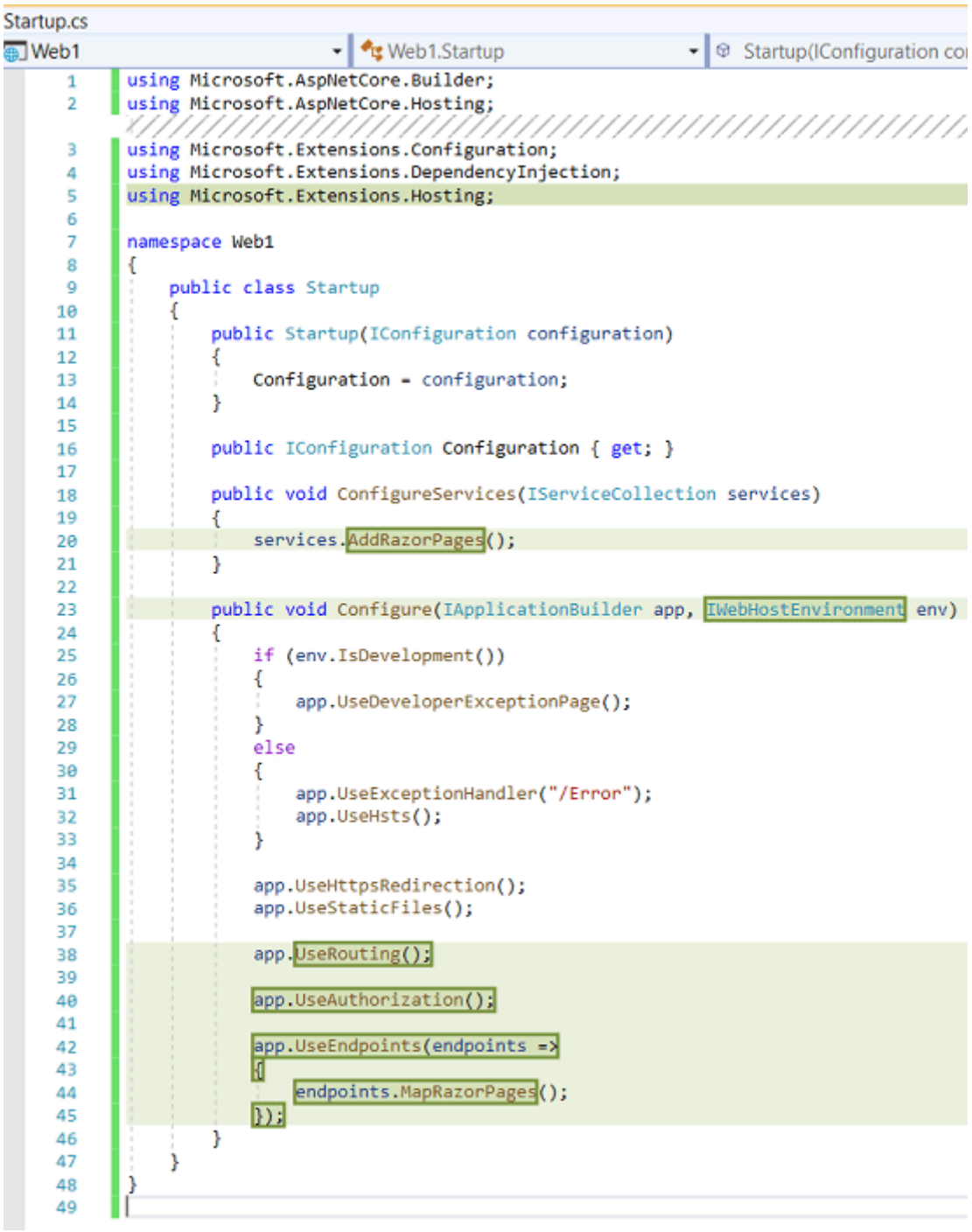
Startup changes

The following image shows the deleted and changed lines in an ASP.NET Core 2.2 Razor Pages Web app:



In the preceding image, deleted code is shown in red. The deleted code doesn't show cookie options code, which was deleted prior to comparing the files.

The following image shows the added and changed lines in an ASP.NET Core 3.0 Razor Pages Web app:



In the preceding image, added code is shown in green. For information on the following changes:

- `services.AddMvc` to `services.AddRazorPages`, see MVC service registration in this document.
- `CompatibilityVersion`, see Compatibility version for ASP.NET Core MVC.
- `IHostingEnvironment` to `IWebHostEnvironment`, see this GitHub announcement.
- `app.UseAuthorization` was added to the templates to show the order authorization middleware must be added. If the app doesn't use authorization, you can safely remove the call to `app.UseAuthorization`.
- `app.UseEndpoints`, see Razor Pages or Migrate Startup.Configure in this document.

Analyzer support

Projects that target `Microsoft.NET.Sdk.Web` implicitly reference analyzers previously shipped as part of the Microsoft.AspNetCore.Mvc.Analyzers package. No additional references are required to enable these.

If your app uses API analyzers previously shipped using the Microsoft.AspNetCore.Mvc.Api.Analyzers package, edit your project file to reference the analyzers shipped as part of the .NET Core Web SDK:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
    <PropertyGroup>
        <TargetFramework>netcoreapp3.0</TargetFramework>
        <IncludeOpenAPIAnalyzers>true</IncludeOpenAPIAnalyzers>
    </PropertyGroup>

    ...
</Project>
```

Razor class library

Razor class library projects that provide UI components for MVC must set the `AddRazorSupportForMvc` property in the project file:

XML

```
<PropertyGroup>
  <AddRazorSupportForMvc>true</AddRazorSupportForMvc>
</PropertyGroup>
```

In-process hosting model

Projects default to the in-process hosting model in ASP.NET Core 3.0 or later. You may optionally remove the `<AspNetCoreHostingModel>` property in the project file if its value is `InProcess`.

Kestrel

Configuration

Migrate Kestrel configuration to the web host builder provided by `ConfigureWebHostDefaults` (`Program.cs`):

C#

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(serverOptions =>
            {
                // Set properties and call methods on options
            })
            .UseStartup<Startup>();
        });
```

If the app creates the host manually with `ConfigureWebHost` instead of `ConfigureWebHostDefaults`, call `UseKestrel` on the web host builder:

C#

```
public static void Main(string[] args)
{
    var host = new HostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder.UseKestrel(serverOptions =>
            {
                // Set properties and call methods on options
            })
            .UseIISIntegration()
            .UseStartup<Startup>();
        })
        .Build();

    host.Run();
}
```

Connection Middleware replaces Connection Adapters

Connection Adapters (`Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal.IConnectionAdapter`) have been removed from Kestrel. Replace Connection Adapters with Connection Middleware. Connection Middleware is similar to HTTP Middleware in the ASP.NET Core pipeline but for lower-level connections. HTTPS and connection logging:

- Have been moved from Connection Adapters to Connection Middleware.
- These extension methods work as in previous versions of ASP.NET Core.

For more information, see the TlsFilterConnectionHandler example in the ListenOptions.Protocols section of the Kestrel article.

Transport abstractions moved and made public

The Kestrel transport layer has been exposed as a public interface in `Connections.Abstractions`. As part of these updates:

- `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions` and associated types have been removed.
- NoDelay was moved from ListenOptions to the transport options.
- `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions.Internal.SchedulingMode` was removed from KestrelServerOptions.

For more information, see the following GitHub resources:

- Client/server networking abstractions (dotnet/AspNetCore #10308)
- Implement new bedrock listener abstraction and re-plat Kestrel on top (dotnet/AspNetCore #10321)

Kestrel Request trailer headers

For apps that target earlier versions of ASP.NET Core:

- Kestrel adds HTTP/1.1 chunked trailer headers into the request headers collection.
- Trailers are available after the request body is read to the end.

This causes some concerns about ambiguity between headers and trailers, so the trailers have been moved to a new collection (`RequestTrailerExtensions`) in 3.0.

HTTP/2 request trailers are:

- Not available in ASP.NET Core 2.2.
- Available in 3.0 as `RequestTrailerExtensions`.

New request extension methods are present to access these trailers. As with HTTP/1.1, trailers are available after the request body is read to the end.

For the 3.0 release, the following `RequestTrailerExtensions` methods are available:

- `GetDeclaredTrailers`: Gets the request `Trailer` header that lists which trailers to expect after the body.
- `SupportsTrailers`: Indicates if the request supports receiving trailer headers.
- `CheckTrailersAvailable`: Checks if the request supports trailers and if they're available to be read. This check doesn't assume that there are trailers to read. There might be no trailers to read even if `true` is returned by this method.
- `GetTrailer`: Gets the requested trailing header from the response. Check `SupportsTrailers` before calling `GetTrailer`, or a NotSupportedException may occur if the request doesn't support trailing headers.

For more information, see Put request trailers in a separate collection (dotnet/AspNetCore #10410).

AllowSynchronousIO disabled

`AllowSynchronousIO` enables or disables synchronous I/O APIs, such as `HttpRequest.Body.Read`, `HttpResponse.Body.Write`, and `Stream.Flush`. These APIs are a source of thread starvation leading to app crashes. In 3.0, `AllowSynchronousIO` is disabled by default. For more information, see the Synchronous I/O section in the Kestrel article.

If synchronous I/O is needed, it can be enabled by configuring the `AllowSynchronousIO` option on the server being used (when calling `ConfigureKestrel`, for example, if using Kestrel). Note that servers (Kestrel, HttpSys, TestServer, etc.) all have their own `AllowSynchronousIO` option that won't affect other servers. Synchronous I/O can be enabled for all servers on a per-request basis using the `IHttpBodyControlFeature.AllowSynchronousIO` option:

C#

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();

if (syncIOFeature != null)
{
    syncIOFeature.AllowSynchronousIO = true;
}
```

If you have trouble with TextWriter implementations or other streams that call synchronous APIs in Dispose, call the new DisposeAsync API instead.

For more information, see [Announcement] AllowSynchronousIO disabled in all servers (dotnet/AspNetCore #7644).

Output formatter buffering

Newtonsoft.Json, XmlSerializer, and DataContractSerializer based output formatters only support synchronous serialization. To allow these formatters to work with the AllowSynchronousIO restrictions of the server, MVC buffers the output of these formatters before writing to disk. As a result of buffering, MVC will include the Content-Length header when responding using these formatters.

System.Text.Json supports asynchronous serialization and consequently the `System.Text.Json` based formatter does not buffer. Consider using this formatter for improved performance.

To disable buffering, applications can configure SuppressOutputFormatterBuffering in their startup:

```
C#

services.AddControllers(options => options.SuppressOutputFormatterBuffering
= true)
```

Note that this may result in the application throwing a runtime exception if `AllowSynchronousIO` isn't also configured.

Microsoft.AspNetCore.Server.Kestrel.Https assembly removed

In ASP.NET Core 2.1, the contents of *Microsoft.AspNetCore.Server.Kestrel.Https.dll* were moved to *Microsoft.AspNetCore.Server.Kestrel.Core.dll*. This was a non-breaking update using `TypeForwardedTo` attributes. For 3.0, the empty *Microsoft.AspNetCore.Server.Kestrel.Https.dll* assembly and the NuGet package have been removed.

Libraries referencing Microsoft.AspNetCore.Server.Kestrel.Https should update ASP.NET Core dependencies to 2.1 or later.

Apps and libraries targeting ASP.NET Core 2.1 or later should remove any direct references to the Microsoft.AspNetCore.Server.Kestrel.Https package.

Newtonsoft.Json (Json.NET) support

As part of the work to improve the ASP.NET Core shared framework, Newtonsoft.Json (Json.NET) has been removed from the ASP.NET Core shared framework.

The default JSON serializer for ASP.NET Core is now System.Text.Json, which is new in .NET Core 3.0. Consider using `System.Text.Json` when possible. It's high-performance and doesn't require an additional library dependency. However, since `System.Text.Json` is new, it might currently be missing features that your app needs. For more information, see How to migrate from Newtonsoft.Json to System.Text.Json.

Use Newtonsoft.Json in an ASP.NET Core 3.0 SignalR project

- Install the Microsoft.AspNetCore.SignalR.Protocols.NewtonsoftJson NuGet package.
- On the client, chain an `AddNewtonsoftJsonProtocol` method call to the `HubConnectionBuilder` instance:

```
C#

new HubConnectionBuilder()
    .WithUrl("/chathub")
    .AddNewtonsoftJsonProtocol(...)
    .Build();
```

- On the server, chain an `AddNewtonsoftJsonProtocol` method call to the `AddSignalR` method call in `Startup.ConfigureServices`:

```
C#

services.AddSignalR()
    .AddNewtonsoftJsonProtocol(...);
```

Use Newtonsoft.Json in an ASP.NET Core 3.0 MVC project

- Install the Microsoft.AspNetCore.Mvc.NewtonsoftJson package.
- Update `Startup.ConfigureServices` to call `AddNewtonsoftJson`.

```
C#

services.AddMvc()
    .AddNewtonsoftJson();
```

`AddNewtonsoftJson` is compatible with the new MVC service registration methods:

- `AddRazorPages`
- `AddControllersWithViews`
- `AddControllers`

```
C#

services.AddControllers()
    .AddNewtonsoftJson();
```

`Newtonsoft.Json` settings can be set in the call to `AddNewtonsoftJson`:

```
C#

services.AddMvc()
    .AddNewtonsoftJson(options =>
           options.SerializerSettings.ContractResolver =
              new CamelCasePropertyNamesContractResolver());
```

Note: If the `AddNewtonsoftJson` method isn't available, make sure that you installed the Microsoft.AspNetCore.Mvc.NewtonsoftJson package. A common error is to install the Newtonsoft.Json package instead of the Microsoft.AspNetCore.Mvc.NewtonsoftJson package.

For more information, see Add Newtonsoft.Json-based JSON format support.

MVC service registration

ASP.NET Core 3.0 adds new options for registering MVC scenarios inside `Startup.ConfigureServices`.

Three new top-level extension methods related to MVC scenarios on `IServiceCollection` are available. Templates use these new methods instead of `AddMvc`. However, `AddMvc` continues to behave as it has in previous releases.

The following example adds support for controllers and API-related features, but not views or pages. The API template uses this code:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
}
```

The following example adds support for controllers, API-related features, and views, but not pages. The Web Application (MVC) template uses this code:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
}
```

The following example adds support for Razor Pages and minimal controller support. The Web Application template uses this code:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
}
```

The new methods can also be combined. The following example is equivalent to calling `AddMvc` in ASP.NET Core 2.2:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddRazorPages();
}
```

Routing startup code

If an app calls `UseMvc` or `UseSignalR`, migrate the app to Endpoint Routing if possible. To improve Endpoint Routing compatibility with previous versions of MVC, we've reverted some of the changes in URL generation introduced in ASP.NET Core 2.2. If you experienced problems using Endpoint Routing in 2.2, expect improvements in ASP.NET Core 3.0 with the following exceptions:

- If the app implements `IRouter` or inherits from `Route`, use DynamicRouteValuesTransformer as the replacement.
- If the app directly accesses `RouteData.Routers` inside MVC to parse URLs, you can replace this with use of LinkParser.ParsePathByEndpointName.
  - Define the route with a route name.

- Use `LinkParser.ParsePathByEndpointName` and pass in the desired route name.

Endpoint Routing supports the same route pattern syntax and route pattern authoring features as `IRouter`. Endpoint Routing supports `IRouteConstraint`. Endpoint routing supports `[Route]`, `[HttpGet]`, and the other MVC routing attributes.

For most applications, only `Startup` requires changes.

Migrate Startup.Configure

General advice:

- Add `UseRouting`.
- If the app calls `UseStaticFiles`, place `UseStaticFiles` **before** `UseRouting`.
- If the app uses authentication/authorization features such as `AuthorizePage` or `[Authorize]`, place the call to `UseAuthentication` and `UseAuthorization`: **after**, `UseRouting` and `UseCors`, but before `UseEndpoints`:

```
C#
```

```
public void Configure(IApplicationBuilder app)
{
  ...

  app.UseStaticFiles();

  app.UseRouting();
  app.UseCors();

  app.UseAuthentication();
  app.UseAuthorization();

  app.UseEndpoints(endpoints => {
     endpoints.MapControllers();
  });
```

- Replace `UseMvc` or `UseSignalR` with `UseEndpoints`.
- If the app uses CORS scenarios, such as `[EnableCors]`, place the call to `UseCors` before any other middleware that use CORS (for example, place `UseCors` before `UseAuthentication`, `UseAuthorization`, and `UseEndpoints`).
- Replace `IHostingEnvironment` with `IWebHostEnvironment` and add a `using` statement for the Microsoft.AspNetCore.Hosting namespace.

- Replace `IApplicationLifetime` with IHostApplicationLifetime (Microsoft.Extensions.Hosting namespace).
- Replace `EnvironmentName` with Environments (Microsoft.Extensions.Hosting namespace).

The following code is an example of `Startup.Configure` in a typical ASP.NET Core 2.2 app:

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    app.UseAuthentication();

    app.UseSignalR(hubs =>
    {
        hubs.MapHub<ChatHub>("/chat");
    });

    app.UseMvc(routes =>
    {
        routes.MapRoute("default", "
{controller=Home}/{action=Index}/{id?}");
    });
}
```

After updating the previous `Startup.Configure` code:

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseStaticFiles();

    app.UseRouting();

    app.UseCors();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>("/chat");
```

```
        endpoints.MapControllerRoute("default", "
{controller=Home}/{action=Index}/{id?}");
    });
}
```

> ⚠ **Warning**
>
> For most apps, calls to `UseAuthentication`, `UseAuthorization`, and `UseCors` must appear between the calls to `UseRouting` and `UseEndpoints` to be effective.

Health Checks

Health Checks use endpoint routing with the Generic Host. In `Startup.Configure`, call `MapHealthChecks` on the endpoint builder with the endpoint URL or relative path:

C#

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHealthChecks("/health");
});
```

Health Checks endpoints can:

- Specify one or more permitted hosts/ports.
- Require authorization.
- Require CORS.

For more information, see Health checks in ASP.NET Core.

Security middleware guidance

Support for authorization and CORS is unified around the middleware approach. This allows use of the same middleware and functionality across these scenarios. An updated authorization middleware is provided in this release, and CORS Middleware is enhanced so that it can understand the attributes used by MVC controllers.

CORS

Previously, CORS could be difficult to configure. Middleware was provided for use in some use cases, but MVC filters were intended to be used **without** the middleware in other use cases. With ASP.NET Core 3.0, we recommend that all apps that require CORS

use the CORS Middleware in tandem with Endpoint Routing. `UseCors` can be provided with a default policy, and `[EnableCors]` and `[DisableCors]` attributes can be used to override the default policy where required.

In the following example:

- CORS is enabled for all endpoints with the `default` named policy.
- The `MyController` class disables CORS with the `[DisableCors]` attribute.

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseCors("default");

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

[DisableCors]
public class MyController : ControllerBase
{
    ...
}
```

Authorization

In earlier versions of ASP.NET Core, authorization support was provided via the `[Authorize]` attribute. Authorization middleware wasn't available. In ASP.NET Core 3.0, authorization middleware is required. We recommend placing the ASP.NET Core Authorization Middleware (`UseAuthorization`) immediately after `UseAuthentication`. The Authorization Middleware can also be configured with a default policy, which can be overridden.

In ASP.NET Core 3.0 or later, `UseAuthorization` is called in `Startup.Configure`, and the following `HomeController` requires a signed in user:

C#

```
public void Configure(IApplicationBuilder app)
{
```

```
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

public class HomeController : Controller
{
    [Authorize]
    public IActionResult BuyWidgets()
    {
        ...
    }
}
```

When using endpoint routing, we recommend against configuring AuthorizeFilter and instead relying on the Authorization middleware. If the app uses an `AuthorizeFilter` as a global filter in MVC, we recommend refactoring the code to provide a policy in the call to `AddAuthorization`.

The `DefaultPolicy` is initially configured to require authentication, so no additional configuration is required. In the following example, MVC endpoints are marked as `RequireAuthorization` so that all requests must be authorized based on the `DefaultPolicy`. However, the `HomeController` allows access without the user signing into the app due to `[AllowAnonymous]`:

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute().RequireAuthorization();
    });
}
```

```
[AllowAnonymous]
public class HomeController : Controller
{
    ...
}
```

Authorization for specific endpoints

Authorization can also be configured for specific classes of endpoints. The following code is an example of converting an MVC app that configured a global `AuthorizeFilter` to an app with a specific policy requiring authorization:

C#

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    static readonly string _RequireAuthenticatedUserPolicy =
                            "RequireAuthenticatedUserPolicy";
    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(
                 options => options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();

        // Pre 3.0:
        // services.AddMvc(options => options.Filters.Add(new
AuthorizeFilter(...));

        services.AddControllersWithViews();
        services.AddRazorPages();
        services.AddAuthorization(o =>
o.AddPolicy(_RequireAuthenticatedUserPolicy,
                        builder => builder.RequireAuthenticatedUser()));

    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
```

```
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute()
                .RequireAuthorization(_RequireAuthenticatedUserPolicy);
            endpoints.MapRazorPages();
        });
    }
}
```

Policies can also be customized. The `DefaultPolicy` is configured to require authentication:

C#

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("DefaultConnection")));
        services.AddDefaultIdentity<IdentityUser>(
                 options => options.SignIn.RequireConfirmedAccount = true)
            .AddEntityFrameworkStores<ApplicationDbContext>();

        services.AddControllersWithViews();
        services.AddRazorPages();
```

```
        services.AddAuthorization(options =>
        {
            options.DefaultPolicy = new AuthorizationPolicyBuilder()
              .RequireAuthenticatedUser()
              .Build();
        });

    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseDatabaseErrorPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute().RequireAuthorization();
            endpoints.MapRazorPages();
        });
    }
}
```

C#

```
[AllowAnonymous]
public class HomeController : Controller
{
```

Alternatively, all endpoints can be configured to require authorization without `[Authorize]` or `RequireAuthorization` by configuring a `FallbackPolicy`. The `FallbackPolicy` is different from the `DefaultPolicy`. The `DefaultPolicy` is triggered by `[Authorize]` or `RequireAuthorization`, while the `FallbackPolicy` is triggered when no other policy is set. `FallbackPolicy` is initially configured to allow requests without authorization.

The following example is the same as the preceding `DefaultPolicy` example but uses the `FallbackPolicy` to always require authentication on all endpoints except when `[AllowAnonymous]` is specified:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddAuthorization(options =>
    {
        options.FallbackPolicy = new AuthorizationPolicyBuilder()
          .RequireAuthenticatedUser()
          .Build();
    });
}

public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
    });
}

[AllowAnonymous]
public class HomeController : Controller
{
    ...
}
```

Authorization by middleware works without the framework having any specific knowledge of authorization. For instance, health checks has no specific knowledge of authorization, but health checks can have a configurable authorization policy applied by the middleware.

Additionally, each endpoint can customize its authorization requirements. In the following example, `UseAuthorization` processes authorization with the `DefaultPolicy`, but the `/healthz` health check endpoint requires an `admin` user:

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints
            .MapHealthChecks("/healthz")
            .RequireAuthorization(new AuthorizeAttribute(){ Roles = "admin",
});
    });
}
```

Protection is implemented for some scenarios. Endpoints Middleware throws an exception if an authorization or CORS policy is skipped due to missing middleware. Analyzer support to provide additional feedback about misconfiguration is in progress.

Custom authorization handlers

If the app uses custom authorization handlers, endpoint routing passes a different resource type to handlers than MVC. Handlers that expect the authorization handler context resource to be of type AuthorizationFilterContext (the resource type provided by MVC filters) will need to be updated to handle resources of type RouteEndpoint (the resource type given to authorization handlers by endpoint routing).

MVC still uses `AuthorizationFilterContext` resources, so if the app uses MVC authorization filters along with endpoint routing authorization, it may be necessary to handle both types of resources.

SignalR

Mapping of SignalR hubs now takes place inside `UseEndpoints`.

Map each hub with `MapHub`. As in previous versions, each hub is explicitly listed.

In the following example, support for the `ChatHub` SignalR hub is added:

C#

```
public void Configure(IApplicationBuilder app)
{
```

```
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHub<ChatHub>();
    });
}
```

There is a new option for controlling message size limits from clients. For example, in `Startup.ConfigureServices`:

C#

```
services.AddSignalR(hubOptions =>
{
    hubOptions.MaximumReceiveMessageSize = 32768;
});
```

In ASP.NET Core 2.2, you could set the `TransportMaxBufferSize` and that would effectively control the maximum message size. In ASP.NET Core 3.0, that option now only controls the maximum size before backpressure is observed.

SignalR assemblies in shared framework

ASP.NET Core SignalR server-side assemblies are now installed with the .NET Core SDK. For more information, see Remove obsolete package references in this document.

MVC controllers

Mapping of controllers now takes place inside `UseEndpoints`.

Add `MapControllers` if the app uses attribute routing. Since routing includes support for many frameworks in ASP.NET Core 3.0 or later, adding attribute-routed controllers is opt-in.

Replace the following:

- `MapRoute` with `MapControllerRoute`
- `MapAreaRoute` with `MapAreaControllerRoute`

Since routing now includes support for more than just MVC, the terminology has changed to make these methods clearly state what they do. Conventional routes such as

`MapControllerRoute`/`MapAreaControllerRoute`/`MapDefaultControllerRoute` are applied in the order that they're added. Place more specific routes (such as routes for an area) first.

In the following example:

- `MapControllers` adds support for attribute-routed controllers.
- `MapAreaControllerRoute` adds a conventional route for controllers in an area.
- `MapControllerRoute` adds a conventional route for controllers.

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapAreaControllerRoute(
            "admin",
            "admin",
            "Admin/{controller=Home}/{action=Index}/{id?}");
        endpoints.MapControllerRoute(
            "default", "{controller=Home}/{action=Index}/{id?}");
    });
}
```

Async suffix removal from controller action names

In ASP.NET Core 3.0, ASP.NET Core MVC removes the `Async` suffix from controller action names. Both routing and link generation are impacted by this new default. For example:

C#

```
public class ProductsController : Controller
{
    public async Task<IActionResult> ListAsync()
    {
        var model = await _dbContext.Products.ToListAsync();
        return View(model);
    }
}
```

Prior to ASP.NET Core 3.0:

- The preceding action could be accessed at the *Products/ListAsync* route.

- Link generation required specifying the `Async` suffix. For example:

```
CSHTML
<a asp-controller="Products" asp-action="ListAsync">List</a>
```

In ASP.NET Core 3.0:

- The preceding action can be accessed at the *Products/List* route.
- Link generation doesn't require specifying the `Async` suffix. For example:

```
CSHTML
<a asp-controller="Products" asp-action="List">List</a>
```

This change doesn't affect names specified using the [ActionName] attribute. The default behavior can be disabled with the following code in `Startup.ConfigureServices`:

```
C#
services.AddMvc(options =>
    options.SuppressAsyncSuffixInActionNames = false);
```

Changes to link generation

There are some differences in link generation (using `Url.Link` and similar APIs, for example). These include:

- By default, when using endpoint routing, casing of route parameters in generated URIs is not necessarily preserved. This behavior can be controlled with the `IOutboundParameterTransformer` interface.
- Generating a URI for an invalid route (a controller/action or page that doesn't exist) will produce an empty string under endpoint routing instead of producing an invalid URI.
- Ambient values (route parameters from the current context) are not automatically used in link generation with endpoint routing. Previously, when generating a link to another action (or page), unspecified route values would be inferred from the *current* routes ambient values. When using endpoint routing, all route parameters must be specified explicitly during link generation.

Razor Pages

Mapping Razor Pages now takes place inside `UseEndpoints`.

Add `MapRazorPages` if the app uses Razor Pages. Since Endpoint Routing includes support for many frameworks, adding Razor Pages is now opt-in.

In the following `Startup.Configure` method, `MapRazorPages` adds support for Razor Pages:

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Use MVC without Endpoint Routing

Using MVC via `UseMvc` or `UseMvcWithDefaultRoute` in ASP.NET Core 3.0 requires an explicit opt-in inside `Startup.ConfigureServices`. This is required because MVC must know whether it can rely on the authorization and CORS Middleware during initialization. An analyzer is provided that warns if the app attempts to use an unsupported configuration.

If the app requires legacy `IRouter` support, disable `EnableEndpointRouting` using any of the following approaches in `Startup.ConfigureServices`:

C#

```
services.AddMvc(options => options.EnableEndpointRouting = false);
```

C#

```
services.AddControllers(options => options.EnableEndpointRouting = false);
```

C#

```
services.AddControllersWithViews(options => options.EnableEndpointRouting =
```

```
false);
```

C#

```
services.AddRazorPages().AddMvcOptions(options =>
options.EnableEndpointRouting = false);
```

Health checks

Health checks can be used as a *router-ware* with Endpoint Routing.

Add `MapHealthChecks` to use health checks with Endpoint Routing. The `MapHealthChecks` method accepts arguments similar to `UseHealthChecks`. The advantage of using `MapHealthChecks` over `UseHealthChecks` is the ability to apply authorization and to have greater fine-grained control over the matching policy.

In the following example, `MapHealthChecks` is called for a health check endpoint at `/healthz`:

C#

```
public void Configure(IApplicationBuilder app)
{
    ...

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHealthChecks("/healthz", new HealthCheckOptions() { });
    });
}
```

HostBuilder replaces WebHostBuilder

The ASP.NET Core 3.0 templates use Generic Host. Previous versions used Web Host. The following code shows the ASP.NET Core 3.0 template generated `Program` class:

C#

```
// requires using Microsoft.AspNetCore.Hosting;
// requires using Microsoft.Extensions.Hosting;

public class Program
{
```

```
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code shows the ASP.NET Core 2.2 template-generated `Program` class:

C#

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

IWebHostBuilder remains in 3.0 and is the type of the `webBuilder` seen in the preceding code sample. WebHostBuilder will be deprecated in a future release and replaced by `HostBuilder`.

The most significant change from `WebHostBuilder` to `HostBuilder` is in dependency injection (DI). When using `HostBuilder`, you can only inject the following into `Startup`'s constructor:

- IConfiguration
- IHostEnvironment
- IWebHostEnvironment

The `HostBuilder` DI constraints:

- Enable the DI container to be built only one time.
- Avoids the resulting object lifetime issues like resolving multiple instances of singletons.

For more information, see Avoiding Startup service injection in ASP.NET Core 3 .

AddAuthorization moved to a different assembly

The ASP.NET Core 2.2 and lower `AddAuthorization` methods in *Microsoft.AspNetCore.Authorization.dll*:

- Have been renamed `AddAuthorizationCore`.
- Have been moved to *Microsoft.AspNetCore.Authorization.Policy.dll*.

Apps that are using both *Microsoft.AspNetCore.Authorization.dll* and *Microsoft.AspNetCore.Authorization.Policy.dll* aren't impacted.

Apps that are not using *Microsoft.AspNetCore.Authorization.Policy.dll* should do one of the following:

- Add a reference to *Microsoft.AspNetCore.Authorization.Policy.dll*. This approach works for most apps and is all that is required.
- Switch to using `AddAuthorizationCore`

For more information, see Breaking change in AddAuthorization(o =>) overload lives in a different assembly #386.

Identity UI

Identity UI updates for ASP.NET Core 3.0:

- Add a package reference to Microsoft.AspNetCore.Identity.UI.
- Apps that don't use Razor Pages must call `MapRazorPages`. See Razor Pages in this document.
- Bootstrap 4 is the default UI framework. Set an `IdentityUIFrameworkVersion` project property to change the default. For more information, see this GitHub announcement.

SignalR

The SignalR JavaScript client has changed from `@aspnet/signalr` to `@microsoft/signalr`. To react to this change, change the references in `package.json` files, `require` statements, and ECMAScript `import` statements.

System.Text.Json is the default protocol

`System.Text.Json` is now the default Hub protocol used by both the client and server.

In `Startup.ConfigureServices`, call `AddJsonProtocol` to set serializer options.

Server:

```
C#

services.AddSignalR(...)
        .AddJsonProtocol(options =>
        {
            options.PayloadSerializerOptions.WriteIndented = false;
        })
```

Client:

```
C#

new HubConnectionBuilder()
    .WithUrl("/chathub")
    .AddJsonProtocol(options =>
    {
        options.PayloadSerializerOptions.WriteIndented = false;
    })
    .Build();
```

Switch to Newtonsoft.Json

If you're using features of Newtonsoft.Json that aren't supported in System.Text.Json, you can switch back to `Newtonsoft.Json`. See Use Newtonsoft.Json in an ASP.NET Core 3.0 SignalR project earlier in this article.

Redis distributed caches

The Microsoft.Extensions.Caching.Redis package isn't available for ASP.NET Core 3.0 or later apps. Replace the package reference with Microsoft.Extensions.Caching.StackExchangeRedis. For more information, see Distributed caching in ASP.NET Core.

Opt in to runtime compilation

Prior to ASP.NET Core 3.0, runtime compilation of views was an implicit feature of the framework. Runtime compilation supplements build-time compilation of views. It allows

the framework to compile Razor views and pages (`.cshtml` files) when the files are modified, without having to rebuild the entire app. This feature supports the scenario of making a quick edit in the IDE and refreshing the browser to view the changes.

In ASP.NET Core 3.0, runtime compilation is an opt-in scenario. Build-time compilation is the only mechanism for view compilation that's enabled by default. The runtime relies on Visual Studio or dotnet-watch in Visual Studio Code to rebuild the project when it detects changes to `.cshtml` files. In Visual Studio, changes to `.cs`, `.cshtml`, or `.razor` files in the project being run (`Ctrl+F5`), but not debugged (`F5`), trigger recompilation of the project.

To enable runtime compilation in your ASP.NET Core 3.0 project:

1. Install the Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation NuGet package.
2. Update `Startup.ConfigureServices` to call `AddRazorRuntimeCompilation`:

   For ASP.NET Core MVC, use the following code:

   ```
   C#

   services.AddControllersWithViews()
       .AddRazorRuntimeCompilation(...);
   ```

   For ASP.NET Core Razor Pages, use the following code:

   ```
   C#

   services.AddRazorPages()
       .AddRazorRuntimeCompilation(...);
   ```

The sample at https://github.com/aspnet/samples/tree/main/samples/aspnetcore/mvc/runtimecompilation shows an example of enabling runtime compilation conditionally in Development environments.

For more information on Razor file compilation, see Razor file compilation in ASP.NET Core.

Migrate libraries via multi-targeting

Libraries often need to support multiple versions of ASP.NET Core. Most libraries that were compiled against previous versions of ASP.NET Core should continue working without issues. The following conditions require the app to be cross-compiled:

- The library relies on a feature that has a binary breaking change.
- The library wants to take advantage of new features in ASP.NET Core 3.0.

For example:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netcoreapp3.0;netstandard2.0</TargetFrameworks>
  </PropertyGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netcoreapp3.0'">
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

  <ItemGroup Condition="'$(TargetFramework)' == 'netstandard2.0'">
    <PackageReference Include="Microsoft.AspNetCore" Version="2.1.0" />
  </ItemGroup>
</Project>
```

Use `#ifdefs` to enable ASP.NET Core 3.0-specific APIs:

C#

```
var webRootFileProvider =
#if NETCOREAPP3_0
    GetRequiredService<IWebHostEnvironment>().WebRootFileProvider;
#elif NETSTANDARD2_0
    GetRequiredService<IHostingEnvironment>().WebRootFileProvider;
#else
#error unknown target framework
#endif
```

For more information on using ASP.NET Core APIs in a class library, see Use ASP.NET Core APIs in a class library.

Miscellaneous changes

The validation system in .NET Core 3.0 and later treats non-nullable parameters or bound properties as if they had a `[Required]` attribute. For more information, see [Required] attribute.

Publish

Delete the *bin* and *obj* folders in the project directory.

TestServer

For apps that use TestServer directly with the Generic Host, create the `TestServer` on an IWebHostBuilder in ConfigureWebHost:

C#

```
[Fact]
public async Task GenericCreateAndStartHost_GetTestServer()
{
    using var host = await new HostBuilder()
        .ConfigureWebHost(webBuilder =>
        {
            webBuilder
                .UseTestServer()
                .Configure(app => { });
        })
    .StartAsync();

    var response = await host.GetTestServer().CreateClient().GetAsync("/");

    Assert.Equal(HttpStatusCode.NotFound, response.StatusCode);
}
```

Breaking API changes

Review breaking changes:

- Complete list of breaking changes in the ASP.NET Core 3.0 release
- Breaking API changes in Antiforgery, CORS, Diagnostics, MVC, and Routing. This list includes breaking changes for compatibility switches.
- For a summary of 2.2-to-3.0 breaking changes across .NET Core, ASP.NET Core, and Entity Framework Core, see Breaking changes for migration from version 2.2 to 3.0.

Endpoint routing with catch-all parameter

.NET Core 3.0 on Azure App Service

The rollout of .NET Core to Azure App Service is finished. .NET Core 3.0 is available in all Azure App Service datacenters.

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Migrate from ASP.NET Core 2.1 to 2.2

Article • 06/03/2022

By Scott Addie

This article explains how to update an existing ASP.NET Core 2.1 project to ASP.NET Core 2.2.

Prerequisites

Visual Studio

- Visual Studio 2019 with the **ASP.NET and web development** workload
- .NET Core SDK 2.2 or later

> ⚠ **Warning**
>
> If you use Visual Studio 2017, see dotnet/sdk issue #3124 for information about .NET Core SDK versions that don't work with Visual Studio.

Update Target Framework Moniker (TFM)

Projects targeting .NET Core should use the TFM of a version greater than or equal to .NET Core 2.2. In the project file, update the `<TargetFramework>` node's inner text with `netcoreapp2.2`:

XML

```
<TargetFramework>netcoreapp2.2</TargetFramework>
```

Projects targeting .NET Framework may continue to use the TFM of a version greater than or equal to .NET Framework 4.6.1:

XML

```
<TargetFramework>net461</TargetFramework>
```

Adopt the IIS in-process hosting model

To adopt the in-process hosting model for IIS, add the `<AspNetCoreHostingModel>` property with a value of `InProcess` to a `<PropertyGroup>` in the project file:

XML

```
<AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
```

The in-process hosting model isn't supported for ASP.NET Core apps targeting .NET Framework.

For more information, see ASP.NET Core Module (ANCM) for IIS.

Update a custom web.config file

For projects that use a custom *web.config* file in the project root to generate their published *web.config* file:

- In the `<handlers>` entry that adds the ASP.NET Core Module (`name="aspNetCore"`), change the `modules` attribute value from `AspNetCoreModule` to `AspNetCoreModuleV2`.
- In the `<aspNetCore>` element, add the hosting model attribute (`hostingModel="InProcess"`).

For more information and example *web.config* files, see ASP.NET Core Module (ANCM) for IIS.

Update package references

If targeting .NET Core, remove the metapackage reference's `Version` attribute in the project file. Inclusion of a `Version` attribute results in the following warning:

Console

```
A PackageReference to 'Microsoft.AspNetCore.App' specified a Version of
`2.2.0`. Specifying the version of this package is not recommended. For more
information, see https://aka.ms/sdkimplicitrefs
```

For more information, see Microsoft.AspNetCore.App metapackage for ASP.NET Core.

The metapackage reference should resemble the following `<PackageReference />` node:

XML

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

If targeting .NET Framework, update each package reference's `Version` attribute to 2.2.0 or later. Here are the package references in a typical ASP.NET Core 2.2 project targeting .NET Framework:

XML

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.CookiePolicy"
Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.HttpsPolicy"
Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.2.0" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles"
Version="2.2.0" />
</ItemGroup>
```

If referencing the Microsoft.AspNetCore.Razor.Design package, update its `Version` attribute to 2.2.0 or later. Failure to do so results in the following error:

Console

```
Detected package downgrade: Microsoft.AspNetCore.Razor.Design from 2.2.0 to
2.1.2. Reference the package directly from the project to select a different
version.
```

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to the 2.2 version installed on your machine:

JSON

```json
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

Update launch settings

If using Visual Studio Code, update the project's launch settings file (`.vscode/launch.json`). The `program` path should reference the new TFM:

JSON

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/bin/Debug/netcoreapp2.2/test-
app.dll",
            "args": [],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "internalConsoleOptions": "openOnSessionStart",
            "launchBrowser": {
                "enabled": true,
                "args": "${auto-detect-url}",
                "windows": {
                    "command": "cmd.exe",
                    "args": "/C start ${auto-detect-url}"
                },
                "osx": {
                    "command": "open"
                },
                "linux": {
                    "command": "xdg-open"
                }
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "sourceFileMap": {
                "/Views": "${workspaceFolder}/Views"
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach",
            "processId": "${command:pickProcess}"
        }
    ]
}
```

Update Kestrel configuration

If the app calls UseKestrel by calling `CreateDefaultBuilder` in the CreateWebHostBuilder method of the `Program` class, call `ConfigureKestrel` to configure Kestrel server instead of `UseKestrel` in order to avoid conflicts with the IIS in-process hosting model:

C#

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        });
```

If the app doesn't call `CreateDefaultBuilder` and builds the host manually in the `Program` class, call UseKestrel **before** calling `ConfigureKestrel`:

C#

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseKestrel()
        .UseIISIntegration()
        .UseStartup<Startup>()
        .ConfigureKestrel((context, options) =>
        {
            // Set properties and call methods on options
        })
        .Build();

    host.Run();
}
```

For more information, see Kestrel web server in ASP.NET Core.

Update compatibility version

Update the compatibility version in `Startup.ConfigureServices` to `Version_2_2`:

C#

```
services.AddMvc()
```

```
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
```

Update CORS policy

In ASP.NET Core 2.2, the CORS middleware responds with a wildcard origin (`*`) if a policy allows any origin and allows credentials. Credentials aren't supported when a wildcard origin (`*`) is specified, and browsers will disallow the CORS request. For more information, including options for correcting the problem on the client, see the MDN web docs.

To correct this problem on the server, take one of the following actions:

- Modify the CORS policy to no longer allow credentials. That is, remove the call to AllowCredentials when configuring the policy.
- If credentials are required for the CORS request to succeed, modify the policy to specify allowed hosts. For example, use `builder.WithOrigins("https://api.example1.com", "https://example2.com")` instead of using AllowAnyOrigin.

Update Docker images

The following table shows the Docker image tag changes:

Expand table

2.1	2.2
`microsoft/dotnet:2.1-aspnetcore-runtime`	`mcr.microsoft.com/dotnet/core/aspnet:2.2`
`microsoft/dotnet:2.1-sdk`	`mcr.microsoft.com/dotnet/core/sdk:2.2`

Change the `FROM` lines in your *Dockerfile* to use the new image tags in the preceding table's 2.2 column.

Build manually in Visual Studio when using IIS in-process hosting

Visual Studio's **Auto build on browser request** experience doesn't function with the IIS in-process hosting model. You must manually rebuild the project when using in-process

hosting. Improvements to this experience are planned for a future release of Visual Studio.

Update logging code

Recommended logging configuration code didn't change from 2.1 to 2.2, but some 1.x coding patterns that still worked in 2.1 no longer work in 2.2.

If your app does logging provider initialization, filtering, and configuration loading in the `Startup` class, move that code to `Program.Main`:

- Provider initialization:

  1.x example:

  ```
  C#

  public void Configure(IApplicationBuilder app, ILoggerFactory
  loggerFactory)
  {
      loggerFactory.AddConsole();
  }
  ```

  2.2 example:

  ```
  C#


  public static void Main(string[] args)
  {
      var webHost = new WebHostBuilder()
          // ...
          .ConfigureLogging((hostingContext, logging) =>
          {
              logging.AddConsole();
          })
          // ...
  }
  ```

- Filtering:

  1.x example:

  ```
  C#

  public void Configure(IApplicationBuilder app, ILoggerFactory
  loggerFactory)
  ```

```
{
    loggerFactory.AddConsole(LogLevel.Information);
    // or
    loggerFactory.AddConsole((category, level) =>
        category == "A" || level == LogLevel.Critical);
}
```

2.2 example:

```C#
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        // ...
        .ConfigureLogging((hostingContext, logging) =>
        {
            logging.AddConsole()
                   .AddFilter<ConsoleLoggerProvider>
                       (category: null, level: LogLevel.Information)
                   // or
                   .AddFilter<ConsoleLoggerProvider>
                       ((category, level) => category == "A" ||
                           level == LogLevel.Critical)
            );
        })
        // ...
}
```

- Configuration loading:

  1.x example:

```C#
public void Configure(IApplicationBuilder app, ILoggerFactory
loggerFactory)
{
    loggerFactory.AddConsole(Configuration);
}
```

  2.2 example:

```C#
public static void Main(string[] args)
{
    var webHost = new WebHostBuilder()
        // ...
        .ConfigureLogging((hostingContext, logging) =>
```

```
        {
            logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
            logging.AddConsole();
        })
        // ...
}
```

For more information, see Logging in .NET Core and ASP.NET Core

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Additional resources

- Compatibility version for ASP.NET Core MVC
- Microsoft.AspNetCore.App metapackage for ASP.NET Core
- Implicit package references

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Migrate from Microsoft.Extensions.Logging 2.1 to 2.2 or 3.0

Article • 06/03/2022

This article outlines the common steps for migrating a non-ASP.NET Core application that uses `Microsoft.Extensions.Logging` from 2.1 to 2.2 or 3.0.

2.1 to 2.2

Manually create `ServiceCollection` and call `AddLogging`.

2.1 example:

C#

```
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

2.2 example:

C#

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddLogging(builder => builder.AddConsole());

using (var serviceProvider = serviceCollection.BuildServiceProvider())
using (var loggerFactory = serviceProvider.GetService<ILoggerFactory>())
{
    // use loggerFactory
}
```

2.1 to 3.0

In 3.0, use `LoggingFactory.Create`.

2.1 example:

C#

```
using (var loggerFactory = new LoggerFactory())
{
    loggerFactory.AddConsole();

    // use loggerFactory
}
```

3.0 example:

C#

```
using (var loggerFactory = LoggerFactory.Create(builder =>
builder.AddConsole()))
{
    // use loggerFactory
}
```

Additional resources

- Microsoft.Extensions.Logging.Console NuGet package.
- Logging in .NET Core and ASP.NET Core

Migrate from ASP.NET Core 2.0 to 2.1

Article • 06/03/2022

By Rick Anderson

See What's new in ASP.NET Core 2.1 for an overview of the new features in ASP.NET Core 2.1.

This article:

- Covers the basics of migrating an ASP.NET Core 2.0 app to 2.1.
- Provides an overview of the changes to the ASP.NET Core web application templates.

A quick way to get an overview of the changes in 2.1 is to:

- Create an ASP.NET Core 2.0 web app named WebApp1.
- Commit the WebApp1 in a source control system.
- Delete WebApp1 and create an ASP.NET Core 2.1 web app named WebApp1 in the same place.
- Review the changes in the 2.1 version.

This article provides an overview on migration to ASP.NET Core 2.1. It doesn't contain a complete list of all changes needed to migrate to version 2.1. Some projects might require more steps depending on the options selected when the project was created and modifications made to the project.

Update the project file to use 2.1 versions

Update the project file:

- Change the target framework to .NET Core 2.1 by updating the project file to `<TargetFramework>netcoreapp2.1</TargetFramework>`.
- Replace the package reference for `Microsoft.AspNetCore.All` with a package reference for `Microsoft.AspNetCore.App`. You may need to add dependencies that were removed from `Microsoft.AspNetCore.All`. For more information, see Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.0 and Microsoft.AspNetCore.App metapackage for ASP.NET Core.
- Remove the "Version" attribute on the package reference to `Microsoft.AspNetCore.App`. Projects that use `<Project Sdk="Microsoft.NET.Sdk.Web">` don't need to set the version. The version is implied

by the target framework and selected to best match the way ASP.NET Core 2.1 works. For more information, see the Rules for projects targeting the shared framework section.

- For apps that target the .NET Framework, update each package reference to 2.1.
- Remove references to <**DotNetCliToolReference**> elements for the following packages. These tools are bundled by default in the .NET CLI and don't need to be installed separately.
  - Microsoft.DotNet.Watcher.Tools (`dotnet watch`)
  - Microsoft.EntityFrameworkCore.Tools.DotNet (`dotnet ef`)
  - Microsoft.Extensions.Caching.SqlConfig.Tools (`dotnet sql-cache`)
  - Microsoft.Extensions.SecretManager.Tools (`dotnet user-secrets`)
- Optional: you can remove the <**DotNetCliToolReference**> element for `Microsoft.VisualStudio.Web.CodeGeneration.Tools`. You can replace this tool with a globally installed version by running `dotnet tool install -g dotnet-aspnet-codegenerator`.
- For 2.1, a Razor class library is the recommended solution to distribute Razor files. If your app uses embedded views, or otherwise relies on runtime compilation of Razor files, add `<CopyRefAssembliesToPublishDirectory>true</CopyRefAssembliesToPublishDirectory>` to a `<PropertyGroup>` in your project file.

The following markup shows the template-generated 2.0 project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="2.0.3" PrivateAssets="All" />
    <PackageReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.4"
PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference
Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.3" />
    <DotNetCliToolReference
Include="Microsoft.Extensions.SecretManager.Tools" Version="2.0.2" />
    <DotNetCliToolReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.4" />
```

```
  </ItemGroup>
</Project>
```

The following markup shows the template-generated 2.1 project file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <UserSecretsId>aspnet-{Project Name}-{GUID}</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1"
PrivateAssets="All" />
  </ItemGroup>

</Project>
```

Rules for projects targeting the shared framework

A *shared framework* is a set of assemblies (*.dll* files) that are not in the app's folders. The shared framework must be installed on the machine to run the app. For more information, see The shared framework.

ASP.NET Core 2.1 includes the following shared frameworks:

- Microsoft.AspNetCore.App
- Microsoft.AspNetCore.All

The version specified by the package reference is the *minimum required* version. For example, a project referencing the 2.1.1 versions of these packages won't run on a machine with only the 2.1.0 runtime installed.

Known issues for projects targeting a shared framework:

- The .NET Core 2.1.300 SDK (first included in Visual Studio 15.6) set the implicit version of `Microsoft.AspNetCore.App` to 2.1.0 which caused conflicts with Entity Framework Core 2.1.1. The recommended solution is to upgrade the .NET Core

SDK to 2.1.301 or later. For more information, see Packages that share dependencies with Microsoft.AspNetCore.App cannot reference patch versions.

- All projects that must use `Microsoft.AspNetCore.All` or `Microsoft.AspNetCore.App` should add a package reference for the package in the project file, even if they contain a project reference to another project using `Microsoft.AspNetCore.All` or `Microsoft.AspNetCore.App`.

  Example:
  - `MyApp` has a package reference to `Microsoft.AspNetCore.App`.
  - `MyApp.Tests` has a project reference to `MyApp.csproj`.

  Add a package reference for `Microsoft.AspNetCore.App` to `MyApp.Tests`. For more information, see Integration testing is hard to set up and may break on shared framework servicing.

Update to the 2.1 Docker images

In ASP.NET Core 2.1, the Docker images migrated to the dotnet/dotnet-docker GitHub repository. The following table shows the Docker image and tag changes:

Expand table

2.0	2.1
microsoft/aspnetcore:2.0	microsoft/dotnet:2.1-aspnetcore-runtime
microsoft/aspnetcore-build:2.0	microsoft/dotnet:2.1-sdk

Change the `FROM` lines in your *Dockerfile* to use the new image names and tags in the preceding table's 2.1 column. For more information, see Migrating from aspnetcore docker repos to dotnet.

Changes to take advantage of the new code-based idioms that are recommended in ASP.NET Core 2.1

Changes to Main

The following images show the changes made to the templated generated `Program.cs` file.

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The preceding image shows the 2.0 version with the deletions in red.

The following image shows the 2.1 code. The code in green replaced the 2.0 version:

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();

    }
}
```

The following code shows the 2.1 version of `Program.cs`:

C#

```
namespace WebApp1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>();
```

```
    }
}
```

The new `Main` replaces the call to `BuildWebHost` with CreateWebHostBuilder. IWebHostBuilder was added to support a new integration test infrastructure.

Changes to Startup

The following code shows the changes to 2.1 template generated code. All changes are newly added code, except that `UseBrowserLink` has been removed:

C#

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace WebApp1
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>(options =>
            {
                // This lambda determines whether user consent for non-
essential cookies is needed for a given request.
                options.CheckConsentNeeded = context => true;
                options.MinimumSameSitePolicy = SameSiteMode.None;
            });


            services.AddMvc()
                .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {
            if (env.IsDevelopment())
            {
```

```
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseCookiePolicy();
            // If the app uses Session or TempData based on Session:
            // app.UseSession();

            app.UseMvc();
        }
    }
}
```

The preceding code changes are detailed in:

- GDPR support in ASP.NET Core for `CookiePolicyOptions` and `UseCookiePolicy`.
- HTTP Strict Transport Security Protocol (HSTS) for `UseHsts`.
- Require HTTPS for `UseHttpsRedirection`.
- SetCompatibilityVersion for `SetCompatibilityVersion(CompatibilityVersion.Version_2_1)`.

Changes to authentication code

ASP.NET Core 2.1 provides ASP.NET Core Identity as a Razor class library (RCL).

The default 2.1 Identity UI doesn't currently provide significant new features over the 2.0 version. Replacing Identity with the RCL package is optional. The advantages to replacing the template generated Identity code with the RCL version include:

- Many files are moved out of your source tree.
- Any bug fixes or new features to Identity are included in the Microsoft.AspNetCore.App metapackage. You automatically get the updated Identity when `Microsoft.AspNetCore.App` is updated.

If you've made non-trivial changes to the template generated Identity code:

- The preceding advantages probably do **not** justify converting to the RCL version.
- You can keep your ASP.NET Core 2.0 Identity code, it's fully supported.

Identity 2.1 exposes endpoints with the `Identity` area. For example, the follow table shows examples of Identity endpoints that change from 2.0 to 2.1:

Expand table

2.0 URL	2.1 URL
/Account/Login	/Identity/Account/Login
/Account/Logout	/Identity/Account/Logout
/Account/Manage	/Identity/Account/Manage

Applications that have code using Identity and replace 2.0 Identity UI with the 2.1 Identity Library need to take into account Identity URLs have `/Identity` segment prepended to the URIs. One way to handle the new Identity endpoints is to set up redirects, for example from `/Account/Login` to `/Identity/Account/Login`.

Update Identity to version 2.1

The following options are available to update Identity to 2.1.

- Use the Identity UI 2.0 code with no changes. Using Identity UI 2.0 code is fully supported. This is a good approach when significant changes have been made to the generated Identity code.
- Delete your existing Identity 2.0 code and Scaffold Identity into your project. Your project will use the ASP.NET Core Identity Razor class library. You can generate code and UI for any of the Identity UI code that you modified. Apply your code changes to the newly scaffolded UI code.
- Delete your existing Identity 2.0 code and Scaffold Identity into your project with the option to **Override all files**.

Replace Identity 2.0 UI with the Identity 2.1 Razor class library

This section outlines the steps to replace the ASP.NET Core 2.0 template-generated Identity code with the ASP.NET Core Identity Razor class library. The following steps are for a Razor Pages project, but the approach for an MVC project is similar.

- Verify the project file is updated to use 2.1 versions
- Delete the following folders and all the files in them:
  - *Controllers*
  - *Pages/Account/*

    - *Extensions*
- Build the project.
- Scaffold Identity into your project:
    - Select the projects exiting *_Layout.cshtml* file.
    - Select the + icon on the right side of the **Data context class**. Accept the default name.
    - Select **Add** to create a new Data context class. Creating a new data context is required for to scaffold. You remove the new data context in the next section.

Update after scaffolding Identity

- Delete the Identity scaffolder generated `IdentityDbContext` derived class in the *Areas/Identity/Data/* folder.

- Delete `Areas/Identity/IdentityHostingStartup.cs`.

- Update the *_LoginPartial.cshtml* file:
    - Move *Pages/_LoginPartial.cshtml* to *Pages/Shared/_LoginPartial.cshtml*.
    - Add `asp-area="Identity"` to the form and anchor links.
    - Update the `<form />` element to `<form asp-area="Identity" asp-page="/Account/Logout" asp-route-returnUrl="@Url.Page("/Index", new { area = "" })" method="post" id="logoutForm" class="navbar-right">`.

    The following code shows the updated *_LoginPartial.cshtml* file:

CSHTML

```
@using Microsoft.AspNetCore.Identity

@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="Identity" asp-page="/Account/Logout" asp-route-
returnUrl="@Url.Page("/Index", new { area = "" })" method="post"
id="logoutForm" class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="Identity" asp-page="/Account/Manage/Index"
title="Manage">Hello @UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn
navbar-link">Log out</button>
            </li>
        </ul>
```

```
        </form>
    }
    else
    {
        <ul class="nav navbar-nav navbar-right">
            <li><a asp-area="Identity" asp-
    page="/Account/Register">Register</a></li>
            <li><a asp-area="Identity" asp-page="/Account/Login">Log in</a>
    </li>
        </ul>
    }
```

Update `ConfigureServices` with the following code:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

    services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    services.AddMvc();

    // Register no-op EmailSender used by account confirmation and password
reset
    // during development
    services.AddSingleton<IEmailSender, EmailSender>();
}
```

Changes to Razor Pages projects Razor files

The layout file

- Move *Pages/_Layout.cshtml* to *Pages/Shared/_Layout.cshtml*
- In *Areas/Identity/Pages/_ViewStart.cshtml*, change `Layout = "/Pages/_Layout.cshtml"` to `Layout = "/Pages/Shared/_Layout.cshtml"`.
- The *_Layout.cshtml* file has the following changes:
  - `<partial name="_CookieConsentPartial" />` is added. For more information, see GDPR support in ASP.NET Core.

    - jQuery changes from 2.2.0 to 3.3.1.

_ValidationScriptsPartial.cshtml

- *Pages/_ValidationScriptsPartial.cshtml* moves to *Pages/Shared/_ValidationScriptsPartial.cshtml*.
- *jquery.validate/1.14.0* changes to *jquery.validate/1.17.0*.

New files

The following files are added:

- `Privacy.cshtml`
- `Privacy.cshtml.cs`

See GDPR support in ASP.NET Core for information on the preceding files.

Changes to MVC projects Razor files

The layout file

The `Layout.cshtml` file has the following changes:

- `<partial name="_CookieConsentPartial" />` is added.
- jQuery changes from 2.2.0 to 3.3.1

_ValidationScriptsPartial.cshtml

jquery.validate/1.14.0 changes to *jquery.validate/1.17.0*

New files and action methods

The following are added:

- `Views/Home/Privacy.cshtml`
- The `Privacy` action method is added to the Home controller.

See GDPR support in ASP.NET Core for information on the preceding files.

Changes to the launchSettings.json file

As ASP.NET Core apps now use HTTPS by default, the `Properties/launchSettings.json` file has changed.

The following JSON shows the earlier 2.0 template-generated `launchSettings.json` file:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:1799/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:1798/"
    }
  }
}
```

The following JSON shows the new 2.1 template-generated `launchSettings.json` file:

JSON

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:39191",
      "sslPort": 44390
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
```

```
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "WebApp1": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

For more information, see Enforce HTTPS in ASP.NET Core.

Breaking changes

FileResult Range header

FileResult no longer processes the Accept-Ranges header by default. To enable the `Accept-Ranges` header, set EnableRangeProcessing to `true`.

ControllerBase.File and PhysicalFile Range header

The following ControllerBase methods no longer processes the Accept-Ranges header by default:

- Overloads of ControllerBase.File
- ControllerBase.PhysicalFile

To enable the `Accept-Ranges` header, set the `EnableRangeProcessing` parameter to `true`.

ASP.NET Core Module (ANCM)

If the ASP.NET Core Module (ANCM) wasn't a selected component when Visual Studio was installed or if a prior version of the ANCM was installed on the system, download the latest .NET Core Hosting Bundle Installer (direct download) and run the installer. For more information, see Hosting Bundle.

Additional changes

- SetCompatibilityVersion
- Libuv transport configuration

Migrate from ASP.NET Core 1.x to 2.0

Article • 06/03/2022

By Scott Addie

In this article, we walk you through updating an existing ASP.NET Core 1.x project to ASP.NET Core 2.0. Migrating your application to ASP.NET Core 2.0 enables you to take advantage of many new features and performance improvements.

Existing ASP.NET Core 1.x applications are based off of version-specific project templates. As the ASP.NET Core framework evolves, so do the project templates and the starter code contained within them. In addition to updating the ASP.NET Core framework, you need to update the code for your application.

Prerequisites

See Get Started with ASP.NET Core.

Update Target Framework Moniker (TFM)

Projects targeting .NET Core should use the TFM of a version greater than or equal to .NET Core 2.0. Search for the `<TargetFramework>` node in the `.csproj` file, and replace its inner text with `netcoreapp2.0`:

XML

```xml
<TargetFramework>netcoreapp2.0</TargetFramework>
```

Projects targeting .NET Framework should use the TFM of a version greater than or equal to .NET Framework 4.6.1. Search for the `<TargetFramework>` node in the `.csproj` file, and replace its inner text with `net461`:

XML

```xml
<TargetFramework>net461</TargetFramework>
```

> **Note**
>
> .NET Core 2.0 offers a much larger surface area than .NET Core 1.x. If you're targeting .NET Framework solely because of missing APIs in .NET Core 1.x, targeting

.NET Core 2.0 is likely to work.

If the project file contains `<RuntimeFrameworkVersion>1.{sub-version}</RuntimeFrameworkVersion>`, see this GitHub issue.

Update .NET Core SDK version in global.json

If your solution relies upon a global.json file to target a specific .NET Core SDK version, update its `version` property to use the 2.0 version installed on your machine:

JSON

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

Update package references

The `.csproj` file in a 1.x project lists each NuGet package used by the project.

In an ASP.NET Core 2.0 project targeting .NET Core 2.0, a single metapackage reference in the `.csproj` file replaces the collection of packages:

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.All" Version="2.0.9" />
</ItemGroup>
```

All the features of ASP.NET Core 2.0 and Entity Framework Core 2.0 are included in the metapackage.

ASP.NET Core 2.0 projects targeting .NET Framework should continue to reference individual NuGet packages. Update the `Version` attribute of each `<PackageReference />` node to 2.0.0.

For example, here's the list of `<PackageReference />` nodes used in a typical ASP.NET Core 2.0 project targeting .NET Framework:

XML

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Authentication.Cookies"
Version="2.0.0" />
  <PackageReference
Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore"
Version="2.0.0" />
  <PackageReference
Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="2.0.0"
/>
  <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.0.0" />
  <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation"
Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.AspNetCore.StaticFiles"
Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
Version="2.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
Version="2.0.0" PrivateAssets="All" />
  <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
Version="2.0.0" />
  <PackageReference
Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.0.0"
PrivateAssets="All" />
</ItemGroup>
```

The package `Microsoft.Extensions.CommandLineUtils` has been retired. It is still available but unsupported.

Update .NET CLI tools

In the `.csproj` file, update the `Version` attribute of each `<DotNetCliToolReference />` node to 2.0.0.

For example, here's the list of CLI tools used in a typical ASP.NET Core 2.0 project targeting .NET Core 2.0:

XML

```
<ItemGroup>
  <DotNetCliToolReference
Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  <DotNetCliToolReference Include="Microsoft.Extensions.SecretManager.Tools"
Version="2.0.0" />
  <DotNetCliToolReference
```

```
Include="Microsoft.VisualStudio.Web.CodeGeneration.Tools" Version="2.0.0" />
</ItemGroup>
```

Rename Package Target Fallback property

The `.csproj` file of a 1.x project used a `PackageTargetFallback` node and variable:

XML

```
<PackageTargetFallback>$(PackageTargetFallback);portable-
net45+win8+wp8+wpa81;</PackageTargetFallback>
```

Rename both the node and variable to `AssetTargetFallback`:

XML

```
<AssetTargetFallback>$(AssetTargetFallback);portable-net45+win8+wp8+wpa81;
</AssetTargetFallback>
```

Update Main method in Program.cs

In 1.x projects, the `Main` method of `Program.cs` looked like this:

C#

```
using System.IO;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore1App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .UseApplicationInsights()
                .Build();

            host.Run();
        }
    }
}
```

In 2.0 projects, the `Main` method of `Program.cs` has been simplified:

C#

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

The adoption of this new 2.0 pattern is highly recommended and is required for product features like Entity Framework (EF) Core Migrations to work. For example, running `Update-Database` from the Package Manager Console window or `dotnet ef database update` from the command line (on projects converted to ASP.NET Core 2.0) generates the following error:

```
Unable to create an object of type '<Context>'. Add an implementation of
'IDesignTimeDbContextFactory<Context>' to the project, or see
https://go.microsoft.com/fwlink/?linkid=851728 for additional patterns
supported at design time.
```

Add configuration providers

In 1.x projects, adding configuration providers to an app was accomplished via the `Startup` constructor. The steps involved creating an instance of `ConfigurationBuilder`, loading applicable providers (environment variables, app settings, etc.), and initializing a member of `IConfigurationRoot`.

C#

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange:
true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:
true);

    if (env.IsDevelopment())
    {
        builder.AddUserSecrets<Startup>();
    }

    builder.AddEnvironmentVariables();
    Configuration = builder.Build();
}

public IConfigurationRoot Configuration { get; }
```

The preceding example loads the `Configuration` member with configuration settings from `appsettings.json` as well as any `appsettings.{Environment}.json` file matching the `IHostingEnvironment.EnvironmentName` property. The location of these files is at the same path as `Startup.cs`.

In 2.0 projects, the boilerplate configuration code inherent to 1.x projects runs behind-the-scenes. For example, environment variables and app settings are loaded at startup. The equivalent `Startup.cs` code is reduced to `IConfiguration` initialization with the injected instance:

C#

```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

To remove the default providers added by `WebHostBuilder.CreateDefaultBuilder`, invoke the `Clear` method on the `IConfigurationBuilder.Sources` property inside of `ConfigureAppConfiguration`. To add providers back, utilize the `ConfigureAppConfiguration` method in `Program.cs`:

C#

```
public static void Main(string[] args)
{
    BuildWebHost(args).Run();
}

public static IWebHost BuildWebHost(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>()
        .ConfigureAppConfiguration((hostContext, config) =>
        {
            // delete all default configuration providers
            config.Sources.Clear();
            config.AddJsonFile("myconfig.json", optional: true);
        })
        .Build();
```

The configuration used by the `CreateDefaultBuilder` method in the preceding code snippet can be seen here.

For more information, see Configuration in ASP.NET Core.

Move database initialization code

In 1.x projects using EF Core 1.x, a command such as `dotnet ef migrations add` does the following:

1. Instantiates a `Startup` instance
2. Invokes the `ConfigureServices` method to register all services with dependency injection (including `DbContext` types)
3. Performs its requisite tasks

In 2.0 projects using EF Core 2.0, `Program.BuildWebHost` is invoked to obtain the application services. Unlike 1.x, this has the additional side effect of invoking `Startup.Configure`. If your 1.x app invoked database initialization code in its `Configure` method, unexpected problems can occur. For example, if the database doesn't yet exist, the seeding code runs before the EF Core Migrations command execution. This problem causes a `dotnet ef migrations list` command to fail if the database doesn't yet exist.

Consider the following 1.x seed initialization code in the `Configure` method of `Startup.cs`:

C#

```
app.UseMvc(routes =>
{
```

```
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

SeedData.Initialize(app.ApplicationServices);
```

In 2.0 projects, move the `SeedData.Initialize` call to the `Main` method of `Program.cs`:

C#

```
var host = BuildWebHost(args);

using (var scope = host.Services.CreateScope())
{
    var services = scope.ServiceProvider;

    try
    {
        // Requires using RazorPagesMovie.Models;
        SeedData.Initialize(services);
    }
    catch (Exception ex)
    {
        var logger = services.GetRequiredService<ILogger<Program>>();
        logger.LogError(ex, "An error occurred seeding the DB.");
    }
}

host.Run();
```

As of 2.0, it's bad practice to do anything in `BuildWebHost` except build and configure the web host. Anything that's about running the application should be handled outside of `BuildWebHost` — typically in the `Main` method of `Program.cs`.

Review Razor view compilation setting

Faster application startup time and smaller published bundles are of utmost importance to you. For these reasons, Razor view compilation is enabled by default in ASP.NET Core 2.0.

Setting the `MvcRazorCompileOnPublish` property to true is no longer required. Unless you're disabling view compilation, the property may be removed from the `.csproj` file.

When targeting .NET Framework, you still need to explicitly reference the Microsoft.AspNetCore.Mvc.Razor.ViewCompilation NuGet package in your `.csproj` file:

XML

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.ViewCompilation"
Version="2.0.0" PrivateAssets="All" />
```

Rely on Application Insights "light-up" features

Effortless setup of application performance instrumentation is important. You can now rely on the new Application Insights "light-up" features available in the Visual Studio 2017 tooling.

ASP.NET Core 1.1 projects created in Visual Studio 2017 added Application Insights by default. If you're not using the Application Insights SDK directly, outside of `Program.cs` and `Startup.cs`, follow these steps:

1. If targeting .NET Core, remove the following `<PackageReference />` node from the `.csproj` file:

   XML

   ```
   <PackageReference Include="Microsoft.ApplicationInsights.AspNetCore"
   Version="2.0.0" />
   ```

2. If targeting .NET Core, remove the `UseApplicationInsights` extension method invocation from `Program.cs`:

   C#

   ```
   public static void Main(string[] args)
   {
       var host = new WebHostBuilder()
           .UseKestrel()
           .UseContentRoot(Directory.GetCurrentDirectory())
           .UseIISIntegration()
           .UseStartup<Startup>()
           .UseApplicationInsights()
           .Build();

       host.Run();
   }
   ```

3. Remove the Application Insights client-side API call from `_Layout.cshtml`. It comprises the following two lines of code:

CSHTML

```
@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet
JavaScriptSnippet
@Html.Raw(JavaScriptSnippet.FullScript)
```

If you are using the Application Insights SDK directly, continue to do so. The 2.0 metapackage includes the latest version of Application Insights, so a package downgrade error appears if you're referencing an older version.

Adopt authentication/Identity improvements

ASP.NET Core 2.0 has a new authentication model and a number of significant changes to ASP.NET Core Identity. If you created your project with Individual User Accounts enabled, or if you have manually added authentication or Identity, see Migrate Authentication and Identity to ASP.NET Core 2.0.

Additional resources

- Breaking Changes in ASP.NET Core 2.0

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Migrate authentication and Identity to ASP.NET Core 2.0

Article • 06/03/2022

By Scott Addie and Hao Kung

ASP.NET Core 2.0 has a new model for authentication and Identity that simplifies configuration by using services. ASP.NET Core 1.x applications that use authentication or Identity can be updated to use the new model as outlined below.

Update namespaces

In 1.x, classes such `IdentityRole` and `IdentityUser` were found in the `Microsoft.AspNetCore.Identity.EntityFrameworkCore` namespace.

In 2.0, the Microsoft.AspNetCore.Identity namespace became the new home for several of such classes. With the default Identity code, affected classes include `ApplicationUser` and `Startup`. Adjust your `using` statements to resolve the affected references.

Authentication Middleware and services

In 1.x projects, authentication is configured via middleware. A middleware method is invoked for each authentication scheme you want to support.

The following 1.x example configures Facebook authentication with Identity in `Startup.cs`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseIdentity();
    app.UseFacebookAuthentication(new FacebookOptions {
        AppId = Configuration["auth:facebook:appid"],
        AppSecret = Configuration["auth:facebook:appsecret"]
```

```
    });
}
```

In 2.0 projects, authentication is configured via services. Each authentication scheme is registered in the `ConfigureServices` method of `Startup.cs`. The `UseIdentity` method is replaced with `UseAuthentication`.

The following 2.0 example configures Facebook authentication with Identity in `Startup.cs`:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>();

    // If you want to tweak Identity cookies, they're no longer part of
IdentityOptions.
    services.ConfigureApplicationCookie(options => options.LoginPath =
"/Account/LogIn");
    services.AddAuthentication()
            .AddFacebook(options =>
            {
                options.AppId = Configuration["auth:facebook:appid"];
                options.AppSecret =
Configuration["auth:facebook:appsecret"];
            });
}

public void Configure(IApplicationBuilder app, ILoggerFactory loggerfactory)
{
    app.UseAuthentication();
}
```

The `UseAuthentication` method adds a single authentication middleware component, which is responsible for automatic authentication and the handling of remote authentication requests. It replaces all of the individual middleware components with a single, common middleware component.

Below are 2.0 migration instructions for each major authentication scheme.

Cookie-based authentication

Select one of the two options below, and make the necessary changes in `Startup.cs`:

1. Use cookies with Identity

- Replace `UseIdentity` with `UseAuthentication` in the `Configure` method:

```
C#

app.UseAuthentication();
```

- Invoke the `AddIdentity` method in the `ConfigureServices` method to add the cookie authentication services.

- Optionally, invoke the `ConfigureApplicationCookie` or `ConfigureExternalCookie` method in the `ConfigureServices` method to tweak the Identity cookie settings.

```
C#

services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

services.ConfigureApplicationCookie(options => options.LoginPath =
"/Account/LogIn");
```

2. Use cookies without Identity

- Replace the `UseCookieAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
C#

app.UseAuthentication();
```

- Invoke the `AddAuthentication` and `AddCookie` methods in the `ConfigureServices` method:

```
C#

// If you don't want the cookie to be automatically authenticated
and assigned to HttpContext.User,
// remove the CookieAuthenticationDefaults.AuthenticationScheme
parameter passed to AddAuthentication.
services.AddAuthentication(CookieAuthenticationDefaults.Authentica
tionScheme)
        .AddCookie(options =>
        {
            options.LoginPath = "/Account/LogIn";
```

```
            options.LogoutPath = "/Account/LogOff";
        });
```

JWT Bearer Authentication

Make the following changes in `Startup.cs`:

- Replace the `UseJwtBearerAuthentication` method call in the `Configure` method with `UseAuthentication`:

  ```
  C#

  app.UseAuthentication();
  ```

- Invoke the `AddJwtBearer` method in the `ConfigureServices` method:

  ```
  C#

  services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
          .AddJwtBearer(options =>
          {
              options.Audience = "http://localhost:5001/";
              options.Authority = "http://localhost:5000/";
          });
  ```

  This code snippet doesn't use Identity, so the default scheme should be set by passing `JwtBearerDefaults.AuthenticationScheme` to the `AddAuthentication` method.

OpenID Connect (OIDC) authentication

Make the following changes in `Startup.cs`:

- Replace the `UseOpenIdConnectAuthentication` method call in the `Configure` method with `UseAuthentication`:

  ```
  C#

  app.UseAuthentication();
  ```

- Invoke the `AddOpenIdConnect` method in the `ConfigureServices` method:

  ```
  C#
  ```

```
services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
})
.AddCookie()
.AddOpenIdConnect(options =>
{
    options.Authority = Configuration["auth:oidc:authority"];
    options.ClientId = Configuration["auth:oidc:clientid"];
});
```

- Replace the `PostLogoutRedirectUri` property in the `OpenIdConnectOptions` action with `SignedOutRedirectUri`:

```
C#

.AddOpenIdConnect(options =>
{
    options.SignedOutRedirectUri = "https://contoso.com";
});
```

Facebook authentication

Make the following changes in `Startup.cs`:

- Replace the `UseFacebookAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
C#

app.UseAuthentication();
```

- Invoke the `AddFacebook` method in the `ConfigureServices` method:

```
C#

services.AddAuthentication()
        .AddFacebook(options =>
        {
            options.AppId = Configuration["auth:facebook:appid"];
            options.AppSecret =
```

```
Configuration["auth:facebook:appsecret"];
        });
```

Google authentication

Make the following changes in `Startup.cs`:

- Replace the `UseGoogleAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
C#

app.UseAuthentication();
```

- Invoke the `AddGoogle` method in the `ConfigureServices` method:

```
C#

services.AddAuthentication()
        .AddGoogle(options =>
        {
            options.ClientId = Configuration["auth:google:clientid"];
            options.ClientSecret =
Configuration["auth:google:clientsecret"];
        });
```

Microsoft Account authentication

For more information on Microsoft account authentication, see this GitHub issue.

Make the following changes in `Startup.cs`:

- Replace the `UseMicrosoftAccountAuthentication` method call in the `Configure` method with `UseAuthentication`:

```
C#

app.UseAuthentication();
```

- Invoke the `AddMicrosoftAccount` method in the `ConfigureServices` method:

```
C#
```

```
services.AddAuthentication()
        .AddMicrosoftAccount(options =>
        {
            options.ClientId =
Configuration["auth:microsoft:clientid"];
            options.ClientSecret =
Configuration["auth:microsoft:clientsecret"];
        });
```

Twitter authentication

Make the following changes in `Startup.cs`:

- Replace the `UseTwitterAuthentication` method call in the `Configure` method with `UseAuthentication`:

  C#

  ```
  app.UseAuthentication();
  ```

- Invoke the `AddTwitter` method in the `ConfigureServices` method:

  C#

  ```
  services.AddAuthentication()
          .AddTwitter(options =>
          {
              options.ConsumerKey =
  Configuration["auth:twitter:consumerkey"];
              options.ConsumerSecret =
  Configuration["auth:twitter:consumersecret"];
          });
  ```

Setting default authentication schemes

In 1.x, the `AutomaticAuthenticate` and `AutomaticChallenge` properties of the AuthenticationOptions base class were intended to be set on a single authentication scheme. There was no good way to enforce this.

In 2.0, these two properties have been removed as properties on the individual `AuthenticationOptions` instance. They can be configured in the `AddAuthentication` method call within the `ConfigureServices` method of `Startup.cs`:

C#

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme
);
```

In the preceding code snippet, the default scheme is set to `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookies").

Alternatively, use an overloaded version of the `AddAuthentication` method to set more than one property. In the following overloaded method example, the default scheme is set to `CookieAuthenticationDefaults.AuthenticationScheme`. The authentication scheme may alternatively be specified within your individual `[Authorize]` attributes or authorization policies.

C#

```
services.AddAuthentication(options =>
{
    options.DefaultScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
OpenIdConnectDefaults.AuthenticationScheme;
});
```

Define a default scheme in 2.0 if one of the following conditions is true:

- You want the user to be automatically signed in
- You use the `[Authorize]` attribute or authorization policies without specifying schemes

An exception to this rule is the `AddIdentity` method. This method adds cookies for you and sets the default authenticate and challenge schemes to the application cookie `IdentityConstants.ApplicationScheme`. Additionally, it sets the default sign-in scheme to the external cookie `IdentityConstants.ExternalScheme`.

Use HttpContext authentication extensions

The `IAuthenticationManager` interface is the main entry point into the 1.x authentication system. It has been replaced with a new set of `HttpContext` extension methods in the `Microsoft.AspNetCore.Authentication` namespace.

For example, 1.x projects reference an `Authentication` property:

C#

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

In 2.0 projects, import the `Microsoft.AspNetCore.Authentication` namespace, and delete the `Authentication` property references:

C#

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Windows Authentication (HTTP.sys / IISIntegration)

There are two variations of Windows authentication:

- The host only allows authenticated users. This variation isn't affected by the 2.0 changes.
- The host allows both anonymous and authenticated users. This variation is affected by the 2.0 changes. For example, the app should allow anonymous users at the IIS or HTTP.sys layer but authorize users at the controller level. In this scenario, set the default scheme in the `Startup.ConfigureServices` method.

  For Microsoft.AspNetCore.Server.IISIntegration, set the default scheme to `IISDefaults.AuthenticationScheme`:

  C#

  ```
  using Microsoft.AspNetCore.Server.IISIntegration;

  services.AddAuthentication(IISDefaults.AuthenticationScheme);
  ```

  For Microsoft.AspNetCore.Server.HttpSys, set the default scheme to `HttpSysDefaults.AuthenticationScheme`:

  C#

  ```
  using Microsoft.AspNetCore.Server.HttpSys;

  services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
  ```

Failure to set the default scheme prevents the authorize (challenge) request from working with the following exception:

> `System.InvalidOperationException`: No authenticationScheme was specified, and there was no DefaultChallengeScheme found.

For more information, see Configure Windows Authentication in ASP.NET Core.

IdentityCookieOptions instances

A side effect of the 2.0 changes is the switch to using named options instead of cookie options instances. The ability to customize the Identity cookie scheme names is removed.

For example, 1.x projects use constructor injection to pass an `IdentityCookieOptions` parameter into `AccountController.cs` and `ManageController.cs`. The external cookie authentication scheme is accessed from the provided instance:

C#

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IOptions<IdentityCookieOptions> identityCookieOptions,
    IEmailSender emailSender,
    ISmsSender smsSender,
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _externalCookieScheme =
identityCookieOptions.Value.ExternalCookieAuthenticationScheme;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

The aforementioned constructor injection becomes unnecessary in 2.0 projects, and the `_externalCookieScheme` field can be deleted:

C#

```
public AccountController(
    UserManager<ApplicationUser> userManager,
    SignInManager<ApplicationUser> signInManager,
    IEmailSender emailSender,
    ISmsSender smsSender,
```

```
    ILoggerFactory loggerFactory)
{
    _userManager = userManager;
    _signInManager = signInManager;
    _emailSender = emailSender;
    _smsSender = smsSender;
    _logger = loggerFactory.CreateLogger<AccountController>();
}
```

1.x projects used the `_externalCookieScheme` field as follows:

C#

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.Authentication.SignOutAsync(_externalCookieScheme);
```

In 2.0 projects, replace the preceding code with the following. The `IdentityConstants.ExternalScheme` constant can be used directly.

C#

```
// Clear the existing external cookie to ensure a clean login process
await HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

Resolve the newly added `SignOutAsync` call by importing the following namespace:

C#

```
using Microsoft.AspNetCore.Authentication;
```

Add IdentityUser POCO navigation properties

The Entity Framework (EF) Core navigation properties of the base `IdentityUser` POCO (Plain Old CLR Object) have been removed. If your 1.x project used these properties, manually add them back to the 2.0 project:

C#

```
/// <summary>
/// Navigation property for the roles this user belongs to.
/// </summary>
public virtual ICollection<IdentityUserRole<int>> Roles { get; } = new
List<IdentityUserRole<int>>();

/// <summary>
```

```
/// Navigation property for the claims this user possesses.
/// </summary>
public virtual ICollection<IdentityUserClaim<int>> Claims { get; } = new
List<IdentityUserClaim<int>>();

/// <summary>
/// Navigation property for this users login accounts.
/// </summary>
public virtual ICollection<IdentityUserLogin<int>> Logins { get; } = new
List<IdentityUserLogin<int>>();
```

To prevent duplicate foreign keys when running EF Core Migrations, add the following to your `IdentityDbContext` class' `OnModelCreating` method (after the `base.OnModelCreating();` call):

C#

```
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);
    // Customize the ASP.NET Core Identity model and override the defaults
if needed.
    // For example, you can rename the ASP.NET Core Identity table names and
more.
    // Add your customizations after calling base.OnModelCreating(builder);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Claims)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Logins)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ApplicationUser>()
        .HasMany(e => e.Roles)
        .WithOne()
        .HasForeignKey(e => e.UserId)
        .IsRequired()
        .OnDelete(DeleteBehavior.Cascade);
}
```

Replace GetExternalAuthenticationSchemes

The synchronous method `GetExternalAuthenticationSchemes` was removed in favor of an asynchronous version. 1.x projects have the following code in `Controllers/ManageController.cs`:

C#

```
var otherLogins =
_signInManager.GetExternalAuthenticationSchemes().Where(auth =>
userLogins.All(ul => auth.AuthenticationScheme !=
ul.LoginProvider)).ToList();
```

This method appears in `Views/Account/Login.cshtml` too:

CSHTML

```
@{
    var loginProviders =
SignInManager.GetExternalAuthenticationSchemes().ToList();
    if (loginProviders.Count == 0)
    {
        <div>
            <p>
                There are no external authentication services configured.
See <a href="https://go.microsoft.com/fwlink/?LinkID=532715">this
article</a>
                for details on setting up this ASP.NET application to
support logging in via external services.
            </p>
        </div>
    }
    else
    {
        <form asp-controller="Account" asp-action="ExternalLogin" asp-route-
returnurl="@ViewData["ReturnUrl"]" method="post" class="form-horizontal">
            <div>
                <p>
                    @foreach (var provider in loginProviders)
                    {
                        <button type="submit" class="btn btn-default"
name="provider" value="@provider.AuthenticationScheme" title="Log in using
your @provider.DisplayName account">@provider.AuthenticationScheme</button>
                    }
                </p>
            </div>
        </form>
    }
}
```

In 2.0 projects, use the GetExternalAuthenticationSchemesAsync method. The change in `ManageController.cs` resembles the following code:

C#

```
var schemes = await _signInManager.GetExternalAuthenticationSchemesAsync();
var otherLogins = schemes.Where(auth => userLogins.All(ul => auth.Name !=
ul.LoginProvider)).ToList();
```

In `Login.cshtml`, the `AuthenticationScheme` property accessed in the `foreach` loop changes to `Name`:

CSHTML

```
@{
    var loginProviders = (await
SignInManager.GetExternalAuthenticationSchemesAsync()).ToList();
    if (loginProviders.Count == 0)
    {
        <div>
            <p>
                There are no external authentication services configured.
See <a href="https://go.microsoft.com/fwlink/?LinkID=532715">this
article</a>
                for details on setting up this ASP.NET application to
support logging in via external services.
            </p>
        </div>
    }
    else
    {
        <form asp-controller="Account" asp-action="ExternalLogin" asp-route-
returnurl="@ViewData["ReturnUrl"]" method="post" class="form-horizontal">
            <div>
                <p>
                    @foreach (var provider in loginProviders)
                    {
                        <button type="submit" class="btn btn-default"
name="provider" value="@provider.Name" title="Log in using your
@provider.DisplayName account">@provider.DisplayName</button>
                    }
                </p>
            </div>
        </form>
    }
}
```

ManageLoginsViewModel property change

A `ManageLoginsViewModel` object is used in the `ManageLogins` action of `ManageController.cs`. In 1.x projects, the object's `OtherLogins` property return type is

`IList<AuthenticationDescription>`. This return type requires an import of `Microsoft.AspNetCore.Http.Authentication`:

C#

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Http.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore1App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationDescription> OtherLogins { get; set; }
    }
}
```

In 2.0 projects, the return type changes to `IList<AuthenticationScheme>`. This new return type requires replacing the `Microsoft.AspNetCore.Http.Authentication` import with a `Microsoft.AspNetCore.Authentication` import.

C#

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Identity;

namespace AspNetCoreDotNetCore2App.Models.ManageViewModels
{
    public class ManageLoginsViewModel
    {
        public IList<UserLoginInfo> CurrentLogins { get; set; }

        public IList<AuthenticationScheme> OtherLogins { get; set; }
    }
}
```

Additional resources

For more information, see the Discussion for Auth 2.0 issue on GitHub.

Collaborate with us on

.NET ASP.NET Core feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Upgrade from ASP.NET Framework to ASP.NET Core

Article • 04/13/2023

Why upgrade to the latest .NET

ASP.NET Core is the modern web framework for .NET. While ASP.NET Core has many similarities to ASP.NET in the .NET Framework, it's a new framework that's completely rewritten. ASP.NET apps updated to ASP.NET Core can benefit from improved performance and access to the latest web development features and capabilities.

ASP.NET Framework update approaches

Most non-trivial ASP.NET Framework apps should consider using the incremental upgrade approach. For more information, see Incremental ASP.NET to ASP.NET Core upgrade.

For ASP.NET MVC and Web API apps, see Learn to upgrade from ASP.NET MVC and Web API to ASP.NET Core MVC. For ASP.NET Framework Web Forms apps, see Learn to upgrade from ASP.NET Web Forms to ASP.NET Core.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

URI decoding differences between ASP.NET to ASP.NET Core

ASP.NET Core has the following URI decoding differences with ASP.NET Framework:

ASCII	Encoded	ASP.NET Core	ASP.NET Framework
`\`	`%5C`	`\`	`/`
`/`	`%2F`	`%2F`	`/`

When decoding `%2F` on ASP.NET Core:

- The entire path gets unescaped except `%2F` because converting it to `/` would change the path structure. It can't be decoded until the path is split into segments.

To generate the value for `HttpRequest.Url`, use `new Uri(this.AspNetCoreHttpRequest.GetEncodedUrl());` to avoid `Uri` misinterpreting the values.

Migrating User Secrets from ASP.NET Framework to ASP.NET Core

See this GitHub issue.

Incremental ASP.NET to ASP.NET Core update

Article • 10/04/2023

Updating an app from ASP.NET Framework to ASP.NET Core is non-trivial for the majority of production apps. These apps often incorporate new technologies as they become available and are often composed of many legacy decisions. This article provides guidance and links to tools for updating ASP.NET Framework apps to ASP.NET Core with as little change as possible.

One of the larger challenges is the pervasive use of HttpContext throughout a code base. Without the incremental approach and tools, a large scale rewrite is required to remove the HttpContext dependency. The adapters in dotnet/systemweb-adapters provide a set of runtime helpers to access the types used in the ASP.NET Framework app in a way that works in ASP.NET Core with minimal changes.

A complete migration may take considerable effort depending on the size of the app, dependencies, and non-portable APIs used. In order to keep deploying an app to production while working on updating, the best pattern is to follow is the Strangler Fig pattern. The *Strangler Fig pattern* allows for continual development on the old system with an incremental approach to replacing specific pieces of functionality with new services. This document describes how to apply the Strangler Fig pattern to an ASP.NET app updating towards ASP.NET Core.

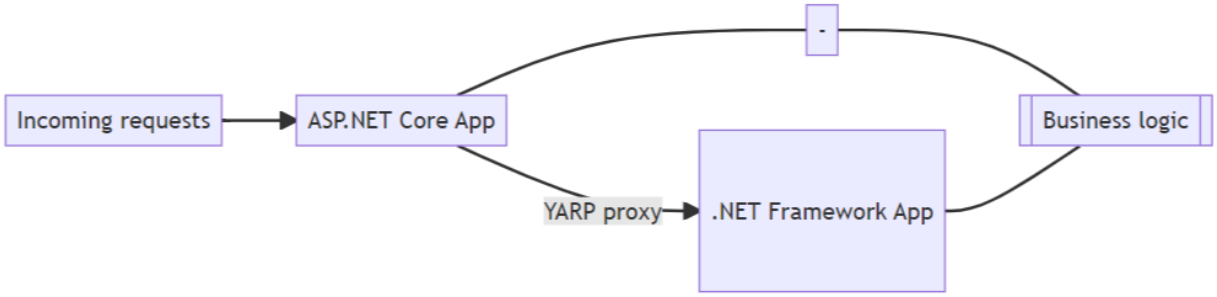
If you'd like to skip this overview article and get started, see Get started.

App migration to ASP.NET Core

Before starting the migration, the app targets ASP.NET Framework and runs on Windows with its supporting libraries:

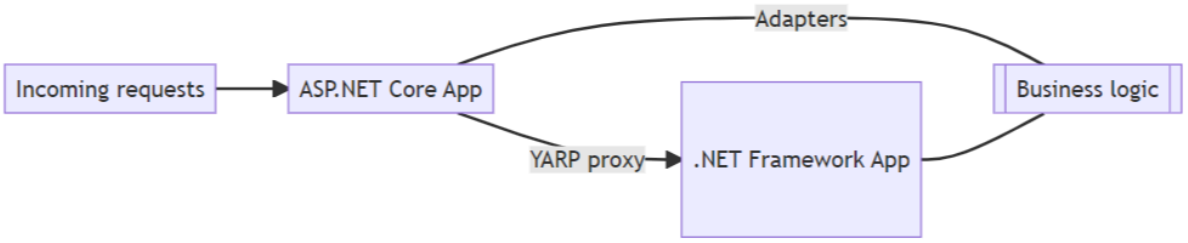


Migration starts by introducing a new app based on ASP.NET Core that becomes the entry point. Incoming requests go to the ASP.NET Core app, which either handles the request or proxies the request to the .NET Framework app via YARP. At first, the majority of code providing responses is in the .NET Framework app, but the ASP.NET Core app is now set up to start migrating routes:



To migrate business logic that relies on `HttpContext`, the libraries need to be built with `Microsoft.AspNetCore.SystemWebAdapters`. Building the libraries with `SystemWebAdapters` allows:

- The libraries to be built against .NET Framework, .NET Core, or .NET Standard 2.0.
- Ensures that the libraries are using APIs that are available on both ASP.NET Framework and ASP.NET Core.

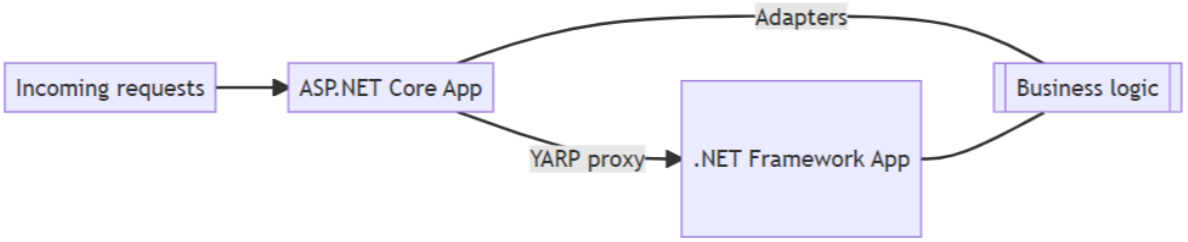


Once the ASP.NET Core app using YARP is set up, you can start updating routes from ASP.NET Framework to ASP.NET Core. For example, WebAPI or MVC controller action methods,handlers, or some other implementation of a route. If the route is available in the ASP.NET Core app, it's matched and served.

During the migration process, additional services and infrastructure are identified that must be updated to run on .NET Core. Options listed in order of maintainability include:

1. Move the code to shared libraries
2. Link the code in the new project
3. Duplicate the code

Eventually, the ASP.NET Core app handles more of the routes than the .NET Framework app:



Once the ASP.NET Framework app is no longer needed and deleted:

- The app is running on the ASP.NET Core app stack, but is still using the adapters.
- The remaining migration work is removing the use of adapters.



The Visual Studio extension .NET Upgrade Assistant can help upgrade ASP.NET Framework web apps to ASP.NET Core. For more information see the blog post Upgrading your .NET projects with Visual Studio.

System.Web Adapters

The `Microsoft.AspNetCore.SystemWebAdapters` namespace is a collection of runtime helpers that facilitate using code written against `System.Web` while moving to ASP.NET Core. There are a few packages that may be used to use features from these adapters:

- `Microsoft.AspNetCore.SystemWebAdapters`: This package is used in supporting libraries and provide the System.Web APIs you may have taken a dependency on, such as `HttpContext` and others. This package targets .NET Standard 2.0, .NET 4.5+, and .NET 6+.
- `Microsoft.AspNetCore.SystemWebAdapters.FrameworkServices`: This package only targets .NET Framework and is intended to provide services to ASP.NET Framework applications that may need to provide incremental migrations. This is generally not expected to be referenced from libraries, but rather from the applications themselves.
- `Microsoft.AspNetCore.SystemWebAdapters.CoreServices`: This package only targets .NET 6+ and is intended to provide services to ASP.NET Core applications to configure behavior of `System.Web` APIs as well as opting into any behaviors for incremental migration. This is generally not expected to be referenced from libraries, but rather from the applications themselves.

- `Microsoft.AspNetCore.SystemWebAdapters.Abstractions`: This package is a supporting package that provides abstractions for services used by both the ASP.NET Core and ASP.NET Framework application such as session state serialization.

For examples of scenarios where this is useful, see the adapters article.

For guidance around usage, see the usage guidance article.

Additional Resources

- Example migration of eShop to ASP.NET Core
- Video:Tooling for Incremental ASP.NET Core Migrations
- Unit Testing

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

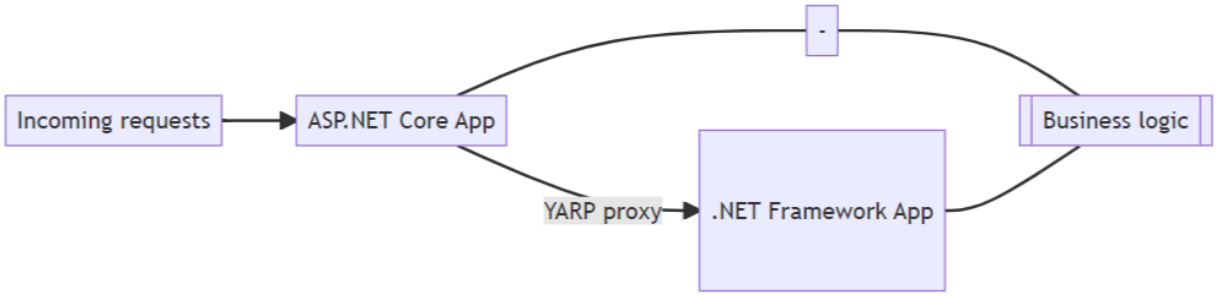
Open a documentation issue

Provide product feedback

Get started with incremental ASP.NET to ASP.NET Core migration

Article • 03/17/2023

For a large migration, we recommend setting up a ASP.NET Core app that proxies to the original .NET Framework app. The new proxy enabled app is shown in the following image:



To understand how this approach is helpful in the migration process, see Incremental ASP.NET to ASP.NET Core migration. The rest of this article provides the steps to proceed with an incremental migration.

Set up ASP.NET Core Project

For ASP.NET MVC and Web API apps, see Learn to upgrade from ASP.NET MVC and Web API to ASP.NET Core MVC. For ASP.NET Framework Web Forms apps, see Learn to upgrade from ASP.NET Web Forms to ASP.NET Core.

Upgrade supporting libraries

If you have supporting libraries in your solution that you will need to use, they should be upgraded to .NET Standard 2.0, if possible. Upgrade Assistant is a great tool for this. If libraries are unable to target .NET Standard, you can target .NET 6 or later either along with the .NET Framework target in the original project or in a new project alongside the original.

The adapters can be used in these libraries to enable support for `System.Web.HttpContext` usage in class libraries. In order to enable `System.Web.HttpContext` usage in a library:

1. Remove reference to `System.Web` in the project file

2. Add the `Microsoft.AspNetCore.SystemWebAdapters` package
3. Enable multi-targeting and add a .NET 6 target or later, or convert the project to .NET Standard 2.0.
4. Ensure the target framework supports .NET Core. Multi-targeting can be used if .NET Standard 2.0 is not sufficient

This step may require a number of projects to change depending on your solution structure. Upgrade Assistant can help you identify which ones need to change and automate a number of steps in the process.

Enable Session Support

Session is a commonly used feature of ASP.NET that shares the name with a feature in ASP.NET Core the APIs are much different. See the documentation on session support.

Enable shared authentication support

It is possible to share authentication between the original ASP.NET app and the new ASP.NET Core app by using the `System.Web` adapters remote authentication feature. This feature allows the ASP.NET Core app to defer authentication to the ASP.NET app. See the remote app connection and remote authentication docs for more details.

General Usage Guidance

There are a number of differences between ASP.NET and ASP.NET Core that the adapters are able to help update. However, there are some features that require an opt-in as they incur some cost. There are also behaviors that cannot be adapted. See usage guidance for a list of these.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

System.Web adapters

Article • 10/04/2023

The main use case of the adapters in the dotnet/systemweb-adapters repository is to help developers who have taken a reliance on `System.Web` types within their class libraries as they want to move to ASP.NET Core.

An important feature of the adapters is the adapters allow the library to be used from ***both*** ASP.NET Framework and ASP.NET Core projects. Updating multiple ASP.NET Framework apps to ASP.NET Core often involves intermediate states where not all the apps have been fully updated. By using the `System.Web` adapters, the library can be used both from ASP.NET Core callers and ASP.NET Framework callers that haven't been upgraded.

Let's take a look at an example using the adapters moving from .NET Framework to ASP.NET Core.

ASP.NET Framework

Consider a controller that does something such as:

cs

```cs
public class SomeController : Controller
{
  public ActionResult Index()
  {
    SomeOtherClass.SomeMethod(HttpContext.Current);
  }
}
```

which then has logic in a separate assembly passing that HttpContext around until finally, some inner method does some logic on it such as:

cs

```cs
public class Class2
{
  public bool PerformSomeCheck(HttpContext context)
  {
    return context.Request.Headers["SomeHeader"] == "ExpectedValue";
  }
}
```

ASP.NET Core

In order to run the above logic in ASP.NET Core, a developer will need to add the `Microsoft.AspNetCore.SystemWebAdapters` package, that will enable the projects to work on both platforms.

The libraries would need to be updated to understand the adapters, but it will be as simple as adding the package and recompiling. If these are the only dependencies a system has on `System.Web.dll`, then the libraries will be able to target .NET Standard 2.0 to facilitate a simpler building process while migrating.

The controller in ASP.NET Core will now look like this:

cs

```cs
public class SomeController : Controller
{
  [Route("/")]
  public IActionResult Index()
  {
    SomeOtherClass.SomeMethod(HttpContext);
  }
}
```

Notice that since there's a HttpContext property, they can pass that through, but it generally looks the same. Using implicit conversions, the HttpContext can be converted into the adapter that could then be passed around through the levels utilizing the code in the same way.

See also

- Unit Testing

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Remote app setup

Article • 05/04/2023

In some incremental upgrade scenarios, it's useful for the new ASP.NET Core app to be able to communicate with the original ASP.NET app.

Specifically, this capability is used, currently, for remote app authentication and remote session features.

Configuration

To enable the ASP.NET Core app to communicate with the ASP.NET app, it's necessary to make a couple small changes to each app.

ASP.NET app configuration

To set up the ASP.NET app to be able to receive requests from the ASP.NET Core app:

1. Install the nuget package Microsoft.AspNetCore.SystemWebAdapters.FrameworkServices
2. Call the `AddRemoteAppServer` extension method on the `ISystemWebAdapterBuilder`:

CSharp

```
SystemWebAdapterConfiguration.AddSystemWebAdapters(this)
    .AddRemoteAppServer(options =>
    {
        // ApiKey is a string representing a GUID
        options.ApiKey =
ConfigurationManager.AppSettings["RemoteAppApiKey"];
    });
```

In the options configuration method passed to the `AddRemoteAppServer` call, an API key must be specified. The API key is:

- Used to secure the endpoint so that only trusted callers can make requests to it.
- The same API key provided to the ASP.NET Core app when it is configured.
- A string and must be parsable as a GUID. Hyphens in the key are optional.

3. **Optional** : Add the `SystemWebAdapterModule` module to the `web.config` if it wasn't already added by NuGet. The `SystemWebAdapterModule` module is not added automatically when using SDK style projects for ASP.NET Core.

diff

```
  <system.webServer>
    <modules>
+       <remove name="SystemWebAdapterModule" />
+       <add name="SystemWebAdapterModule"
type="Microsoft.AspNetCore.SystemWebAdapters.SystemWebAdapterModule,
Microsoft.AspNetCore.SystemWebAdapters.FrameworkServices"
preCondition="managedHandler" />
    </modules>
</system.webServer>
```

ASP.NET Core app

To set up the ASP.NET Core app to be able to send requests to the ASP.NET app, you need to make a similar change, calling `AddRemoteApp` after registering System.Web adapter services with `AddSystemWebAdapters`.

CSharp

```
builder.Services.AddSystemWebAdapters()
    .AddRemoteAppClient(options =>
    {
        options.RemoteAppUrl =
new(builder.Configuration["ReverseProxy:Clusters:fallbackCluster:Destination
s:fallbackApp:Address"]);
        options.ApiKey = builder.Configuration["RemoteAppApiKey"];
    });
```

In the preceding code:

- The `AddRemoteApp` call is used to configure the remote app's URL and the shared secret API key.
- The `RemoteAppUrl` property specifies the URL of the ASP.NET Framework app that the ASP.NET Core app communicates with. In this example, the URL is read from an existing configuration setting used by the YARP proxy that proxies requests to the ASP.NET Framework app as part of the incremental migration's *strangler fig pattern*.

With both the ASP.NET and ASP.NET Core app updated, extension methods can now be used to set up remote app authentication or remote session, as needed.

Securing the remote app connection

Because remote app features involve serving requests on new endpoints from the ASP.NET app, it's important that communication to and from the ASP.NET app be secure.

First, make sure that the API key string used to authenticate the ASP.NET Core app with the ASP.NET app is unique and kept secret. It is a best practice to not store the API key in source control. Instead, load it at runtime from a secure source such as Azure Key Vault or other secure runtime configuration. In order to encourage secure API keys, remote app connections require that the keys be non-empty GUIDs (128-bit hex numbers).

Second, because it's important for the ASP.NET Core app to be able to trust that it is requesting information from the correct ASP.NET app, the ASP.NET app should use HTTPS in any production scenarios so that the ASP.NET Core app can know responses are being served by a trusted source.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Usage Guidance

Article • 03/17/2023

`Microsoft.AspNetCore.SystemWebAdapters` provides an emulation layer to mimic behavior from ASP.NET framework on ASP.NET Core. Below are some guidelines for some of the considerations when using them:

`HttpContext` lifetime

The adapters are backed by HttpContext which cannot be used past the lifetime of a request. Thus, HttpContext when run on ASP.NET Core cannot be used past a request as well, while on ASP.NET Framework it would work at times. An ObjectDisposedException will be thrown in cases where it is used past a request end.

Recommendation: Store the values needed into a POCO and hold onto that.

Conversion to HttpContext

There are two ways to convert an HttpContext to a HttpContext:

- Implicit casting
- Constructor usage

Recommendation: For the most cases, implicit casting should be preferred as this will cache the created instance and ensure only a single HttpContext per request.

CurrentCulture is not set by default

In ASP.NET Framework, CurrentCulture was set for a request, but this is not done automatically in ASP.NET Core. Instead, you must add the appropriate middleware to your pipeline.

Recommendation: See ASP.NET Core Localization for details on how to enable this.

Simplest way to enable this with similar behavior as ASP.NET Framework would be to add the following to your pipeline:

C#

```
app.UseRequestLocalization();
```

CurrentPrincipal

In ASP.NET Framework, CurrentPrincipal and Current would be set to the current user. This is not available on ASP.NET Core out of the box. Support for this is available with these adapters by adding the `ISetThreadCurrentPrincipal` to the endpoint (available to controllers via the `SetThreadCurrentPrincipalAttribute`). However, it should only be used if the code cannot be refactored to remove usage.

Recommendation: If possible, use the property User or User instead by passing it through to the call site. If not possible, enable setting the current user and also consider setting the request to be a logical single thread (see below for details).

Request thread does not exist in ASP.NET Core

In ASP.NET Framework, a request had thread-affinity and Current would only be available if on that thread. ASP.NET Core does not have this guarantee so Current will be available within the same async context, but no guarantees about threads are made.

Recommendation: If reading/writing to the HttpContext, you must ensure you are doing so in a single-threaded way. You can force a request to never run concurrently on any async context by setting the `ISingleThreadedRequestMetadata`. This will have performance implications and should only be used if you can't refactor usage to ensure non-concurrent access. There is an implementation available to add to controllers with `SingleThreadedRequestAttribute`:

C#

```
[SingleThreadedRequest]
public class SomeController : Controller
{
    ...
}
```

Request may need to be prebuffered

By default, the incoming request is not always seekable nor fully available. In order to get behavior seen in .NET Framework, you can opt into prebuffering the input stream. This will fully read the incoming stream and buffer it to memory or disk (depending on settings).

Recommendation: This can be enabled by applying endpoint metadata that implements the `IPreBufferRequestStreamMetadata` interface. This is available as an attribute

`PreBufferRequestStreamAttribute` that can be applied to controllers or methods.

To enable this on all MVC endpoints, there is an extension method that can be used as follows:

```cs
app.MapDefaultControllerRoute()
    .PreBufferRequestStream();
```

Response may require buffering

Some APIs on Response require that the output stream is buffered, such as Output, End(), Clear(), and SuppressContent.

Recommendation: In order to support behavior for Response that requires buffering the response before sending, endpoints must opt-into it with endpoint metadata implementing `IBufferResponseStreamMetadata`.

To enable this on all MVC endpoints, there is an extension method that can be used as follows:

```cs
app.MapDefaultControllerRoute()
    .BufferResponseStream();
```

Shared session state

In order to support Session, endpoints must opt-into it via metadata implementing `ISessionMetadata`.

Recommendation: To enable this on all MVC endpoints, there is an extension method that can be used as follows:

```cs
app.MapDefaultControllerRoute()
    .RequireSystemWebAdapterSession();
```

This also requires some implementation of a session store. For details of options here, see here.

Remote session exposes additional endpoint for application

The remote session support exposes an endpoint that allows the core app to retrieve session information. This may cause a potentially long-lived request to exist between the core app and the framework app, but will time out with the current request or the session timeout (by default is 20 minutes).

Recommendation: Ensure the API key used is a strong one and that the connection with the framework app is done over SSL.

Virtual directories must be identical for framework and core applications

The virtual directory setup is used for route generation, authorization, and other services within the system. At this point, no reliable method has been found to enable different virtual directories due to how ASP.NET Framework works.

Recommendation: Ensure your two applications are on different sites (hosts and/or ports) with the same application/virtual directory layout.

Wrapped ASP.NET Core session state

Article • 08/01/2023

This implementation wraps the session provided on ASP.NET Core so that it can be used with the adapters. The session will be using the same backing store as `Microsoft.AspNetCore.Http.ISession` but will provide strongly-typed access to its members.

Configuration for ASP.NET Core would look similar to the following:

C#

```
builder.Services.AddSystemWebAdapters()
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be
registered to a type
        options.RegisterKey<int>("test-value");
        options.RegisterKey<SessionDemoModel>("SampleSessionItem");
    })
    .WrapAspNetCoreSession();
```

The framework app would not need any changes to enable this behavior.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

ASP.NET to ASP.NET Core incremental session state migration

Article • 08/01/2023

Session State

Session state in ASP.NET Framework provided a number of features that ASP.NET Core does not provide. In order to update from ASP.NET Framework to Core, the adapters provide mechanisms to enable populating session state with similar behavior as `System.Web` did. Some of the differences between framework and core are:

- ASP.NET Framework would lock session usage within a session, so subsequent requests in a session are handled in a serial fashion. This is different than ASP.NET Core that does not provide any of these guarantees.
- ASP.NET Framework would serialize and deserialize objects automatically (unless being done in-memory). ASP.NET Core provides a mechanism to store a `byte[]` given a key. Any object serialization/deserialization has to be done manually by the user.

The adapter infrastructure exposes two interfaces that can be used to implement any session storage system. These are:

- `Microsoft.AspNetCore.SystemWebAdapters.ISessionManager`: This has a single method that gets passed an HttpContext and the session metadata and expects an `ISessionState` object to be returned.
- `Microsoft.AspNetCore.SystemWebAdapters.ISessionState`: This describes the state of a session object. It is used as the backing of the HttpSessionState type.

Serialization

Since the adapters provide the ability to work with strongly-typed session state, we must be able to serialize and deserialize types. This is customized through the `Microsoft.AspNetCore.SystemWebAdapters.SessionState.Serialization.ISessionKeySerializer`.

A default JSON implementation is provided that is configured via the `JsonSessionSerializerOptions`:

- `RegisterKey<T>(string)` - Registers a session key to a known type. This is required in order to serialize/deserialize the session state correctly. If a key is found that there is no registration for, an error will be thrown and session will not be available.

C#

```
builder.Services.AddSystemWebAdapters()
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be
registered to a type
        options.RegisterKey<int>("test-value");
    });
```

Implementations

There are two available implementations of the session state object that currently ship, each with some trade offs of features. The best choice for an app may depend on which part of the migration it is in, and may change over time.

- Strongly typed: Provides the ability to access an object and can be cast to the expected type
- Locking: Ensures multiple requests within a single session are queued up and aren't accessing the session at the same time
- Standalone: Use when you're not sharing session between ASP.NET Framework and ASP.NET Core to avoid modifying code in class libraries that references SessionState

Below are the available implementations:

Expand table

Implementation	Strongly typed	Locking	Standalone
Remote app	✔	✔	⛔
Wrapped ASP.NET Core	✔	⛔	✔

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see our contributor guide.

Open a documentation issue

Provide product feedback

ASP.NET to ASP.NET Core incremental IHttpModule migration

Article • 03/20/2024

Modules are types that implement IHttpModule and are used in ASP.NET Framework to hook into the request pipeline at various events. In an ASP.NET Core application, these should ideally be migrated to middleware. However, there are times when this cannot be done. In order to support migration scenarios in which modules are required and cannot be moved to middleware, System.Web adapters support adding them to ASP.NET Core.

IHttpModule Example

In order to support modules, an instance of HttpApplication must be available. If no custom HttpApplication is used, a default one will be used to add the modules to. Events declared in a custom application (including `Application_Start`) will be registered and run accordingly.

C#

```
using System.Web;
using Microsoft.AspNetCore.OutputCaching;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSystemWebAdapters()
    .AddHttpApplication<MyApp>(options =>
    {
        // Size of pool for HttpApplication instances. Should be what the
expected concurrent requests will be
        options.PoolSize = 10;

        // Register a module (optionally) by name
        options.RegisterModule<MyModule>("MyModule");
    });

// Only available in .NET 7+
builder.Services.AddOutputCache(options =>
{
    options.AddHttpApplicationBasePolicy(_ => new[] { "browser" });
});

builder.Services.AddAuthentication();
builder.Services.AddAuthorization();

var app = builder.Build();
```

```
app.UseAuthentication();
app.UseAuthenticationEvents();

app.UseAuthorization();
app.UseAuthorizationEvents();

app.UseSystemWebAdapters();
app.UseOutputCache();

app.MapGet("/", () => "Hello World!")
    .CacheOutput();

app.Run();

class MyApp : HttpApplication
{
    protected void Application_Start()
    {
    }

    public override string? GetVaryByCustomString(System.Web.HttpContext
context, string custom)
    {
        // Any custom vary-by string needed

        return base.GetVaryByCustomString(context, custom);
    }
}

class MyModule : IHttpModule
{
    public void Init(HttpApplication application)
    {
        application.BeginRequest += (s, e) =>
        {
            // Handle events at the beginning of a request
        };

        application.AuthorizeRequest += (s, e) =>
        {
            // Handle events that need to be authorized
        };
    }

    public void Dispose()
    {
    }
}
```

Global.asax migration

This infrastructure can be used to migrate usage of `Global.asax` if needed. The source from `Global.asax` is a custom HttpApplication and the file can be included in an ASP.NET Core application. Since it is named `Global`, the following code can be used to register it:

C#

```
builder.Services.AddSystemWebAdapters()
    .AddHttpApplication<Global>();
```

As long as the logic within it is available in ASP.NET Core, this approach can be used to incrementally migrate reliance on `Global.asax` to ASP.NET Core.

Authentication/Authorization events

In order for the authentication and authorization events to run at the desired time, the following pattern should be used:

C#

```
app.UseAuthentication();
app.UseAuthenticationEvents();

app.UseAuthorization();
app.UseAuthorizationEvents();
```

If this is not done, the events will still run. However, it will be during the call of `.UseSystemWebAdapters()`.

HTTP Module pooling

Because modules and applications in ASP.NET Framework were assigned to a request, a new instance is needed for each request. However, since they can be expensive to create, they are pooled using ObjectPool<T>. In order to customize the actual lifetime of the HttpApplication instances, a custom pool can be used:

C#

```
builder.Services.TryAddSingleton<ObjectPool<HttpApplication>>(sp =>
{
    // Recommended to use the in-built policy as that will ensure everything
is initialized correctly and is not intended to be replaced
    var policy = sp.GetRequiredService<IPooledObjectPolicy<HttpApplication>>
```

```
();

    // Can use any provider needed
    var provider = new DefaultObjectPoolProvider();

    // Use the provider to create a custom pool that will then be used for
the application.
    return provider.Create(policy);
});
```

Additional resources

- HTTP Module Migration
- HTTP Handlers and HTTP Modules Overview

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Remote app session state

Article • 08/01/2023

Remote app session state will enable communication between the ASP.NET Core and ASP.NET app to retrieve the session state. This is enabled by exposing an endpoint on the ASP.NET app that can be queried to retrieve and set the session state.

HttpSessionState serialization

The HttpSessionState object must be serialized for remote app session state to be enabled. This is accomplished through implementation of the type `Microsoft.AspNetCore.SystemWebAdapters.SessionState.Serialization.ISessionSerializer`, of which a default binary writer implementation is provided. This is added by the following code:

C#

```
builder.Services.AddSystemWebAdapters()
    .AddSessionSerializer(options =>
    {
        // Customize session serialization here
    });
```

Configuration

First, follow the remote app setup instructions to connect the ASP.NET Core and ASP.NET apps. Then, there are just a couple extra extension methods to call to enable remote app session state.

Configuration for ASP.NET Core involves calling `AddRemoteAppSession` and `AddJsonSessionSerializer` to register known session item types. The code should look similar to the following:

C#

```
builder.Services.AddSystemWebAdapters()
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be
registered to a type
        options.RegisterKey<int>("test-value");
        options.RegisterKey<SessionDemoModel>("SampleSessionItem");
    })
```

```
    .AddRemoteAppClient(options =>
    {
        // Provide the URL for the remote app that has enabled session
querying
        options.RemoteAppUrl =
new(builder.Configuration["ReverseProxy:Clusters:fallbackCluster:Destination
s:fallbackApp:Address"]);

        // Provide a strong API key that will be used to authenticate the
request on the remote app for querying the session
        options.ApiKey = builder.Configuration["RemoteAppApiKey"];
    })
    .AddSessionClient();
```

Session support requires additional work for the ASP.NET Core pipeline, and is not turned on by default. It can be configured on a per-route basis via ASP.NET Core metadata.

For example, session support requires either to annotate a controller:

C#

```
[Session]
public class SomeController : Controller
{
}
```

or to enable for all endpoints by default:

C#

```
app.MapDefaultControllerRoute()
    .RequireSystemWebAdapterSession();
```

The framework equivalent would look like the following change in `Global.asax.cs`:

C#

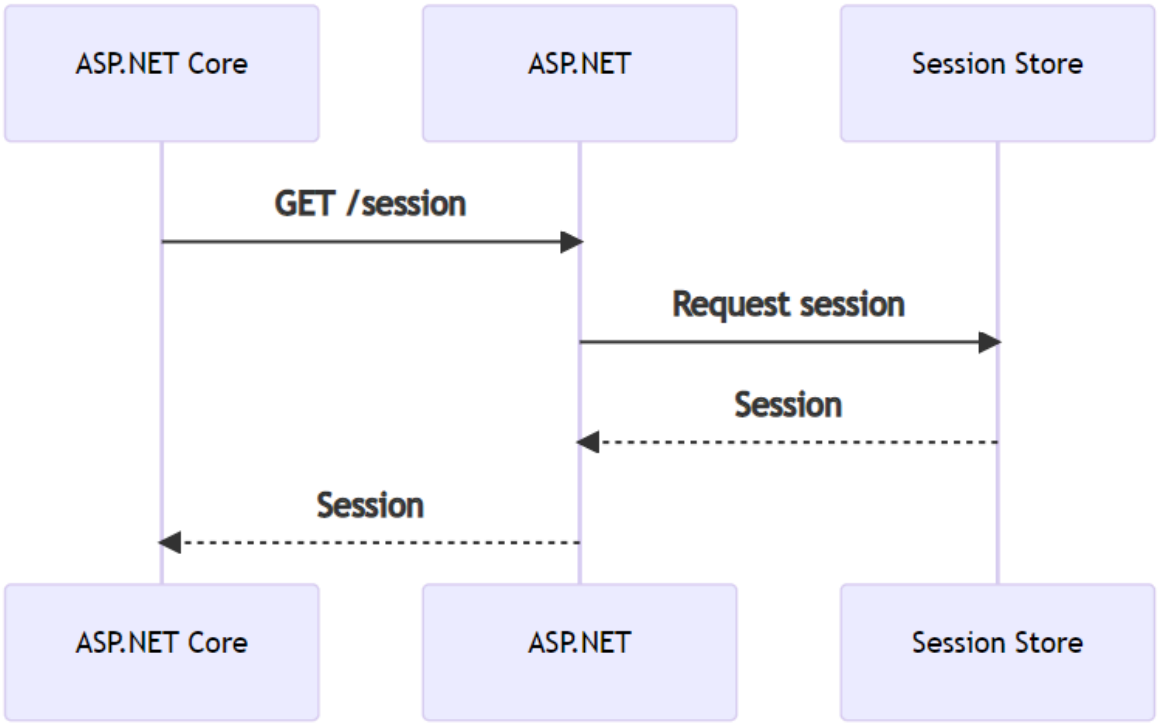
```
SystemWebAdapterConfiguration.AddSystemWebAdapters(this)
    .AddJsonSessionSerializer(options =>
    {
        // Serialization/deserialization requires each session key to be
registered to a type
        options.RegisterKey<int>("test-value");
        options.RegisterKey<SessionDemoModel>("SampleSessionItem");
    })
    // Provide a strong API key that will be used to authenticate the
request on the remote app for querying the session
    // ApiKey is a string representing a GUID
```

```
    .AddRemoteAppServer(options => options.ApiKey =
ConfigurationManager.AppSettings["RemoteAppApiKey"])
    .AddSessionServer();
```

Protocol

Readonly

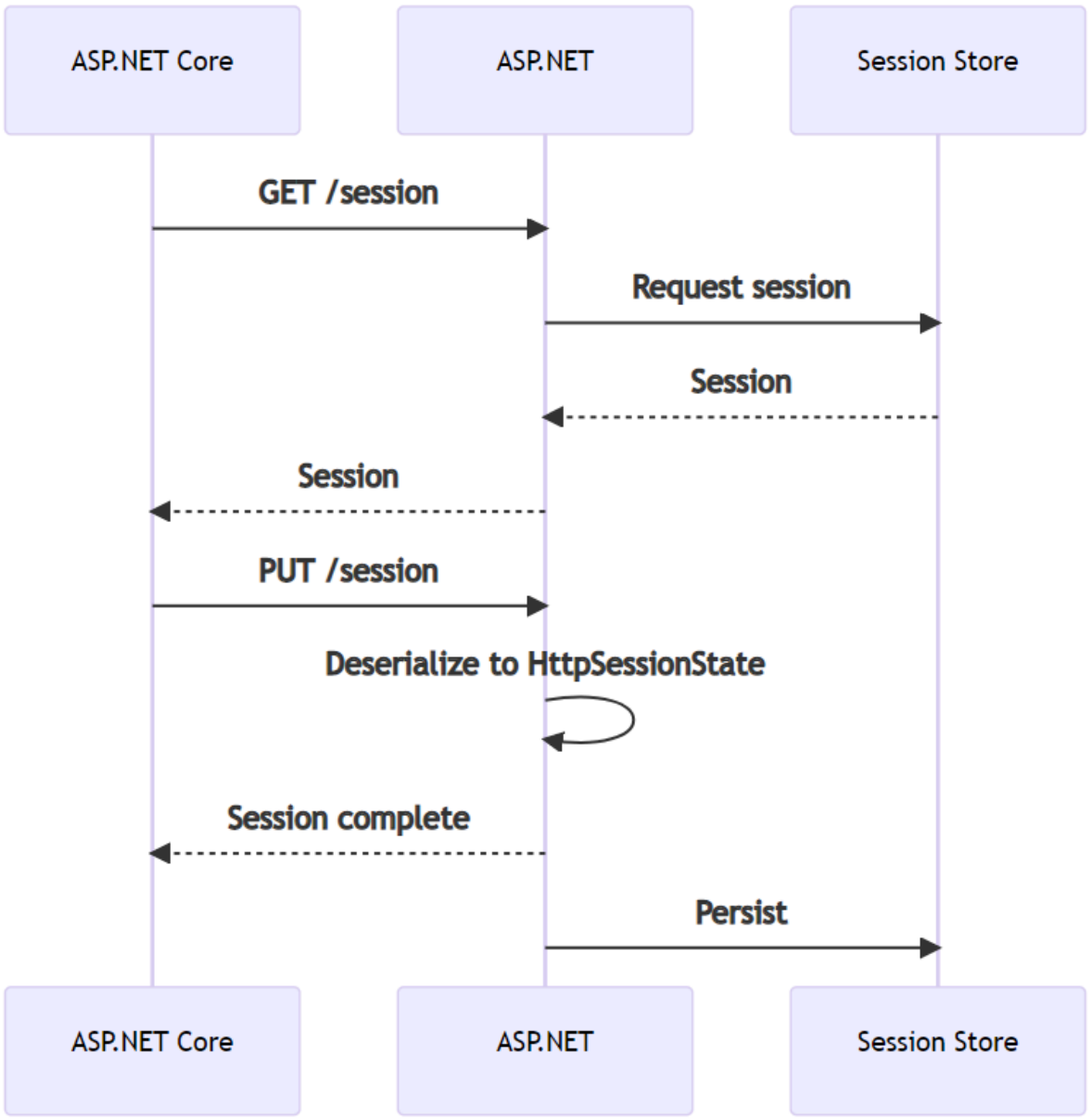
Readonly session will retrieve the session state from the framework app without any sort of locking. This consists of a single `GET` request that will return a session state and can be closed immediately.



Writeable

Writeable session state protocol starts with the same as the readonly, but differs in the following:

- Requires an additional `PUT` request to update the state
- The initial `GET` request must be kept open until the session is done; if closed, the session will not be able to be updated



Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Remote Authentication

Article • 08/05/2023

The System.Web adapters' remote authentication feature allows an ASP.NET Core app to determine a user's identity (authenticate an HTTP request) by deferring to an ASP.NET app. Enabling the feature adds an endpoint to the ASP.NET app that returns a serialized ClaimsPrincipal representing the authenticated user for any requests made to the endpoint. The ASP.NET Core app, then, registers a custom authentication handler that will (for endpoints with remote authentication enabled) determine a user's identity by calling that endpoint on the ASP.NET app and passing selected headers and cookies from the original request received by the ASP.NET Core app.

Configuration

There are just a few small code changes needed to enable remote authentication in a solution that's already set up according to the Getting Started.

First, follow the remote app setup instructions to connect the ASP.NET Core and ASP.NET apps. Then, there are just a couple extra extension methods to call to enable remote app authentication.

ASP.NET app configuration

The ASP.NET app needs to be configured to add the authentication endpoint. Adding the authentication endpoint is done by calling the `AddAuthenticationServer` extension method to set up the HTTP module that watches for requests to the authentication endpoint. Note that remote authentication scenarios typically want to add proxy support as well, so that any authentication related redirects correctly route to the ASP.NET Core app rather than the ASP.NET one.

C#

```
SystemWebAdapterConfiguration.AddSystemWebAdapters(this)
    .AddProxySupport(options => options.UseForwardedHeaders = true)
    .AddRemoteAppServer(options =>
    {
        options.ApiKey =
ConfigurationManager.AppSettings["RemoteAppApiKey"];
    })
    .AddAuthenticationServer();
```

ASP.NET Core app configuration

Next, the ASP.NET Core app needs to be configured to enable the authentication handler that will authenticate users by making an HTTP request to the ASP.NET app. Again, this is done by calling `AddAuthenticationClient` when registering System.Web adapters services:

```
C#

builder.Services.AddSystemWebAdapters()
    .AddRemoteAppClient(options =>
    {
        options.RemoteAppUrl = new Uri(builder.Configuration

["ReverseProxy:Clusters:fallbackCluster:Destinations:fallbackApp:Address"]);
        options.ApiKey = builder.Configuration["RemoteAppApiKey"];
    })
    .AddAuthenticationClient(true);
```

The boolean that is passed to the `AddAuthenticationClient` call specifies whether remote app authentication should be the default authentication scheme. Passing `true` will cause the user to be authenticated via remote app authentication for all requests, whereas passing `false` means that the user will only be authenticated with remote app authentication if the remote app scheme is specifically requested (with `[Authorize(AuthenticationSchemes = RemoteAppAuthenticationDefaults.AuthenticationScheme)]` on a controller or action method, for example). Passing false for this parameter has the advantage of only making HTTP requests to the original ASP.NET app for authentication for endpoints that require remote app authentication but has the disadvantage of requiring annotating all such endpoints to indicate that they will use remote app auth.

In addition to the require boolean, an optional callback may be passed to `AddAuthenticationClient` to modify some other aspects of the remote authentication process's behavior:

- `RequestHeadersToForward`: This property contains headers that should be forwarded from a request when calling the authenticate API. By default, the only headers forwarded are `Authorization` and `Cookie`. Additional headers can be forwarded by adding them to this list. Alternatively, if the list is cleared (so that no headers are specified), then all headers will be forwarded.
- `ResponseHeadersToForward`: This property lists response headers that should be propagated back from the authenticate request to the original call that prompted

authentication in scenarios where identity is challenged. By default, this includes `Location`, `Set-Cookie`, and `WWW-Authenticate` headers.

- `AuthenticationEndpointPath`: The endpoint on the ASP.NET app where authenticate requests should be made. This defaults to `/systemweb-adapters/authenticate` and must match the endpoint specified in the ASP.NET authentication endpoint configuration.

Finally, if the ASP.NET Core app didn't previously include authentication middleware, that will need to be enabled (after routing middleware, but before authorization middleware):

```
C#

app.UseAuthentication();
```

Design

1. When requests are processed by the ASP.NET Core app, if remote app authentication is the default scheme or specified by the request's endpoint, the `RemoteAuthenticationAuthHandler` will attempt to authenticate the user.
   a. The handler will make an HTTP request to the ASP.NET app's authenticate endpoint. It will copy configured headers from the current request onto this new one in order to forward auth-relevant data. As mentioned above, default behavior is to copy the `Authorize` and `Cookie` headers. The API key header is also added for security purposes.
2. The ASP.NET app will serve requests sent to the authenticate endpoint. As long as the API keys match, the ASP.NET app will return either the current user's ClaimsPrincipal serialized into the response body or it will return an HTTP status code (like 401 or 302) and response headers indicating failure.
3. When the ASP.NET Core app's `RemoteAuthenticationAuthHandler` receives the response from the ASP.NET app:
   a. If a ClaimsPrincipal was successfully returned, the auth handler will deserialize it and use it as the current user's identity.
   b. If a ClaimsPrincipal was not successfully returned, the handler will store the result and if authentication is challenged (because the user is accessing a protected resource, for example), the request's response will be updated with the status code and selected response headers from the response from the authenticate endpoint. This enables challenge responses (like redirects to a login page) to be propagated to end users.

i. Because results from the ASP.NET app's authenticate endpoint may include data specific to that endpoint, users can register `IRemoteAuthenticationResultProcessor` implementations with the ASP.NET Core app which will run on any authentication results before they are used. As an example, the one built-in `IRemoteAuthenticationResultProcessor` is `RedirectUrlProcessor` which looks for `Location` response headers returned from the authenticate endpoint and ensures that they redirect back to the host of the ASP.NET Core app and not the ASP.NET app directly.

Known limitations

This remote authentication approach has a couple known limitations:

1. Because Windows authentication depends on a handle to a Windows identity, Windows authentication is not supported by this feature. Future work is planned to explore how shared Windows authentication might work. See dotnet/systemweb-adapters#246 for more information.
2. This feature allows the ASP.NET Core app to make use of an identity authenticated by the ASP.NET app, but all actions related to users (logging on, logging off, etc.) still need to be routed through the ASP.NET app.

Alternatives

If authentication in the ASP.NET app is done using `Microsoft.Owin` Cookie Authentication Middleware, an alternative solution to sharing identity is to configure the ASP.NET and ASP.NET Core apps so that they are able to share an authentication cookie. Sharing an authentication cookie enables:

- Both apps to determine the user identity from the same cookie.
- Signing in or out of one app signs the user in or out of the other app.

Note that because signing in typically depends on a specific database, not all authentication functionality will work in both apps:

- Users should sign in through only one of the apps, either the ASP.NET or ASP.NET Core app, whichever the database is setup to work with.
- Both apps are able to see the users' identity and claims.
- Both apps are able to sign the user out.

Details on how to configure sharing auth cookies between ASP.NET and ASP.NET Core apps are available in cookie sharing documentation. The following samples in the

System.Web adapters GitHub repo demonstrates remote app authentication with shared cookie configuration enabling both apps to sign users in and out :

- ASP.NET app
- ASP.NET Core app

Sharing authentication is a good option if both the following are true:

- The ASP.NET app is already using `Microsoft.Owin` cookie authentication.
- It's possible to update the ASP.NET app and ASP.NET Core apps to use matching data protection settings. Matching shared data protection settings includes a shared file path, Redis cache, or Azure Blob Storage for storing data protection keys.

For other scenarios, the remote authentication approach described previously in this doc is more flexible and is probably a better fit.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Unit Testing

Article • 10/04/2023

In most cases, there's no need to set up additional components for running tests. But if the component being tested uses HttpRuntime, it might be necessary to start up the `SystemWebAdapters` service, as shown in the following example:

C#

```
namespace TestProject1;

/// <summary>
/// This demonstrates an xUnit feature that ensures all tests
/// in classes marked with this collection are run sequentially.
/// </summary>
[CollectionDefinition(nameof(SystemWebAdaptersHostedTests),
    DisableParallelization = true)]
public class SystemWebAdaptersHostedTests
{
}
[Collection(nameof(SystemWebAdaptersHostedTests))]
public class RuntimeTests
{
    /// <summary>
    /// This method starts up a host in the background that
    /// makes it possible to initialize <see cref="HttpRuntime"/>
    /// and <see cref="HostingEnvironment"/> with values needed
    /// for testing with the <paramref name="configure"/> option.
    /// </summary>
    /// <param name="configure">
    /// Configuration for the hosting and runtime options.
    /// </param>
    public static async Task<IDisposable> EnableRuntimeAsync(
        Action<SystemWebAdaptersOptions>? configure = null,
        CancellationToken token = default)
        => await new HostBuilder()
           .ConfigureWebHost(webBuilder =>
           {
               webBuilder
                   .UseTestServer()
                   .ConfigureServices(services =>
                   {
                       services.AddSystemWebAdapters();
                       if (configure is not null)
                       {
                           services.AddOptions
                               <SystemWebAdaptersOptions>()
                               .Configure(configure);
                       }
                   })
                   .Configure(app =>
```

```
                    {
                        // No need to configure pipeline for tests
                    });
            })
            .StartAsync(token);
    [Fact]
    public async Task RuntimeEnabled()
    {
        using (await EnableRuntimeAsync(options =>
            options.AppDomainAppPath = "path"))
        {
            Assert.True(HostingEnvironment.IsHosted);
            Assert.Equal("path", HttpRuntime.AppDomainAppPath);
        }
        Assert.False(HostingEnvironment.IsHosted);
    }
}
```

The tests must be executed in sequence, not in parallel. The preceding example illustrates how to achieve this by setting XUnit's DisableParallelization option to `true`. This setting disables parallel execution for a specific test collection, ensuring that the tests within that collection run one after the other, without concurrency.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

A/B Testing endpoints during migration

Article • 03/04/2023

During incremental migration, new endpoints are brought over to a YARP enabled ASP.NET Core app. With the default setup, these endpoints are automatically served for all requests once deployed. In order to test these endpoints, or be able to turn them off if needed, additional setup is needed.

This document describes how to setup a conditional endpoint selection system to enable A/B testing during incremental migration. It assumes a setup as described in incremental migration overview as a starting point.

Conditional endpoint selection

To enable conditional endpoint selection, a few services need to be defined:

1. Metadata that can be added to an endpoint to turn on any conditional related logic. Endpoints include controllers, minimal APIs, Razor Page, etc. If this metadata isn't added to an endpoint, no conditional checks are performed for it.

   CSharp

   ```
   [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
   public sealed class ConditionalRouteAttribute : Attribute
   {
       public bool IsConditional { get; set; } = true;
   }
   ```

   The property `IsConditional` is useful to set it `true` at a global level, and then allow lower levels, such as controllers or routes, to turn off the conditional checks.

2. The API we want to implement to make a decision per request and endpoint selection:

   CSharp

   ```
   public interface IConditionalEndpointSelector
   {
       ValueTask<bool> IsEnabledAsync(HttpContext context, Endpoint
   candidate);
   }
   ```

3. A `MatcherPolicy` that is used to hook into routing and call the custom selector:

CSharp

```
public static class ConditionalEndpointExtensions
{
    /// <summary>
    /// Registers a <see cref="IConditionalEndpointSelector"/> that is
called when selecting
    /// an endpoint that has been marked as conditional by <see
cref="WithConditionalRoute{TBuilder}(TBuilder)"/>.
    /// </summary>
    /// <typeparam name="T">Type of selector</typeparam>
    /// <param name="services">Service collection to add to.</param>
    public static void AddConditionalEndpoints<T>(this
IServiceCollection services)
        where T : class, IConditionalEndpointSelector
    {
        services.AddTransient<IConditionalEndpointSelector, T>();

services.TryAddEnumerable(ServiceDescriptor.Singleton<MatcherPolicy,
ConditionalEndpointMatcherPolicy>());
    }

    /// <summary>
    /// Enable conditional behavior for supplied endpoints. An
implementation of <see cref="IConditionalEndpointSelector"/> must be
registered as a service for this to be enabled at runtime.
    /// </summary>
    /// <param name="builder">The endpoint convention builder</param>
    /// <returns>The original convention builder.</returns>
    public static TBuilder WithConditionalRoute<TBuilder>(this TBuilder
builder)
        where TBuilder : IEndpointConventionBuilder
    {
        ArgumentNullException.ThrowIfNull(builder);

        return builder.WithMetadata(new ConditionalRouteAttribute());
    }

    private sealed class ConditionalEndpointMatcherPolicy :
MatcherPolicy, IEndpointSelectorPolicy
    {
        private readonly IConditionalEndpointSelector _selector;

        public
ConditionalEndpointMatcherPolicy(IConditionalEndpointSelector selector)
        {
            _selector = selector;
        }

        public override int Order => 0;

        public bool AppliesToEndpoints(IReadOnlyList<Endpoint>
endpoints)
            => endpoints.Any(e =>
```

```
e.Metadata.GetMetadata<ConditionalRouteAttribute>() is { IsConditional:
true });

        public async Task ApplyAsync(HttpContext httpContext,
CandidateSet candidates)
        {
            for (int i = 0; i < candidates.Count; i++)
            {
                var endpoint = candidates[i].Endpoint;

                if
(endpoint.Metadata.GetMetadata<ConditionalRouteAttribute>() is {
IsConditional: true })
                {
                    if (await _selector.IsEnabledAsync(httpContext,
endpoint) == false)
                    {
                        candidates.SetValidity(i, false);
                    }
                }
            }
        }
    }
}
```

4. Implement a custom selector. As an example, this defines a check for the presence of a query parameter (`IgnoreLocal`) that turns off the local endpoint and uses the YARP endpoint instead.

CSharp

```
public class QueryConditionalSelector : IConditionalEndpointSelector
{
    public ValueTask<bool> IsEnabledAsync(HttpContext context, Endpoint
candidate)
    {
        var result = context.Request.Query.TryGetValue("IgnoreLocal",
out var values) &&
            values is { Count: 1 } &&
            bool.TryParse(values[0], out var skip)
            && skip;

        return ValueTask.FromResult(result);
    }
```

5. Register the services in the program startup and mark the controllers for conditional selection. This marking can also be done using attribute marking on controllers or routes if needed.

diff

```
using Microsoft.AspNetCore.SystemWebAdapters;

var builder = WebApplication.CreateBuilder();
builder.Services.AddReverseProxy().LoadFromConfig(builder.Configuration
.GetSection("ReverseProxy"));

// Add services to the container.
builder.Services.AddControllersWithViews();
builder.Services.AddSystemWebAdapters();
+ builder.Services.AddConditionalEndpoint<QueryConditionalSelector>();

var app = builder.Build();

app.UseSystemWebAdapters();

- app.MapDefaultControllerRoute();
+ app.MapDefaultControllerRoute().WithConditionalRoute();
app.MapReverseProxy();

app.Run();
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Upgrade from ASP.NET MVC and Web API to ASP.NET Core MVC

Article • 03/10/2023

This article shows how to upgrade an ASP.NET Framework MVC or Web API app to ASP.NET Core MVC using the Visual Studio .NET Upgrade Assistant and the incremental update approach.

Upgrade using the .NET Upgrade Assistant

If your .NET Framework project has supporting libraries in the solution that are required, they should be upgraded to .NET Standard 2.0, if possible. For more information, see Upgrade supporting libraries.

1. Install the .NET Upgrade Assistant Visual Studio extension.
2. Open the ASP.NET MVC or Web API solution in Visual Studio.
3. In **Solution Explorer**, right click on the project to upgrade and select **Upgrade**. Select **Side-by-side incremental project upgrade**, which is the only upgrade option.
4. For the upgrade target, select **New project**.
5. Name the project and select the template. If the project you're migrating is a API project, select **ASP.NET Core Web API**. If it's an MVC project or MVC and Web API, select **ASP.NET Core MVC**.
6. Select **Next**
7. Select the target framework version and then select **Next**. For more information, see .NET and .NET Core Support Policy.
8. Review the **Summary of changes**, then select **Finish**.
9. The **Summary** step displays `<Framework Project>` **is now connected to** `<Framework ProjectCore>` **via Yarp proxy.** and a pie chart showing the migrated endpoints. Select **Upgrade Controller** and then select a controller to upgrade.
10. Select the component to upgrade, then select **Upgrade selection**.

Incremental update

Follow the steps in Get started with incremental ASP.NET to ASP.NET Core migration to continue the update process.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

The ASP.NET Core documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

Upgrade an ASP.NET Framework Web Forms app to ASP.NET Core MVC

Article • 05/23/2023

This article shows how to upgrade an ASP.NET Framework Web Forms to ASP.NET Core MVC using the Visual Studio .NET Upgrade Assistant and the incremental update approach.

If your .NET Framework project has supporting libraries in its solution that are required, they should be upgraded to .NET Standard 2.0, if possible. For more information, see Upgrade supporting libraries.

1. Install the .NET Upgrade Assistant Visual Studio extension.
2. Open the ASP.NET Web Forms solution in Visual Studio.
3. In **Solution Explorer**, right click on the project to upgrade and select **Upgrade**. Select **Side-by-side incremental project upgrade**, which is the only upgrade option.
4. For the upgrade target, select **New project**.
5. Name the project and select the **ASP.NET Core** template and then select **Next**
6. Select the target framework version and then select **Next**. For more information, see .NET and .NET Core Support Policy.
7. Select **Done**, then select **Finish**.
8. The **Summary** step displays **`<Framework Project>` is now connected to `<Framework ProjectCore>` via Yarp proxy.**.
9. Select the component to upgrade, then select **Upgrade selection**.

Incremental update

Follow the steps in Get started with incremental ASP.NET to ASP.NET Core migration to continue the update process.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

more information, see our contributor guide.

Provide product feedback

Migrate from ASP.NET Web API to ASP.NET Core

Article • 09/28/2023

ASP.NET Core combines ASP.NET 4.x's MVC and Web API app models into a single programming model known as ASP.NET Core MVC.

This article shows how to migrate the Products controller created in Getting Started with ASP.NET Web API 2 to ASP.NET Core.

Prerequisites

Visual Studio

- Visual Studio 2022 with the **ASP.NET and web development** workload.
- .NET 6.0 SDK

Create the new ASP.NET Core Web API project

Visual Studio

1. From the **File** menu, select **New** > **Project**.
2. Enter *Web API* in the search box.
3. Select the **ASP.NET Core Web API** template and select **Next**.
4. In the **Configure your new project** dialog, name the project *ProductsCore* and select **Next**.
5. In the **Additional information** dialog:
   a. Confirm the **Framework** is **.NET 6.0 (Long-term support)**.
   b. Confirm the checkbox for **Use controllers(uncheck to use minimal APIs)** is checked.
   c. Uncheck **Enable OpenAPI support**.
   d. Select **Create**.

Remove the *WeatherForecast* template files

1. Remove the `WeatherForecast.cs` and `Controllers/WeatherForecastController.cs` example files from the new *ProductsCore* project.
2. Open *Properties\launchSettings.json*.
3. Change `launchUrl` properties from `weatherforcast` to `productscore`.

The configuration for ASP.NET Core Web API

ASP.NET Core doesn't use the *App_Start* folder or the *Global.asax* file. The *web.config* file is added at publish time. For more information, see web.config file.

The `Program.cs` file:

- Replaces *Global.asax*.
- Handles all app startup tasks.

For more information, see App startup in ASP.NET Core.

The following shows the application startup code in the ASP.NET Core `Program.cs` file:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Copy the *Product* model

Visual Studio

1. In **Solution Explorer**, right-click the project. Select **Add** > **New Folder**. Name the folder *Models*.

2. Right-click the **Models** folder. Select **Add** > **Class**. Name the class *Product* and select **Add**.
3. Replace the template model code with the following:

C#

```
namespace ProductsCore.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

The preceding highlighted code changes the following:

- The `?` annotation has been added to declare the `Name` and `Category` properties as nullable reference types.

By utilizing the Nullable feature introduced in C# 8, ASP.NET Core can provide additional code flow analysis and compile-time safety in the handling of reference types. For example, protecting against `null` reference exceptions.

In this case, the intent is that the `Name` and `Category` can be nullable types.

ASP.NET Core 6.0 projects enable nullable reference types by default. For more information, see Nullable reference types.

Copy the *ProductsController*

Visual Studio

1. Right-click the **Controllers** folder.
2. Select **Add** > **Controller...**.
3. In **Add New Scaffolded Item** dialog, select **Mvc Controller - Empty** then select **Add**.
4. Name the controller *ProductsController* and select **Add**.
5. Replace the template controller code with the following:

C#

```
using Microsoft.AspNetCore.Mvc;
using ProductsCore.Models;

namespace ProductsCore.Controllers;

[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    Product[] products = new Product[]
    {
            new Product
            {
                Id = 1, Name = "Tomato Soup", Category = "Groceries", Price
= 1
            },
            new Product
            {
                Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M
            },
            new Product
            {
                Id = 3, Name = "Hammer", Category = "Hardware", Price =
16.99M
            }
    };

    [HttpGet]
    public IEnumerable<Product> GetAllProducts()
    {
        return products;
    }

    [HttpGet("{id}")]
    public ActionResult<Product> GetProduct(int id)
    {
        var product = products.FirstOrDefault((p) => p.Id == id);
        if (product == null)
        {
            return NotFound();
        }
        return product;
    }
}
```

The preceding highlighted code changes the following, to migrate to ASP.NET Core:

- Removes using statements for the following ASP.NET 4.x components that don't exist in ASP.NET Core:
  - `ApiController` class

    - `System.Web.Http` namespace
    - `IHttpActionResult` interface
- Changes the `using ProductsApp.Models;` statement to `using ProductsCore.Models;`.
- Sets the root namespace to `ProductsCore`.
- Changes `ApiController` to ControllerBase.
- Adds `using Microsoft.AspNetCore.Mvc;` to resolve the `ControllerBase` reference.
- Changes the `GetProduct` action's return type from `IHttpActionResult` to `ActionResult<Product>`. For more info, see Controller action return types.
- Simplifies the `GetProduct` action's `return` statement to the following statement:

```
C#

return product;
```

- Adds the following attributes which are explained in the next sections:
    - `[Route("api/[controller]")]`
    - `[ApiController]`
    - `[HttpGet]`
    - `[HttpGet("{id}")]`

Routing

ASP.NET Core provides a minimal hosting model in which the endpoint routing middleware wraps the entire middleware pipeline, therefore routes can be added directly to the WebApplication without an explicit call to UseEndpoints or UseRouting to register routes.

`UseRouting` can still be used to specify where route matching happens, but `UseRouting` doesn't need to be explicitly called if routes should be matched at the beginning of the middleware pipeline.

```
C#

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
```

```
builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Note: Routes added directly to the WebApplication execute at the ***end*** of the pipeline.

Routing in the migrated `ProductsController`

The migrated `ProductsController` contains the following highlighted attributes:

C#

```
using Microsoft.AspNetCore.Mvc;
using ProductsCore.Models;

namespace ProductsCore.Controllers;

[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    Product[] products = new Product[]
    {
            new Product
            {
                Id = 1, Name = "Tomato Soup", Category = "Groceries", Price
= 1
            },
            new Product
            {
                Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M
            },
            new Product
            {
                Id = 3, Name = "Hammer", Category = "Hardware", Price =
16.99M
            }
    };

    [HttpGet]
    public IEnumerable<Product> GetAllProducts()
```

```
    {
        return products;
    }

    [HttpGet("{id}")]
    public ActionResult<Product> GetProduct(int id)
    {
        var product = products.FirstOrDefault((p) => p.Id == id);
        if (product == null)
        {
            return NotFound();
        }
        return product;
    }
}
```

- The [Route] attribute configures the controller's attribute routing pattern.
- The [ApiController] attribute makes attribute routing a requirement for all actions in this controller.
- Attribute routing supports tokens, such as `[controller]` and `[action]`. At runtime, each token is replaced with the name of the controller or action, respectively, to which the attribute has been applied. The tokens:
  - Reduces or eliminates the need to use hard coded strings for the route.
  - Ensure routes remain synchronized with the corresponding controllers and actions when automatic rename refactorings are applied.
- HTTP Get requests are enabled for `ProductController` actions with the following attributes:
  - [HttpGet] attribute applied to the `GetAllProducts` action.
  - `[HttpGet("{id}")]` attribute applied to the `GetProduct` action.

Run the migrated project, and browse to `/api/products`. For example: https://localhost:`<port>`/api/products. A full list of three products appears. Browse to `/api/products/1`. The first product appears.

View or download sample code (how to download)

Additional resources

- Create web APIs with ASP.NET Core
- Controller action return types in ASP.NET Core web API
- Compatibility version for ASP.NET Core MVC

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

- Open a documentation issue
- Provide product feedback

Migrate configuration to ASP.NET Core

Article • 10/04/2024

In the previous article, we began to migrate an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate configuration.

View or download sample code (how to download)

Setup configuration

ASP.NET Core no longer uses the *Global.asax* and *web.config* files that previous versions of ASP.NET utilized. In the earlier versions of ASP.NET, application startup logic was placed in an `Application_StartUp` method within *Global.asax*. Later, in ASP.NET MVC, a `Startup.cs` file was included in the root of the project; and, it was called when the application started. ASP.NET Core has adopted this approach completely by placing all startup logic in the `Startup.cs` file.

The *web.config* file has also been replaced in ASP.NET Core. Configuration itself can now be configured, as part of the application startup procedure described in `Startup.cs`. Configuration can still utilize XML files, but typically ASP.NET Core projects will place configuration values in a JSON-formatted file, such as `appsettings.json`. ASP.NET Core's configuration system can also easily access environment variables, which can provide a more secure and robust location for environment-specific values. This is especially true for secrets like connection strings and API keys that shouldn't be checked into source control. See Configuration to learn more about configuration in ASP.NET Core.

> ⚠ **Warning**
>
> This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

For this article, we are starting with the partially migrated ASP.NET Core project from the previous article. To setup configuration, add the following constructor and property to the `Startup.cs` file located in the root of the project:

```
C#
```

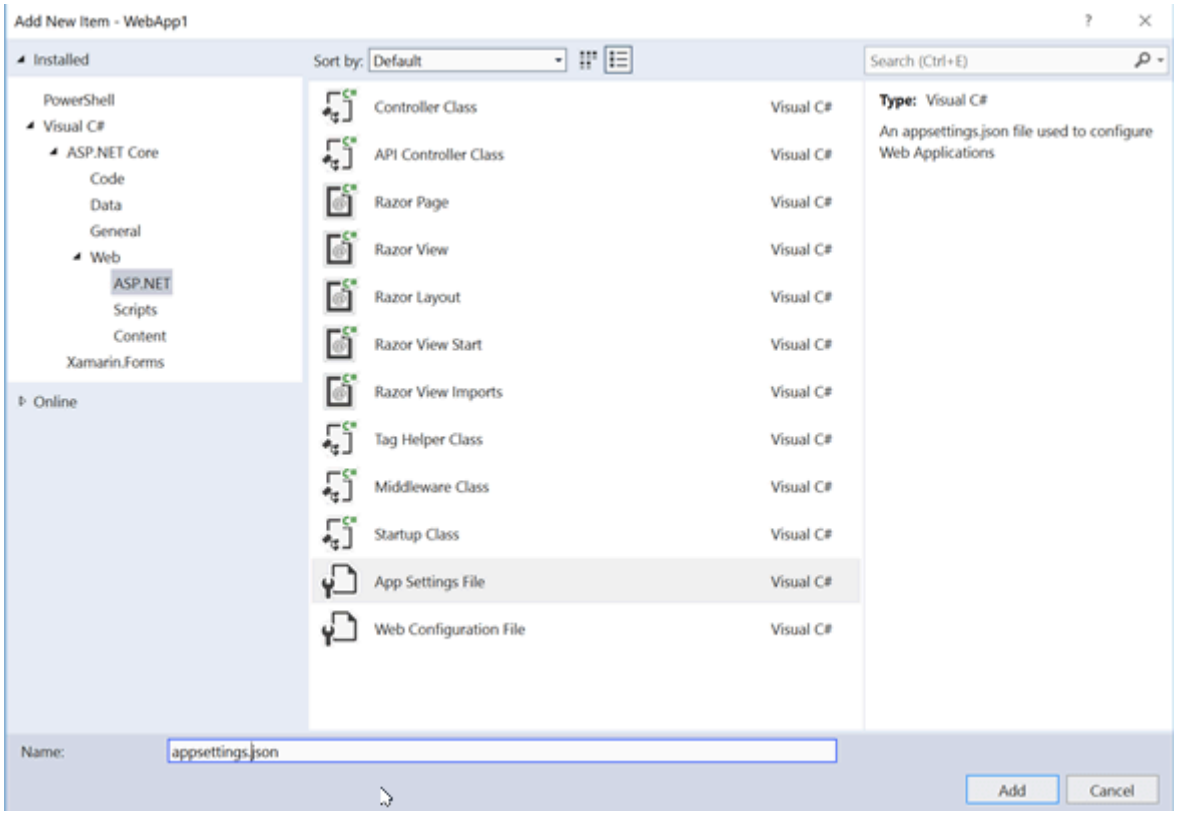
```
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

public IConfiguration Configuration { get; }
```

Note that at this point, the `Startup.cs` file won't compile, as we still need to add the following `using` statement:

C#

```
using Microsoft.Extensions.Configuration;
```

Add an `appsettings.json` file to the root of the project using the appropriate item template:



Migrate configuration settings from web.config

Our ASP.NET MVC project included the required database connection string in *web.config*, in the `<connectionStrings>` element. In our ASP.NET Core project, we are going to store this information in the `appsettings.json` file. Open `appsettings.json`, and note that it already includes the following:

```
```

JSON

```
{
    "Data": {
        "DefaultConnection": {
            "ConnectionString": "Server=
(localdb)\\MSSQLLocalDB;Database=_CHANGE_ME;Trusted_Connection=True;"
        }
    }
}
```

In the highlighted line depicted above, change the name of the database from **_CHANGE_ME** to the name of your database.

Summary

ASP.NET Core places all startup logic for the application in a single file, in which the necessary services and dependencies can be defined and configured. It replaces the *web.config* file with a flexible configuration feature that can leverage a variety of file formats, such as JSON, as well as environment variables.

Migrate Authentication and Identity to ASP.NET Core

Article • 10/30/2024

By Steve Smith

In the previous article, we migrated configuration from an ASP.NET MVC project to ASP.NET Core MVC. In this article, we migrate the registration, login, and user management features.

Configure Identity and Membership

In ASP.NET MVC, authentication and identity features are configured using ASP.NET Identity in `Startup.Auth.cs` and `IdentityConfig.cs`, located in the *App_Start* folder. In ASP.NET Core MVC, these features are configured in `Startup.cs`.

Install the following NuGet packages:

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`
- `Microsoft.AspNetCore.Authentication.Cookies`
- `Microsoft.EntityFrameworkCore.SqlServer`

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

In `Startup.cs`, update the `Startup.ConfigureServices` method to use Entity Framework and Identity services:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    // Add EF services to the services container.
```

```
    services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"))
);

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

     services.AddMvc();
}
```

At this point, there are two types referenced in the above code that we haven't yet migrated from the ASP.NET MVC project: `ApplicationDbContext` and `ApplicationUser`. Create a new *Models* folder in the ASP.NET Core project, and add two classes to it corresponding to these types. You will find the ASP.NET MVC versions of these classes in `/Models/IdentityModels.cs`, but we will use one file per class in the migrated project since that's more clear.

`ApplicationUser.cs`:

C#

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace NewMvcProject.Models
{
    public class ApplicationUser : IdentityUser
    {
    }
}
```

`ApplicationDbContext.cs`:

C#

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.Data.Entity;

namespace NewMvcProject.Models
{
    public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options)
            : base(options)
        {
        }
```

```
        protected override void OnModelCreating(ModelBuilder builder)
        {
            base.OnModelCreating(builder);
            // Customize the ASP.NET Core Identity model and override the
defaults if needed.
            // For example, you can rename the ASP.NET Core Identity table
names and more.
            // Add your customizations after calling
base.OnModelCreating(builder);
        }
    }
}
```

The ASP.NET Core MVC Starter Web project doesn't include much customization of users, or the `ApplicationDbContext`. When migrating a real app, you also need to migrate all of the custom properties and methods of your app's user and `DbContext` classes, as well as any other Model classes your app utilizes. For example, if your `DbContext` has a `DbSet<Album>`, you need to migrate the `Album` class.

With these files in place, the `Startup.cs` file can be made to compile by updating its `using` statements:

C#

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
```

Our app is now ready to support authentication and Identity services. It just needs to have these features exposed to users.

Migrate registration and login logic

With Identity services configured for the app and data access configured using Entity Framework and SQL Server, we're ready to add support for registration and login to the app. Recall that earlier in the migration process we commented out a reference to *_LoginPartial* in `_Layout.cshtml`. Now it's time to return to that code, uncomment it, and add in the necessary controllers and views to support login functionality.

Uncomment the `@Html.Partial` line in `_Layout.cshtml`:

CSHTML

```
      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
    </ul>
    @*@Html.Partial("_LoginPartial")*@
  </div>
</div>
```

Now, add a new Razor view called *_LoginPartial* to the *Views/Shared* folder:

Update `_LoginPartial.cshtml` with the following code (replace all of its contents):

CSHTML

```
@inject SignInManager<ApplicationUser> SignInManager
@inject UserManager<ApplicationUser> UserManager

@if (SignInManager.IsSignedIn(User))
{
    <form asp-area="" asp-controller="Account" asp-action="Logout"
method="post" id="logoutForm" class="navbar-right">
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-area="" asp-controller="Manage" asp-action="Index"
title="Manage">Hello @UserManager.GetUserName(User)!</a>
            </li>
            <li>
                <button type="submit" class="btn btn-link navbar-btn navbar-
link">Log out</button>
            </li>
        </ul>
    </form>
}
else
{
    <ul class="nav navbar-nav navbar-right">
        <li><a asp-area="" asp-controller="Account" asp-
action="Register">Register</a></li>
        <li><a asp-area="" asp-controller="Account" asp-action="Login">Log
in</a></li>
    </ul>
}
```

At this point, you should be able to refresh the site in your browser.

Summary

ASP.NET Core introduces changes to the ASP.NET Identity features. In this article, you have seen how to migrate the authentication and user management features of ASP.NET Identity to ASP.NET Core.

Migrate from ClaimsPrincipal.Current

Article • 06/03/2022

In ASP.NET 4.x projects, it was common to use ClaimsPrincipal.Current to retrieve the current authenticated user's identity and claims. In ASP.NET Core, this property is no longer set. Code that was depending on it needs to be updated to get the current authenticated user's identity through a different means.

Context-specific state instead of static state

When using ASP.NET Core, the values of both `ClaimsPrincipal.Current` and `Thread.CurrentPrincipal` aren't set. These properties both represent static state, which ASP.NET Core generally avoids. Instead, ASP.NET Core uses dependency injection (DI) to provide dependencies such as the current user's identity. Getting the current user's identity from DI is more testable, too, since test identities can be easily injected.

Retrieve the current user in an ASP.NET Core app

There are several options for retrieving the current authenticated user's `ClaimsPrincipal` in ASP.NET Core in place of `ClaimsPrincipal.Current`:

- **ControllerBase.User**. MVC controllers can access the current authenticated user with their User property.
- **HttpContext.User**. Components with access to the current `HttpContext` (middleware, for example) can get the current user's `ClaimsPrincipal` from HttpContext.User.
- **Passed in from caller**. Libraries without access to the current `HttpContext` are often called from controllers or middleware components and can have the current user's identity passed as an argument.
- **IHttpContextAccessor**. The project being migrated to ASP.NET Core may be too large to easily pass the current user's identity to all necessary locations. In such cases, IHttpContextAccessor can be used as a workaround. `IHttpContextAccessor` is able to access the current `HttpContext` (if one exists). If DI is being used, see Access HttpContext in ASP.NET Core. A short-term solution to getting the current

user's identity in code that hasn't yet been updated to work with ASP.NET Core's DI-driven architecture would be:

- Make `IHttpContextAccessor` available in the DI container by calling AddHttpContextAccessor in `Startup.ConfigureServices`.
- Get an instance of `IHttpContextAccessor` during startup and store it in a static variable. The instance is made available to code that was previously retrieving the current user from a static property.
- Retrieve the current user's `ClaimsPrincipal` using `HttpContextAccessor.HttpContext?.User`. If this code is used outside of the context of an HTTP request, the `HttpContext` is null.

The final option, using an `IHttpContextAccessor` instance stored in a static variable, is contrary to the ASP.NET Core principle of preferring injected dependencies to static dependencies. Plan to eventually retrieve `IHttpContextAccessor` instances from DI instead. A static helper can be a useful bridge, though, when migrating large existing ASP.NET apps that use `ClaimsPrincipal.Current`.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback

Migrate from ASP.NET Membership authentication to ASP.NET Core 2.0 Identity

Article • 10/29/2024

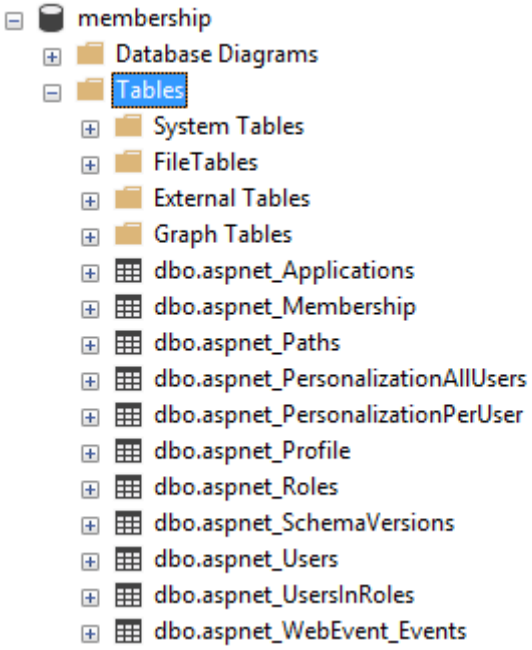
By Isaac Levin

This article demonstrates migrating the database schema for ASP.NET apps using Membership authentication to ASP.NET Core 2.0 Identity.

> **Note**
>
> This document provides the steps needed to migrate the database schema for ASP.NET Membership-based apps to the database schema used for ASP.NET Core Identity. For more information about migrating from ASP.NET Membership-based authentication to ASP.NET Identity, see Migrate an existing app from SQL Membership to ASP.NET Identity. For more information about ASP.NET Core Identity, see Introduction to Identity on ASP.NET Core.

Review of Membership schema

Prior to ASP.NET 2.0, developers were tasked with creating the entire authentication and authorization process for their apps. With ASP.NET 2.0, Membership was introduced, providing a boilerplate solution to handling security within ASP.NET apps. Developers were now able to bootstrap a schema into a SQL Server database with the ASP.NET SQL Server Registration Tool (`Aspnet_regsql.exe`) (no longer supported). After running this command, the following tables were created in the database.



To migrate existing apps to ASP.NET Core 2.0 Identity, the data in these tables needs to be migrated to the tables used by the new Identity schema.

ASP.NET Core Identity 2.0 schema

ASP.NET Core 2.0 follows the Identity principle introduced in ASP.NET 4.5. Though the principle is shared, the implementation between the frameworks is different, even between versions of ASP.NET Core (see Migrate authentication and Identity to ASP.NET Core 2.0).

The fastest way to view the schema for ASP.NET Core 2.0 Identity is to create a new ASP.NET Core 2.0 app. Follow these steps in Visual Studio 2017:

1. Select **File** > **New** > **Project**.
2. Create a new **ASP.NET Core Web Application** project named *CoreIdentitySample*.
3. Select **ASP.NET Core 2.0** in the dropdown and then select **Web Application**. This template produces a Razor Pages app. Before clicking **OK**, click **Change Authentication**.
4. Choose **Individual User Accounts** for the Identity templates. Finally, click **OK**, then **OK**. Visual Studio creates a project using the ASP.NET Core Identity template.
5. Select **Tools** > **NuGet Package Manager** > **Package Manager Console** to open the **Package Manager Console** (PMC) window.
6. Navigate to the project root in PMC, and run the Entity Framework (EF) Core `Update-Database` command.

ASP.NET Core 2.0 Identity uses EF Core to interact with the database storing the authentication data. In order for the newly created app to work, there needs to be a database to store this data. After creating a new app, the fastest way to inspect the schema in a database environment is to create the database using EF Core Migrations. This process creates a database, either locally or elsewhere, which mimics that schema. Review the preceding documentation for more information.

EF Core commands use the connection string for the database specified in `appsettings.json`. The following connection string targets a database on *localhost* named *asp-net-core-identity*. In this setting, EF Core is configured to use the `DefaultConnection` connection string.

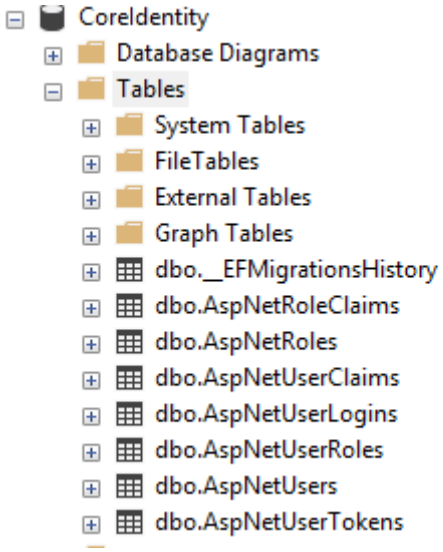
JSON

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=aspnet-core-
identity;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

> ⚠ **Warning**
>
> This article shows the use of connection strings. With a local database the user doesn't have to be authenticated, but in production, connection strings sometimes include a password to authenticate. A resource owner password credential (ROPC) is a security risk that should be avoided in production databases. Production apps should use the most secure authentication flow available. For more information on authentication for apps deployed to test or production environments, see Secure authentication flows.

1. Select **View** > **SQL Server Object Explorer**. Expand the node corresponding to the database name specified in the `ConnectionStrings:DefaultConnection` property of `appsettings.json`.

   The `Update-Database` command created the database specified with the schema and any data needed for app initialization. The following image depicts the table structure that's created with the preceding steps.



Migrate the schema

There are subtle differences in the table structures and fields for both Membership and ASP.NET Core Identity. The pattern has changed substantially for authentication/authorization with ASP.NET and ASP.NET Core apps. The key objects that are still used with Identity are *Users* and *Roles*. Here are mapping tables for *Users*, *Roles*, and *UserRoles*.

Users

Expand table

Identity (`dbo.AspNetUsers`) column	**Type**	**Membership (`dbo.aspnet_Users` / `dbo.aspnet_Membership`) column**	**Type**
`Id`	`string`	`aspnet_Users.UserId`	`string`
`UserName`	`string`	`aspnet_Users.UserName`	`string`
`Email`	`string`	`aspnet_Membership.Email`	`string`
`NormalizedUserName`	`string`	`aspnet_Users.LoweredUserName`	`string`
`NormalizedEmail`	`string`	`aspnet_Membership.LoweredEmail`	`string`
`PhoneNumber`	`string`	`aspnet_Users.MobileAlias`	`string`
`LockoutEnabled`	`bit`	`aspnet_Membership.IsLockedOut`	`bit`

`IsLockedOut` doesn't map to `LockoutEnabled`. `IsLockedOut` is set if a user had too many failed logins and is locked out for a set time. `LockoutEnabled` enables locking out a user

with too many failed sign in attempts. When the user has too many failed sign in attempts, `LockoutEnd` is set to a date in the future, and the user can't sign in until that date. If `LockoutEnabled` is false then a user is never locked out for too many failed sign in attempts. Per OWASP, *Temporary account lockout after several failed attempts is too simple of a target for DoS attacks against legitimate users*.

For more information on lock out, see OWASP Testing for Weak Lock Out Mechanism.

Apps migrating to Identity that wish to enable failed login lockout should set `LockoutEnabled` to true as part of the migration.

> **Note**
>
> Not all the field mappings resemble one-to-one relationships from Membership to ASP.NET Core Identity. The preceding table takes the default Membership User schema and maps it to the ASP.NET Core Identity schema. Any other custom fields that were used for Membership need to be mapped manually. In this mapping, there's no map for passwords, as both password criteria and password salts don't migrate between the two. **It's recommended to leave the password as null and to ask users to reset their passwords.** In ASP.NET Core Identity, `LockoutEnd` should be set to some date in the future if the user is locked out. This is shown in the migration script.

Roles

Expand table

Identity (`dbo.AspNetRoles`) column	Type	Membership (`dbo.aspnet_Roles`) column	Type
`Id`	`string`	`RoleId`	`string`
`Name`	`string`	`RoleName`	`string`
`NormalizedName`	`string`	`LoweredRoleName`	`string`

User Roles

Expand table

Identity (`dbo.AspNetUserRoles`) column	Type	Membership (`dbo.aspnet_UsersInRoles`) column	Type
`RoleId`	`string`	`RoleId`	`string`
`UserId`	`string`	`UserId`	`string`

Reference the preceding mapping tables when creating a migration script for *Users* and *Roles*. The following example assumes you have two databases on a database server. One database contains the existing ASP.NET Membership schema and data. The other *CoreIdentitySample* database was created using steps described earlier. Comments are included inline for more details.

SQL

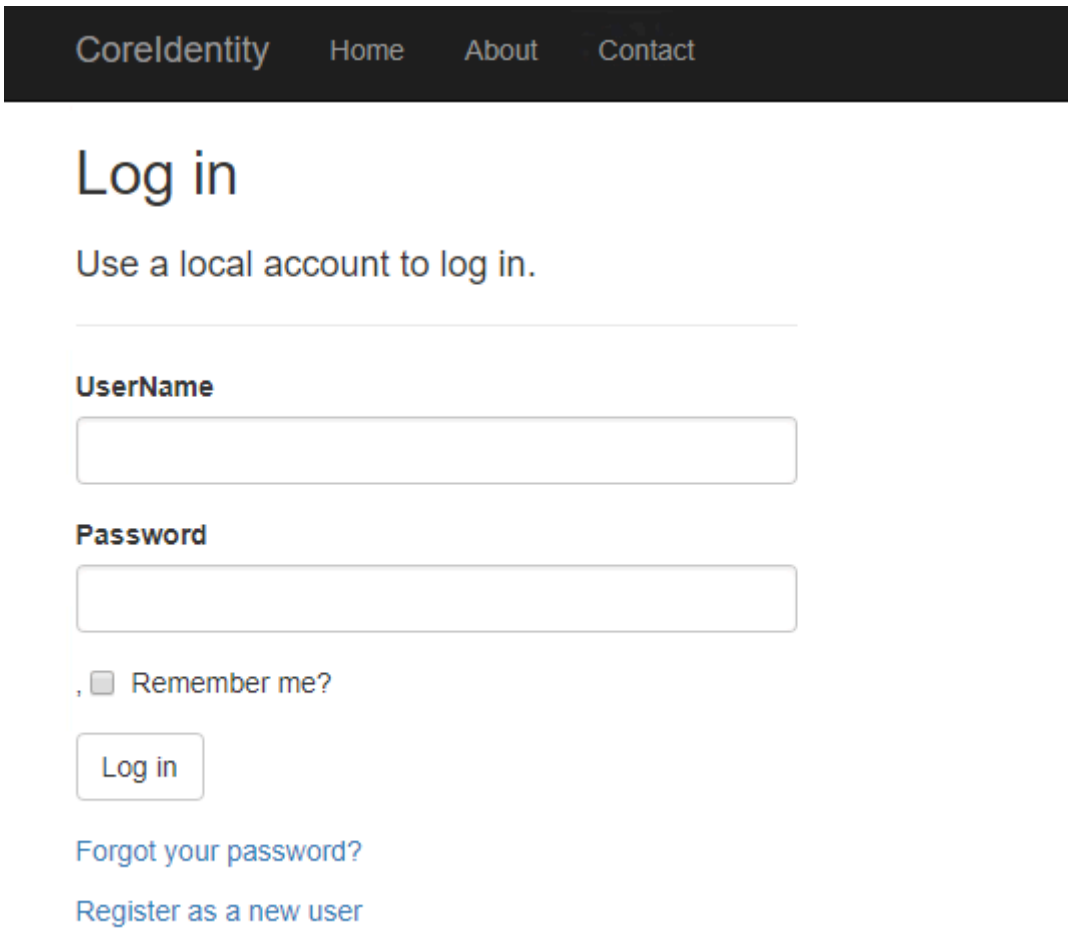
```
-- THIS SCRIPT NEEDS TO RUN FROM THE CONTEXT OF THE MEMBERSHIP DB
BEGIN TRANSACTION MigrateUsersAndRoles
USE aspnetdb

-- INSERT USERS
INSERT INTO CoreIdentitySample.dbo.AspNetUsers
            (Id,
             UserName,
             NormalizedUserName,
             PasswordHash,
             SecurityStamp,
             EmailConfirmed,
             PhoneNumber,
             PhoneNumberConfirmed,
             TwoFactorEnabled,
             LockoutEnd,
             LockoutEnabled,
             AccessFailedCount,
             Email,
             NormalizedEmail)
SELECT aspnet_Users.UserId,
       aspnet_Users.UserName,
       -- The NormalizedUserName value is upper case in ASP.NET Core
Identity
       UPPER(aspnet_Users.UserName),
       -- Creates an empty password since passwords don't map between the 2
schemas
       '',
       /*
        The SecurityStamp token is used to verify the state of an account
and
        is subject to change at any time. It should be initialized as a new
ID.
       */
       NewID(),
       /*
        EmailConfirmed is set when a new user is created and confirmed via
```

```
email.
        Users must have this set during migration to reset passwords.
       */
       1,
       aspnet_Users.MobileAlias,
       CASE
         WHEN aspnet_Users.MobileAlias IS NULL THEN 0
         ELSE 1
       END,
       -- 2FA likely wasn't setup in Membership for users, so setting as
false.
       0,
       CASE
         -- Setting lockout date to time in the future (1,000 years)
         WHEN aspnet_Membership.IsLockedOut = 1 THEN Dateadd(year, 1000,
                                                     Sysutcdatetime())
         ELSE NULL
       END,
       aspnet_Membership.IsLockedOut,
       /*
        AccessFailedAccount is used to track failed logins. This is stored
in
        Membership in multiple columns. Setting to 0 arbitrarily.
       */
       0,
       aspnet_Membership.Email,
       -- The NormalizedEmail value is upper case in ASP.NET Core Identity
       UPPER(aspnet_Membership.Email)
FROM   aspnet_Users
       LEFT OUTER JOIN aspnet_Membership
                    ON aspnet_Membership.ApplicationId =
                       aspnet_Users.ApplicationId
                       AND aspnet_Users.UserId = aspnet_Membership.UserId
       LEFT OUTER JOIN CoreIdentitySample.dbo.AspNetUsers
                    ON aspnet_Membership.UserId = AspNetUsers.Id
WHERE  AspNetUsers.Id IS NULL

-- INSERT ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetRoles(Id, Name)
SELECT RoleId, RoleName
FROM aspnet_Roles;

-- INSERT USER ROLES
INSERT INTO CoreIdentitySample.dbo.AspNetUserRoles(UserId, RoleId)
SELECT UserId, RoleId
FROM aspnet_UsersInRoles;

IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRANSACTION MigrateUsersAndRoles
    RETURN
  END

COMMIT TRANSACTION MigrateUsersAndRoles
```

After completion of the preceding script, the ASP.NET Core Identity app created earlier is populated with Membership users. Users need to change their passwords before logging in.

> **Note**
>
> If the Membership system had users with user names that didn't match their email address, changes are required to the app created earlier to accommodate this. The default template expects `UserName` and `Email` to be the same. For situations in which they're different, the login process needs to be modified to use `UserName` instead of `Email`.

In the `PageModel` of the Login Page, located at , remove the `[EmailAddress]` attribute from the *Email* property. Rename it to *UserName*. This requires a change wherever `EmailAddress` is mentioned, in the *View* and *PageModel*. The result looks like the following:



Next steps

In this tutorial, you learned how to port users from SQL membership to ASP.NET Core 2.0 Identity. For more information regarding ASP.NET Core Identity, see Introduction to Identity.

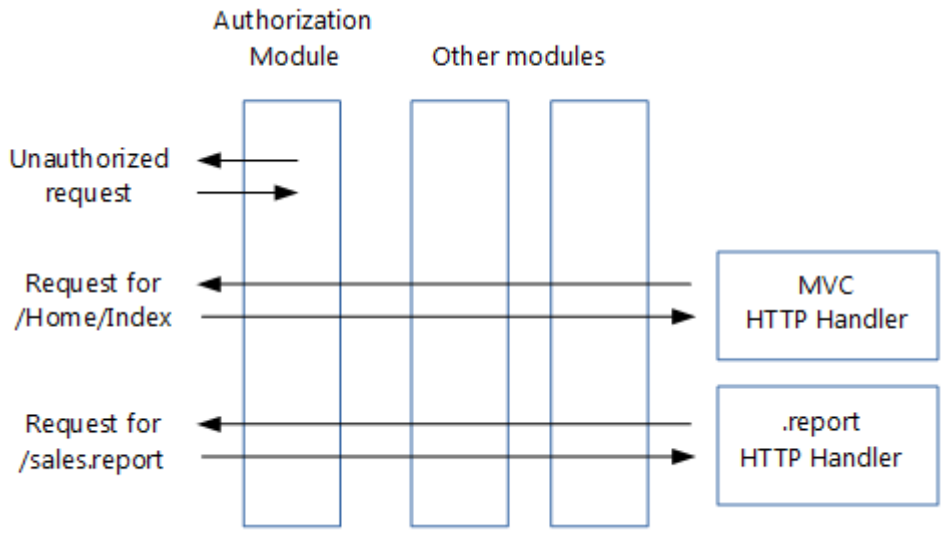
Migrate HTTP handlers and modules to ASP.NET Core middleware

Article • 03/22/2024

This article shows how to migrate existing ASP.NET HTTP modules and handlers from system.webserver to ASP.NET Core middleware.

Modules and handlers revisited

Before proceeding to ASP.NET Core middleware, let's first recap how HTTP modules and handlers work:



Handlers are:

- Classes that implement IHttpHandler
- Used to handle requests with a given file name or extension, such as *.report*
- Configured in *Web.config*

Modules are:

- Classes that implement IHttpModule
- Invoked for every request
- Able to short-circuit (stop further processing of a request)
- Able to add to the HTTP response, or create their own

- Configured in *Web.config*

The order in which modules process incoming requests is determined by:

1. A series events fired by ASP.NET, such as BeginRequest and AuthenticateRequest. For a complete list, see System.Web.HttpApplication. Each module can create a handler for one or more events.
2. For the same event, the order in which they're configured in *Web.config*.

In addition to modules, you can add handlers for the life cycle events to your `Global.asax.cs` file. These handlers run after the handlers in the configured modules.

From handlers and modules to middleware

Middleware are simpler than HTTP modules and handlers:

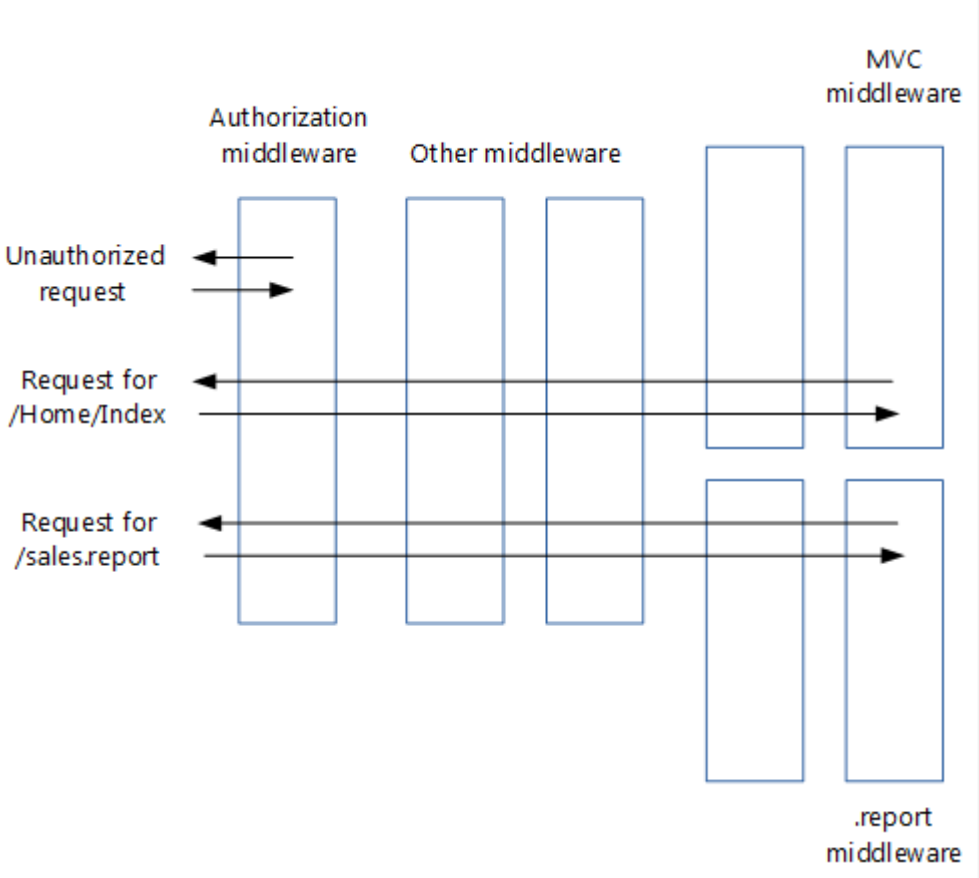
- Modules, handlers, `Global.asax.cs`, *Web.config* (except for IIS configuration) and the application life cycle are gone
- The roles of both modules and handlers have been taken over by middleware
- Middleware are configured using code rather than in *Web.config*
- Pipeline branching lets you send requests to specific middleware, based on not only the URL but also on request headers, query strings, etc.

Middleware are very similar to modules:

- Invoked in principle for every request
- Able to short-circuit a request, by not passing the request to the next middleware
- Able to create their own HTTP response

Middleware and modules are processed in a different order:

- Order of middleware is based on the order in which they're inserted into the request pipeline, while order of modules is mainly based on System.Web.HttpApplication events.
- Order of middleware for responses is the reverse from that for requests, while order of modules is the same for requests and responses
- See Create a middleware pipeline with IApplicationBuilder



Note how in the image above, the authentication middleware short-circuited the request.

Migrating module code to middleware

An existing HTTP module will look similar to this:

C#

```
// ASP.NET 4 module

using System;
using System.Web;

namespace MyApp.Modules
{
    public class MyModule : IHttpModule
    {
        public void Dispose()
        {
        }

        public void Init(HttpApplication application)
        {
            application.BeginRequest += (new
EventHandler(this.Application_BeginRequest));
            application.EndRequest += (new
EventHandler(this.Application_EndRequest));
```

```
        }

        private void Application_BeginRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the beginning of request
processing.
        }

        private void Application_EndRequest(Object source, EventArgs e)
        {
            HttpContext context = ((HttpApplication)source).Context;

            // Do something with context near the end of request processing.
        }
    }
}
```

As shown in the Middleware page, an ASP.NET Core middleware is a class that exposes an `Invoke` method taking an `HttpContext` and returning a `Task`. Your new middleware will look like this:

C#

```
// ASP.NET Core middleware

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyMiddleware
    {
        private readonly RequestDelegate _next;

        public MyMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            // Do something with context near the beginning of request
processing.

            await _next.Invoke(context);

            // Clean up.
        }
    }
```

```
    public static class MyMiddlewareExtensions
    {
        public static IApplicationBuilder UseMyMiddleware(this
IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyMiddleware>();
        }
    }
}
```

The preceding middleware template was taken from the section on writing middleware.

The *MyMiddlewareExtensions* helper class makes it easier to configure your middleware in your `Startup` class. The `UseMyMiddleware` method adds your middleware class to the request pipeline. Services required by the middleware get injected in the middleware's constructor.

Your module might terminate a request, for example if the user isn't authorized:

C#

```
// ASP.NET 4 module that may terminate the request

private void Application_BeginRequest(Object source, EventArgs e)
{
    HttpContext context = ((HttpApplication)source).Context;

    // Do something with context near the beginning of request processing.

    if (TerminateRequest())
    {
        context.Response.End();
        return;
    }
}
```

A middleware handles this by not calling `Invoke` on the next middleware in the pipeline. Keep in mind that this doesn't fully terminate the request, because previous middlewares will still be invoked when the response makes its way back through the pipeline.

C#

```
// ASP.NET Core middleware that may terminate the request

public async Task Invoke(HttpContext context)
{
    // Do something with context near the beginning of request processing.
```

```
    if (!TerminateRequest())
        await _next.Invoke(context);

    // Clean up.
}
```

When you migrate your module's functionality to your new middleware, you may find that your code doesn't compile because the `HttpContext` class has significantly changed in ASP.NET Core. Later on, you'll see how to migrate to the new ASP.NET Core HttpContext.

Migrating module insertion into the request pipeline

HTTP modules are typically added to the request pipeline using *Web.config*:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <modules>
      <add name="MyModule" type="MyApp.Modules.MyModule"/>
    </modules>
  </system.webServer>
</configuration>
```

Convert this by adding your new middleware to the request pipeline in your `Startup` class:

C#

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
```

```
            app.UseExceptionHandler("/Home/Error");
        }

        app.UseMyMiddleware();

        app.UseMyMiddlewareWithParams();

        var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptio
ns>();
        var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOpti
ons>();
        app.UseMyMiddlewareWithParams(myMiddlewareOptions);
        app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

        app.UseMyTerminatingMiddleware();

        // Create branch to the MyHandlerMiddleware.
        // All requests ending in .report will follow this branch.
        app.MapWhen(
            context => context.Request.Path.ToString().EndsWith(".report"),
            appBranch => {
                // ... optionally add more middleware to this branch
                appBranch.UseMyHandler();
            });

        app.MapWhen(
            context => context.Request.Path.ToString().EndsWith(".context"),
            appBranch => {
                appBranch.UseHttpContextDemoMiddleware();
            });

        app.UseStaticFiles();

        app.UseMvc(routes =>
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
```

The exact spot in the pipeline where you insert your new middleware depends on the event that it handled as a module (`BeginRequest`, `EndRequest`, etc.) and its order in your list of modules in *Web.config*.

As previously stated, there's no application life cycle in ASP.NET Core and the order in which responses are processed by middleware differs from the order used by modules. This could make your ordering decision more challenging.

If ordering becomes a problem, you could split your module into multiple middleware components that can be ordered independently.

Migrating handler code to middleware

An HTTP handler looks something like this:

C#

```
// ASP.NET 4 handler

using System.Web;

namespace MyApp.HttpHandlers
{
    public class MyHandler : IHttpHandler
    {
        public bool IsReusable { get { return true; } }

        public void ProcessRequest(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            context.Response.Output.Write(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.QueryString["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }
}
```

In your ASP.NET Core project, you would translate this to a middleware similar to this:

C#

```
// ASP.NET Core middleware migrated from a handler

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
```

```
using System.Threading.Tasks;

namespace MyApp.Middleware
{
    public class MyHandlerMiddleware
    {

        // Must have constructor with this signature, otherwise exception at
run time
        public MyHandlerMiddleware(RequestDelegate next)
        {
            // This is an HTTP Handler, so no need to store next
        }

        public async Task Invoke(HttpContext context)
        {
            string response = GenerateResponse(context);

            context.Response.ContentType = GetContentType();
            await context.Response.WriteAsync(response);
        }

        // ...

        private string GenerateResponse(HttpContext context)
        {
            string title = context.Request.Query["title"];
            return string.Format("Title of the report: {0}", title);
        }

        private string GetContentType()
        {
            return "text/plain";
        }
    }

    public static class MyHandlerExtensions
    {
        public static IApplicationBuilder UseMyHandler(this
IApplicationBuilder builder)
        {
            return builder.UseMiddleware<MyHandlerMiddleware>();
        }
    }
}
```

This middleware is very similar to the middleware corresponding to modules. The only real difference is that here there's no call to `_next.Invoke(context)`. That makes sense, because the handler is at the end of the request pipeline, so there will be no next middleware to invoke.

Migrating handler insertion into the request pipeline

Configuring an HTTP handler is done in *Web.config* and looks something like this:

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--ASP.NET 4 web.config-->
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandler" verb="*" path="*.report"
type="MyApp.HttpHandlers.MyHandler" resourceType="Unspecified"
preCondition="integratedMode"/>
    </handlers>
  </system.webServer>
</configuration>
```

You could convert this by adding your new handler middleware to the request pipeline in your `Startup` class, similar to middleware converted from modules. The problem with that approach is that it would send all requests to your new handler middleware. However, you only want requests with a given extension to reach your middleware. That would give you the same functionality you had with your HTTP handler.

One solution is to branch the pipeline for requests with a given extension, using the `MapWhen` extension method. You do this in the same `Configure` method where you add the other middleware:

C#

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseMyMiddleware();
```

```
    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddlewareOptio
ns>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewareOpti
ons>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

`MapWhen` takes these parameters:

1. A lambda that takes the `HttpContext` and returns `true` if the request should go down the branch. This means you can branch requests not just based on their extension, but also on request headers, query string parameters, etc.
2. A lambda that takes an `IApplicationBuilder` and adds all the middleware for the branch. This means you can add additional middleware to the branch in front of your handler middleware.

Middleware added to the pipeline before the branch will be invoked on all requests; the branch will have no impact on them.

Loading middleware options using the options pattern

Some modules and handlers have configuration options that are stored in *Web.config*. However, in ASP.NET Core a new configuration model is used in place of *Web.config*.

The new configuration system gives you these options to solve this:

- Directly inject the options into the middleware, as shown in the next section.
- Use the options pattern:

1. Create a class to hold your middleware options, for example:

   C#

   ```
   public class MyMiddlewareOptions
   {
       public string Param1 { get; set; }
       public string Param2 { get; set; }
   }
   ```

2. Store the option values

   The configuration system allows you to store option values anywhere you want. However, most sites use `appsettings.json`, so we'll take that approach:

   JSON

   ```
   {
     "MyMiddlewareOptionsSection": {
       "Param1": "Param1Value",
       "Param2": "Param2Value"
     }
   }
   ```

   MyMiddlewareOptionsSection here is a section name. It doesn't have to be the same as the name of your options class.

3. Associate the option values with the options class

   The options pattern uses ASP.NET Core's dependency injection framework to associate the options type (such as `MyMiddlewareOptions`) with a `MyMiddlewareOptions` object that has the actual options.

   Update your `Startup` class:

a. If you're using `appsettings.json`, add it to the configuration builder in the `Startup` constructor:

```
C#

public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true,
reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json",
optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

b. Configure the options service:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
    services.AddMvc();
}
```

c. Associate your options with your options class:

```
C#

public void ConfigureServices(IServiceCollection services)
{
    // Setup options service
    services.AddOptions();

    // Load options from section "MyMiddlewareOptionsSection"
    services.Configure<MyMiddlewareOptions>(
        Configuration.GetSection("MyMiddlewareOptionsSection"));

    // Add framework services.
```

```
        services.AddMvc();
    }
```

4. Inject the options into your middleware constructor. This is similar to injecting options into a controller.

```
C#

public class MyMiddlewareWithParams
{
    private readonly RequestDelegate _next;
    private readonly MyMiddlewareOptions _myMiddlewareOptions;

    public MyMiddlewareWithParams(RequestDelegate next,
        IOptions<MyMiddlewareOptions> optionsAccessor)
    {
        _next = next;
        _myMiddlewareOptions = optionsAccessor.Value;
    }

    public async Task Invoke(HttpContext context)
    {
        // Do something with context near the beginning of request
processing
        // using configuration in _myMiddlewareOptions

        await _next.Invoke(context);

        // Do something with context near the end of request processing
        // using configuration in _myMiddlewareOptions
    }
}
```

The UseMiddleware extension method that adds your middleware to the `IApplicationBuilder` takes care of dependency injection.

This isn't limited to `IOptions` objects. Any other object that your middleware requires can be injected this way.

Loading middleware options through direct injection

The options pattern has the advantage that it creates loose coupling between options values and their consumers. Once you've associated an options class with the actual options values, any other class can get access to the options through the dependency injection framework. There's no need to pass around options values.

This breaks down though if you want to use the same middleware twice, with different options. For example an authorization middleware used in different branches allowing different roles. You can't associate two different options objects with the one options class.

The solution is to get the options objects with the actual options values in your `Startup` class and pass those directly to each instance of your middleware.

1. Add a second key to `appsettings.json`

   To add a second set of options to the `appsettings.json` file, use a new key to uniquely identify it:

   JSON

   ```
   {
     "MyMiddlewareOptionsSection2": {
       "Param1": "Param1Value2",
       "Param2": "Param2Value2"
     },
     "MyMiddlewareOptionsSection": {
       "Param1": "Param1Value",
       "Param2": "Param2Value"
     }
   }
   ```

2. Retrieve options values and pass them to middleware. The `Use...` extension method (which adds your middleware to the pipeline) is a logical place to pass in the option values:

   C#

   ```
   public void Configure(IApplicationBuilder app, IHostingEnvironment env,
   ILoggerFactory loggerFactory)
   {
       loggerFactory.AddConsole(Configuration.GetSection("Logging"));
       loggerFactory.AddDebug();

       if (env.IsDevelopment())
       {
           app.UseDeveloperExceptionPage();
           app.UseBrowserLink();
       }
       else
       {
           app.UseExceptionHandler("/Home/Error");
       }

       app.UseMyMiddleware();
   ```

```
    app.UseMyMiddlewareWithParams();

    var myMiddlewareOptions =
Configuration.GetSection("MyMiddlewareOptionsSection").Get<MyMiddleware
Options>();
    var myMiddlewareOptions2 =
Configuration.GetSection("MyMiddlewareOptionsSection2").Get<MyMiddlewar
eOptions>();
    app.UseMyMiddlewareWithParams(myMiddlewareOptions);
    app.UseMyMiddlewareWithParams(myMiddlewareOptions2);

    app.UseMyTerminatingMiddleware();

    // Create branch to the MyHandlerMiddleware.
    // All requests ending in .report will follow this branch.
    app.MapWhen(
        context => context.Request.Path.ToString().EndsWith(".report"),
        appBranch => {
            // ... optionally add more middleware to this branch
            appBranch.UseMyHandler();
        });

    app.MapWhen(
        context =>
context.Request.Path.ToString().EndsWith(".context"),
        appBranch => {
            appBranch.UseHttpContextDemoMiddleware();
        });

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

3. Enable middleware to take an options parameter. Provide an overload of the `Use...` extension method (that takes the options parameter and passes it to `UseMiddleware`). When `UseMiddleware` is called with parameters, it passes the parameters to your middleware constructor when it instantiates the middleware object.

```
C#

public static class MyMiddlewareWithParamsExtensions
{
    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder)
```

```
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>();
    }

    public static IApplicationBuilder UseMyMiddlewareWithParams(
        this IApplicationBuilder builder, MyMiddlewareOptions
myMiddlewareOptions)
    {
        return builder.UseMiddleware<MyMiddlewareWithParams>(
            new OptionsWrapper<MyMiddlewareOptions>
(myMiddlewareOptions));
    }
}
```

Note how this wraps the options object in an `OptionsWrapper` object. This implements `IOptions`, as expected by the middleware constructor.

Migrating to the new HttpContext

You saw earlier that the `Invoke` method in your middleware takes a parameter of type `HttpContext`:

C#

```
public async Task Invoke(HttpContext context)
```

`HttpContext` has significantly changed in ASP.NET Core. This section shows how to translate the most commonly used properties of System.Web.HttpContext to the new `Microsoft.AspNetCore.Http.HttpContext`.

HttpContext

HttpContext.Items translates to:

C#

```
IDictionary<object, object> items = httpContext.Items;
```

Unique request ID (no System.Web.HttpContext counterpart)

Gives you a unique id for each request. Very useful to include in your logs.

C#

```
string requestId = httpContext.TraceIdentifier;
```

HttpContext.Request

HttpContext.Request.HttpMethod translates to:

C#

```
string httpMethod = httpContext.Request.Method;
```

HttpContext.Request.QueryString translates to:

C#

```
IQueryCollection queryParameters = httpContext.Request.Query;

// If no query parameter "key" used, values will have 0 items
// If single value used for a key (...?key=v1), values will have 1 item
("v1")
// If key has multiple values (...?key=v1&key=v2), values will have 2 items
("v1" and "v2")
IList<string> values = queryParameters["key"];

// If no query parameter "key" used, value will be ""
// If single value used for a key (...?key=v1), value will be "v1"
// If key has multiple values (...?key=v1&key=v2), value will be "v1,v2"
string value = queryParameters["key"].ToString();
```

HttpContext.Request.Url and **HttpContext.Request.RawUrl** translate to:

C#

```
// using Microsoft.AspNetCore.Http.Extensions;
var url = httpContext.Request.GetDisplayUrl();
```

HttpContext.Request.IsSecureConnection translates to:

C#

```
var isSecureConnection = httpContext.Request.IsHttps;
```

HttpContext.Request.UserHostAddress translates to:

C#

```
var userHostAddress = httpContext.Connection.RemoteIpAddress?.ToString();
```

HttpContext.Request.Cookies translates to:

C#

```
IRequestCookieCollection cookies = httpContext.Request.Cookies;
string unknownCookieValue = cookies["unknownCookie"]; // will be null (no
exception)
string knownCookieValue = cookies["cookie1name"];     // will be actual
value
```

HttpContext.Request.RequestContext.RouteData translates to:

C#

```
var routeValue = httpContext.GetRouteValue("key");
```

HttpContext.Request.Headers translates to:

C#

```
// using Microsoft.AspNetCore.Http.Headers;
// using Microsoft.Net.Http.Headers;

IHeaderDictionary headersDictionary = httpContext.Request.Headers;

// GetTypedHeaders extension method provides strongly typed access to many
headers
var requestHeaders = httpContext.Request.GetTypedHeaders();
CacheControlHeaderValue cacheControlHeaderValue =
requestHeaders.CacheControl;

// For unknown header, unknownheaderValues has zero items and
unknownheaderValue is ""
IList<string> unknownheaderValues = headersDictionary["unknownheader"];
string unknownheaderValue = headersDictionary["unknownheader"].ToString();

// For known header, knownheaderValues has 1 item and knownheaderValue is
the value
IList<string> knownheaderValues =
headersDictionary[HeaderNames.AcceptLanguage];
string knownheaderValue =
headersDictionary[HeaderNames.AcceptLanguage].ToString();
```

HttpContext.Request.UserAgent translates to:

C#

```
string userAgent = headersDictionary[HeaderNames.UserAgent].ToString();
```

HttpContext.Request.UrlReferrer translates to:

C#

```
string urlReferrer = headersDictionary[HeaderNames.Referer].ToString();
```

HttpContext.Request.ContentType translates to:

C#

```
// using Microsoft.Net.Http.Headers;

MediaTypeHeaderValue mediaHeaderValue = requestHeaders.ContentType;
string contentType = mediaHeaderValue?.MediaType.ToString();   // ex.
application/x-www-form-urlencoded
string contentMainType = mediaHeaderValue?.Type.ToString();    // ex.
application
string contentSubType = mediaHeaderValue?.SubType.ToString();  // ex. x-www-
form-urlencoded

System.Text.Encoding requestEncoding = mediaHeaderValue?.Encoding;
```

HttpContext.Request.Form translates to:

C#

```
if (httpContext.Request.HasFormContentType)
{
    IFormCollection form;

    form = httpContext.Request.Form; // sync
    // Or
    form = await httpContext.Request.ReadFormAsync(); // async

    string firstName = form["firstname"];
    string lastName = form["lastname"];
}
```

> ⚠ **Warning**
>
> Read form values only if the content sub type is *x-www-form-urlencoded* or *form-data*.

HttpContext.Request.InputStream translates to:

C#

```
string inputBody;
using (var reader = new System.IO.StreamReader(
    httpContext.Request.Body, System.Text.Encoding.UTF8))
{
    inputBody = reader.ReadToEnd();
}
```

> ⚠ **Warning**
>
> Use this code only in a handler type middleware, at the end of a pipeline.
>
> You can read the raw body as shown above only once per request. Middleware trying to read the body after the first read will read an empty body.
>
> This doesn't apply to reading a form as shown earlier, because that's done from a buffer.

HttpContext.Response

HttpContext.Response.Status and **HttpContext.Response.StatusDescription** translate to:

C#

```
// using Microsoft.AspNetCore.Http;
httpContext.Response.StatusCode = StatusCodes.Status200OK;
```

HttpContext.Response.ContentEncoding and **HttpContext.Response.ContentType** translate to:

C#

```
// using Microsoft.Net.Http.Headers;
var mediaType = new MediaTypeHeaderValue("application/json");
mediaType.Encoding = System.Text.Encoding.UTF8;
httpContext.Response.ContentType = mediaType.ToString();
```

HttpContext.Response.ContentType on its own also translates to:

C#

```
httpContext.Response.ContentType = "text/html";
```

HttpContext.Response.Output translates to:

```
C#

string responseContent = GetResponseContent();
await httpContext.Response.WriteAsync(responseContent);
```

HttpContext.Response.TransmitFile

Serving up a file is discussed in Request Features in ASP.NET Core.

HttpContext.Response.Headers

Sending response headers is complicated by the fact that if you set them after anything has been written to the response body, they will not be sent.

The solution is to set a callback method that will be called right before writing to the response starts. This is best done at the start of the `Invoke` method in your middleware. It's this callback method that sets your response headers.

The following code sets a callback method called `SetHeaders`:

```
C#

public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetHeaders` callback method would look like this:

```
C#

// using Microsoft.AspNet.Http.Headers;
// using Microsoft.Net.Http.Headers;

private Task SetHeaders(object context)
{
    var httpContext = (HttpContext)context;

    // Set header with single value
    httpContext.Response.Headers["ResponseHeaderName"] = "headerValue";

    // Set header with multiple values
    string[] responseHeaderValues = new string[] { "headerValue1",
```

```
"headerValue1" };
    httpContext.Response.Headers["ResponseHeaderName"] =
responseHeaderValues;

    // Translating ASP.NET 4's HttpContext.Response.RedirectLocation
    httpContext.Response.Headers[HeaderNames.Location] =
"http://www.example.com";
    // Or
    httpContext.Response.Redirect("http://www.example.com");

    // GetTypedHeaders extension method provides strongly typed access to
many headers
    var responseHeaders = httpContext.Response.GetTypedHeaders();

    // Translating ASP.NET 4's HttpContext.Response.CacheControl
    responseHeaders.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = new System.TimeSpan(365, 0, 0, 0)
        // Many more properties available
    };

    // If you use .NET Framework 4.6+, Task.CompletedTask will be a bit
faster
    return Task.FromResult(0);
}
```

HttpContext.Response.Cookies

Cookies travel to the browser in a *Set-Cookie* response header. As a result, sending cookies requires the same callback as used for sending response headers:

C#

```
public async Task Invoke(HttpContext httpContext)
{
    // ...
    httpContext.Response.OnStarting(SetCookies, state: httpContext);
    httpContext.Response.OnStarting(SetHeaders, state: httpContext);
```

The `SetCookies` callback method would look like the following:

C#

```
private Task SetCookies(object context)
{
    var httpContext = (HttpContext)context;

    IResponseCookies responseCookies = httpContext.Response.Cookies;

    responseCookies.Append("cookie1name", "cookie1value");
    responseCookies.Append("cookie2name", "cookie2value",
```

```
        new CookieOptions { Expires = System.DateTime.Now.AddDays(5),
HttpOnly = true });

    // If you use .NET Framework 4.6+, Task.CompletedTask will be a bit
faster
    return Task.FromResult(0);
}
```

Additional resources

- Incremental HTTP module migration
- HTTP Handlers and HTTP Modules Overview
- Configuration
- Application Startup
- Middleware

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback